第7章

搜索树

从本章开始，讨论的重点将逐步转入查找技术。实际上，此前的若干章节已经就此做过一些讨论，在向量与列表等结构中，甚至已经提供并实现了对应的ADT接口。然而遗憾的是，此前这类接口的总体效率均无法令人满意。

以31页代码2.1中的向量模板类Vector为例，其中针对无序和有序向量的查找，分别提供了find()和search()接口。前者的实现策略只不过是将目标对象与向量内存放的对象逐个比对，故最坏情况下需要运行$O(n)$时间。后者利用二分查找策略尽管可以确保在$O(\log n)$时间内完成单次查找，但一旦向量本身需要修改，无论是插入还是删除，在最坏情况下每次仍需$O(n)$时间。而就代码3.2中的列表模板类List（70页）而言，情况反而更糟：即便不考虑对列表本身的修改，无论find()或search()接口，在最坏情况或平均情况下都需要线性的时间。另外，基于向量或列表实现的栈和队列，一般地甚至不提供对任意成员的查找接口。总之，若既要求对象集合的组成可以高效率地动态调整，同时也要求能够高效率地查找，则以上线性结构均难以胜任。

那么，高效率的动态修改和高效率的静态查找，究竟能否同时兼顾？如有可能，又应该采用什么样的数据结构？接下来的两章，将逐步回答这两个层次的问题。

因为这部分内容所涉及的数据结构变种较多，它们各具特色、各有所长，也有其各自的适用范围，故按基本和高级两章分别讲解，相关内容之间的联系如图7.1所示。

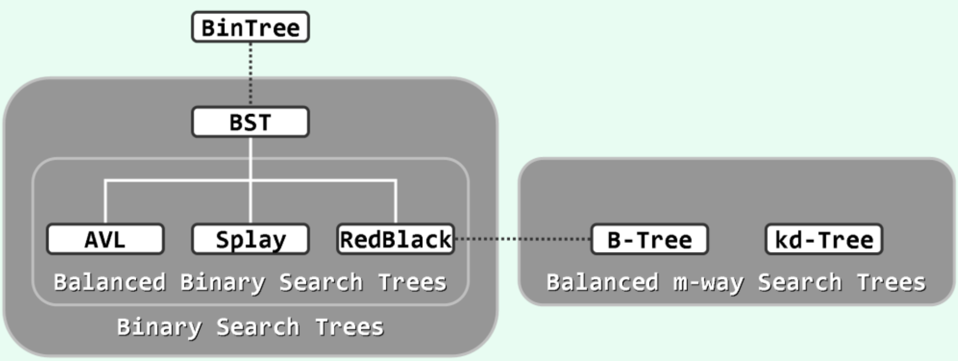

图7.1 第7章和第8章内容纵览

本章将首先介绍树式查找的总体构思、基本算法以及数据结构，通过对二分查找策略的抽象与推广，定义并实现二叉搜索树结构。尽管就最坏情况下的渐进时间复杂度而言，这一方法与此前相比并无实质的改进，但这部分内容依然十分重要——基于半线性的树形结构的这一总体构思，正是后续内容的立足点和出发点。比如，本章的后半部分将据此提出理想平衡和适度平衡等概念，并相应地引入和实现AVL树这一典型的平衡二叉搜索树。借助精巧的平衡调整算法，AVL树可以保证，即便是在最坏情况下，单次动态修改和静态查找也均可以在$O(\log n)$时间内完成。这样，以上关于兼顾动态修改与静态查找操作效率的问题，就从正面得到了较为圆满的回答。接下来的第8章将在此基础上，针对更为具体的应用需求和更为精细的性能指标，介绍平衡搜索树家族的其它典型成员。

§7.1 查找

7.1.1 循关键码访问

所谓的查找或搜索（search），指从一组数据对象中找出符合特定条件者，这是构建算法的一种基本而重要的操作。其中的数据对象，统一地表示和实现为词条（entry）的形式；不同词条之间，依照各自的关键码（key）彼此区分。根据身份证号查找特定公民，根据车牌号查找特定车辆，根据国际统一书号查找特定图书，均属于根据关键码查找特定词条的实例。

请注意，与此前的“循秩访问”和“循位置访问”等完全不同，这一新的访问方式，与数据对象的物理位置或逻辑次序均无关。实际上，查找的过程与结果，仅仅取决于目标对象的关键码，故这种方式亦称作循关键码访问（call-by-key）。

7.1.2 词条

一般地，查找集内的元素，均可视作如代码7.1所示词条模板类Entry的实例化对象。

```
template <typename K, typename V> struct Entry { //词条模板类
   K key; V value; //关键码、数值
   Entry ( K k = K(), V v = V() ) : key ( k ), value ( v ) {}; //默认构造函数
   Entry ( Entry<K, V> const& e ) : key ( e.key ), value ( e.value ) {}; //基于克隆的构造函数
   bool operator< ( Entry<K, V> const& e ) { return key <  e.key; }  //比较器：小于
   bool operator> ( Entry<K, V> const& e ) { return key >  e.key; }  //比较器：大于
   bool operator== ( Entry<K, V> const& e ) { return key == e.key; } //判等器：等于
   bool operator!= ( Entry<K, V> const& e ) { return key != e.key; } //判等器：不等于
}; //得益于比较器和判等器，从此往后，不必严格区分词条及其对应的关键码
```

代码7.1 词条模板类Entry

词条对象拥有成员变量key和value。前者作为特征，是词条之间比对和比较的依据；后者为实际的数据。若词条对应于商品的销售记录，则key为其条形扫描码，value可以是其单价或库存量等信息。设置词条类只为保证查找算法接口的统一，故不必过度封装。

7.1.3 序与比较器

由代码7.1可见，通过重载对应的操作符，可将词条的判等与比较等操作转化为关键码的判等与比较（故在不致歧义时，往往无需严格区分词条及其关键码）。当然，这里隐含地做了一个假定——所有词条构成一个全序关系，可以相互比对和比较。需指出的是，这一假定条件不见得总是满足。比如在人事数据库中，作为姓名的关键码之间并不具有天然的大小次序。另外，在任务相对单纯但更加讲求效率的某些场合，并不允许花费过多时间来维护全序关系，只能转而付出有限的代价维护一个偏序关系。后者的一个实例，即第10章将要介绍的优先级队列——根据其ADT接口规范，只需高效地跟踪全局的极值元素，其它元素一般无需直接访问。

实际上，任意词条之间可相互比较大小，也是此前（2.6.5节至2.6.8节）有序向量得以定义，以及二分查找算法赖以成立的基本前提。以下将基于同样的前提，讨论如何将二分查找的技巧融入二叉树结构，进而借助二叉搜索树以实现高效的查找。

§7.2 二叉搜索树

7.2.1 顺序性

若二叉树中各节点所对应的词条之间支持大小比较，则在不致歧义的情况下，我们可以不必严格区分树中的节点、节点对应的词条以及词条内部所存的关键码。

如图7.2所示，在所谓的二叉搜索树（binary search tree）中，处处都满足顺序性：

任一节点r的左（右）子树中，所有节点（若存在）均不大于（不小于）r

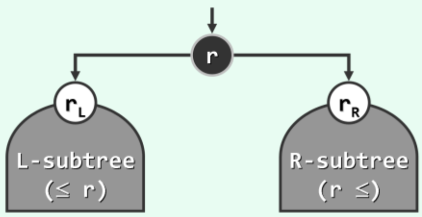

图7.2 二叉搜索树即处处满足顺序性的二叉树

为回避边界情况，这里不妨暂且假定所有节点互不相等。于是，上述顺序性便可简化表述为：

任一节点r的左（右）子树中，所有节点（若存在）均小于（大于）r

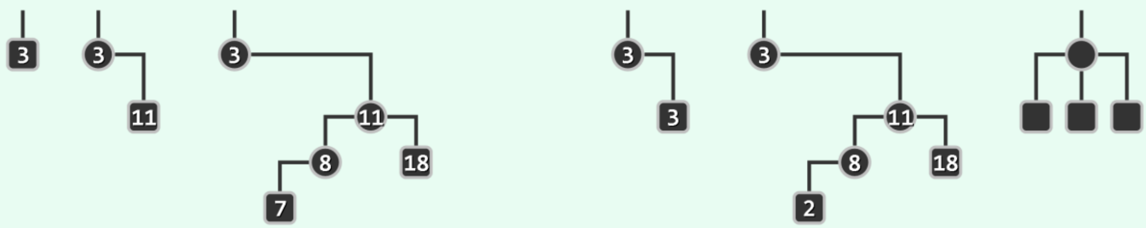

图7.3 二叉搜索树的三个实例（左），以及三个反例（右）

当然，在实际应用中，对相等元素的禁止既不自然也不必要。读者可在本书所给代码的基础上继续扩展，使得二叉搜索树的接口支持相等词条的同时并存（习题[7-10]）。比如，在去除掉这一限制之后，图7.3中原先的第一个反例，将转而成为合法的二叉搜索树。

7.2.2 中序遍历序列

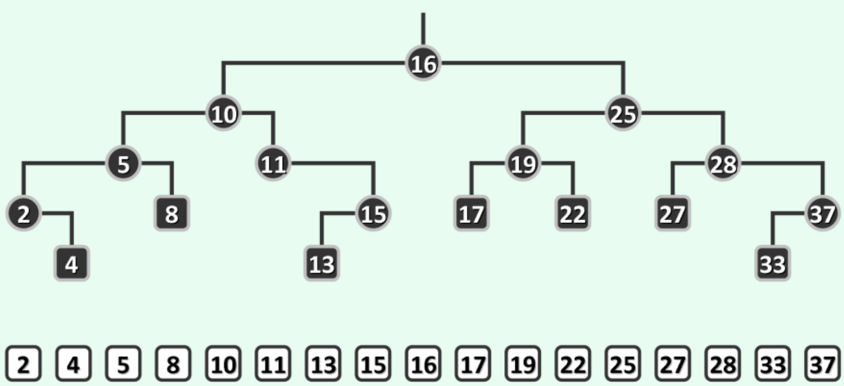

图7.4 二叉搜索树（上）的中序遍历序列（下），必然单调非降

顺序性是一项很强的条件。实际上，搜索树中节点之间的全序关系，已完全“蕴含”于这一条件之中。以如图7.4所示的二叉搜索树为例，只需采用5.4.3节的算法对该树做一次中序遍历，即可将该树转换为一个线性序列，且该序列中的节点严格按照其大小次序排列。

这一现象，并非巧合。借助数学归纳法，可以证明更具一般性的结论（习题[7-1]）：

任何一棵二叉树是二叉搜索树，当且仅当其中序遍历序列单调非降

7.2.3 BST模板类

既然二叉搜索树属于二叉树的特例，故自然可以基于BinTree模板类（121页代码5.5），派生出如代码7.2所示的BST模板类。

```
#include "../BinTree/BinTree.h" //引入BinTree

template <typename T> class BST : public BinTree<T> { //由BinTree派生BST模板类
protected:
   BinNodePosi(T) _hot; // “命中”节点的父亲
   BinNodePosi(T) connect34 ( //按照“3 + 4”结构，联接3个节点及四棵子树
      BinNodePosi(T), BinNodePosi(T), BinNodePosi(T),
      BinNodePosi(T), BinNodePosi(T), BinNodePosi(T), BinNodePosi(T) );
   BinNodePosi(T) rotateAt ( BinNodePosi(T) x ); //对x及其父亲、祖父做统一旋转调整
public: //基本接口：以virtual修饰，强制要求所有派生类（BST变种）根据各自的规则对其重写
   virtual BinNodePosi(T) & search ( const T& e ); //查找
   virtual BinNodePosi(T) insert ( const T& e ); //插入
   virtual bool remove ( const T& e ); //删除
};
```

代码7.2 由BinTree派生的二叉搜索树模板类BST

可见，在继承原模板类BinTree的同时，BST内部也继续沿用了二叉树节点模板类BinNode。

按照二叉搜索树的接口规范定义，这里新增了三个标准的对外接口search()、insert()和remove()，分别对应于基本的查找、插入和删除操作。这三个标准接口的调用参数，都是属于元素类型T的对象引用——这正是此类结构“循关键码访问”方式的具体体现。

另外，既然这些操作接口的语义均涉及词条的大小和相等关系，故这里也假定基本元素类型T或者直接支持比较和判等操作，或者已经重载过对应的操作符。

本章以及下一章还将以BST为基类，进一步派生出二叉搜索树的多个变种。无论哪一变种，既必须支持上述三个基本接口，同时在内部的具体实现方式又有所不同。因此，它们均被定义为虚成员函数，从而强制要求派生的所有变种，根据各自的规则对其重写。

7.2.4 查找算法及其实现

■ 算法

二叉搜索树的查找算法，亦采用了减而治之的思路与策略，其执行过程可描述为：

从树根出发，逐步地缩小查找范围，直到发现目标（成功）或缩小至空树（失败）

例如，在图7.5中查找关键码22的过程如下。

首先，经与根节点16比较确认目标关键码更大，故深入右子树25递归查找；经比较发现目标关键码更小，故继续深入左子树19递归查找；经再次比较确认目标关键码更大后，深入右子树22递归查找；最终在节点22处匹配，查找成功。

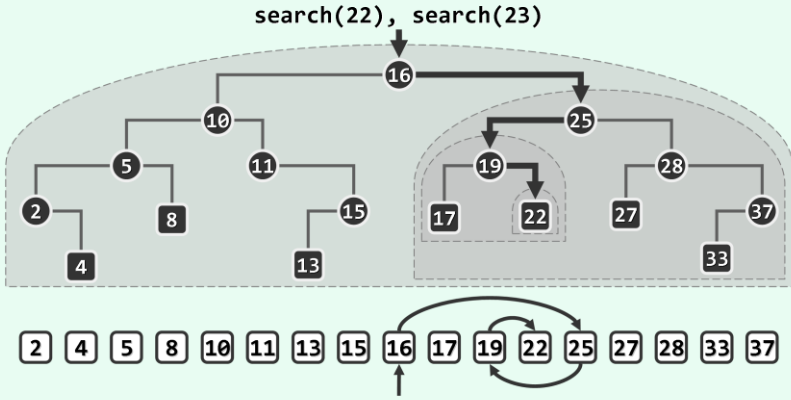

图7.5 二叉搜索树的查找过程（查找所经过的通路，以粗线条示意）

当然，查找未必成功。比如针对关键码20的查找也会经过同一查找通路并抵达节点22，但在因目标关键码更小而试图继续向左深入时发现左子树为空[1]，至此即可确认查找失败。

一般地，在上述查找过程中，一旦发现当前节点为NULL，即说明查找范围已经缩小至空，查找失败；否则，视关键码比较结果，向左（更小）或向右（更大）深入，或者报告成功（相等）。

对照中序遍历序列可见，整个过程与有序向量的二分查找过程等效，故可视作后者的推广。

■ searchIn()算法与search()接口

一般地，在子树v中查找关键码e的过程，可实现为如代码7.3所示的算法searchIn()。

```
template <typename T> //在以v为根的（AVL、SPLAY、rbTree等）BST子树中查找关键码e
static BinNodePosi(T) & searchIn ( BinNodePosi(T) & v, const T& e, BinNodePosi(T) & hot ) {
   if ( !v || ( e == v->data ) ) return v; //递归基：在节点v（或假想的通配节点）处命中
   hot = v; //一般情况：先记下当前节点，然后再
   return searchIn ( ( ( e < v->data ) ? v->lc : v->rc ), e, hot ); //深入一层，递归查找
} //返回时，返回值指向命中节点（或假想的通配哨兵），hot指向其父亲（退化时为初始值NULL）
```

代码7.3 二叉搜索树searchIn()算法的递归实现

节点的插入和删除操作，都需要首先调用查找算法，并根据查找结果确定后续的处理方式。因此，这里以引用方式传递（子）树根节点，以为后续操作提供必要的信息。

如代码7.4所示，通过调用searchIn()算法，即可实现二叉搜索树的标准接口search()。

```
template <typename T> BinNodePosi(T) & BST<T>::search ( const T& e ) //在BST中查找关键码e
{ return searchIn ( _root, e, _hot = NULL ); } //返回目标节点位置的引用，以便后续插入、删除操作
```

代码7.4 二叉搜索树search()接口

[1] 此类空节点通常对应于空孩子指针或引用，也可假想地等效为“真实”节点，后一方式不仅可简化算法描述以及退化情况的处理，也可直观地解释（B-树之类）纵贯多级存储层次的搜索树。故在后一场合，空节点也称作外部节点（external node），并等效地当作叶节点的“孩子”。这里暂采用前一方式，故空节点不在插图中出现。

■ 语义约定

以上查找算法之所以如此实现，是为了统一并简化后续不同搜索树的各种操作接口的实现。其中的技巧，主要体现于返回值和hot变量（即BinTree对象内部的_hot变量）的语义约定。

若查找成功，则searchIn()以及search()的返回值都将如图7.6(a)所示，指向一个关键码为e且真实存在的节点；若查找失败，则返回值的数值虽然为NULL，但是它作为引用将如图(b)所示，指向最后一次试图转向的空节点。对于后一情况，不妨假想地将此空节点转换为一个数值为e的哨兵节点——如此，无论成功与否，查找的返回值总是等效地指向“命中节点”。

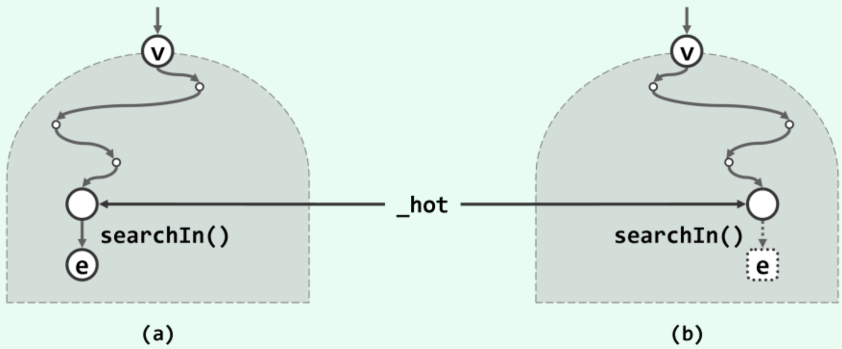

图7.6 searchIn()算法对返回值和_hot的语义定义：(a) 查找成功；(b) 查找失败

在调用searchIn()算法之前，search()接口首先将内部变量_hot初始化为NULL，然后作为引用型参数hot传递给searchIn()。在整个查找的过程中，hot变量始终指向当前节点的父亲。因此在算法返回时，按照如上定义，_hot亦将统一指向“命中节点”的父亲。

请注意，_hot节点是否拥有另一个孩子，与查找成功与否无关。查找成功时，节点e可能是叶子，也可能是内部节点；查找失败时，假想的哨兵e等效于叶节点，但可能有兄弟。

同时也请读者对照代码7.3验证，即便在退化的情况下（比如查找终止并返回于树根处），算法searchIn()的输出依然符合以上语义约定。

在7.2.6节将要介绍的删除操作中，也首先要进行查找（不妨设查找成功）。按照如上语义，命中节点必然就是待摘除节点；该节点与其父亲_hot，联合指示了删除操作的位置。7.2.5节将要介绍的插入操作，亦首先需做查找（不妨设查找失败）。按照如上语义，假想的“命中节点”也就是待插入的新节点；_hot所指向的，正是该节点可行的接入位置。

■ 效率

在二叉搜索树的每一层，查找算法至多访问一个节点，且只需常数时间，故总体所需时间应线性正比于查找路径的长度，或最终返回节点的深度。在最好情况下，目标关键码恰好出现在树根处（或其附近），此时只需$\mathcal{O}(1)$时间。然而不幸的是，对于规模为n的二叉搜索树，深度在最坏情况下可达$\Omega(n)$。比如，当该树退化为（接近于）一条单链时，发生此类情况的概率将很高。此时的单次查找可能需要线性时间并不奇怪，因为实际上这样的一棵“二分”搜索树，已经退化成了一个不折不扣的一维有序列表，而此时的查找则等效于顺序查找。

由此我们可得到启示：若要控制单次查找在最坏情况下的运行时间，须从控制二叉搜索树的高度入手。后续章节将要讨论的平衡二叉搜索树，正是基于这一思路而做的改进。

7.2.5 插入算法及其实现

■ 算法

为了在二叉搜索树中插入一个节点，首先需要利用查找算法search()确定插入的位置及方向，然后才能将新节点作为叶子插入。

以如图7.7(a)所示的二叉搜索树为例。若欲插入关键码40，则在执行search(40)之后，如图(b)所示，_hot将指向比较过的最后一个节点46，同时返回其左孩子（此时为空）的位置。于是接下来如图(c)所示，只需创建新节点40，并将其作为46的左孩子接入，拓扑意义上的节点插入即告完成。不过，为保持二叉搜索树作为数据结构的完整性和一致性，还需从节点_hot（46）出发，自底而上地逐个更新新节点40历代祖先的高度。

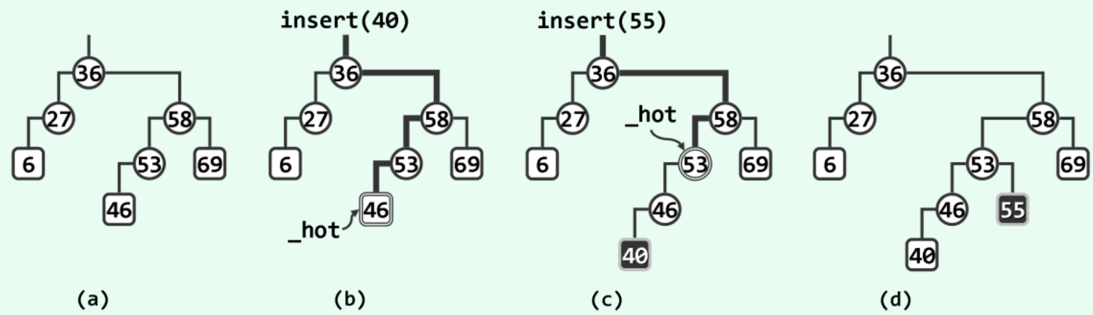

图7.7 二叉搜索树节点插入算法实例

接下来若欲插入关键码55，则在执行search(55)之后如图(c)所示，_hot将指向比较过的最后一个节点53，同时返回其右孩子（此时为空）的位置。于是如图(d)所示，创建新节点55，并将其作为53的右孩子接入。当然，此后同样需从节点_hot出发，逐代更新祖先的高度。

■ insert()接口的实现

一般地，在二叉搜索树中插入新节点e的过程，可描述为代码7.5中的函数insert()。

```
template <typename T> BinNodePosi(T) BST<T>::insert ( const T& e ) { //将关键码e插入BST树中
   BinNodePosi(T) & x = search ( e ); if ( x ) return x; //确认目标不存在（留意对_hot的设置）
   x = new BinNode<T> ( e, _hot ); //创建新节点x：以e为关键码，以_hot为父
   _size++; //更新全树规模
   updateHeightAbove ( x ); //更新x及其历代祖先的高度
   return x; //新插入的节点，必为叶子
} //无论e是否存在于原树中，返回时总有x->data == e
```

代码7.5 二叉搜索树insert()接口

首先调用search()查找e。若返回位置x非空，则说明已有雷同节点，插入操作失败。否则，x必是_hot节点的某一空孩子，于是创建这个孩子并存入e。此后，更新全树的规模记录，并调用代码5.6中的updateHeightAbove()更新x及其历代祖先的高度。

注意，按照以上实现方式，无论插入操作成功与否，都会返回一个非空位置，且该处的节点与拟插入的节点相等。如此可以确保一致性，以简化后续的操作。另外，也请对照代码7.3和代码7.4中的查找算法，体会这里对“首个节点插入空树”等特殊情况的处理手法。

■ 效率

由上可见，节点插入操作所需的时间，主要消耗于对算法search()及updateHeightAbove()的调用。后者与前者一样，在每一层次至多涉及一个节点，仅消耗$O(1)$时间，故其时间复杂度也同样取决于新节点的深度，在最坏情况下不超过全树的高度。

7.2.6 删除算法及其实现

为从二叉搜索树中删除节点，首先也需要调用算法BST::search()，判断目标节点是否的确存在于树中。若存在，则需返回其位置，然后方能相应地具体实施删除操作。

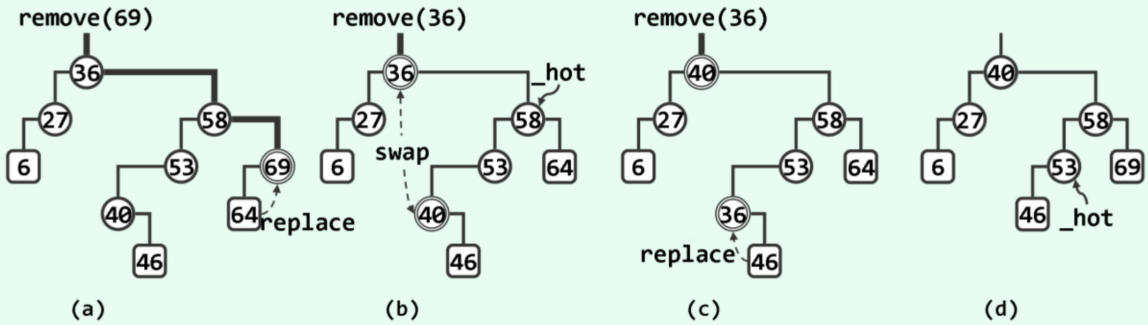

图7.8 二叉搜索树节点删除算法实例

■ 单分支情况

以如图7.8(a)所示二叉搜索树为例。若欲删除节点69，需首先通过search(69)定位待删除节点（69）。因该节点的右子树为空，故只需如图(b)所示，将其替换为左孩子（64），则拓扑意义上的节点删除即告完成。当然，为保持二叉搜索树作为数据结构的完整性和一致性，还需更新全树的规模记录，释放被摘除的节点（69），并自下而上地逐个更新替代节点（64）历代祖先的高度。注意，首个需要更新高度的祖先（58），恰好由_hot指示。

不难理解，对于没有左孩子的目标节点，也可以对称地予以处理。当然，以上同时也已涵盖了左、右孩子均不存在（即目标节点为叶节点）的情况。

那么，当目标节点的左、右孩子双全时，删除操作又该如何实施呢？

■ 双分支情况

继续上例，设拟再删除二度节点36。如图7.8(b)所示，首先调用BinNode::succ()算法，找到该节点的直接后继（40）。然后，只需如图(c)所示交换二者的数据项，则可将后继节点等效地视作待删除的目标节点。不难验证，该后继节点必无左孩子，从而相当于转化为此前相对简单的情况。于是最后可如图(d)所示，将新的目标节点（36）替换为其右孩子（46）。

请注意，在中途互换数据项之后，这一局部如图(c)所示曾经一度并不满足顺序性。但这并不要紧——不难验证，在按照上述方法完成整个删除操作之后，全树的顺序性必然又将恢复。

同样地，除了更新全树规模记录和释放被摘除节点，此时也要更新一系列祖先节点的高度。不难验证，此时首个需要更新高度的祖先（53），依然恰好由_hot指示。

■ remove()

一般地，删除关键码e的过程，可描述为如代码7.6所示的函数remove()。

```
template <typename T> bool BST<T>::remove ( const T& e ) { //从BST树中删除关键码e
   BinNodePosi(T) & x = search ( e ); if ( !x ) return false; //确认目标存在（留意_hot的设置）
   removeAt ( x, _hot ); _size--; //实施删除
   updateHeightAbove ( _hot ); //更新_hot及其历代祖先的高度
   return true;
} //删除成功与否，由返回值指示
```

代码7.6 二叉搜索树remove()接口

首先调用search()查找e。若返回位置x为空，则说明树中不含目标节点，故删除操作随即可以失败返回。否则，调用removeAt()删除目标节点x。同样，此后还需更新全树的规模，并调用函数updateHeightAbove(_hot)（121页代码5.6），更新被删除节点历代祖先的高度。

■ removeAt()

这里，实质的删除操作由removeAt()负责分情况实施，其具体实现如代码7.7所示。

```
/******************************************************************************************
 * BST节点删除算法：删除位置x所指的节点（全局静态模板函数，适用于AVL、Splay、RedBlack等各种BST）
 * 目标x在此前经查找定位，并确认非NULL，故必删除成功；与searchIn不同，调用之前不必将hot置空
 * 返回值指向实际被删除节点的接替者，hot指向实际被删除节点的父亲——二者均有可能是NULL
 ******************************************************************************************/
template <typename T>
static BinNodePosi(T) removeAt ( BinNodePosi(T) & x, BinNodePosi(T) & hot ) {
   BinNodePosi(T) w = x; //实际被摘除的节点，初值同x
   BinNodePosi(T) succ = NULL; //实际被删除节点的接替者
   if ( !HasLChild ( *x ) ) //若*x的左子树为空，则可
      succ = x = x->rc; //直接将*x替换为其右子树
   else if ( !HasRChild ( *x ) ) //若右子树为空，则可
      succ = x = x->lc; //对称地处理——注意：此时succ != NULL
   else { //若左右子树均存在，则选择x的直接后继作为实际被摘除节点，为此需要
      w = w->succ(); //（在右子树中）找到*x的直接后继*w
      swap ( x->data, w->data ); //交换*x和*w的数据元素
      BinNodePosi(T) u = w->parent;
      ( ( u == x ) ? u->rc : u->lc ) = succ = w->rc; //隔离节点*w
   }
   hot = w->parent; //记录实际被删除节点的父亲
   if ( succ ) succ->parent = hot; //并将被删除节点的接替者与hot相联
   release ( w->data ); release ( w ); return succ; //释放被摘除节点，返回接替者
}
```

代码7.7 二叉搜索树removeAt()算法

■ 效率

删除操作所需的时间，主要消耗于对search()、succ()和updateHeightAbove()的调用。在树中的任一高度，它们至多消耗$O(1)$时间。故总体的渐进时间复杂度，亦不超过全树的高度。

§7.3 平衡二叉搜索树

7.3.1 树高与性能

根据7.2节对二叉搜索树的实现与分析，search()、insert()和remove()等主要接口的运行时间，均线性正比于二叉搜索树的高度。而在最坏情况下，二叉搜索树可能彻底地退化为列表，此时的查找效率甚至会降至$\mathcal{O}(n)$，线性正比于数据集的规模。因此，若不能有效地控制树高，则就实际的性能而言，较之此前的向量和列表，二叉搜索树将无法体现出明显优势。

那么，出现上述最坏（或较坏）情况的概率有多大？或者，至少从平均复杂度的角度来看，二叉搜索树的性能是否还算令人满意？

以下，将按照两种常用的随机统计口径，就二叉搜索树的平均性能做一比较。

■ 随机生成

不妨设各节点对应于n个互异关键码{ e_1, e_2, ..., e_n }。于是按照每一排列：

$$\sigma = (e_{i_1}, e_{i_2}, \ldots, e_{i_n})$$

只要从空树开始，通过依次执行insert(e_{i_k})，即可得到这n个关键码的一棵二叉搜索树T(σ)。与随机排列σ如此相对应的二叉搜索树T(σ)，称作由σ“随机生成”（randomly generated）。

图7.9以三个关键码{ 1, 2, 3 }为例，列出了由其所有排列所生成的二叉搜索树。

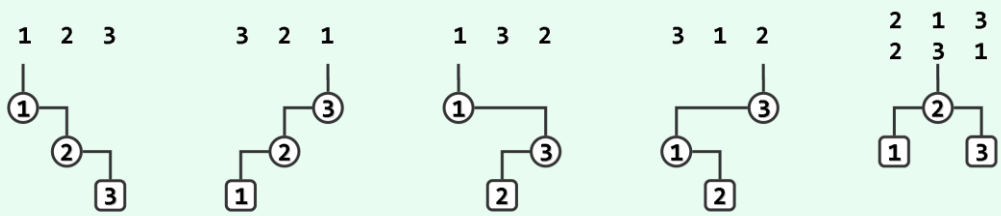

图7.9 由三个关键码{ 1, 2, 3 }的6种全排列生成的二叉搜索树

显然，任意的n个互异关键码，都可以构成n!种全排列。若各排列作为输入序列的概率均等，则只要将它们各自所生成二叉搜索树的平均查找长度进行平均，即可在一定程度上反映二叉搜索树的平均查找性能。可以证明[29][30]，在这一随机意义下，二叉搜索树的平均高度为$\Theta(\log n)$。

■ 随机组成

另一随机策略是，假定n个互异节点同时给定，然后在遵守顺序性的前提下，随机确定它们之间的拓扑联接。如此，称二叉搜索树由这组节点“随机组成”（randomly composed）。

实际上，由n个互异节点组成的二叉搜索树，总共可能有(2n)!/n!/(n + 1)!棵（习题[7-2]）。若这些树出现的概率均等，则通过对其高度做平均可知[30]，平均查找长度为$\Theta(\sqrt{n})$。

■ 比较

前一口径的$\Theta(\log n)$与后一口径的$\Theta(\sqrt{n})$之间，就渐进意义而言有实质的差别。原因何在？

读者也许已经发现，同一组关键码的不同排列所生成的二叉搜索树，未必不同。仍以图7.9为例，排列(2, 1, 3)与(2, 3, 1)生成的，实际上就是同一棵二叉搜索树。而在按照前一口径估计平均树高时，这棵树被统计了两次。实际上一般而言，越是平衡的树，被统计的次数亦越多。从这个角度讲，前一种平均的方式，在无形中高估了二叉搜索树的平均性能。因此相对而言，按照后一口径所得的估计值更加可信。

■ 树高与平均树高

实际上，即便按照以上口径统计出平均树高，仍不足以反映树高的随机分布情况。实际上，树高较大情况的概率依然可能很大。另外，理想的随机并不常见，实际应用中的情况恰恰相反，一组关键码往往会按照（接近）单调次序出现，因此频繁出现极高的搜索树也不足为怪。

另外，若removeAt()操作的确如代码7.7所示，总是固定地将待删除的二度节点与其直接后继交换，则随着操作次数的增加，二叉搜索树向左侧倾斜的趋势将愈发明显（习题[7-9]）。

7.3.2 理想平衡与适度平衡

■ 理想平衡

既然二叉搜索树的性能主要取决于高度，故在节点数目固定的前提下，应尽可能地降低高度。相应地，应尽可能地使兄弟子树的高度彼此接近，即全树尽可能地平衡。当然，包含n个节点的二叉树，高度不可能小于$\lfloor \log_2 n \rfloor$（习题[7-3]）。若树高恰好为$\lfloor \log_2 n \rfloor$，则称作理想平衡树。例如，如图5.26所示的完全二叉树，甚至如图5.27所示的满二叉树，均属此列。

遗憾的是，完全二叉树“叶节点只能出现于最底部两层”的限制过于苛刻。略做简单的组合统计不难发现，相对于二叉树所有可能的形态，此类二叉树所占比例极低；而随着二叉树规模的增大，这一比例还将继续锐减（习题[7-2]）。由此可见，从算法可行性的角度来看，有必要依照某种相对宽松的标准，重新定义二叉搜索树的平衡性。

■ 适度平衡

在渐进意义下适当放松标准之后的平衡性，称作适度平衡。

幸运的是，适度平衡的标准的确存在。比如，若将树高限制为“渐进地不超过$O(\log n)$”，则下节将要介绍的AVL树，以及下一章将要介绍的伸展树、红黑树、kd-树等，都属于适度平衡。这些变种，因此也都可归入平衡二叉搜索树（balanced binary search tree, BBST）之列。

7.3.3 等价变换

■ 等价二叉搜索树

如图7.10所示，若两棵二叉搜索树的中序遍历序列相同，则称它们彼此等价；反之亦然。

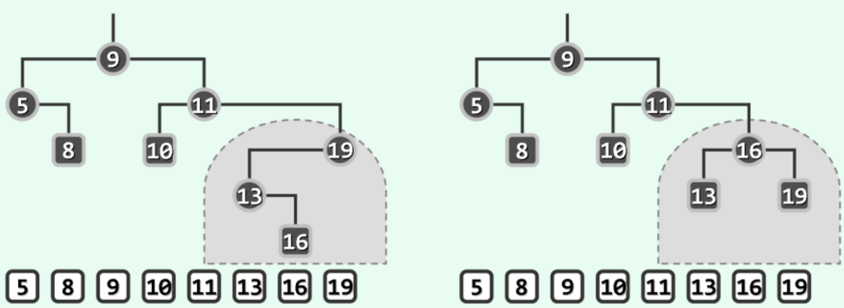

图7.10 由同一组共11个节点组成，相互等价的两棵二叉搜索树（二者在拓扑上的差异，以阴影圈出）

由该图也不难看出，虽然等价二叉搜索树中各节点的垂直高度可能有所不同，但水平次序完全一致。这一特点可概括为“上下可变，左右不乱”，它也是以下等价变换的基本特性。

■ 局部性

平衡二叉搜索树的适度平衡性，都是通过对树中每一局部增加某种限制条件来保证的。比如，在红黑树中，从树根到叶节点的通路，总是包含一样多的黑节点；在AVL树中，兄弟节点的高度相差不过1。事实上，这些限制条件设定得非常精妙，除了适度平衡性，还具有如下局部性：

1）经过单次动态修改操作后，至多只有$O(1)$处局部不再满足限制条件
2）总可在$O(\log n)$时间内，使这$O(1)$处局部（以至全树）重新满足限制条件

这就意味着：刚刚失去平衡的二叉搜索树，必然可以迅速转换为一棵等价的平衡二叉搜索树。等价二叉搜索树之间的上述转换过程，也称作等价变换。

这里的局部性至关重要。比如，尽管任何二叉搜索树都可等价变换至理想平衡的完全二叉树，然而鉴于二者的拓扑结构可能相去甚远，在最坏情况下我们为此将不得不花费$O(n)$时间。反观图7.10中相互等价的两棵二叉搜索树，右侧属于AVL树，而左侧不是。鉴于二者的差异仅限于某一局部（阴影区域），故可轻易地将后者转换为前者。

那么，此类局部性的失衡，具体地可以如何修复？如何保证修复的速度？

7.3.4 旋转调整

最基本的修复手段，就是通过围绕特定节点的旋转，实现等价前提下的局部拓扑调整。

■ zig和zag

如图7.11(a)所示，设c和Z是v的左孩子、右子树，X和Y是c的左、右子树。所谓以v为轴的zig旋转，即如图(b)所示，重新调整这两个节点与三棵子树的联接关系：将X和v作为c的左子树、右孩子，Y和Z分别作为v的左、右子树。

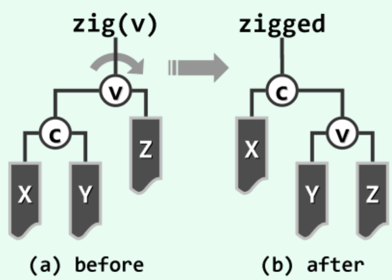

图7.11 zig(v)：顺时针旋转

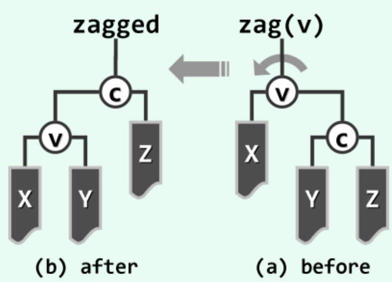

图7.12 zag(v)：逆时针旋转

可见，尽管局部结构以及子树根均有变化，但中序遍历序列仍是{ ..., X, c, Y, v, Z, ... }，故zig旋转属于等价变换。

对称地如图7.12(a)所示，设X和c是v的左子树、右孩子，Y和Z分别是c的左、右子树。所谓以v为轴的zag旋转，即如图(b)所示，重新调整这两个节点与三棵子树的联接关系：将v和Z作为c的左孩子、右子树，X和Y分别作为v的左、右子树。

同样地，旋转之后中序遍历序列依然不变，故zag旋转亦属等价变换。

■ 效率与效果

zig和zag旋转均属局部操作，仅涉及常数个节点及其之间的联接关系，故均可在常数时间内完成。正因如此，在后面实现各种二叉搜索树平衡化算法时，它们都是支撑性的基本操作。

就与树相关的指标而言，经一次zig或zag旋转之后，节点v的深度加一，节点c的深度减一；这一局部子树（乃至全树）的高度可能发生变化，但上、下幅度均不超过一层。

§7.4 AVL树

通过合理设定适度平衡的标准，并借助以上等价变换，AVL树（AVL tree）②可以实现近乎理想的平衡。在渐进意义下，AVL树可始终将其高度控制在$O(\log n)$以内，从而保证每次查找、插入或删除操作，均可在$O(\log n)$的时间内完成。

7.4.1 定义及性质

■ 平衡因子

任一节点v的平衡因子（balance factor）定义为“其左、右子树的高度差”，即

balFac(v) = height(lc(v)) - height(rc(v))

请注意，本书中空树高度取-1，单节点子树（叶节点）高度取0，与以上定义没有冲突。

所谓AVL树，即平衡因子受限的二叉搜索树——其中各节点平衡因子的绝对值均不超过1。

■ 接口定义

基于BST模板类（185页代码7.2），可直接派生出AVL模板类如代码7.8所示。

```
#include "../BST/BST.h" //基于BST实现AVL树
template <typename T> class AVL : public BST<T> { //由BST派生AVL树模板类
public:
   BinNodePosi(T) insert ( const T& e ); //插入（重写）
   bool remove ( const T& e ); //删除（重写）
// BST::search()等其余接口可直接沿用
};
```

代码7.8 基于BST定义的AVL树接口

可见，这里直接沿用了BST模板类的search()等接口，并根据AVL树的重平衡规则与算法，重写了insert()和remove()接口，其具体实现将在后续数节陆续给出。

另外，为简化对节点平衡性的判断，算法实现时可借用以下宏定义：

```
#define Balanced(x) ( stature( (x).lc ) == stature( (x).rc ) ) //理想平衡条件
#define BalFac(x) ( stature( (x).lc ) - stature( (x).rc ) ) //平衡因子
#define AvlBalanced(x) ( ( -2 < BalFac(x) ) && ( BalFac(x) < 2 ) ) //AVL平衡条件
```

代码7.9 用于简化AVL树算法描述的宏

■ 平衡性

在完全二叉树中各节点的平衡因子非0即1，故完全二叉树必是AVL树；不难举例说明，反之不然。完全二叉树的平衡性可以自然保证（习题[7-3]），那AVL树的平衡性又如何呢？可以证明，高度为h的AVL树至少包含fib(h + 3) - 1个节点。为此需做数学归纳。

作为归纳基，当h = 0时，T中至少包含fib(3) - 1 = 2 - 1 = 1个节点，命题成立；当h = 1时，T中至少包含fib(4) - 1 = 3 - 1 = 2个节点，命题也成立。

② 由G. M. Adelson-Velsky和E. M. Landis与1962年发明[36]，并以他们名字的首字母命名

假设对于高度低于h的任何AVL树，以上命题均成立。现考查高度为h的所有AVL树，并取S为其中节点最少的任何一棵（请注意，这样的S可能不止一棵）。

如图7.13，设S的根节点为r，r的左、右子树分别为S_L和S_R，将其高度记作h_L和h_R，其规模记作$|S_L|$和$|S_R|$。于是就有：

$$|S| = 1 + |S_L| + |S_R|$$

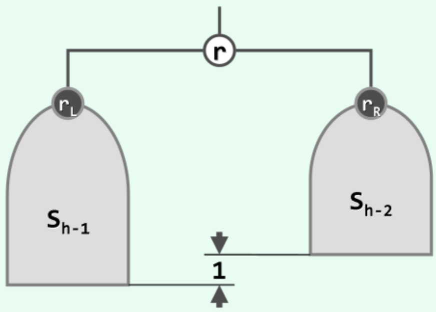

图7.13 在高度固定为h的前提下，节点最少的AVL树

作为S的子树，S_L和S_R也都是AVL树，而且高度不超过h - 1。进一步地，在考虑到AVL树有关平衡因子的要求的同时，既然S中的节点数最少，故S_L和S_R的高度只能是一个为h - 1，另一个为h - 2不失一般性，设h_L = h - 1，h_R = h - 2。而且，在所有高度为h_L（h_R）的AVL树中，S_L（S_R）中包含的节点也应该最少。因此，根据归纳假设，可得如下关系：

$$|S| = 1 + (fib(h + 2) - 1) + (fib(h + 1) - 1)$$

根据Fibonacci数列的定义，可得：

$$|S| = fib(h + 2) + fib(h + 1) - 1 = fib(h + 3) - 1$$

总而言之，高度为h的AVL树的确至少包含fib(h + 3) - 1个节点。于是反过来，包含n个节点的AVL树的高度应为$\mathcal{O}(\log n)$。因此就渐进意义而言，AVL树的确是平衡的。

■ 失衡与重平衡

AVL树与常规的二叉搜索树一样，也应支持插入、删除等动态修改操作。但经过这类操作之后，节点的高度可能发生变化，以致于不再满足AVL树的条件。

以插入操作为例，考查图7.14(b)中的AVL树，其中的关键码为字符类型。现按代码7.5中二叉搜索树的通用算法BST::insert()插入关键码'M'，于是如图(c)所示，节点'N'、'R'和'G'都将失衡。类似地，按代码7.6中二叉搜索树的通用算法BST::remove()摘除关键码'Y'之后，也会如图(a)所示导致节点'R'的失衡。

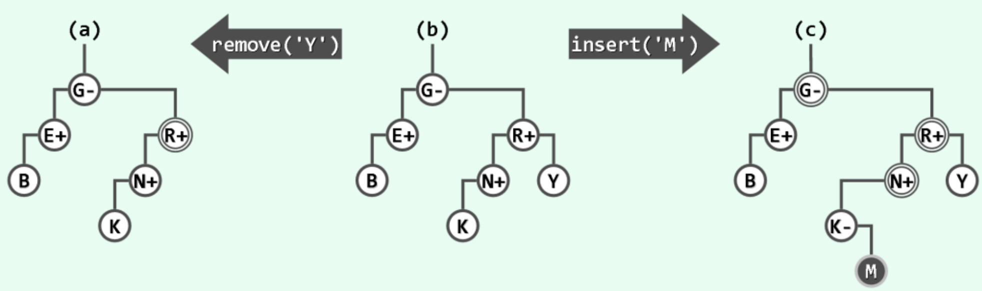

图7.14 经节点删除和插入操作后，AVL树可能失衡（加减号示意平衡因子，双圈表示失衡节点）

如此因节点x的插入或删除而暂时失衡的节点，构成失衡节点集，记作UT(x)。请注意，若x为被摘除的节点，则UT(x)仅含单个节点；但若x为被引入的节点，则UT(x)可能包含多个节点（习题[7-13]）。由以上实例，也可验证这一性质。

以下，我们从对UT(x)的分析入手，分别介绍使失衡搜索树重新恢复平衡的调整算法。

7.4.2 节点插入

■ 失衡节点集

不难看出，新引入节点x后，UT(x)中的节点都是x的祖先，且高度不低于x的祖父。以下，将其中的最深者记作g(x)。在x与g(x)之间的通路上，设p为g(x)的孩子，v为p的孩子。注意，既然g(x)不低于x的祖父，则p必是x的真祖先。

■ 重平衡

首先，需要找到如上定义的g(x)。为此，可从x出发沿parent指针逐层上行并核对平衡因子，首次遇到的失衡祖先即为g(x)。既然原树是平衡的，故这一过程只需$\mathcal{O}(\log n)$时间。

请注意，既然g(x)是因x的引入而失衡，则p和v的高度均不会低于其各自的兄弟。因此，借助如代码7.10所示的宏tallerChild()，即可反过来由g(x)找到p和v。

```
/******************************************************************************
 * 在左、右孩子中取更高者
 * 在AVL平衡调整前，借此确定重构方案
 ******************************************************************************/
#define tallerChild(x) ( \
   stature( (x)->lc ) > stature( (x)->rc ) ? (x)->lc : ( /*左高*/ \
   stature( (x)->lc ) < stature( (x)->rc ) ? (x)->rc : ( /*右高*/ \
   IsLChild( * (x) ) ? (x)->lc : (x)->rc /*等高：与父亲x同侧者（zIg-zIg或zAg-zAg）优先*/ \
   ) \
   ) \
)
```

代码7.10 恢复平衡的调整方案，决定于失衡节点的更高孩子、更高孙子节点的方向

这里并未显式地维护各节点的平衡因子，而是在需要时通过比较子树的高度直接计算。

以下，根据节点g(x)、p和v之间具体的联接方向，将采用不同的局部调整方案。分述如下。

■ 单旋

如图7.15(a)所示，设v是p的右孩子，且p是g的右孩子。

这种情况下，必是由于在子树v中刚插入某节点x，而使g(x)不再平衡。图中以虚线联接的每一对灰色方块中，其一对应于节点x，另一为空。

此时，可采用7.3.4节的技巧，做逆时针旋转zag(g(x))，得到如图(b)所示的另一棵等价二叉搜索树。

可见，经如此调整之后，g(x)必将恢复平衡。不难验证，通过zig(g(x))可以处理对称的失衡。

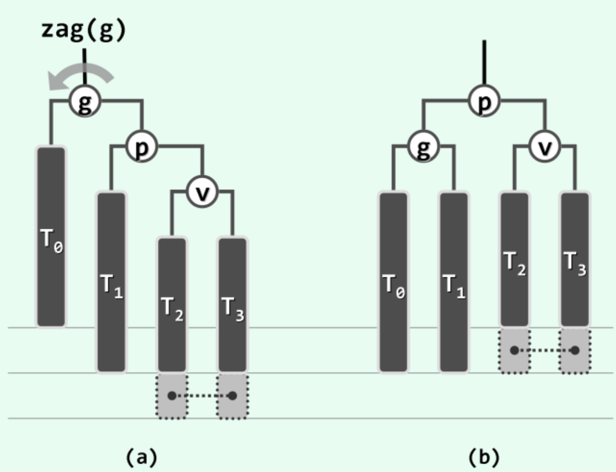

图7.15 节点插入后，通过单旋操作使AVL树重新平衡

双旋

如图7.16(a)所示，设节点v是p的左孩子，而p是g(x)的右孩子。

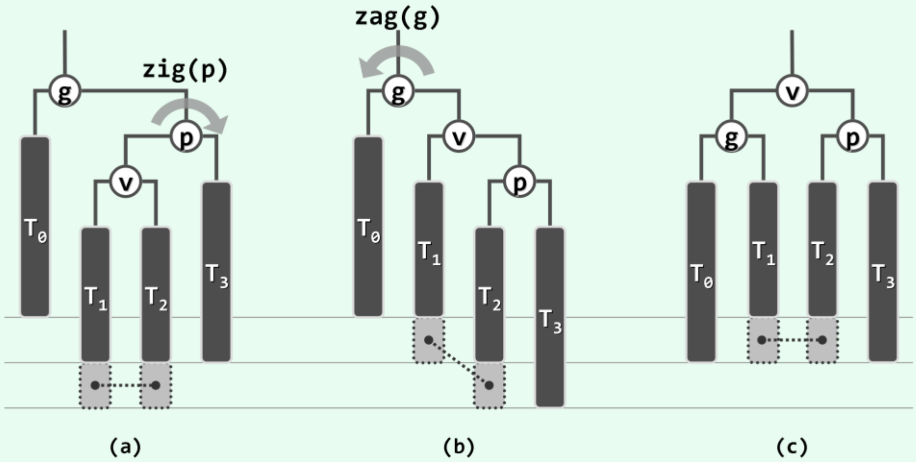

图7.16 节点插入后通过连续的两次旋转操作使AVL树重新平衡

这种情况，也必是由于在子树v中插入了新节点x，而致使g(x)不再平衡。同样地，在图中以虚线联接的每一对灰色方块中，其一对应于新节点x，另一为空。

此时，可先做顺时针旋转zig(p)，得到如图(b)所示的一棵等价二叉搜索树。再做逆时针旋转zag(g(x))，得到如图(c)所示的另一棵等价二叉搜索树。

此类分别以父子节点为轴、方向互逆的连续两次旋转，合称“双旋调整”。可见，经如此调整之后，g(x)亦必将重新平衡。不难验证，通过zag(p)和zig(g(x))可以处理对称的情况。

高度复原

纵观图7.15和图7.16可见，无论单旋或双旋，经局部调整之后，不仅g(x)能够重获平衡，而且局部子树的高度也必将复原。这就意味着，g(x)以上所有祖先的平衡因子亦将统一地复原——换而言之，在AVL树中插入新节点后，仅需不超过两次旋转，即可使整树恢复平衡。

实现

```
template <typename T> BinNodePosi(T) AVL<T>::insert ( const T& e ) { //将关键码e插入AVL树中
   BinNodePosi(T) & x = search ( e ); if ( x ) return x; //确认目标节点不存在
   BinNodePosi(T) xx = x = new BinNode<T> ( e, _hot ); _size++; //创建新节点x
// 此时，x的父亲_hot若增高，则其祖父有可能失衡
   for ( BinNodePosi(T) g = _hot; g; g = g->parent ) { //从x之父出发向上，逐层检查各代祖先g
      if ( !AvlBalanced ( *g ) ) { //一旦发现g失衡，则（采用“3 + 4”算法）使之复衡，并将子树
         FromParentTo ( *g ) = rotateAt ( tallerChild ( tallerChild ( g ) ) ); //重新接入原树
         break; //g复衡后，局部子树高度必然复原；其祖先亦必如此，故调整随即结束
      } else //否则（g依然平衡），只需简单地
         updateHeight ( g ); //更新其高度（注意：即便g未失衡，高度亦可能增加）
   } //至多只需一次调整；若果真做过调整，则全树高度必然复原
   return xx; //返回新节点位置
} //无论e是否存在于原树中，总有AVL::insert(e)->data == e
```

代码7.11 AVL树节点的插入

■ 效率

如代码7.11所示，该算法首先按照二叉搜索树的常规算法，在$O(\log n)$时间内插入新节点x。既然原树是平衡的，故至多检查$O(\log n)$个节点即可确定g(x)；如有必要，至多旋转两次，即可使局部乃至全树恢复平衡。由此可见，AVL树的节点插入操作可以在$O(\log n)$时间内完成。

7.4.3 节点删除

■ 失衡节点集

与插入操作十分不同，在摘除节点x后，以及随后的调整过程中，失衡节点集UT(x)始终至多只含一个节点（习题[7-13]）。而且若该节点g(x)存在，其高度必与失衡前相同。

另外还有一点重要的差异是，g(x)有可能就是x的父亲。

■ 重平衡

与插入操作同理，从_hot节点（7.2.6节）出发沿parent指针上行，经过$O(\log n)$时间即可确定g(x)位置。作为失衡节点的g(x)，在不包含x的一侧，必有一个非空孩子p，且p的高度至少为1。于是，可按以下规则从p的两个孩子（其一可能为空）中选出节点v：若两个孩子不等高，则v取作其中的更高者；否则，优先取v与p同向者（亦即，v与p同为左孩子，或者同为右孩子）。

以下不妨假定失衡后g(x)的平衡因子为+2（为-2的情况完全对称）。根据祖孙三代节点g(x)、p和v的位置关系，通过以g(x)和p为轴的适当旋转，同样可以使得这一局部恢复平衡。

■ 单旋

如图7.17(a)所示，由于在T_3中删除了节点而致使g(x)不再平衡，但p的平衡因子非负时，通过以g(x)为轴顺时针旋转一次即可恢复局部的平衡。平衡后的局部子树如图(b)所示。

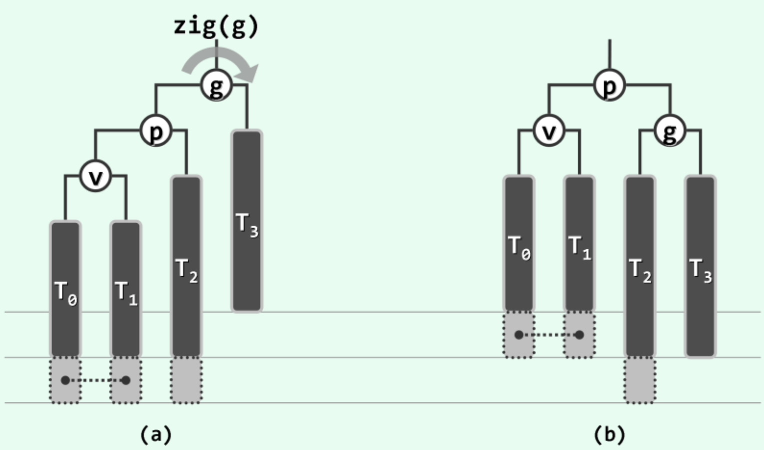

图7.17 节点删除后经一次旋转恢复局部平衡

同样地这里约定，图中以虚线联接的灰色方块所对应的节点，不能同时为空；T_2底部的灰色方块所对应的节点，可能为空，也可能非空。

■ 双旋

如图7.18(a)所示，g(x)失衡时若p的平衡因子为-1，则经过以p为轴的一次逆时针旋转之后（图(b)），即可转化为图7.17(a)的情况。

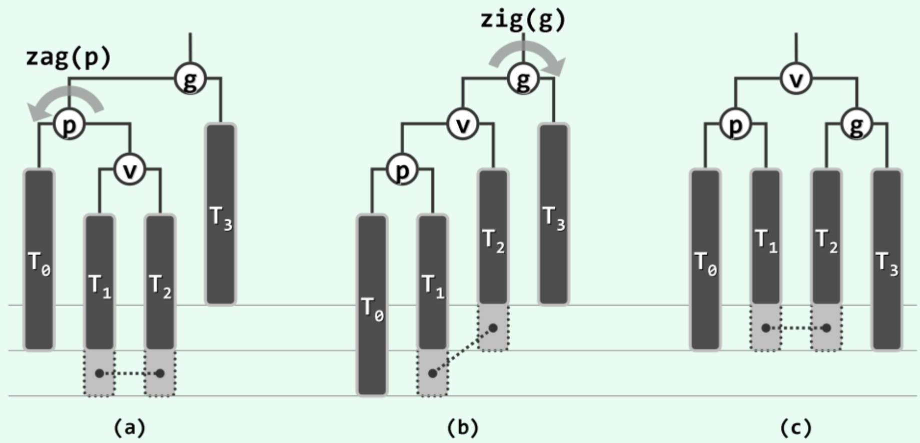

图7.18 节点删除后通过两次旋转恢复局部平衡

接着再套用上一情况的处理方法，以g(x)为轴顺时针旋转，即可恢复局部平衡（图(c)）。

■ 失衡传播

与插入操作不同，在删除节点之后，尽管也可通过单旋或双旋调整使局部子树恢复平衡，但复平衡之后，局部子树的高就全局而言，依然可能再次失衡。若能仔细观察图7.17(b)和图7.18(c)，则不难发现：g(x)恢度却可能降低。这与引入节点之后的重平衡后完全不同——在上一节我们已看到，后者不仅能恢复子树的平衡性，也同时能恢复子树的高度。

设g(x)复衡之后，局部子树的高度的确降低。此时，若g(x)原本属于某一更高祖先的更短分支，则因为该分支现在又进一步缩短，从而会致使该祖先失衡。在摘除节点之后的调整过程中，这种由于低层失衡节点的重平衡而致使其更高层祖先失衡的现象，称作“失衡传播”。

请注意，失衡传播的方向必然自底而上，而不致于影响到后代节点。在此过程中的任一时刻，至多只有一个失衡的节点；高层的某一节点由平衡转为失衡，只可能发生在下层失衡节点恢复平衡之后。因此，可沿parent指针逐层遍历所有祖先，每找到一个失衡的祖先节点，即可套用以上方法使之恢复平衡（习题[7-19]）。

■ 实现

以上算法过程，可描述并实现如代码7.12所示。

```
template <typename T> bool AVL<T>::remove ( const T& e ) { //从AVL树中删除关键码e
   BinNodePosi(T) & x = search ( e ); if ( !x ) return false; //确认目标存在（留意_hot的设置）
   removeAt ( x, _hot ); _size--; //先按BST规则删除之（此后，原节点之父_hot及其祖先均可能失衡）
   for ( BinNodePosi(T) g = _hot; g; g = g->parent ) { //从_hot出发向上，逐层检查各代祖先g
      if ( !AvlBalanced ( *g ) ) //一旦发现g失衡，则（采用“3 + 4”算法）使之复衡，并将该子树联至
         g = FromParentTo ( *g ) = rotateAt ( tallerChild ( tallerChild ( g ) ) ); //原父亲
      updateHeight ( g ); //并更新其高度（注意：即便g未失衡，高度亦可能降低）
   } //可能需做Omega(logn)次调整——无论是否做过调整，全树高度均可能降低
   return true; //删除成功
} //若目标节点存在且被删除，返回true；否则返回false
```

代码7.12 AVL树节点的删除

■ 效率

由上可见，较之插入操作，删除操作可能需在重平衡方面多花费一些时间。不过，既然需做重平衡的节点都是x的祖先，故重平衡过程累计只需不过$\mathcal{O}(\log n)$时间（习题[7-17]）。综合各方面的消耗，AVL树的节点删除操作总体的时间复杂度依然是$\mathcal{O}(\log n)$。

7.4.4 统一重平衡算法

上述重平衡的方法，需要根据失衡节点及其孩子节点、孙子节点的相对位置关系，分别做单旋或双旋调整。按照这一思路直接实现调整算法，代码量大且流程繁杂，必然导致调试困难且容易出错。为此，本节将引入一种更为简明的统一处理方法。

无论对于插入或删除操作，新方法也同样需要从刚发生修改的位置x出发逆行而上，直至遇到最低的失衡节点g(x)。于是在g(x)更高一侧的子树内，其孩子节点p和孙子节点v必然存在，而且这一局部必然可以g(x)、p和v为界，分解为四棵子树——按照图7.15至图7.18中的惯例，将它们按中序遍历次序重命名为T_0至T_3。

若同样按照中序遍历次序，重新排列g(x)、p和v，并将其命名为a、b和c，则这一局部的中序遍历序列应为：

$$\{ T_0,\ a,\ T_1,\ b,\ T_2,\ c,\ T_3 \}$$

这就意味着，这一局部应等价于如图7.19所示的子树。更重要的是，纵观图7.15至图7.18可见，这四棵子树的高度相差不超过一层，故只需如图7.19所示将这三个节点与四棵子树重新“组装”起来，恰好即是一棵AVL树！

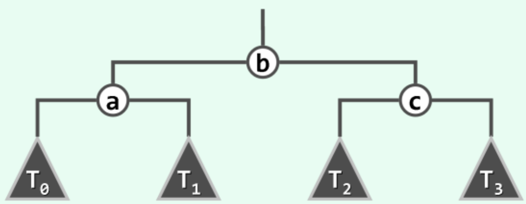

图7.19 节点插入后的统一重新平衡

实际上，这一理解涵盖了此前两节所有的单旋和双旋情况。相应的重构过程，仅涉及局部的三个节点及其四棵子树，故称作“3 + 4”重构。其具体实现如代码7.13所示。

```
/******************************************************************************************
 * 按照“3 + 4”结构联接3个节点及其四棵子树，返回重组之后的局部子树根节点位置（即b）
 * 子树根节点与上层节点之间的双向联接，均须由上层调用者完成
 * 可用于AVL和RedBlack的局部平衡调整
 ******************************************************************************************/
template <typename T> BinNodePosi(T) BST<T>::connect34 (
   BinNodePosi(T) a, BinNodePosi(T) b, BinNodePosi(T) c,
   BinNodePosi(T) T0, BinNodePosi(T) T1, BinNodePosi(T) T2, BinNodePosi(T) T3
) {
   a->lc = T0; if ( T0 ) T0->parent = a;
   a->rc = T1; if ( T1 ) T1->parent = a; updateHeight ( a );
```

```
   c->lc = T2; if ( T2 ) T2->parent = c;
   c->rc = T3; if ( T3 ) T3->parent = c; updateHeight ( c );
   b->lc = a; a->parent = b;
   b->rc = c; c->parent = b; updateHeight ( b );
   return b; //该子树新的根节点
}
```

代码7.13 “3 + 4”重构

利用以上connect34()算法，即可视不同情况，按如下具体方法完成重平衡：

```
/******************************************************************************************
 * BST节点旋转变换统一算法（3节点 + 4子树），返回调整之后局部子树根节点的位置
 * 注意：尽管子树根会正确指向上层节点（如果存在），但反向的联接须由上层函数完成
 ******************************************************************************************/
template <typename T> BinNodePosi(T) BST<T>::rotateAt ( BinNodePosi(T) v ) { //v为非空孙辈节点
   BinNodePosi(T) p = v->parent; BinNodePosi(T) g = p->parent; //视v、p和g相对位置分四种情况
   if ( IsLChild ( *p ) ) /* zig */
      if ( IsLChild ( *v ) ) { /* zig-zig */
         p->parent = g->parent; //向上联接
         return connect34 ( v, p, g, v->lc, v->rc, p->rc, g->rc );
      } else { /* zig-zag */
         v->parent = g->parent; //向上联接
         return connect34 ( p, v, g, p->lc, v->lc, v->rc, g->rc );
      }
   else  /* zag */
      if ( IsRChild ( *v ) ) { /* zag-zag */
         p->parent = g->parent; //向上联接
         return connect34 ( g, p, v, g->lc, p->lc, v->lc, v->rc );
      } else { /* zag-zig */
         v->parent = g->parent; //向上联接
         return connect34 ( g, v, p, g->lc, v->lc, v->rc, p->rc );
      }
}
```

代码7.14 AVL树的统一重平衡

将图7.19与图7.15至图7.18做一比对即可看出，统一调整算法的效果，的确与此前的单旋、双旋算法完全一致。另外不难验证，新算法的复杂度也依然是$O(1)$。