**深度理解跳跃链表：一种基于概率选择的平衡树 - CSDN博客**

| | |
|---|---|
| **笔记本：** | data-structure |
| **创建时间：** | 2018/8/24 10:53 |
| **标签：** | skiplist |
| **URL：** | https://blog.csdn.net/u013011841/article/details/39158585 |

# 原 深度理解跳跃链表：一种基于概率选择的平衡树

2014年09月09日 17:29:22　阅读数：1966　标签：跳跃链表　更多



# 跳跃链表：一种基于概率选择的平衡树

作者：林子
Blog：**http://blog.csdn.net/u013011841**
时间：2014年9月
声明：欢迎指出错误，转载不要去掉出处

### 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。

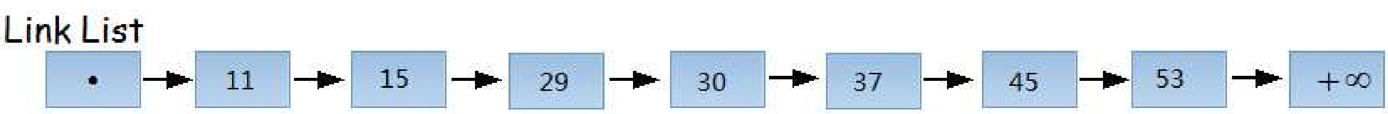
跳跃链表的最大优势在于无论是查找、插入和删除都是O(logn)，不过由于跳跃链表的操作是基于概率形成的，那么它操作复杂度大于O(logn)的概率为，可以看出当n越大的时候失败的概率越小。

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?

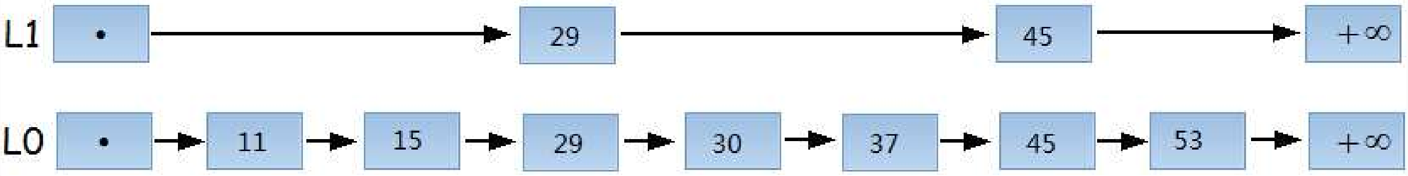
我们先看看一个普通的链表




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

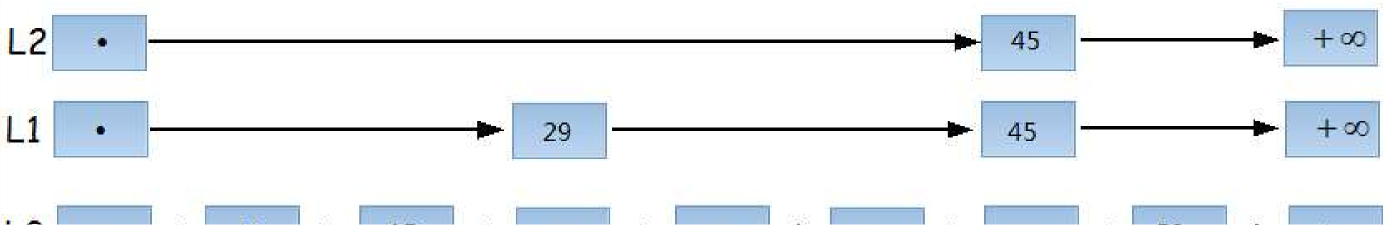
$$\min mize \quad |L_1| + \frac{n}{|L_1|}$$

我们对|L1|的长度求导得

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

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 $k\sqrt[k]{n}$

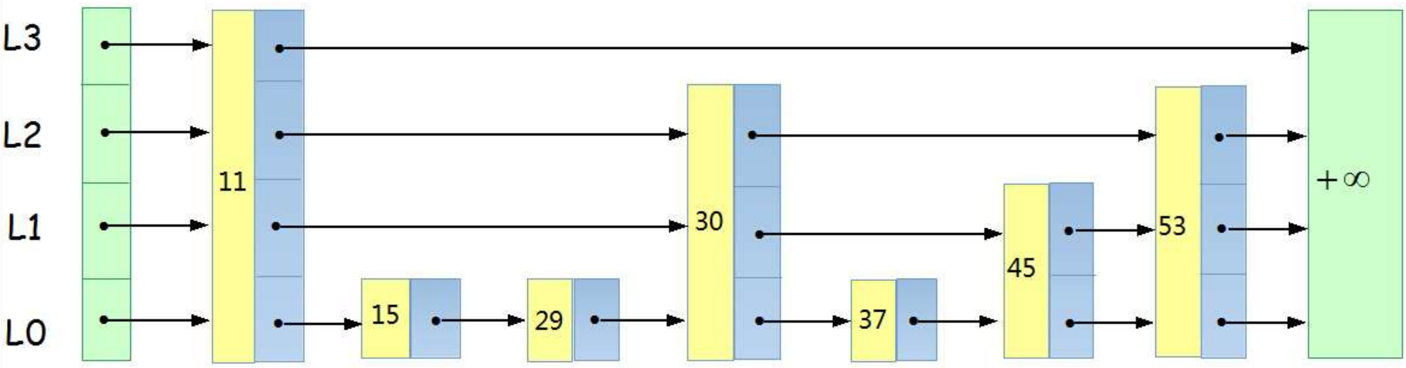
我们这里取k=logn，代入的查找次数为2logn.

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 -∞(其实不需要)
- 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。




结点结构源代码

```
struct node
{
        int key;
        struct node *forward[MAXlevel];
};
```

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. 从最上层的链（Lh）的开头开始

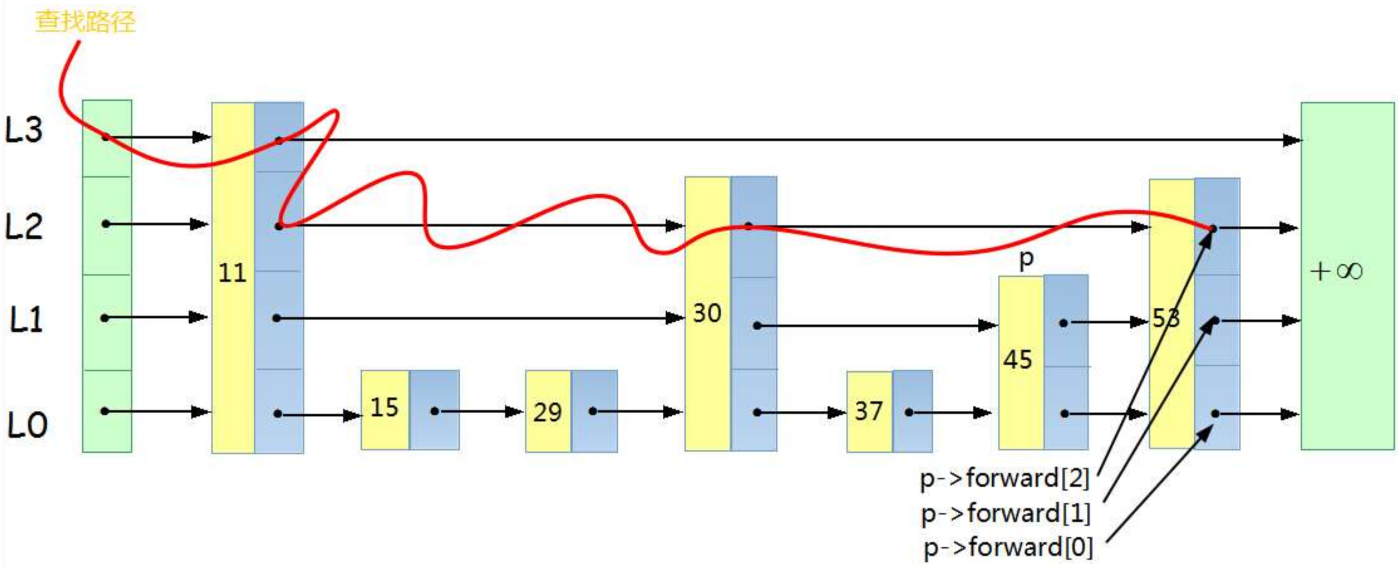
2. 假设当前位置为p，它向右指向的节点为q（p与q不一定相邻），且q的值为y。将y与x作比较

   (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

3. 如果当前位置在最底层的链中（L0），且还要往下移动的话，则输出查询失败




元素53的查找路径

```
struct node* search(struct node*head, int key, int level)
{
        int i;
        struct node *p;
        p=head;
        for(i=level;i>=0;i--)
                while((p->forward[i]!=0)&&(p->forward[i]->key<key))
                        p=p->foward[i];
        p==p->forward[0];//回到0级链，当前p或者空或者指向比搜索关键字小
的前一个结点
        if(p==0)//这时若p为空则推出检索,返回0
                return 0;
        else if(p->key==key)//找到，返回成功
                return p;
        else
                return 0;//否则仍然检索失败,返回0
}
```

## 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如果r小于一个常数p(通常取0.25或0.5)，则执行方案A，否则，执行方案B.

```
int randX(int &level)
{
        int i,j,t;
        t=rand();
        for(i=0;j=2;i<MAXlevel;i++,j+=j)
                if(t>RAND_MAX/j)
                        break;
        if(i>level)
                level=i;
        return i;
}
```

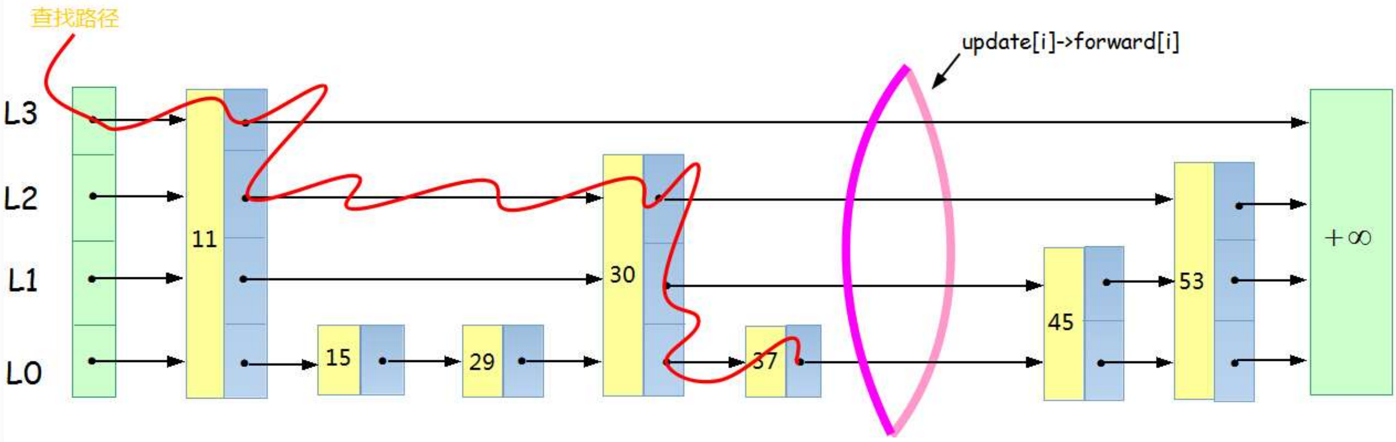
RADN_MAX为随机函数rand()能随机到的最大值,每个结点的高度都由该随机函数决定

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。

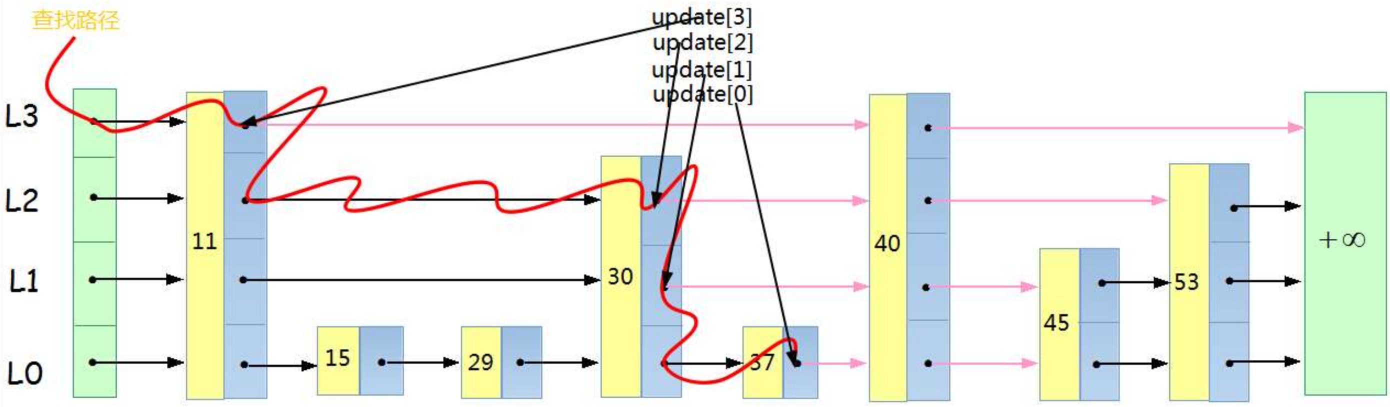
我们来看一个例子：

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




紫色的箭头表示更新过的指针

```
void insert(struct node*head,int key,int &level)
{
        struct node*p,*update[MAXlevel];
        int i,newlevel;
        p=head;
        newlevel=randX(level);
        for(i=level;i>=0;i---)
        {
                while((p->forward[i]!=0)&&(p->forward[i]->key<key))
                        p=p->forward[i];
                update[i]=p;//update[i]记录了搜索过程中在各层中走过的最大的结点位置
        }//设置新结点
        p=new(struct node);
        p->key=key;
        for(i=0;i<MAXlevel;i++)
                p->forward[i]=0;
        for(i=0;i<=newlevel;i++)//插入是从最高的newlevel层直至0层
```

```
        {
                p->forward[i]=update[i]->forward[i];//插入到分配到的层数
                update[i]->forward[i]=p;
        }
}
```

## 跳跃链表的删除

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：

在跳跃表中查找到这个元素的位置，如果未找到，则退出

将该元素所在整列从表中删除

将多余的“空链”删除

删除的过程即为插入的逆操作

```
int deletenode(struct node*head,int &level)
{
        int delkey;
        struct node*r;
        cout<<"请输入要删除的数字:";
        cin>>delkey;
        r=search(head,delkey,level);
        if(r)
        {
                int i=level;
                struct node*p,*q;
                while(i>=0)
                {
                        p=q=head;
                        while(p!=r&&p!=null)
                        {
                                q=p;
                                p=p->forward[i];
                        }
                        if(p)
                        {
```

```
                        {
                                if(i==level&&q=head&&p->forward[i]==0)
                                        level--;
                                else
                                        q->forward[i]=p->forward[i];
                        }
                        i--;
                }
                delete r;
                return 1;
        }
        else
                return 0;
}
```

## 跳跃链表的搜索时间复杂度为O(logn)

**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(logn).

**高概率:**事件E以很高的概率发生意味着对于a>=1,存在一个合适的常数使得事件E发生的概率Pr{E}>=1-O(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*O(logn),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(logn),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Ei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

$$
\begin{aligned}
&\Pr\{E_1 \cup E_2 \cup \cdots \cup E_k\} \\
&\le \Pr\{E_1\} + \Pr\{E_2\} + \ldots + \Pr\{E_k\} \\
&= k \times \frac{1}{n^{\alpha}} \\
&\xlongequal{令k=n^c} \frac{1}{n^{\alpha-c}}
\end{aligned}
$$

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(logn)

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

$$n \times \Pr\{\text{某一结点抬升的层数} \geq c \cdot \log_2 n\}$$

$$\leq n \cdot \left(\frac{1}{2}\right)^{c \cdot \log_2 n}$$

$$= n \cdot \left(\frac{1}{2^{\log_2 n^c}}\right)$$

$$= \frac{n}{n^c}$$

$$= \frac{1}{n^{c-1}}$$

令a=c-1,则只要a>=1时，就有高概率的可能使得层数为O(logn)

## 跳跃链表单次查找复杂度大于O(logn)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clogn层,那么扔正面的次数为clogn-1次.

我们假设扔了clogn个正面，超过10*clogn次是反面，则有

$$\Pr\{\text{扔了} c\log n \text{次正面，超过} 10c\log n \text{次反面}\}$$

$$\leq \binom{10c\log n}{c\log n}\left(\frac{1}{2}\right)^{9c\log n}, \text{因为} \binom{y}{x} \leq \left(e\frac{y}{x}\right)^{x}$$

$$\le e\left(\frac{10c\log n}{c\log n}\right)^{c\log n}\left(\frac{1}{2}\right)^{9\log n}$$

$$=\frac{(10e)^{c\log n}}{2^{9c\log n}}$$

$$=\frac{2^{\log(10e)c\log n}}{2^{9c\log n}}$$

$$=2^{[\log(10e)-9]c\log n}$$

$$=\frac{1}{2^{[9-\log(10e)]c\log n}}$$

$$\underline{\underline{令\alpha=[9-\log(10e)]c}}\qquad\frac{1}{n^{\alpha}}$$

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10clogn的概率随着10的增长变得越来越.所以全部的操作都在10logn以内，我们使用k替代10作为常数，即查找次数为klogn，为O(logn).