# 第 3 章　伸展树与跳表

**字典**（dictionary）是词条的集合，词条包括**关键字**（key）和其他信息。字典作为一种数据结构，主要包括搜索、插入和删除等基本运算，所以字典是数据元素（词条）的动态集（dynamic set）。用于表示字典的数据结构很多，**线性表**（linear list）、**散列表**（hash table）和**搜索树**（search tree）都可用于表示字典。本章讨论两种用于表示字典的高级数据结构：伸展树和跳表。

伸展树具有较好的平均分摊代价（$O(\log n)$），跳表利用随机性可获得较好的平均情况时间复杂度（$O(\log n)$）。这两种数据结构的性能分析，在数据结构的效率分析中具有很好的代表性。

## 3.1　伸展树

### 3.1.1　二叉搜索树

**二叉搜索树**（binary search tree）[①]是一棵二叉树，它要求根结点的左子树上所有结点的值都小于根结点的值，右子树上所有结点的值都大于根结点的值，并且左、右子树都是二叉搜索树。二叉搜索树用于表示动态集，实现动态集上定义的下列基本运算：

ResultCode Search(K key, T& x)const

**后置条件**：在集合中搜索关键字值为 key 的元素。若存在该元素，则将其值赋给 x，返回 Success；否则，返回 NotPresent。

ResultCode Insert(T x)

**后置条件**：在集合中搜索关键字值为 x.key 的元素。若存在该元素，则返回 Duplicate；否则，若集合已满，则返回 Overflow，若未满，则插入 x，并返回 Success。

ResultCode Remove(K key)

**后置条件**：在集合中搜索关键字值为 key 的元素。若存在该元素，则从集合中删除之，返回 Success；否则，返回 NotPresent。

**函数返回类型**：

enum ResultCode{ Underflow, Overflow, Success, Duplicate, NotPresent, …};

字典可用二叉搜索树表示，但该结构容易出现退化树形，使得搜索和修改代价增大。**二叉平衡树**（binary balanced tree）是一种平衡搜索树，它需在每次插入或删除元素之后，按规则重新平衡树形，使之始终保持平衡，从而限制树形的高度，避免退化。平衡搜索树能保证好的性能，但也增加了实现难度。

伸展树由 Sleator 和 Tarjan 于 1985 年提出，它是一种自调节搜索树。若对伸展树执行一系列运算，会有良好的时间性能。在伸展树上，执行一个包含 $m$ 个运算（搜索、插入和删除）的序列，总的时间复杂度为 $O(m\log n)$，因而有良好的平均分摊代价。伸展树被认为是平衡搜索树很好的替代结构。

### 3.1.2　自调节树和伸展树

**伸展树**（splay tree）是一棵二叉搜索树，它要求每访问一个元素后，将最新访问的元素移至

① 二叉搜索树也称二叉排序树，有关二叉搜索树和二叉平衡树的更详细知识见文献[16～20]。

二叉搜索树的根结点，从而保证经常被访问的元素靠近根结点，而较少访问的元素位于该搜索树的下层，所以这是一种**自调节搜索树**（self-adjusting search tree）。将一个元素移至根结点的操作称为一次**伸展**（splay）。

事实上，对伸展树，并不仅仅在成功搜索一个元素后需要做一次伸展，在插入和删除之后同样需要做一次伸展操作。伸展树的每次单独的伸展不一定会产生一棵更平衡的树，但执行一系列运算（搜索、插入和删除）时，在每次运算后加一次伸展操作，可在总体上使得伸展树趋向于平衡。

稍后的分析表明，在伸展树中，虽然某次运算（搜索、插入或删除）可能很费时，需要 $O(n)$ 时间，但执行一个包含 $m$（$m \geqslant n$）个运算的长运算序列，花费的总时间为 $O(m\log n)$（$n$ 是树中元素个数），每个运算的平均分摊代价为 $O(\log n)$。伸展操作的重要意义还在于，它可使访问频率较高的元素靠近根结点，较少访问的元素位于树的下层，它是一种自调节树。

### 3.1.3 伸展操作

伸展树的搜索、插入和删除运算的算法与普通二叉搜索树完全相同，只是在每个运算后，需紧跟一次伸展操作。伸展操作的作用在于将树中某个结点 $x$ 移至根结点，这个结点称为**伸展结点**（splay node）。伸展操作结束，伸展结点成为树的根结点。可以按下列方式来确定伸展结点。

（1）搜索运算：搜索成功的结点 $x$ 为伸展结点。

（2）插入运算：新插入的结点 $x$ 为伸展结点。

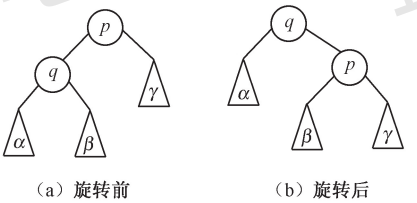
（3）删除运算：被删除的结点 $x$ 的双亲（结点）为伸展结点。

（4）若上述运算失败终止，则搜索过程中遇到的最后一个结点为伸展结点。

由于对伸展树的每次运算结束总会将最近访问的结点移至根结点，这样可使频繁访问的结点紧靠根结点，很少访问的结点远离根结点。

一次伸展操作由一组**旋转**（rotation）动作组成，可分为**单一旋转**（single rotation）和**双重旋转**（double rotation）两类。伸展操作结束，伸展结点被移至根结点处。

设 $q$ 是本次伸展操作的伸展结点。下面分两种情况讨论如何实现伸展操作的旋转动作。



（a）旋转前　　（b）旋转后

图 3-1　伸展树的 zig 旋转

（1）单一旋转

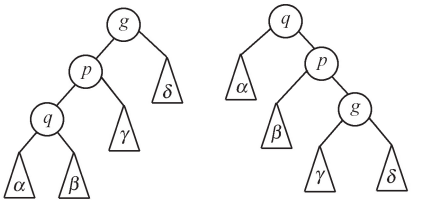
若 $q$ 是 $p$ 的左孩子（结点）或 $q$ 是 $p$ 的右孩子，则执行单一旋转。前者称为 **zig 旋转**（右旋转），后者称为 **zag 旋转**（左旋转）。图 3-1 所示为 zig 旋转。经过一次单一旋转，树的高度并未减小，只是将伸展结点向上移了一层。

（2）双重旋转

第一种双重旋转称为一字旋转。如果伸展结点 $q$ 是其祖父（结点）的左孩子的左孩子，或是其祖父的右孩子的右孩子，则执行双重旋转的一字旋转。前者称为 **zigzig 旋转**，后者称为 **zagzag 旋转**。图 3-2 所示为 zigzig 旋转。经过两个旋转步后，伸展结点 $q$ 的位置提升到原来 $g$ 的位置。经过一次一字旋转，树的高度并未减小，只是把伸展结点 $q$ 的位置向上移了两层。

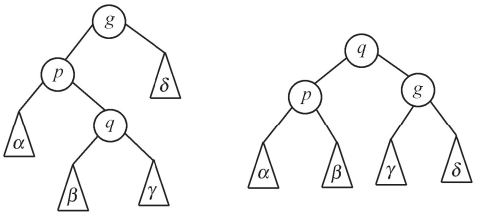
第二种双重旋转称为之字旋转。如果伸展结点 $q$ 是其祖父的左孩子的右孩子，或是其祖父的右孩子的左孩子，则执行双重旋转的之字旋转。前者称为 **zigzag 旋转**，后者称为 **zagzig 旋转**。图 3-3 所示为 zigzag 旋转。经过两个旋转步后，伸展结点 $q$ 的位置提升到原来 $g$ 的位置。经过一次之字旋转，树的高度减 1，且伸展结点 $q$ 的位置向上移，离根的距离减少了两层。

从前面的讨论可知，一次双重旋转将伸展结点的层次提升两层，而一次单一旋转只将伸展结点提升一层。所以为了将伸展结点提升为根结点，可能需要进行多次旋转，直到将伸展结点提升为根结点为止。




（a）旋转前　　（b）旋转后

图 3-2　伸展树的 zigzig 旋转



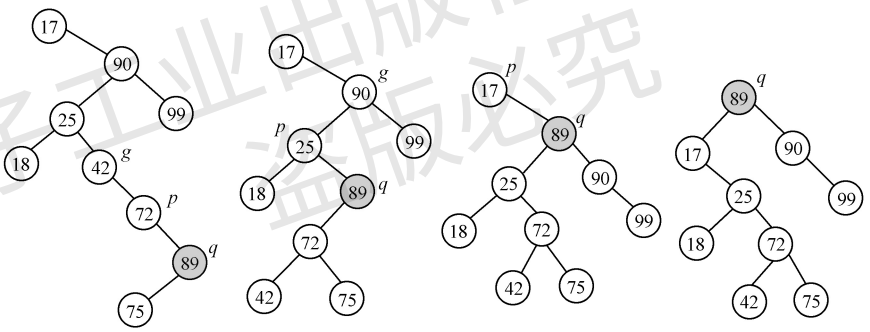

（a）旋转前　　（b）旋转后

图 3-3　伸展树的 zigzag 旋转

设伸展操作开始时，伸展结点 $q$ 位于伸展树的第 $k+1$ 层（根结点的层次为 1），$k$ 是从根结点到伸展结点的路径长度。那么，若 $k$ 是偶数，则需执行 $k/2$ 次双重旋转；若 $k$ 是奇数，则除了执行 $k/2$ 次双重旋转，还需执行一次单一旋转，才能最终将伸展结点 $q$ 提升到根结点处。这也就是说，每次伸展操作可能需要若干次双重旋转，但至多一次单一旋转才能实现。单一旋转可安排在伸展过程开始时执行，也可安排在最后执行，其效果是相同的。伸展操作的结果总是将伸展结点移至根结点。

此外，伸展树的伸展操作可以**自底向上**（bottom-up）进行，也可以**自顶向下**（top-down）进行。下面讨论自底向上的伸展过程。

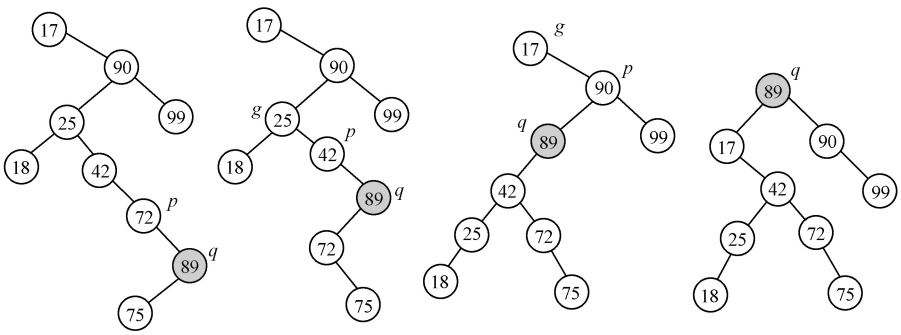
图 3-4 所示是在伸展树中搜索 89 的例子，其过程与普通二叉搜索树的完全一样。图中，$q=89$ 是伸展结点，从根结点到 89 的路径长度为奇数 5，需执行两次双重旋转和一次单一旋转才能将 $q$ 提升为根结点。图中采取将单一旋转最后执行的做法，即自底向上，先执行两次双重旋转，最后执行一次单一旋转。




（a）搜索89　　（b）一字旋转后　　（c）之字旋转后　　（d）单一旋转后

图 3-4　伸展树的伸展操作示例 1

图 3-5 所示也是自底向上进行伸展的，但它采取先执行一次单一旋转，再执行多次双重旋转的做法。注意，在稍后的程序实现中，我们采用的是图 3-5 的做法。




（a）搜索89　　（b）单一旋转后　　（c）一字旋转后　　（d）之字旋转后

图 3-5　伸展树的伸展操作示例 2

### 3.1.4 伸展树类

伸展树采用一般二叉树的存储方式存储，每个结点有三个域：element、lChild 和 rChild。代码 3-1 定义了伸展树类，给出了它的数据成员和部分成员函数。类型 ResultCode 为函数返回类型。下面以插入运算为例阐明伸展树的伸展操作。插入运算的实现中包含搜索过程，因而不难实现伸展树的搜索和删除运算。

**代码 3-1** 伸展树类。

```
#include <iostream.h>
enum ResultCode{Underflow, Overflow, Success, Duplicate, Fail, NotPresent};
template<class T>
struct BTNode
{//二叉树结点类
    BTNode(const T& x)
    {
        element=x; lChild=rChild=NULL;
    }
    T element;
    BTNode* lChild,*rChild;
};
template<class T, class K>
class SPTree
{//伸展树类
public:
    SPTree(){root=NULL;}
    ResultCode Insert(T x);
    …
protected:
    BTNode<T>* root;
private:
    ResultCode Insert(BTNode<T>* &p, T x);
    void   LRot(BTNode<T>* &p);
    void   RRot(BTNode<T>* &p);
    …
};
```

### 3.1.5 旋转的实现

代码 3-2 中的左旋转函数 LRot 和右旋转函数 RRot 分别实现一次向左和向右的旋转步。zig 向右旋转，zag 向左旋转。一次单一旋转调用一次旋转函数。双重旋转需要采用下列方式两次调用旋转函数：

（1）zigzig 旋转，执行两次函数 RRot；

（2）zagzag 旋转，执行两次函数 LRot；

（3）zigzag 旋转，执行一次函数 LRot，再执行一次函数 RRot；

（4）zagzig 旋转，执行一次函数 RRot，再执行一次函数 LRot。

**代码 3-2** 旋转函数。

```
template <class T>
void SPTree<T>::LRot(BTNode<T>*& p)
{ //前置条件：p 有右孩子，实现向左旋转
    BTNode<T>* r=p->rChild;
    p->rChild=r->lChild;
    r->lChild=p; p=r;                          //p 的右孩子成为子树根结点
}
template <class T>
void SPTree<T>::RRot(BTNode<T>*& p)
{ //前置条件：p 有左孩子，实现向右旋转
    BTNode<T>* r=p->lChild;
    p->lChild=r->rChild;
    r->rChild=p; p=r;                          //p 的左孩子成为子树根结点
}
```

### 3.1.6 插入运算的实现

代码 3-3 实现了伸展树的插入运算。其中，递归函数 Insert 是私有的，它完成伸展树插入运算的基本工作。一旦搜索到重复结点，该重复结点便作为伸展操作的伸展结点；否则，插入的新结点作为伸展操作的伸展结点。伸展操作由至多一次单一旋转和若干次双重旋转组成。若从根结点运算到伸展结点的路径长度为奇数，则先执行一次单一旋转；否则，仅执行若干次双重旋转。

**代码 3-3** 伸展树插入运算。

```
template <class T, class K>
ResultCode SPTree<T, K>::Insert(T x)
{
    return Insert(root, x);
}
template <class T, class K>
ResultCode SPTree<T, K>::Insert(BTNode<T>* &p, T x)
{//假定 T 类上已重载了关系运算符或类型转换运算符①
    ResultCode result=Success;
    BTNode<T>* r;
    if (p==NULL) {                                    //插入新结点
        p=new BTNode<T>(x); return result;
    }
    if(x==p->element) {
```

① 在一个类上通过重载关系运算符或类型转换运算符，可将结构间的比较视为关键字间的比较，见附录 B（前言二维码）。

```
            result=Duplicate; return result;
        }
        if (x<p->element) {
            r=p->lChild;
            if(r==NULL)     {                                   //zig 旋转
                r=new BTNode<T>(x); r->rChild=p; p=r;
                return result;
            }
            else if(x==r->element) {                            //zig 旋转
                RRot(p); result=Duplicate; return result;
            }
            if(x<r->element){                                   //zigzig 旋转
                result=Insert(r->lChild, x);
                RRot(p);
            }
            else{                                               //zigzag 旋转
                result=Insert(r->rChild, x);
                LRot(r); p->lChild=r;
            }
            RRot(p);
        }
        else {
            r=p->rChild;
            if(r==NULL)     {
                r=new BTNode<T>(x); r->lChild=p; p=r;
                return result;
            }
            else if(x==r->element) {
                LRot(p);                                        //zag 旋转
                result=Duplicate; return result;
            }
            if(x>r->element){                                   //zagzag 旋转
                result=Insert(r->rChild, x);
                LRot(p);
            }
            else{                                               //zagzig 旋转
                result=Insert(r->lChild, x);
                RRot(r); p->rChild=r;
            }
            LRot(p);
        }
```

```
        return result;
    }
```

### 3.1.7 分摊分析

从第 2 章的讨论可知，一般的算法分析是针对某个运算的一次执行而言的。最坏情况时间复杂度是指某个运算对某个输入有最长的执行时间。平均情况时间复杂度是指对各种可能的输入，执行某个运算所需时间的概率平均值。分摊分析是对一个长的运算序列在最坏情况下所需的总的时间求平均值。本节运用势能分析方法来分析伸展树运算的时间性能。

假定对一棵伸展树执行了 $m$ 次运算（搜索、插入或删除），当然，每次运算都需要进行伸展。分摊分析将 $m$ 次运算在最坏情况下的总时间除以 $m$，得到每次运算的平均时间。一个伸展操作所需的时间可以以旋转步数来度量，一次单一旋转记为一个旋转步，一次双重旋转记为两个旋转步。

**定义 3-1** （秩）设 $x$ 是伸展树 $T$ 中一个结点，$s(x)$是以 $x$ 为根的子树的结点数，结点 $x$ 的**秩**（rank）$r(x)$定义如下：

$$r(x) = \log s(x) \tag{3-1}$$

**定义 3-2** （势能）设 $x$ 是伸展树 $T$ 中一个结点，伸展树 $T$ 的**势能**（potential）$\Phi$ 定义为树中所有结点的秩之和：

$$\Phi = \sum_x r(x) \tag{3-2}$$

**定义 3-3** （分摊代价）设对伸展树 $T$ 执行 $m$ 次运算，第 $i$ 次运算的分摊代价 $\hat{c}_i$ 定义如下：

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

式中，$c_i$ 为第 $i$ 次运算的实际代价，$\Phi_{i-1}$ 是该运算执行前伸展树的势能，$\Phi_i$ 是该运算执行后伸展树的势能。$\Phi_i - \Phi_{i-1}$ 是该运算执行后的势能增加值。

从定义 3-3 可以计算这 $m$ 次运算的总分摊代价如下：

$$\sum_{i=1}^{m} \hat{c}_i = \sum_{i=1}^{m} c_i + \Phi_m - \Phi_0 \tag{3-3}$$

式中，$\Phi_0$ 是 $m$ 次运算执行前的伸展树的势能，$\Phi_m$ 是 $m$ 次运算执行后的势能。

$m$ 次运算的总实际代价可表示如下：

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} \hat{c}_i + \Phi_0 - \Phi_m \tag{3-4}$$

通过计算每次运算的分摊代价 $\hat{c}_i$，可以得到 $m$ 次运算总实际代价的上界。

我们已经知道，伸展树的搜索、插入和删除运算结束时，都需要做一次伸展操作。每个伸展操作由若干次双重旋转和至多一次单一旋转组成，设第 $i$ 次运算的伸展操作包括 $h$ 个旋转，设每个旋转的分摊代价为 $b_j^i$（$j = 1, 2, \cdots, h$），因此，可以将 $\hat{c}_i$ 进一步细分为

$$\hat{c}_i = \sum_{j=1}^{h} b_j^i, \quad b_j^i = c_j^i + \Phi_j^i - \Phi_{j-1}^i \tag{3-5}$$

式中，$c_j^i$ 是在第 $i$ 次运算的伸展操作的第 $j$ 次旋转的实际代价，$\Phi_{j-1}^i$ 是旋转前的势能，$\Phi_j^i$ 是旋转后的势能。一次双重旋转包含两个旋转步，一次单一旋转包含一个旋转步，即 $c_j^i$ 等于 1 或 2。

为了计算每种旋转的分摊代价，先给出引理 3-1。

**引理 3-1** 如果$\alpha$、$\beta$和$\gamma$是正实数，$\alpha + \beta \leqslant \gamma$，则 $\log\alpha + \log\beta \leqslant 2\log\gamma - 2$。

**证明** 因为$(\sqrt{\alpha} - \sqrt{\beta})^2 \geqslant 0$，所以，$\sqrt{\alpha\beta} \leqslant \dfrac{\alpha + \beta}{2} \leqslant \gamma / 2$，两边取对数，公式成立。证毕。

下面区分不同的旋转种类来计算 $b_j^i$ 的值。引理 3-2 给出一字旋转的分摊代价，引理 3-3 给出之字旋转的分摊代价，引理 3-4 给出单一旋转的分摊代价。

**引理 3-2** 对一字旋转，有

$$b_j^i < 3\,r_j^i(q) - 3\,r_{j-1}^i(q) \tag{3-6}$$

**证明** 从图 3-2 可得

$$b_j^i = 2 + r_j^i(q) + r_j^i(p) + r_j^i(g) - r_{j-1}^i(q) - r_{j-1}^i(p) - r_{j-1}^i(g)$$

式中，$r_j^i(q) = r_{j-1}^i(g)$，因为旋转前以 $g$ 为根结点的子树和旋转后以 $q$ 为根结点的子树有相同的结点数，即 $s_j^i(q) = s_{j-1}^i(g)$。所以有

$$b_j^i = 2 + r_j^i(p) + r_j^i(g) - r_{j-1}^i(q) - r_{j-1}^i(p)$$

从图 3-2 可知，第 $j$ 次旋转执行前，以 $q$ 为根结点的子树上包含结点 $q$ 及两棵子树$\alpha$和$\beta$，而旋转后，以 $g$ 为根结点的子树包含结点 $g$ 及两棵子树$\delta$ 和$\gamma$。这两棵子树的结点合并起来，仅比旋转后以 $q$ 为根结点的子树少一个结点 $p$。于是

$$s_{j-1}^i(q) + s_j^i(g) < s_j^i(q)$$

根据引理 3-1，$r_{j-1}^i(q) + r_j^i(g) < 2\,r_j^i(q) - 2$，于是

$$b_j^i < 2\,r_j^i(q) - 2\,r_{j-1}^i(q) + r_j^i(p) - r_{j-1}^i(p)$$

又因为 $s_{j-1}^i(p) > s_{j-1}^i(q)$和 $s_j^i(q) > s_j^i(p)$，所以

$$b_j^i < 3\,r_j^i(q) - 3\,r_{j-1}^i(q)$$

证毕。

**引理 3-3** 对之字旋转，有

$$b_j^i < 2r_j^i(q) - 2r_{j-1}^i(q) \leqslant 3r_j^i(q) - 3r_{j-1}^i(q) \tag{3-7}$$

证明略。

**引理 3-4** 对单一旋转，有

$$b_j^i < 1 + r_j^i(q) - r_{j-1}^i(q) \tag{3-8}$$

证明略。

**定理 3-1** 在一个有 $n$ 个结点的伸展树上，执行第 $i$ 次运算（搜索、插入或删除）时，其伸展结点为 $q$，所需的分摊代价为

$$\hat{c}_i \leqslant 1 + 3\log n \tag{3-9}$$

**证明** 因为最多只有一次单一旋转，其余均为双重旋转，设最后一步是单一旋转，所以有

$$\hat{c}_i = \sum_{j=1}^{h} b_j^i = \sum_{j=1}^{h-1} b_j^i + b_h^i \leqslant \sum_{j=1}^{h-1} [3r_j^i(q) - 3r_{j-1}^i(q)] + [1 + 3r_h^i(q) - 3r_{h-1}^i(q)]$$

$$= 1 + 3r_h^i(q) - 3r_0^i(q) \leqslant 1 + 3r_h^i(q) = 1 + 3\log n$$

证毕。

**定理 3-2** 对一棵结点数不超过 $n$ 的伸展树，执行 $m$ 次运算（搜索、插入或删除）的总实际代价不超过

$$m(1 + 3\log n) + n\log n \tag{3-10}$$

**证明** 因为 $m$ 次运算的总实际代价为

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} \hat{c}_i + \Phi_0 - \Phi_m \leqslant m(1 + 3\log n) + \Phi_0 - \Phi_m$$

$$\leqslant m(1 + 3\log n) + \Phi_0 \leqslant m(1 + 3\log n) + n\log n$$

最后一步因为每个结点 $x$ 的秩 $r(x)=\log s(x)\leqslant \log n$，所以 $\Phi_0\leqslant n\log n$。证毕。

于是，当 $m$ 很大时，伸展树的平均分摊代价为 $O(\log n)$。

## 3.2 跳表

跳表是 William Pugh 在 1989 年提出的，被认为是可以代替平衡搜索树的另外一种选择。平均来说，跳表具有很好的搜索、插入和删除的时间效率，并且它比平衡搜索树更容易实现，因此，它在实现的难度和性能之间做了很好的折中。

### 3.2.1 什么是跳表

#### 1．跳表的结构

有序表（指已排序表）的二分搜索有很高的搜索效率，但是这种搜索方法不能在链表上进行，不宜表示动态集。这是因为对链表结构，难以有效计算中间结点的地址。但如果将一个有序表组成如图 3-6（c）所示的结构形式，就可以提高链表的搜索效率，这种结构称为**跳表**（skip list）。用它可替代平衡搜索树，并获得良好的运算性能。

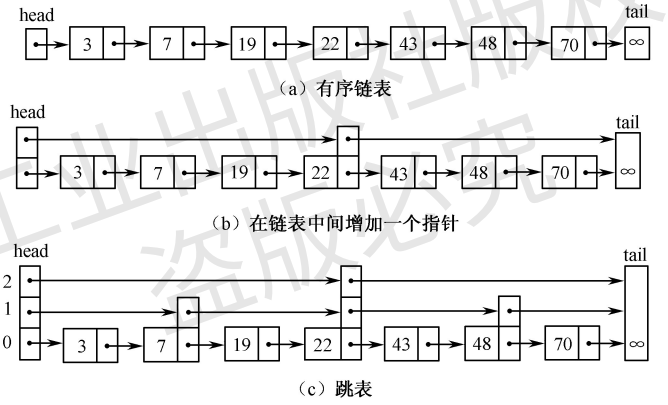



图 3-6　跳表结构

如图 3-6（a）所示是一个简单的链表，其结点按元素关键字值的顺序排列。搜索一个链表需要沿着链表一次一个结点地移动，在最坏情况下，其需要比较 $n=7$ 次，搜索时间为 $O(n)$。如果采用如图 3-6（b）所示的方法，其在最坏情况下的比较次数可减少近一半，为 4 次。搜索一个元素时，首先将它与中间元素进行比较，然后根据比较结果决定或者在前半部分搜索或者在后半部分搜索。还可进一步用如图 3-6（c）所示的方法，分别在链表的前半部分和后半部分的中间再增加一个指针，这样，有三条链，第 0 层链是图 3-6（a）的初始链，包括 $n$ 个元素，第 1 层链包括 $n/2$ 个元素，第 2 层链包括 $n/4$ 个元素。

跳表是一个有序链表，每个结点包含可变数目的链（指针），结点中的第 $i$ 层链，跳过那些只包含低于第 $i$ 层链的结点，构成一个单链表。每隔 $2^i$ 个元素有一个第 $i$ 层指针。第 0 层链是包含所有元素的有序链表，第 1 层链包含的元素是第 0 层链的子集，……，第 $i$ 层链包含的元素是第 $i-1$ 层链的子集。在理想情况下，跳表的层数为 $\lceil \log n \rceil$。

#### 2．跳表的搜索

在跳表上的搜索从最高层表头指针开始，顺着指针向右搜索，遇到某个关键字值大于或等于

待查关键字值时，则下降一层，再沿该层的指针向右搜索，逐步逼近待查元素，直到第 0 层指针所指的关键字值大于或等于待查关键字值，搜索终止。这时，如果指针所指元素的关键字值等于待查关键字值，则搜索成功终止，否则搜索失败终止。由此可见，跳表的搜索总是在最下层处结束。

例如，要在图 3-6（c）的跳表中搜索关键字值 43。首先由第 2 层表头指针开始向右搜索，令 22 与 43 比较，因为 22＜43，向右搜索；令∞与 43 比较，现满足∞≥43，所以下降到第 1 层；令 48 与 43 比较，这时有 48≥43，再次下降到第 0 层；最后令 43 与 43 比较，这时有 43≥43。在第 0 层的元素关键字值与待查关键字值比较后，还需进行最后一次比较，以确定两个关键字值是否相等，若二者相等，则搜索成功，否则搜索失败。所以，在搜索过程中，与待查关键字值 43 比较的跳表中的关键字值依次为 22、∞、48、43 和 43。

要在图 3-6（c）的跳表中搜索关键字值 46，与 46 比较的跳表的关键字值依次为 22、∞、48、43、48 和 48。

### 3．跳表的插入

在图 3-6（c）所示的跳表中，约有一半的结点只有 1 个指针，四分之一的结点有 2 个指针，八分之一的结点有 3 个指针，其余类推。也就是说，有 $n/2^i$ 个元素为第 $i$ 层链元素。这样的跳表称为理想的跳表或“完全平衡”的跳表。在插入和删除过程中，如果要始终保持跳表的这种理想状态，其代价是很大的。

实用的跳表按一定的概率分布为新结点指定层次。例如，可使新结点有 1 个指针的概率是 1/2，有 2 个指针的概率是 1/4，其余类推。这样构建的跳表称为**随机跳表**（randomized skip list）。

为了在图 3-6（c）的跳表中插入新元素 56，设随机分配的层次为 1，这意味着在将 56 插入 48 与 70 之间时，还需建立它在第 0 层和第 1 层的链接指针，如图 3-7 所示。

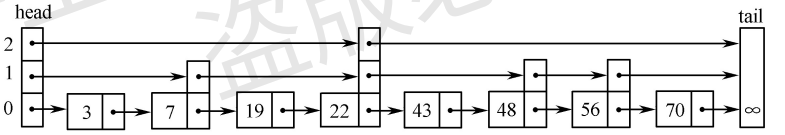



图 3-7　跳表的插入

### 4．跳表的删除

对删除运算，将无法控制删除后表的层链分布。例如，删除图 3-7 中的元素 56 之后，跳表又成为图 3-6（c）的形式。删除一个第 $i$ 层链的结点，删除后需要改变第 0～$i$ 层的链指针，使这些指针均指向 56 后面的结点。

## 3.2.2　跳表类

代码 3-4 的跳表结点类 SNode<T>有两个数据成员：element 和 link。element 是元素域，link 是长度可变的动态一维指针数组，用于存储跳表的链指针。

从图 3-6 中可以看到，跳表由结点连接而成，其中，跳表的头结点需要有足够多的指针域，以满足构造最大层链数目的需要，但它并不需要元素（元素）域；在尾结点中存放了一个大值（∞），作为搜索终止条件，但它并不需要指针域。每个存有元素的结点都有一个元素域和层次加 1 个指针域。所需的指针域的个数比层次多 1。指针域由一维指针数组 link 表示，其中 link[i]表示第 i 层链指针。

**代码 3-4** 跳表结点类。

```
template <class T>
struct SNode
{
    SNode(int mSize)
    {
        link=new SNode* [mSize];
    }
    ~SNode(){ delete[] link;}
    T element;
    SNode<T>**link;
};
```

代码 3-5 定义了跳表类，表中定义的公有函数的功能见 3.1.1 节中的描述。head 是头结点指针，tail 是尾结点指针。last 是指针数组，它用于保存在插入和删除运算之前的搜索过程中每条链上遇到的最后一个结点的地址。maxLevel 为最大层次号，整数 levels 是当前已存在的最大层次，链指针数为 levels + 1。

代码 3-6 为跳表类的构造函数。构造函数同时为头结点、尾结点和 last 指针数组分配空间。头结点中有 maxLevel+1 个用于指向各层链的指针，它们被初始化为指向尾结点。尾结点中保存了一个作为哨兵值的最大值。

**代码 3-5** 跳表类。

```
template<class T>
class SkipList
{
public:
    SkipList(T large, int mLev);
    ~SkipList();
    ResultCode Insert(T x);              //函数定义见 3.1.1 节
        …
private:
    int Level();
    SNode<T>* SaveSearch(T x);
    int maxLevel, levels;
    SNode<T> *head, *tail, **last;
};
```

**代码 3-6** 构造函数。

```
template <class T>
SkipList<T>::SkipList(T large, int mLev)
{
    maxLevel=mLev; levels=0;
    head=new SNode<T>(maxLevel+1);     //指向包括元素域和 maxLevel+1 个指针的头结点
    tail=new SNode<T>(0);              //指向只有元素域，不包含指针域的尾结点
```

```
        last=new SNode<T>*[maxLevel+1];      //maxLevel+1 个指针
        tail->element.key=large;             //尾结点中保存作为哨兵值的最大值 large
        for (int i=0; i<=maxLevel; i++)
                head->link[i]=tail;          //头结点的所有指针均指向尾结点
}
```

### 3.2.3 层次分配

设 $S_0$ 为第 0 层链上元素的集合，$S_1$ 是第 1 层链上元素的集合，……，$S_h$ 是第 $h$ 层链上元素的集合，必有 $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_h$。一个元素的层次为 $i$，是指该元素属于 $S_0, S_1, \cdots, S_i$，但不属于 $S_{i+1}$。可以采用连续抛掷硬币的方式为新元素分配层次，假定出现正面的概率为 $p$。具体试验方法是，抛掷一枚硬币，如果为正面，则继续抛掷，直到出现反面为止，将连续出现正面的抛掷次数作为新结点的层次。这种方式可由下列语句实现：

```
int lev=0;
while (rand()<=CutOff) lev++;
```

其中，rand()为伪随机数产生函数，它返回一个伪随机数。常数 RAND_MAX 是可由该函数返回的最大值，令 CutOff=$p$×RAND_MAX，则 rand()产生的随机数小于或等于 CutOff 的概率为 $p$。

新结点的层次由上述程序段生成。若第一个随机数小于或等于 CutOff，则说明新元素属于 $S_1$。继续确定新元素是否属于 $S_2$。若下一个随机数仍小于或等于 CutOff，则表明新元素也属于 $S_2$。因此，$S_{i-1}$ 元素属于 $S_i$ 的概率为 $p$。重复这一过程，直到得到一个随机数大于 CutOff。令 $i$ = lev 作为新元素的层次意味着新元素属于 $S_0, S_1, \cdots, S_i$，但不属于 $S_{i+1}$。

采用上述方法分配层次，可能导致一些结点的层次过大。为避免这种情况，可对 lev 设定一个上限，例如，令 maxLevel=$\lceil \log n \rceil - 1$。具体做法如下：

```
lev=(lev<=maxLevel)？ lev:maxLevel;
```

代码 3-7 的函数 Level 产生一个随机级数 lev，如果它大于 maxLevel，则返回 maxLevel 作为准备分配给新结点的层次。

**代码 3-7** 层次分配。

```
template <class T>
int SkipList<T>::Level()
{
    int lev=0;
    while (rand()<= RAND_MAX/2) lev++;
    return (lev<=maxLevel)？ lev:maxLevel;
}
```

即使采取这一上限，但还可能出现下列情况。在插入一个新结点之前有 3 条链，即已存在层次为 0～2 的链。假如现在分配给新结点的层次是 9，则插入新结点后便有了 10 条链。也就是说，在这之前尚未插入第 3～8 层的结点，这些空链对当前的搜索没有好处，因此可将新结点的层次调整为 3。

设跳表当前的层次为 levels，为了避免出现上述空链，代码 3-8 中的具体实现如下：

```
if (lev>levels) lev=++levels;
```

最终以这种方式修正的 lev 值作为新结点的层次。

### 3.2.4 插入运算的实现

函数 Insert 通过调用私有成员函数 Level 和 SaveSearch 实现插入运算。

私有成员函数 SaveSearch 定义如下：

```
SNode<T>* SaveSearch(T x);
```

前置条件：x 的关键字值小于最大值。

后置条件：函数返回大于或等于 x 的关键字值的结点的地址，并且每个指针 last[i]（i 的取值范围为 0～levels）都指向该结点在相应层的链中前一个结点。

此函数的搜索从表头结点 head 开始，令指针 p 指向头结点，并从最高层出发，顺着指针向右搜索，遇到某个关键字值大于或等于待查关键字值，则下降一层，沿该层的指针向右搜索，逐步逼近待查元素，直到第 0 层指针所指的关键字值大于或等于待查关键字值，搜索终止。函数 SaveSearch 使用语句“last[i]=p;”，在下降到下一层前，把该层（设为第 i 层）遇到的最后一个结点的地址存放在指针数组 last[i]中。

函数 Insert 调用函数 SaveSearch 搜索待查元素 x，如果表中不存在与 x 的关键字值相同的元素，表示没有重复关键字值，则构造一个元素值为 x 的新结点。

函数 Level 为新结点分配层次，设为 lev，新结点将被链接到第 0～lev 层的各层链中。新结点在第 i（i 的取值范围为 0～lev）层链中的位置位于指针 last[i]指示的结点后面。由于在搜索中使用了 last[i]数组，因此插入新结点时，建立各层的链接很容易实现。

如果待查元素的关键字值大于或等于最大值，则 RangeError 出错；如果表中已存在与待查元素关键字值相同的元素，则输出 Duplicate 信息。

**代码 3-8** 插入运算。

```
enum ResultCode{Underflow, Overflow, Success, Duplicate, RangeError, NotPresent};
template <class T, class K>
SNode<T>* SkipList<T, K>::SaveSearch(T x)
{                                          //假定类 T 已重载了关系运算符或类型转换运算符
    SNode<T>*p=head;
    for(int i=levels; i>=0; i--){
        while(p->link[i]->element <x) p=p->link[i];
        last[i]=p;                         //将最后搜索到的第 i 层结点的地址保存在 last[i]中
    }
    return (p->link[0]);
}
template <class T, class K>
ResultCode SkipList<T, K>::Insert(T x)
{
    if (x>=tail->element) return RangeError;
    SNode<T>* p=SaveSearch(x);
    if (p->element==x)   return Duplicate;          //表明有重复元素
    int lev=Level();                                //计算层次
    if(lev>levels){
        lev=++levels;last[lev]=head;
```

```
        }
        SNode<T>* y=new SNode<T>(lev+1);              //构造新结点
        y->element=x;
        for(int i=0; i<=lev; i++){                    //新结点插入各层链中
            y->link[i]=last[i]->link[i];
            last[i]->link[i]=y;
        }
        return Success;
    }
```

### 3.2.5 性能分析

假定采用上述抛掷硬币的层次分配方案，并假定每次抛掷硬币时，出现正面的概率为 1/2。现在分析跳表的时间和空间性能。对一个有 $n$ 个结点的跳表有如下结论。

**引理 3-5** 第 $k$ 层链中至少有一个元素的概率至多为 $n/2^k$。

**证明** 采用上述抛掷硬币的层次分配方案，一个新元素属于第 $k$ 层链的概率为 $1/2^k$，所以第 $k$ 层链中至少有一个元素的概率至多为 $n/2^k$，因为 $n$ 个不同事件中任意一个事件发生的概率至多是其中每个事件发生的概率之和。证毕。

**定理 3-3** 跳表的高度（最大层次）大于 $k$ 的概率至多为 $n/2^k$。

**证明** 有 $n$ 个结点的跳表的高度大于 $k$ 的概率等于在第 $k$ 层链中至少有一个元素的概率，所以跳表的高度大于 $k$ 的概率至多为 $n/2^k$。证毕。

令 $k$=3log$n$，则跳表的高度大于 3log$n$ 的概率至多为

$$\frac{n}{2^k}=\frac{n}{2^{3\log n}}=\frac{n}{n^3}=\frac{1}{n^2} \tag{3-11}$$

更一般地，给定一个常数 $c>1$，跳表的高度大于 $c\log n$ 的概率至多为 $1/n^{c-1}$。这就是说，有 $n$ 个元素的跳表的高度为 $O(\log n)$的概率很高。

**定理 3-4** 有 $n$ 个元素的跳表的平均空间复杂度为 $O(n)$。

**证明** 跳表的实际空间需考虑元素所占的空间和结点中包含的指针数。可以将一个跳表视为一组链表 $S_0, S_1, \cdots, S_h$ 的集合，$S=S_0\supseteq S_1\supseteq\cdots\supseteq S_h$。跳表的空间复杂度依赖于所有这些集合中的元素总数。第 $i$ 层链包含的元素个数的期望值为 $n/2^i$，则有

$$\sum_{i=0}^{h}\frac{n}{2^i}=n\sum_{i=0}^{h}\frac{1}{2^i}<2n \tag{3-12}$$

因此，有 $n$ 个元素的跳表的平均空间复杂度为 $O(n)$。证毕。

## 本 章 小 结

伸展树和跳表都可用于表示字典结构。伸展树是一种平衡搜索树的替代结构。对长的运算序列，伸展树有很好的平均时间性能。对伸展树运算所做的分摊分析也与传统数据结构的时间分析不同，很有代表性。跳表是建立在概率平衡基础上的具有随机性的数据结构，它有很好的平均时间性能。跳表运算实现简单，被认为可以替代平衡搜索树。跳表搜索时间的分析方法与传统的数据结构有区别，具有特殊性。

# 习题 3

3-1　指明图 3-4 和图 3-5 中每步执行的是何种单一旋转或双重旋转。

3-2　在图 3-8 所示的二叉搜索树上完成下列运算及随后的伸展操作，画出每次运算加伸展操作后的结果伸展树。采用最先执行和最后执行单一旋转两种方式实现之。

（1）搜索 80；（2）插入 80；（3）删除 30。

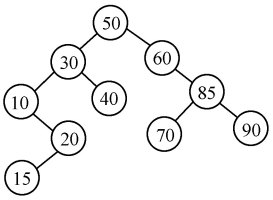


图 3-8　习题 3-2 的图

3-3　证明引理 3-3 和引理 3-4。

3-4　编写程序实现伸展树的搜索（Search）和删除（Remove）运算。

3-5　设跳表采用抛掷硬币的层次分配方案。对任意 $k$（$0 \leqslant k \leqslant n-1$），$n$ 是跳表的元素个数，$x$ 是跳表中的某个元素，随机变量 $h(x)$是 $x$ 的层次，求 $h(x)$的平均值。

3-6　计算一个有 $n$ 个结点的跳表第 $k$ 层链中至少有一个元素的概率。

3-7　计算跳表中各层链上元素集合 $S_i$ 中的平均元素个数。

3-8　求从空表开始，向跳表中插入 $n$ 个元素的平均空间复杂度。

3-9　实现在跳表中搜索一个给定元素和从跳表中删除给定元素的运算。

3-10　既然跳表的第 0 层链是普通的有序链表，那么，可以在跳表上实现线性表的一般运算。设计两个函数 Before 和 Next，实现求指定关键字值的前驱和后继元素的运算。

3-11　修改跳表使之允许插入重复元素。

3-12　扩充跳表类，增加删除最大、最小元素的函数，以及以升序输出全部元素的函数。

3-13　比较几种常用的表示字典的数据结构的性能：二叉搜索树、二叉平衡树、伸展树、跳表和散列表。