第8章

高级搜索树

除了AVL树，本章将按照如图7.1所示的总体框架，继续介绍平衡二叉搜索树家族中的其它成员。首先，鉴于数据访问的局部性在实际应用中普遍存在，将按照“最常用者优先”的启发策略，引入并实现伸展树。尽管最坏情况下其单次操作需要$\mathcal{O}(n)$时间，但分摊而言仍在$\mathcal{O}(\log n)$以内。构思巧妙，实现简洁，加上适用广泛，这些特点都使得伸展树具有别样的魅力。

接下来，通过对平衡二叉搜索树的推广，引入平衡多路搜索树，并着重讨论作为其中典型代表的B-树。借助此类结构，可以有效地弥合不同存储级别之间，在访问速度上的巨大差异。

对照4阶B-树，还将引入并实现红黑树。红黑树不仅能保持全树的适度平衡，从而有效地控制单次操作的时间成本，而且可以将每次重平衡过程执行的结构性调整，控制在常数次数以内。后者也是该树有别于其它变种的关键特性，它不仅保证了红黑树更高的实际计算效率，更为持久性结构（persistent structure）之类高级数据结构的实现，提供了直接而有效的方法。

最后，将针对平面范围查询应用，介绍基于平面子区域正交划分的kd-树结构。该结构是对四叉树（quadtree）和八叉树（octree）等结构的一般性推广，它也为计算几何类应用问题的求解，提供了一种基本的模式和有效的方法。

§8.1 伸展树

与前一章的AVL树一样，伸展树（splay tree）[①]也是平衡二叉搜索树的一种形式。相对于前者，后者的实现更为简捷。伸展树无需时刻都严格地保持全树的平衡，但却能够在任何足够长的真实操作序列中，保持分摊意义上的高效率。伸展树也不需要对基本的二叉树节点结构，做任何附加的要求或改动，更不需要记录平衡因子或高度之类的额外信息，故适用范围更广。

8.1.1 局部性

信息处理的典型模式是，将所有数据项视作一个集合，并将其组织为某种适宜的数据结构，进而借助操作接口高效访问。本书介绍的搜索树、词典和优先级队列等，都可归于此类。

为考查和评价各操作接口的效率，除了从最坏情况的角度出发，也可假定所有操作彼此独立、次序随机且概率均等，并从平均情况的角度出发。然而，后一尺度所依赖的假定条件，往往并不足以反映真实的情况。实际上，通常在任意数据结构的生命期内，不仅执行不同操作的概率往往极不均衡，而且各操作之间具有极强的相关性，并在整体上多呈现出极强的规律性。其中最为典型的，就是所谓的“数据局部性”（data locality），这包括两个方面的含义：

1）刚刚被访问过的元素，极有可能在不久之后再次被访问到
2）将被访问的下一元素，极有可能就处于不久之前被访问过的某个元素的附近

充分利用好此类特性，即可进一步地提高数据结构和算法的效率。比如习题[3-6]中的自调

① 由D. D. Sleator和R. E. Tarjan于1985年发明[41]

整列表，就是通过“即用即前移”的启发式策略，将最为常用的数据项集中于列表的前端，从而使得单次操作的时间成本大大降低。同样地，类似的策略也可应用于二叉搜索树。

就二叉搜索树而言，数据局部性具体表现为：

1）刚刚被访问过的节点，极有可能在不久之后再次被访问到
2）将被访问的下一节点，极有可能就处于不久之前被访问过的某个节点的附近

因此，只需将刚被访问的节点，及时地“转移”至树根（附近），即可加速后续的操作。当然，转移前后的搜索树必须相互等价，故为此仍需借助7.3.4节中等价变换的技巧。

8.1.2 逐层伸展

■ 简易伸展树

一种直接方式是：每访问过一个节点之后，随即反复地以它的父节点为轴，经适当的旋转将其提升一层，直至最终成为树根。以图8.1为例，若深度为3的节点E刚被访问——无论查找或插入，甚至“删除”——都可通过3次旋转，将该树等价变换为以E为根的另一棵二叉搜索树。

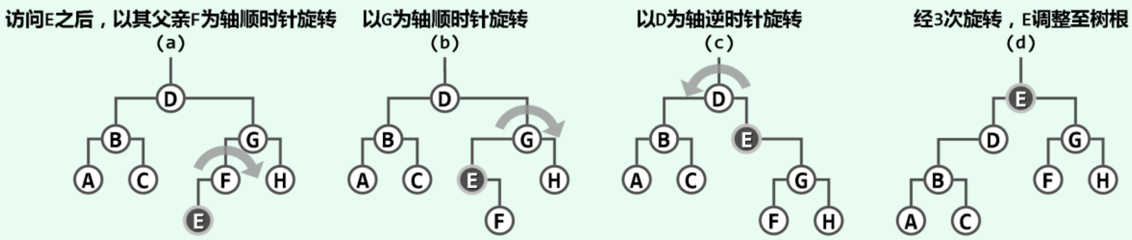

图8.1 通过自下而上的一系列等价变换，可使任一节点上升至树根

随着节点E的逐层上升，两侧子树的结构也不断地调整，故这一过程也形象地称作伸展（splaying），而采用这一调整策略的二叉搜索树也因此得名。不过，为实现真正意义上的伸展树，还须对以上策略做点微妙而本质的改进。之所以必须改进，是因为目前的策略仍存在致命的缺陷——对于很多访问序列，单次访问的分摊时间复杂度在极端情况下可能高达$\Omega(n)$。

■ 最坏情况

不难验证，若从空树开始依次插入关键码{ 1, 2, 3, 4, 5 }，且其间采用如上调整策略，则可得到如图8.2(a)所示的二叉搜索树。

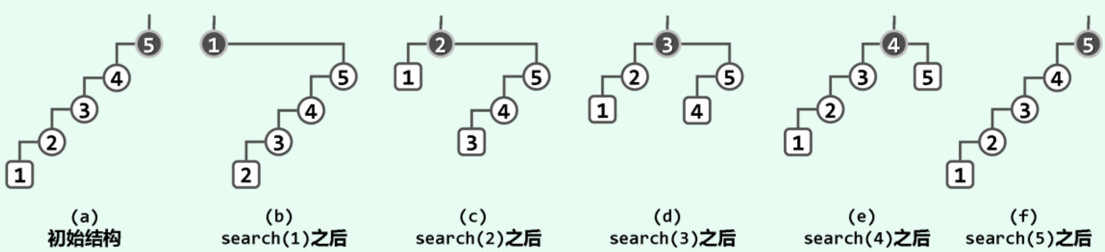

图8.2 简易伸展树的最坏情况

接下来，若通过search()接口，再由小到大地依次访问各节点一次，则该树在各次访问之后的结构形态将如图(b~f)所示。

可见，在各次访问之后，为将对应节点伸展调整至树根，分别需做4、4、3、2和1次旋转。

一般地，若节点总数为n，则旋转操作的总次数应为：

$$(n - 1) + \{ (n - 1) + (n - 2) + ... + 1 \}$$
$$= (n^2 + n - 2)/2 = \Omega(n^2)$$

如此分摊下来，每次访问平均需要Ω(n)时间。很遗憾，这一效率不仅远远低于AVL树，而且甚至与原始的二叉搜索树的最坏情况相当。而事实上，问题还远不止于此。

稍做比对即不难发现，图8.2(a)与(f)中二叉搜索树的结构完全相同。也就是说，经过以上连续的5次访问之后，全树的结构将会复原！这就意味着，以上情况可以持续地再现。

当然，这一实例，完全可以推广至规模任意的二叉搜索树。于是对于规模为任意n的伸展树，只要按关键码单调的次序，周期性地反复进行查找，则无论总的访问次数m >> n有多大，就分摊意义而言，每次访问都将需要Ω(n)时间！

那么，这类最坏的访问序列能否回避？具体地，又应该如何回避？

8.1.3 双层伸展

为克服上述伸展调整策略的缺陷，一种简便且有效的方法就是：将逐层伸展改为双层伸展。具体地，每次都从当前节点v向上追溯两层（而不是仅一层），并根据其父亲p以及祖父g的相对位置，进行相应的旋转。以下分三类情况，分别介绍具体的处理方法。

■ **zig-zig/zag-zag**

如图8.3(a)所示，设v是p的左孩子，且p也是g的左孩子；设W和X分别是v的左、右子树，Y和Z分别是p和g的右子树。

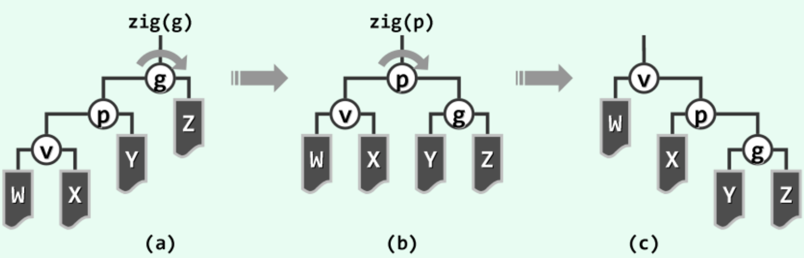

图8.3 通过zig-zig操作，将节点v上推两层

针对这种情况，首先以节点g为轴做顺时针旋转zig(g)，其效果如图(b)所示。然后，再以p为轴做顺时针旋转zig(p)，其效果如图(c)所示。如此连续的两次zig旋转，合称zig-zig调整。

自然地，另一完全对称的情形——v是p的右孩子，且p也是g的右孩子——则可通过连续的两次逆时针旋转实现调整，合称zag-zag操作。这一操作的具体过程，请读者独立绘出。

■ **zig-zag/zag-zig**

如图8.4(a)所示，设v是p的左孩子，而p是g的右孩子；设W是g的左子树，X和Y分别是v的左、右子树，Z是p的右子树。

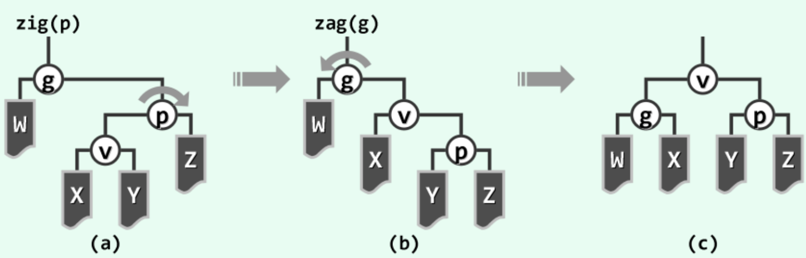

图8.4 通过zig-zag操作，将节点v上推两层

针对这种情况，首先以节点p为轴做顺时针旋转zig(p)，其效果如(b)所示。然后，再以g为轴做逆时针旋转zag(g)，其效果如图(c)所示。如此zig旋转再加zag旋转，合称zig-zag调整。

同样地，另一完全对称的情形——v是p的右孩子，而p是g的左孩子——则可通过zag旋转再加zig旋转实现调整，合称zag-zig操作。这一操作的具体过程，请读者独立绘出。

zig/zag

如图8.5(a)所示，若v最初的深度为奇数，则经过若干次双层调整至最后一次调整时，v的父亲p即是树根r。将v的左、右子树记作X和Y，节点p = r的另一子树记作Z。

此时，只需围绕p = r做顺时针旋转zig(p)，即可如图(b)所示，使v最终攀升至树根，从而结束整个伸展调整的过程。

zag调整与之对称，其过程请读者独立绘出。

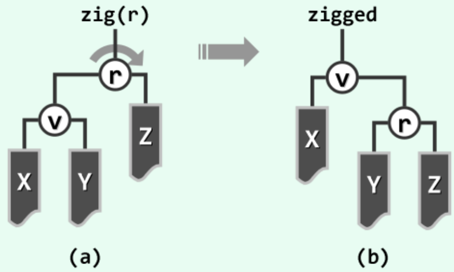

图8.5 通过zig操作，将节点v上推一层，成为树根

效果与效率

综合以上各种情况，每经过一次双层调整操作，节点v都会上升两层。若v的初始深度depth(v)为偶数，则最终v将上升至树根。若depth(v)为奇数，则当v上升至深度为1时，不妨最后再相应地做一次zig或zag单旋操作。无论如何，经过depth(v)次旋转后，v最终总能成为树根。

重新审视图8.2的最坏实例不难发现，这一访问序列导致Ω(n)平均单次访问时间的原因，可以解释为：在这一可持续重复的过程中，二叉搜索树的高度始终不小于⌊n/2⌋；而且，至少有一半的节点在接受访问时，不仅没有如最初设想的那样靠近树根，而且反过来恰恰处于最底层。从树高的角度看，问题根源也可再进一步地解释为：在持续访问的过程中，树高依算术级数逐步从n - 1递减至⌊n/2⌋，然后再逐步递增回到n - 1。那么，采用上述双层伸展的策略将每一刚被访问过的节点推至树根，可否避免如图8.2所示的最坏情况呢？

稍作对比不难看出，就调整之后的局部结构而言，zig-zag和zag-zig调整与此前的逐层伸展完全一致（亦等效于AVL树的双旋调整），而zig-zig和zag-zag调整则有所不同。事实上，后者才是双层伸展策略优于逐层伸展策略的关键所在。

以如图8.6(b)所示的二叉搜索树为例，在find(1)操作之后，采用逐层调整策略与双层调整策略的效果，分别如图(a)和图(c)所示。

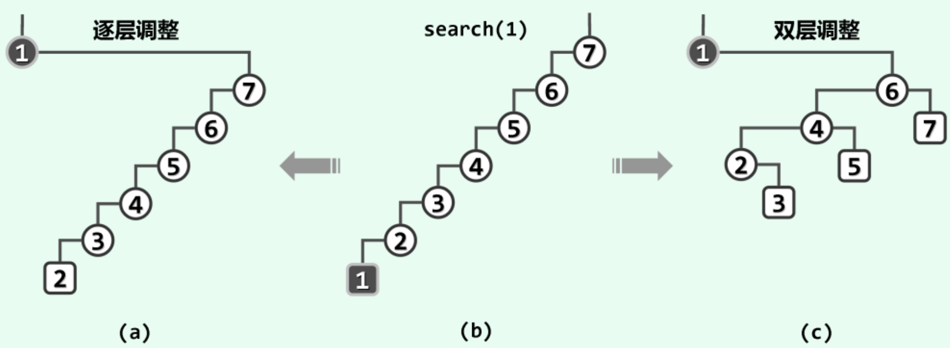

图8.6 双层调整策略的高度折半效果

可见，最深节点（1）被访问之后再经过双层调整，不仅同样可将该节点伸展至树根，而且同时可使树的高度接近于减半。就树的形态而言，双层伸展策略可“智能”地“折叠”被访问的子树分支，从而有效地避免对长分支的连续访问。这就意味着，即便节点v的深度为Ω(n)，双层伸展策略既可将v推至树根，亦可令对应分支的长度以几何级数（大致折半）的速度收缩。

图8.7则给出了一个节点更多、更具一般性的例子，从中可更加清晰地看出这一效果。

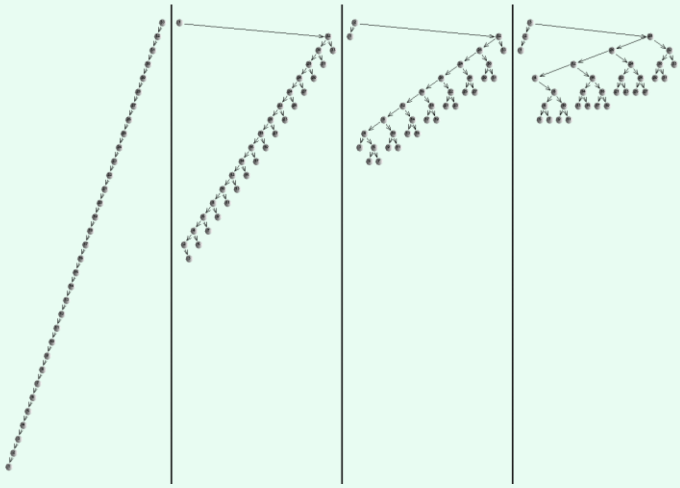

图8.7 伸展树中较深的节点一旦被访问到，对应分支的长度将随即减半

尽管在任一时刻伸展树中都可能存在很深的节点，但与含羞草类似地，一旦这类“坏”节点被“碰触”到，经过随后的双层伸展，其对应的分支都会收缩至长度大致折半。于是，即便每次都“恶意地”试图访问最底层节点，最坏情况也不会持续发生。可见，伸展树虽不能杜绝最坏情况的发生，却能有效地控制最坏情况发生的频度，从而在分摊意义下保证整体的高效率。

更准确地，Tarjan等人[41]采用势能分析法（potential analysis）业已证明，在改用“双层伸展”策略之后，伸展树的单次操作均可在分摊的$\mathcal{O}(\log n)$时间内完成（习题[8-2]）。

8.1.4 伸展树的实现

■ 伸展树接口定义

基于BST类，可定义伸展树模板类Splay如代码8.1所示。

```
#include "../BST/BST.h" //基于BST实现Splay
template <typename T> class Splay : public BST<T> { //由BST派生的Splay树模板类
protected:
   BinNodePosi(T) splay ( BinNodePosi(T) v ); //将节点v伸展至根
public:
   BinNodePosi(T) & search ( const T& e ); //查找（重写）
   BinNodePosi(T) insert ( const T& e ); //插入（重写）
   bool remove ( const T& e ); //删除（重写）
};
```

代码8.1 基于BST定义的伸展树接口

可见，这里直接沿用二叉搜索树类，并根据伸展树的平衡规则，重写了三个基本操作接口search()、insert()和remove()，另外，针对伸展调整操作，设有一个内部保护型接口splay()。

这些接口的具体实现将在以下数节陆续给出。需强调的是，与一般的二叉搜索树不同，伸展树的查找也会引起整树的结构调整，故search()操作也需重写。

■ 伸展算法的实现

8.1.3节所述的伸展调整方法，可具体实现如代码8.2所示。

```
template <typename NodePosi> inline //在节点*p与*lc（可能为空）之间建立父（左）子关系
void attachAsLChild ( NodePosi p, NodePosi lc ) { p->lc = lc; if ( lc ) lc->parent = p; }

template <typename NodePosi> inline //在节点*p与*rc（可能为空）之间建立父（右）子关系
void attachAsRChild ( NodePosi p, NodePosi rc ) { p->rc = rc; if ( rc ) rc->parent = p; }

template <typename T> //Splay树伸展算法：从节点v出发逐层伸展
BinNodePosi(T) Splay<T>::splay ( BinNodePosi(T) v ) { //v为因最近访问而需伸展的节点位置
   if ( !v ) return NULL; BinNodePosi(T) p; BinNodePosi(T) g; //*v的父亲与祖父
   while ( ( p = v->parent ) && ( g = p->parent ) ) { //自下而上，反复对*v做双层伸展
      BinNodePosi(T) gg = g->parent; //每轮之后*v都以原曾祖父（great-grand parent）为父
      if ( IsLChild ( *v ) )
         if ( IsLChild ( *p ) ) { //zig-zig
            attachAsLChild ( g, p->rc ); attachAsLChild ( p, v->rc );
            attachAsRChild ( p, g ); attachAsRChild ( v, p );
         } else { //zig-zag
            attachAsLChild ( p, v->rc ); attachAsRChild ( g, v->lc );
            attachAsLChild ( v, g ); attachAsRChild ( v, p );
         }
      else if ( IsRChild ( *p ) ) { //zag-zag
         attachAsRChild ( g, p->lc ); attachAsRChild ( p, v->lc );
         attachAsLChild ( p, g ); attachAsLChild ( v, p );
      } else { //zag-zig
         attachAsRChild ( p, v->lc ); attachAsLChild ( g, v->rc );
         attachAsRChild ( v, g ); attachAsLChild ( v, p );
      }
      if ( !gg ) v->parent = NULL; //若*v原先的曾祖父*gg不存在，则*v现在应为树根
      else //否则，*gg此后应该以*v作为左或右孩子
         ( g == gg->lc ) ? attachAsLChild ( gg, v ) : attachAsRChild ( gg, v );
      updateHeight ( g ); updateHeight ( p ); updateHeight ( v );
   } //双层伸展结束时，必有g == NULL，但p可能非空
   if ( p = v->parent ) { //若p果真非空，则额外再做一次单旋
      if ( IsLChild ( *v ) ) { attachAsLChild ( p, v->rc ); attachAsRChild ( v, p ); }
      else                   { attachAsRChild ( p, v->lc ); attachAsLChild ( v, p ); }
      updateHeight ( p ); updateHeight ( v );
   }
   v->parent = NULL; return v;
} //调整之后新树根应为被伸展的节点，故返回该节点的位置以便上层函数更新树根
```

代码8.2 伸展树节点的调整

查找算法的实现

在伸展树中查找任一关键码e的过程，可实现如代码8.3所示。

```
template <typename T> BinNodePosi(T) & Splay<T>::search ( const T& e ) { //在伸展树中查找e
   BinNodePosi(T) p = searchIn ( _root, e, _hot = NULL );
   _root = splay ( ( p ? p : _hot ) ); //将最后一个被访问的节点伸展至根
   return _root;
} //与其它BST不同，无论查找成功与否，_root都指向最后被访问的节点
```

代码8.3 伸展树节点的查找

首先，调用二叉搜索树的通用算法searchIn()（代码7.3）尝试查找具有关键码e的节点。无论查找是否成功，都继而调用splay()算法，将查找终止位置处的节点伸展到树根。

插入算法的实现

为将节点插至伸展树中，固然可以调用二叉搜索树的标准插入算法BST::insert()（188页代码7.5），再通过双层伸展，将新插入的节点提升至树根。

然而，以上接口Splay::search()已集成了splay()伸展功能，故查找返回后，树根节点要么等于查找目标（查找成功），要么就是_hot，而且恰为拟插入节点的直接前驱或直接后继（查找失败）。因此，不妨改用如下方法实现Splay::insert()接口。

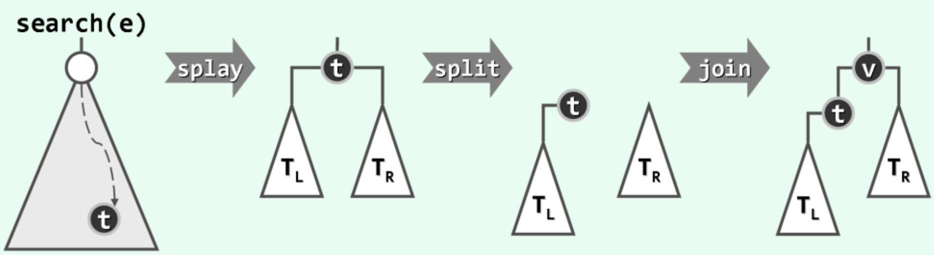

图8.8 伸展树的节点插入

如图8.8所示，为将关键码e插至伸展树T中，首先调用伸展树查找接口Splay::search(e)，查找该关键码（图(a)）。于是，其中最后被访问的节点t，将通过伸展被提升为树根，其左、右子树分别记作T_L和T_R（图(b)）。

接下来，根据e与t的大小关系（不妨排除二者相等的情况），以t为界将T分裂为两棵子树。比如，不失一般性地设e大于t。于是，可切断t与其右孩子之间的联系（图(c)），再将以e为关键码的新节点v作为树根，并以t作为其左孩子，以T_R作为其右子树（图(d)）。

v小于t的情况与此完全对称，请读者独立做出分析。

上述算法过程，可具体实现如代码8.4所示。

```
template <typename T> BinNodePosi(T) Splay<T>::insert ( const T& e ) { //将关键码e插入伸展树中
   if ( !_root ) { _size++; return _root = new BinNode<T> ( e ); } //处理原树为空的退化情况
   if ( e == search ( e )->data ) return _root; //确认目标节点不存在
   _size++; BinNodePosi(T) t = _root; //创建新节点。以下调整<=7个指针以完成局部重构
```

```
    if ( _root->data < e ) { //插入新根，以t和t->rc为左、右孩子
        t->parent = _root = new BinNode<T> ( e, NULL, t, t->rc ); //2 + 3个
        if ( HasRChild ( *t ) ) { t->rc->parent = _root; t->rc = NULL; } //<= 2个
    } else { //插入新根，以t->lc和t为左、右孩子
        t->parent = _root = new BinNode<T> ( e, NULL, t->lc, t ); //2 + 3个
        if ( HasLChild ( *t ) ) { t->lc->parent = _root; t->lc = NULL; } //<= 2个
    }
    updateHeightAbove ( t ); //更新t及其祖先（实际上只有_root一个）的高度
    return _root; //新节点必然置于树根，返回之
} //无论e是否存在于原树中，返回时总有_root->data == e
```

代码8.4 伸展树节点的插入

尽管伸展树并不需要记录和维护节点高度，为与其它平衡二叉搜索树的实现保持统一，这里还是对节点的高度做了及时的更新。出于效率的考虑，实际应用中可视情况，省略这类更新。

■ 删除算法的实现

为从伸展树中删除节点，固然也可以调用二叉搜索树标准的节点删除算法BST::remove()（190页代码7.6），再通过双层伸展，将该节点此前的父节点提升至树根。

然而同样地，在实施删除操作之前，通常都需要调用Splay::search()定位目标节点，而该接口已经集成了splay()伸展功能，从而使得在成功返回后，树根节点恰好就是待删除节点。因此，亦不妨改用如下策略，以实现Splay::remove()接口。

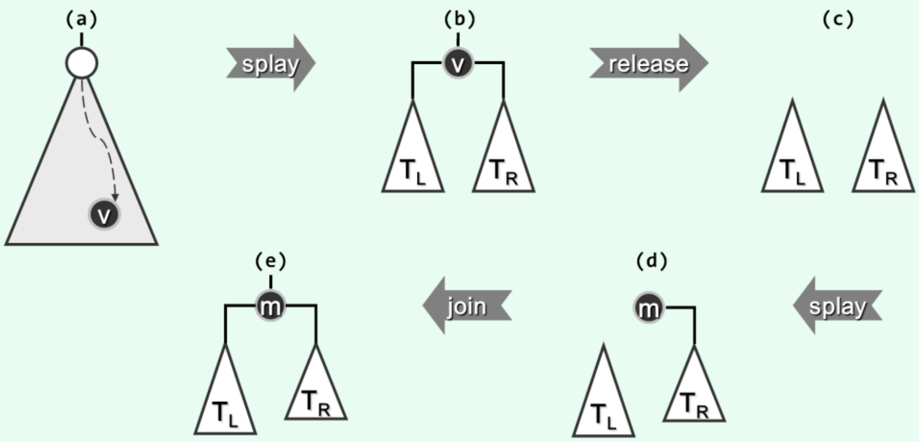

图8.9 伸展树的节点删除

如图8.9所示，为从伸展树T中删除关键码为e的节点，首先亦调用接口Splay::search(e)，查找该关键码，且不妨设命中节点为v（图(a)）。于是，v将随即通过伸展被提升为树根，其左、右子树分别记作T_L和T_R（图(b)）。接下来，将v摘除（图(c)）。然后，在T_R中再次查找关键码e。尽管这一查找注定失败，却可以将T_R中的最小节点m，伸展提升为该子树的根。

得益于二叉搜索树的顺序性，此时节点m的左子树必然为空；同时，T_L中所有节点都应小于m（图(d)）。于是，只需将T_L作为左子树与m相互联接，即可得到一棵完整的二叉搜索树（图(e)）。如此不仅删除了v，而且既然新树根m在原树中是v的直接后继，故数据局部性也得到了利用。

上述算法过程，可具体实现如代码8.5所示。

```
template <typename T> bool Splay<T>::remove ( const T& e ) { //从伸展树中删除关键码e
   if ( !_root || ( e != search ( e )->data ) ) return false; //若树空或目标不存在，则无法删除
   BinNodePosi(T) w = _root; //assert: 经search()后节点e已被伸展至树根
   if ( !HasLChild ( *_root ) ) { //若无左子树，则直接删除
      _root = _root->rc; if ( _root ) _root->parent = NULL;
   } else if ( !HasRChild ( *_root ) ) { //若无右子树，也直接删除
      _root = _root->lc; if ( _root ) _root->parent = NULL;
   } else { //若左右子树同时存在，则
      BinNodePosi(T) lTree = _root->lc;
      lTree->parent = NULL; _root->lc = NULL; //暂时将左子树切除
      _root = _root->rc; _root->parent = NULL; //只保留右子树
      search ( w->data ); //以原树根为目标，做一次（必定失败的）查找
///// assert: 至此，右子树中最小节点必伸展至根，且（因无雷同节点）其左子树必空，于是
      _root->lc = lTree; lTree->parent = _root; //只需将原左子树接回原位即可
   }
   release ( w->data ); release ( w ); _size--; //释放节点，更新规模
   if ( _root ) updateHeight ( _root ); //此后，若树非空，则树根的高度需要更新
   return true; //返回成功标志
} //若目标节点存在且被删除，返回true；否则返回false
```

代码8.5 伸展树节点的删除

当然，其中的第二次查找也可在T_L（若非空）中进行。读者不妨独立实现这一对称的版本。

§8.2 B-树

8.2.1 多路平衡查找

■ 分级存储

现代电子计算机发展速度空前。就计算能力而言，ENIAC[2]每秒只能够执行5000次加法运算，而今天的超级计算机每秒已经能够执行3×10^{16}次以上的浮点运算[3]。就存储能力而言，情况似乎也是如此：ENIAC只有一万八千个电子管，而如今容量以TB计的硬盘也不过数百元，内存的常规容量也已达到GB量级。

然而从实际应用的需求来看，问题规模的膨胀却远远快于存储能力的增长。以数据库为例，在20世纪80年代初，典型数据库的规模为10~100 MB，而三十年后的今天，典型数据库的规模已需要以TB为单位来计量。计算机存储能力提高速度相对滞后，是长期存在的现象，而且随着时间的推移，这一矛盾将日益凸显。鉴于在同等成本下，存储器的容量越大（小）则访问速度越慢（快），因此一味地提高存储器容量，亦非解决这一矛盾的良策。

② 第一台电子计算机， 1946年2月15日诞生于美国宾夕法尼亚大学工学院

③ 2013年6月，天河-2以此运算速度，荣登世界超级计算机500强榜首

实践证明，分级存储才是行之有效的方法。在由内存与外存（磁盘）组成的二级存储系统中，数据全集往往存放于外存中，计算过程中则可将内存作为外存的高速缓存，存放最常用数据项的复本。借助高效的调度算法，如此便可将内存的“高速度”与外存的“大容量”结合起来。

两个相邻存储级别之间的数据传输，统称I/O操作。各级存储器的访问速度相差悬殊，故应尽可能地减少I/O操作。仍以内存与磁盘为例，其单次访问延迟大致分别在纳秒（ns）和毫秒（ms）级别，相差5至6个数量级。也就是说，对内存而言的一秒/一天，相当于磁盘的一星期/两千年。因此，为减少对外存的一次访问，我们宁愿访问内存百次、千次甚至万次。也正因为此，在衡量相关算法的性能时，基本可以忽略对内存的访问，转而更多地关注对外存的访问次数。

多路搜索树

当数据规模大到内存已不足以容纳时，常规平衡二叉搜索树的效率将大打折扣。其原因在于，查找过程对外存的访问次数过多。例如，若将10^9个记录在外存中组织为AVL树，则每次查找大致需做30次外存访问。那么，如何才能有效减少外存操作呢？

为此，需要充分利用磁盘之类外部存储器的另一特性：就时间成本而言，读取物理地址连续的一千个字节，与读取单个字节几乎没有区别。既然外部存储器更适宜于批量式访问，不妨通过时间成本相对极低的多次内存操作，来替代时间成本相对极高的单次外存操作。相应地，需要将通常的二叉搜索树，改造为多路搜索树——在中序遍历的意义下，这也是一种等价变换。

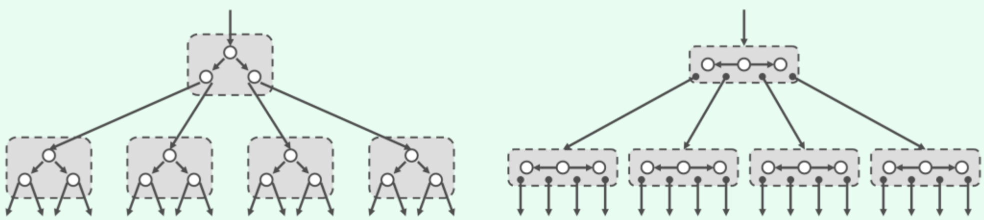

图8.10 二叉搜索树与四路搜索树

具体地如图8.10所示，比如可以两层为间隔，将各节点与其左、右孩子合并为“大节点”：原节点及其孩子的共三个关键码予以保留；孩子节点原有的四个分支也予以保留并按中序遍历次序排列；节点到左、右孩子的分支转化为“大节点”内部的搜索，在图中表示为水平分支。如此改造之后，每个“大节点”拥有四个分支，故称作四路搜索树。

这一策略还可进一步推广，比如以三层为间隔，将各节点及其两个孩子、四个孙子合并为含有七个关键码、八个分支的“大节点”，进而得到八路搜索树。一般地，以k层为间隔如此重组，可将二叉搜索树转化为等价的2^k路搜索树，统称多路搜索树（multi-way search tree）。

不难验证，多路搜索树同样支持查找等操作，且效果与原二叉搜索树完全等同；然而重要的是，其对外存的访问方式已发生本质变化。实际上，在此时的搜索每下降一层，都以“大节点”为单位从外存读取一组（而不再是单个）关键码。更为重要的是，这组关键码在逻辑上与物理上都彼此相邻，故可以批量方式从外存一次性读出，且所需时间与读取单个关键码几乎一样。

当然，每组关键码的最佳数目，取决于不同外存的批量访问特性。比如旋转式磁盘的读写操作多以扇区为单位，故可根据扇区的容量和关键码的大小，经换算得出每组关键码的最佳规模。例如若取k = 8，则每个“大节点”将拥有255个关键码和256个分支，此时同样对于1G个记录，每次查找所涉及的外存访问将减至4~5次。

■ 多路平衡搜索树

所谓m阶B-树④（B-tree），即m路平衡搜索树（m ≥ 2），其宏观结构如图8.11所示。

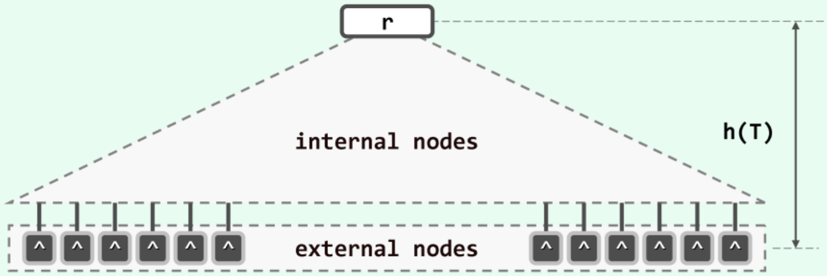

图8.11 B-树的宏观结构（外部节点以深色示意，深度完全一致，且都同处于最底层）

其中，所有外部节点均深度相等。同时，每个内部节点都存有不超过m - 1个关键码，以及用以指示对应分支的不超过m个引用。具体地，存有n ≤ m - 1个关键码：

$$K_1 < K_2 < K_3 < K_4 < \ldots < K_n$$

的内部节点，同时还配有n + 1 ≤ m个引用：

$$A_0 < A_1 < A_2 < A_3 < A_4 < \ldots < A_n$$

反过来，各内部节点的分支数也不能太少。具体地，除根以外的所有内部节点，都应满足：

$$n + 1 \geq \lceil m/2 \rceil$$

而在非空的B-树中，根节点应满足：

$$n + 1 \geq 2$$

由于各节点的分支数介于⌈m/2⌉至m之间，故m阶B-树也称作(⌈m/2⌉, m)-树，如(2, 3)-树、(3, 6)-树或(7, 13)-树等。

B-树的外部节点（external node）更加名副其实——它们实际上未必意味着查找失败，而可能表示目标关键码存在于更低层次的某一外部存储系统中，顺着该节点的指示，即可深入至下一级存储系统并继续查找。正因为如此，不同于常规的搜索树，如图8.11所示，在计算B-树高度时，还需要计入其最底层的外部节点。

例如，图8.12(a)即为一棵由9个内部节点、15个外部节点以及14个关键码组成的4阶B-树，其高度h = 3，其中每个节点包含1~3个关键码，拥有2~4个分支。

作为与二叉搜索树等价的“扁平化”版本，B-树的宽度（亦即最底层外部节点的数目）往往远大于其高度。因此在以图形描述B-树的逻辑结构时，我们往往需要简化其中分支的画法，并转而采用如图(b)所示的紧凑形式。

另外，既然外部节点均同处于最底层，且深度完全一致，故在将它们省略之后，通常还不致造成误解。因此，还可以将B-树的逻辑结构，进一步精简为如图(c)所示的最紧凑形式。

由这种最紧凑的表示形式，也可同时看出，B-树叶节点（即最深的内部节点）的深度也必然完全一致，比如[7]、[19, 22]、[28]、[37, 40, 41]、[46]和[52]。

---

④ 由R. Bayer和E. McCreight于1970年合作发明[43]

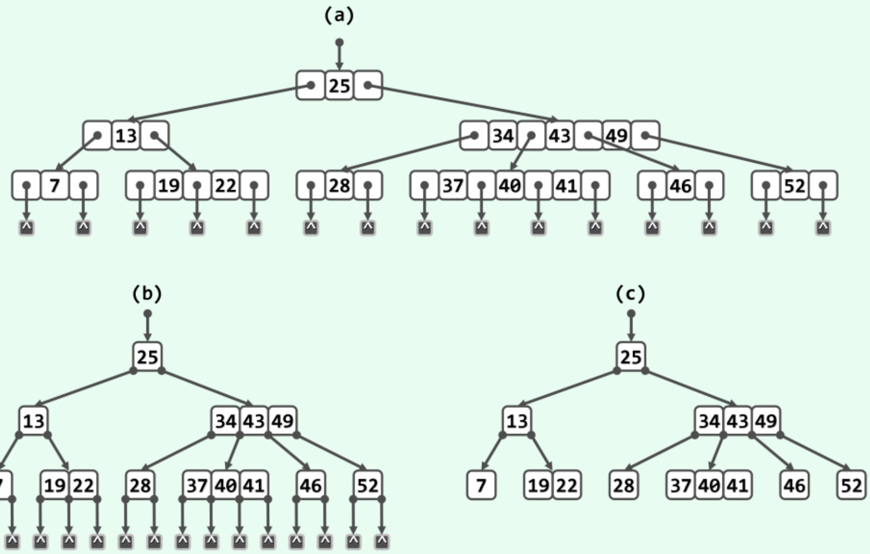

图8.12 (a) 4阶B-树;(b) B-树的紧凑表示;(c) B-树的最紧凑表示

8.2.2 ADT接口及其实现

按照以上定义，可以模板类的形式描述并实现B-树节点以及B-树结构本身如下。

■ 节点

B-树节点BTNode类，可实现如代码8.6所示。

```
#include "../vector/vector.h"
#define BTNodePosi(T) BTNode<T>* //B-树节点位置

template <typename T> struct BTNode { //B-树节点模板类
// 成员（为简化描述起见统一开放，读者可根据需要进一步封装）
   BTNodePosi(T) parent; //父节点
   Vector<T> key; //关键码向量
   Vector<BTNodePosi(T)> child; //孩子向量（其长度总比key多一）
// 构造函数（注意：BTNode只能作为根节点创建，而且初始时有0个关键码和1个空孩子指针）
   BTNode() { parent = NULL; child.insert ( 0, NULL ); }
   BTNode ( T e, BTNodePosi(T) lc = NULL, BTNodePosi(T) rc = NULL ) {
      parent = NULL; //作为根节点，而且初始时
      key.insert ( 0, e ); //只有一个关键码，以及
      child.insert ( 0, lc ); child.insert ( 1, rc ); //两个孩子
      if ( lc ) lc->parent = this; if ( rc ) rc->parent = this;
   }
};
```

代码8.6 B-树节点

这里，同一节点的所有孩子组织为一个向量，各相邻孩子之间的关键码也组织为一个向量。当然，按照B-树的定义，孩子向量的实际长度总是比关键码向量多一。

■ B-树

B-树模板类，可实现如代码8.7所示。

```
#include "BTNode.h" //引入B-树节点类

template <typename T> class BTree { //B-树模板类
protected:
   int _size; //存放的关键码总数
   int _order; //B-树的阶次，至少为3——创建时指定，一般不能修改
   BTNodePosi(T) _root; //根节点
   BTNodePosi(T) _hot; //BTree::search()最后访问的非空（除非树空）的节点位置
   void solveOverflow ( BTNodePosi(T) ); //因插入而上溢之后的分裂处理
   void solveUnderflow ( BTNodePosi(T) ); //因删除而下溢之后的合并处理
public:
   BTree ( int order = 3 ) : _order ( order ), _size ( 0 ) //构造函数：默认为最低的3阶
   { _root = new BTNode<T>(); }
   ~BTree() { if ( _root ) release ( _root ); } //析构函数：释放所有节点
   int const order() { return _order; } //阶次
   int const size() { return _size; } //规模
   BTNodePosi(T) & root() { return _root; } //树根
   bool empty() const { return !_root; } //判空
   BTNodePosi(T) search ( const T& e ); //查找
   bool insert ( const T& e ); //插入
   bool remove ( const T& e ); //删除
}; //BTree
```

代码8.7 B-树

后面将会看到，B-树的关键码插入操作和删除操作，可能会引发节点的上溢和下溢。因此，这里设有内部接口solveOverflow()和solveUnderflow()，分别用于修正此类问题。在稍后的8.2.6节和8.2.8节中，将分别讲解其具体原理及实现。

8.2.3 关键码查找

■ 算法

如前述，B-树结构非常适宜于在相对更小的内存中，实现对大规模数据的高效操作。

一般地如图8.13所示，可以将大数据集组织为B-树并存放于外存。对于活跃的B-树，其根节点会常驻于内存；此外，任何时刻通常只有另一节点（称作当前节点）留驻于内存。

B-树的查找过程，与二叉搜索树的查找过程基本类似。

首先以根节点作为当前节点，然后再逐层深入。若在当前节点（所包含的一组关键码）中能够找到目标关键码，则成功返回。否则（在当前节点中查找“失败”），则必可在当前节点中确定某一个引用（“失败”位置），并通过它转至逻辑上处于下一层的另一节点。若该节点不是外部节点，则将其载入内存，并更新为当前节点，然后继续重复上述过程。

整个过程如图8.13所示，从根节点开始，通过关键码的比较不断深入至下一层，直到某一关键码命中（查找成功），或者到达某一外部节点（查找失败）。

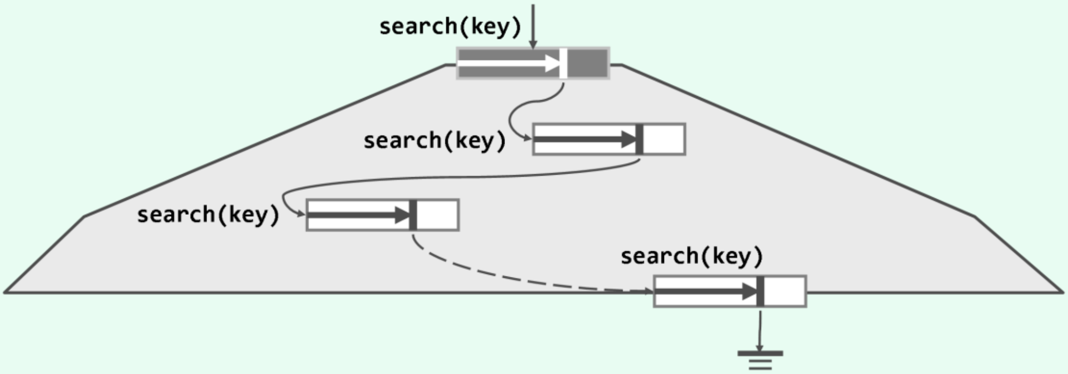

图8.13 B-树的查找过程

与二叉搜索树的不同之处在于，因此时各节点内通常都包含多个关键码，故有可能需要经过（在内存中的）多次比较，才能确定应该转向下一层的哪个节点并继续查找。

仍以如图8.12所示的4阶B-树为例，查找关键码41的过程大致如下：在根节点处经过一次关键码比较（25）之后，即可确定应转入第2个分支；再经过两次比较（34, 43）之后，确定转入第2个分支；最后经过三次比较（37, 40, 41）之后，才成功地找到目标关键码。查找关键码42的过程与之类似，只是在最底层的内部节点内，需要经过三次关键码比较（37, 40, 41）之后，才确定应转入关键码41右侧的外部节点，从而最终确定查找失败。

可见，只有在切换和更新当前节点时才会发生I/O操作，而在同一节点内部的查找则完全在内存中进行。因内存的访问速度远远高于外存，再考虑到各节点所含关键码数量通常在128~512之间，故可直接使用顺序查找策略，而不必采用二分查找之类的复杂策略。

■ 实现

如代码8.8所示，为简化代码，节点内部的查找直接借用了有序向量的search()接口。

```
template <typename T> BTNodePosi(T) BTree<T>::search ( const T& e ) { //在B-树中查找关键码e
   BTNodePosi(T) v = _root; _hot = NULL; //从根节点出发
   while ( v ) { //逐层查找
      Rank r = v->key.search ( e ); //在当前节点中，找到不大于e的最大关键码
      if ( ( 0 <= r ) && ( e == v->key[r] ) ) return v; //成功：在当前节点中命中目标关键码
      _hot = v; v = v->child[r + 1]; //否则，转入对应子树（_hot指向其父）——需做I/O，最费时间
   } //这里在向量内是二分查找，但对通常的_order可直接顺序查找
   return NULL; //失败：最终抵达外部节点
}
```

代码8.8 B-树关键码的查找

与二叉搜索树的实现类似，这里也约定查找结果由返回的节点位置指代：成功时返回目标关键码所在的节点，上层调用过程可在该节点内进一步查找以确定准确的命中位置；失败时返回对应外部节点，其父节点则由变量_hot指代。

8.2.4 性能分析

由上可见，B-树查找操作所需的时间不外乎消耗于两个方面：将某一节点载入内存，以及在内存中对当前节点进行查找。鉴于内存、外存在访问速度上的巨大差异，相对于前一类时间消耗，后一类时间消耗可以忽略不计。也就是说，B-树查找操作的效率主要取决于查找过程中的外存访问次数。那么，至多需要访问多少次外存呢？

由前节分析可见，与二叉搜索树类似，B-树的每一次查找过程中，在每一高度上至多访问一个节点。这就意味着，对于高度为h的B-树，外存访问不超过$\mathcal{O}(h - 1)$次。

B-树节点的分支数并不固定，故其高度h并不完全取决于树中关键码的总数n。对于包含N个关键码的m阶B-树，高度h具体可在多大范围内变化？就渐进意义而言，h与m及N的关系如何？

■ **树高**

可以证明，若存有N个关键码的m阶B-树高度为h，则必有：

$$\log_m(N + 1) \le h \le \log_{\lceil m/2 \rceil}\lfloor (N + 1) / 2 \rfloor + 1 \quad \text{（式8-1）}$$

首先证明$h \le \log_{\lceil m/2 \rceil}\lfloor (N+1)/2 \rfloor + 1$。关键码总数固定时，为使B-树更高，各内部节点都应包含尽可能少的关键码。于是按照B-树的定义，各高度层次上节点数目至少是：

$$
\begin{aligned}
n_0 &= 1 \\
n_1 &= 2 \\
n_2 &= 2 \times \lceil m / 2 \rceil \\
n_3 &= 2 \times \lceil m / 2 \rceil^2 \\
&\cdots \\
n_{h-1} &= 2 \times \lceil m / 2 \rceil^{h-2} \\
n_h &= 2 \times \lceil m / 2 \rceil^{h-1}
\end{aligned}
$$

现考查外部节点。这些节点对应于失败的查找，故其数量n_h应等于失败查找可能情形的总数，即应比成功查找可能情形的总数恰好多1，而后者等于关键码的总数N。于是有

$$N + 1 = n_h \ge 2 \times (\lceil m / 2 \rceil)^{h-1}, \quad h \ge 1$$

即 $$h \le 1 + \log_{\lceil m/2 \rceil}\lfloor (N + 1) / 2 \rfloor = \mathcal{O}(\log_m N)$$

再来证明$h \ge \log_m(N + 1)$。同理，关键码总数固定时，为使B-树更矮，每个内部节点都应该包含尽可能多的关键码。按照B-树的定义，各高度层次上的节点数目至多是：

$$
\begin{aligned}
n_0 &= 1 \\
n_1 &= m \\
n_2 &= m^2 \\
&\cdots \\
n_{h-1} &= m^{h-1} \\
n_h &= m^h
\end{aligned}
$$

与上同理，有

$$N + 1 = n_h \le m^h$$

即 $$h \ge \log_m(N + 1) = \Omega(\log_m N)$$

总之，式8-1必然成立。也就是说，存有N个关键码的m阶B-树的高度$h = \Theta(\log_m N)$。

■ 复杂度

因此，每次查找过程共需访问$O(\log_m N)$个节点，相应地需要做$O(\log_m N)$次外存读取操作。由此可知，对存有N个关键码的m阶B-树的每次查找操作，耗时不超过$O(\log_m N)$。

需再次强调的是，尽管没有渐进意义上的改进，但相对而言极其耗时的I/O操作的次数，却已大致缩减为原先的$1/\log_2 m$。鉴于m通常取值在256至1024之间，较之此前大致降低一个数量级，故使用B-树后，实际的访问效率将有十分可观的提高。

8.2.5 关键码插入

B-树的关键码插入算法，可实现如代码8.9所示。

```
template <typename T> bool BTree<T>::insert ( const T& e ) { //将关键码e插入B树中
   BTNodePosi(T) v = search ( e ); if ( v ) return false; //确认目标节点不存在
   Rank r = _hot->key.search ( e ); //在节点_hot的有序关键码向量中查找合适的插入位置
   _hot->key.insert ( r + 1, e ); //将新关键码插至对应的位置
   _hot->child.insert ( r + 2, NULL ); //创建一个空子树指针
   _size++; //更新全树规模
   solveOverflow ( _hot ); //如有必要，需做分裂
   return true; //插入成功
}
```

代码8.9 B-树关键码的插入

为在B-树中插入一个新的关键码e，首先调用search(e)在树中查找该关键码。若查找成功，则按照“禁止重复关键码”的约定不予插入，操作即告完成并返回false。

否则，按照代码8.8的出口约定，查找过程必然终止于某一外部节点v，且其父节点由变量_hot指示。当然，此时的_hot必然指向某一叶节点（可能同时也是根节点）。接下来，在该节点中再次查找目标关键码e。尽管这次查找注定失败，却可以确定e在其中的正确插入位置r。最后，只需将e插至这一位置。

至此，_hot所指的节点中增加了一个关键码。若该节点内关键码的总数依然合法（即不超过m - 1个），则插入操作随即完成。否则，称该节点发生了一次上溢（overflow），此时需要通过适当的处理，使该节点以及整树重新满足B-树的条件。由代码8.9可见，这项任务将借助调整算法solveOverflow(_hot)来完成。

8.2.6 上溢与分裂

■ 算法

一般地，刚发生上溢的节点，应恰好含有m个关键码。若取$s = \lfloor m/2 \rfloor$，则它们依次为：

$$\{ k_0, \ldots, k_{s-1};\quad k_s;\quad k_{s+1}, \ldots, k_{m-1} \}$$

可见，以k_s为界，可将该节点分前、后两个子节点，且二者大致等长。于是，可令关键码k_s上升一层，归入其父节点（若存在）中的适当位置，并分别以这两个子节点作为其左、右孩子。这一过程，称作节点的分裂（split）。

不难验证，如此分裂所得的两个孩子节点，均符合m阶B-树关于节点分支数的条件。

■ 可能的情况

以如图8.14(a1)所示的6阶B-树局部为例，其中节点{ 17, 20, 31, 37, 41, 56 }，因所含关键码增至6个而发生上溢。为完成修复，可以关键码37为界，将该节点分裂为{ 17, 20, 31 }和{ 41, 56 }；关键码37则上升一层，并以分裂出来的两个子节点作为左、右孩子。

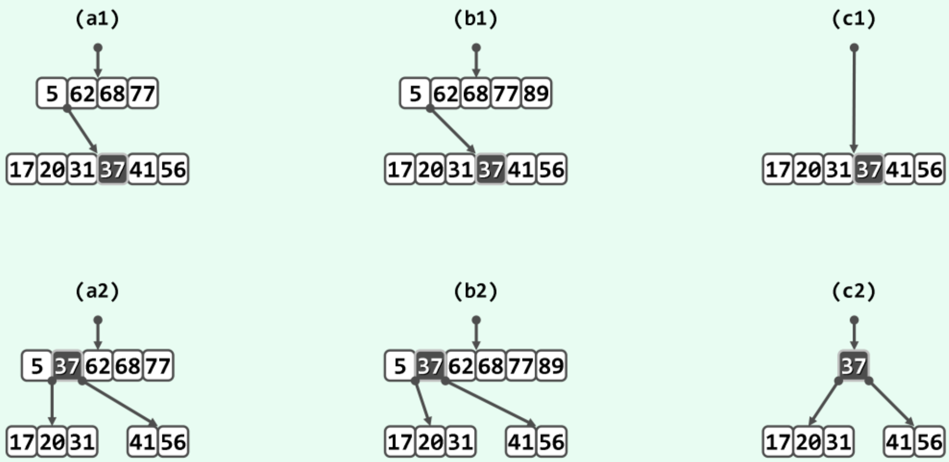

图8.14 通过分裂修复上溢节点

被提升的关键码，可能有三种进一步的处置方式。首先如图(a1)所示，设原上溢节点的父节点存在，且足以接纳一个关键码。此种情况下，只需将被提升的关键码（37）按次序插入父节点中，修复即告完成，修复后的局部如图(a2)所示。

其次如图(b1)所示，尽管上溢节点的父节点存在，但业已处于饱和状态。此时如图(b2)，在强行将被提升的关键码插入父节点之后，尽管上溢节点也可得到修复，却会导致其父节点继而发生上溢——这种现象称作上溢的向上传递。好在每经过一次这样的修复，上溢节点的高度都必然上升一层。这意味着上溢的传递不至于没有尽头，最远不至超过树根。

最后如图(c1)所示，若上溢果真传递至根节点，则可令被提升的关键码（37）自成一个节点，并作为新的树根。于是如图(c2)所示，至此上溢修复完毕，全树增高一层。可见，整个过程中所做分裂操作的次数，必不超过全树的高度——根据8.2.4节结论，即$\mathcal{O}(\log_m N)$。

■ 实现

以上针对上溢的处理算法，可实现如代码8.10所示。

```
template <typename T> //关键码插入后若节点上溢，则做节点分裂处理
void BTree<T>::solveOverflow ( BTNodePosi(T) v ) {
   if ( _order >= v->child.size() ) return; //递归基：当前节点并未上溢
   Rank s = _order / 2; //轴点（此时应有_order = key.size() = child.size() - 1）
   BTNodePosi(T) u = new BTNode<T>(); //注意：新节点已有一个空孩子
   for ( Rank j = 0; j < _order - s - 1; j++ ) { //v右侧_order-s-1个孩子及关键码分裂为右侧节点u
      u->child.insert ( j, v->child.remove ( s + 1 ) ); //逐个移动效率低
      u->key.insert ( j, v->key.remove ( s + 1 ) ); //此策略可改进
   }
   u->child[_order - s - 1] = v->child.remove ( s + 1 ); //移动v最靠右的孩子
```

```
    if ( u->child[0] ) //若u的孩子们非空，则
       for ( Rank j = 0; j < _order - s; j++ ) //令它们的父节点统一
          u->child[j]->parent = u; //指向u
    BTNodePosi(T) p = v->parent; //v当前的父节点p
    if ( !p ) { _root = p = new BTNode<T>(); p->child[0] = v; v->parent = p; } //若p空则创建之
    Rank r = 1 + p->key.search ( v->key[0] ); //p中指向u的指针的秩
    p->key.insert ( r, v->key.remove ( s ) ); //轴点关键码上升
    p->child.insert ( r + 1, u );  u->parent = p; //新节点u与父节点p互联
    solveOverflow ( p ); //上升一层，如有必要则继续分裂——至多递归O(logn)层
}
```

代码8.10 B-树节点的上溢处理

请特别留意上溢持续传播至根的情况：原树根分裂之后，新创建的树根仅含单关键码。由此也可看出，就B-树节点分支数的下限要求而言，树根节点的确应该作为例外。

■ 实例

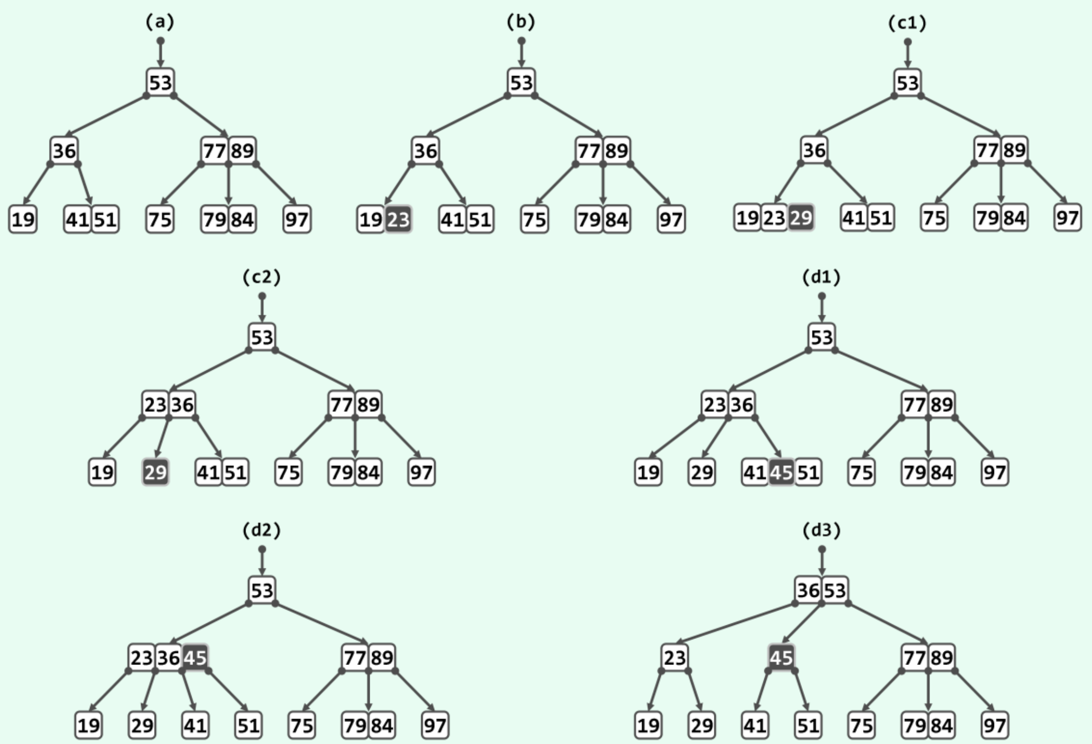

图8.15 3阶B-树插入操作实例（I）

考查如图8.15(a)所示的3阶B-树。执行insert(23)后未发生任何上溢；如(b)所示不必做任何调整。接下来执行insert(29)后，如图(c1)所示发生上溢；经一次分裂即完全修复，结果如图(c2)所示。继续执行insert(45)后，如图(d1)所示发生上溢；经分裂做局部修复之后，如图(d2)所示上一层再次发生上溢；经再次分裂后，方得以实现全树的修复，结果如图(d3)所示。

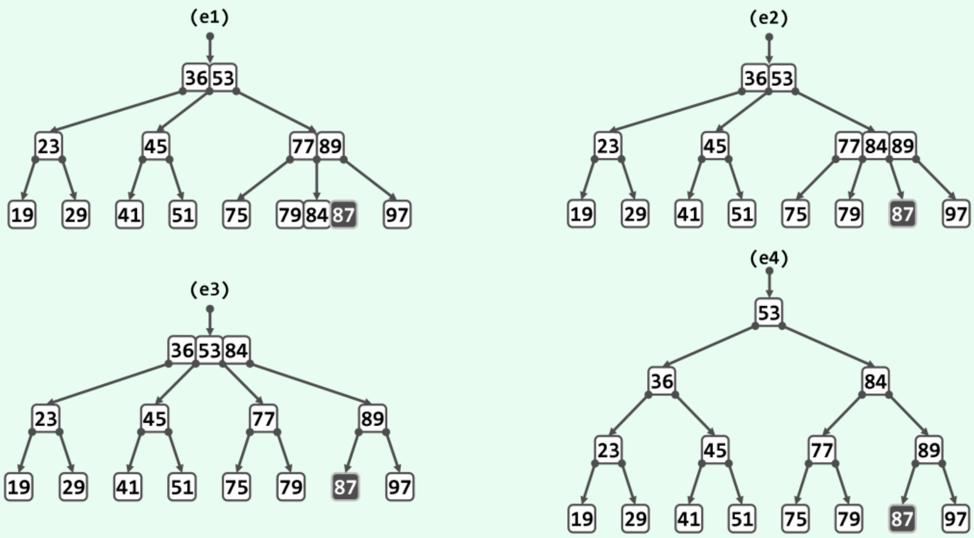

图8.16 3阶B-树插入操作实例（II）

最后，执行insert(87)后如图8.16(e1)所示亦发生上溢；经局部分裂调整后，在更高层将持续发生上溢，故如图(e2)、(e3)和(e4)所示，先后总共经三次分裂，方得以实现全树的修复。此时因一直分裂至根节点，故最终全树高度增加一层——这也是B-树长高的唯一可能。

■ 复杂度

若将B-树的阶次m视作为常数，则关键码的移动和复制操作所需的时间都可以忽略。至于solveOverflow()算法，其每一递归实例均只需常数时间，递归层数不超过B-树高度。由此可知，对于存有N个关键码的m阶B-树，每次插入操作都可在$\mathcal{O}(\log_m N)$时间内完成。

实际上，因插入操作而导致$\Omega(\log_m N)$次分裂的情况极为罕见，单次插入操作平均引发的分裂次数，远远低于这一估计（习题[8-6]），故时间通常主要消耗于对目标关键码的查找。

8.2.7 关键码删除

```
template <typename T> bool BTree<T>::remove ( const T& e ) { //从BTree树中删除关键码e
   BTNodePosi(T) v = search ( e ); if ( !v ) return false; //确认目标关键码存在
   Rank r = v->key.search ( e ); //确定目标关键码在节点v中的秩（由上，肯定合法）
   if ( v->child[0] ) { //若v非叶子，则e的后继必属于某叶节点
      BTNodePosi(T) u = v->child[r+1]; //在右子树中一直向左，即可
      while ( u->child[0] ) u = u->child[0]; //找出e的后继
      v->key[r] = u->key[0]; v = u; r = 0; //并与之交换位置
   } //至此，v必然位于最底层，且其中第r个关键码就是待删除者
   v->key.remove ( r ); v->child.remove ( r + 1 ); _size--; //删除e，以及其下两个外部节点之一
   solveUnderflow ( v ); //如有必要，需做旋转或合并
   return true;
}
```

代码8.11 B-树关键码的删除

B-树的关键码删除算法的实现如代码8.11所示。

为从B-树中删除关键码e，也首先需要调用search(e)查找e所属的节点。倘若查找失败，则说明关键码e尚不存在，删除操作即告完成；否则按照代码8.8的出口约定，目标关键码所在的节点必由返回的位置v指示。此时，通过顺序查找，即可进一步确定e在节点v中的秩r。

不妨假定v是叶节点——否则，e的直接前驱（后继）在其左（右）子树中必然存在，而且可在$\mathcal{O}$(height(v))时间内确定它们的位置，其中height(v)为节点v的高度。此处不妨选用直接后继。于是，e的直接后继关键码所属的节点u必为叶节点，且该关键码就是其中的最小者u[0]。既然如此，只要令e与u[0]互换位置，即可确保待删除的关键码e所属的节点v是叶节点。

于是，接下来可直接将e（及其左侧的外部空节点）从v中删去。如此，节点v中所含的关键码以及（空）分支将分别减少一个。

此时，若该节点所含关键码的总数依然合法（即不少于$\lceil m/2 \rceil - 1$），则删除操作随即完成。否则，称该节点发生了下溢（underflow），并需要通过适当的处置，使该节点以及整树重新满足B-树的条件。由代码8.11可见，这项任务将借助调整算法solveUnderflow(v)来完成。

8.2.8 下溢与合并

由上，在m阶B-树中，刚发生下溢的节点V必恰好包含$\lceil m/2 \rceil - 2$个关键码和$\lceil m/2 \rceil - 1$个分支。以下将根据其左、右兄弟所含关键码的数目，分三种情况做相应的处置。

■ **V的左兄弟L存在，且至少包含$\lceil m/2 \rceil$个关键码**

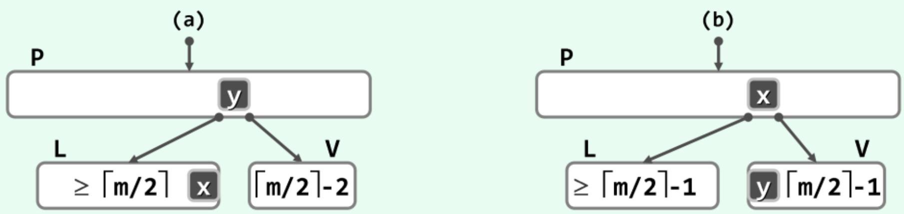

图8.17 下溢节点向父亲“借”一个关键码，父亲再向左兄弟“借”一个关键码

如图8.17(a)所示，不妨设L和V分别是其父节点P中关键码y的左、右孩子，L中最大关键码为x（$x \leq y$）。此时可如图(b)所示，将y从节点P转移至节点V中（作为最小关键码），再将x从L转移至P中（取代原关键码y）。至此，局部乃至整树都重新满足B-树条件，下溢修复完毕。

■ **V的右兄弟R存在，且至少包含$\lceil m/2 \rceil$个关键码**

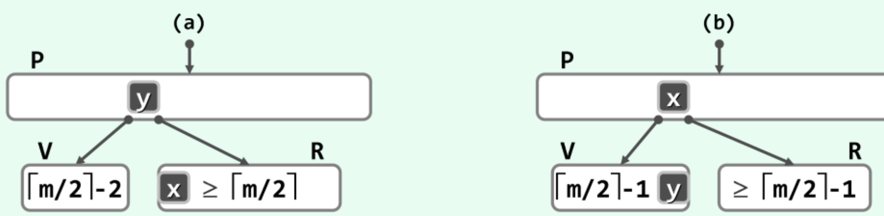

图8.18 下溢节点向父亲“借”一个关键码，父亲再向右兄弟“借”一个关键码

如图8.18所示，可参照前一情况对称地修复，不再赘述。

■ V的左、右兄弟L和R或者不存在，或者其包含的关键码均不足⌈m/2⌉个

实际上，此时的L和R不可能同时不存在。如图8.19(a)所示，不失一般性地设左兄弟节点L存在。当然，此时节点L应恰好包含⌈m/2⌉ - 1个关键码。

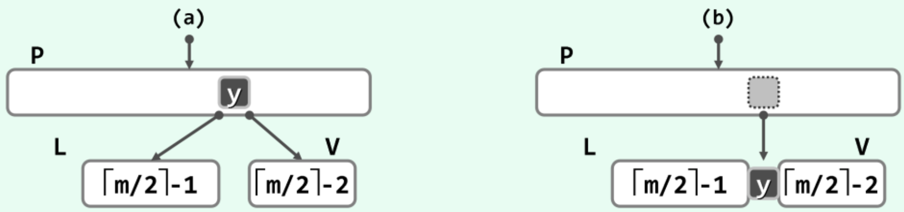

图8.19　下溢节点向父亲“借”一个关键码，然后与左兄弟“粘接”成一个节点

于是为修复节点V的下溢缺陷，可如图(b)所示，从父节点P中抽出介于L和V之间的关键码y，并通过该关键码将节点L和V“粘接”成一个节点——这一过程称作节点的合并（merge）。注意，在经如此合并而得新节点中，关键码总数应为：

$$(\lceil m/2 \rceil - 1) + 1 + (\lceil m/2 \rceil - 2) \quad = \quad 2\times\lceil m/2 \rceil - 2 \quad \leq \quad m - 1$$

故原节点V的下溢缺陷得以修复，而且同时也不致于反过来引发上溢。

接下来，还须检查父节点P——关键码y的删除可能致使该节点出现下溢。好在，即便如此，也尽可套用上述三种方法继续修复节点P。当然，修复之后仍可能导致祖父节点以及更高层节点的下溢——这种现象称作下溢的传递。特别地，当下溢传递至根节点且其中不再含有任何关键码时，即可将其删除并代之以其唯一的孩子节点，全树高度也随之下降一层。

与上溢传递类似地，每经过一次下溢修复，新下溢节点的高度都必然上升一层。再次由8.2.4节的式8-1可知，整个下溢修复的过程中至多需做$\mathcal{O}(\log_m N)$次节点合并操作。

■ 实现

对下溢节点的整个处理过程，如代码8.12所示。

```
template <typename T> //关键码删除后若节点下溢，则做节点旋转或合并处理
void BTree<T>::solveUnderflow ( BTNodePosi(T) v ) {
   if ( ( _order + 1 ) / 2 <= v->child.size() ) return; //递归基：当前节点并未下溢
   BTNodePosi(T) p = v->parent;
   if ( !p ) { //递归基：已到根节点，没有孩子的下限
      if ( !v->key.size() && v->child[0] ) {
         //但倘若作为树根的v已不含关键码，却有（唯一的）非空孩子，则
         _root = v->child[0]; _root->parent = NULL; //这个节点可被跳过
         v->child[0] = NULL; release ( v ); //并因不再有用而被销毁
      } //整树高度降低一层
      return;
   }
   Rank r = 0; while ( p->child[r] != v ) r++;
   //确定v是p的第r个孩子——此时v可能不含关键码，故不能通过关键码查找
   //另外，在实现了孩子指针的判等器之后，也可直接调用Vector::find()定位
// 情况1：向左兄弟借关键码
```

```
   if ( 0 < r ) { //若v不是p的第一个孩子，则
      BTNodePosi(T) ls = p->child[r - 1]; //左兄弟必存在
      if ( ( _order + 1 ) / 2 < ls->child.size() ) { //若该兄弟足够“胖”，则
         v->key.insert ( 0, p->key[r - 1] ); //p借出一个关键码给v（作为最小关键码）
         p->key[r - 1] = ls->key.remove ( ls->key.size() - 1 ); //ls的最大关键码转入p
         v->child.insert ( 0, ls->child.remove ( ls->child.size() - 1 ) );
         //同时ls的最右侧孩子过继给v
         if ( v->child[0] ) v->child[0]->parent = v; //作为v的最左侧孩子
         return; //至此，通过右旋已完成当前层（以及所有层）的下溢处理
      }
   } //至此，左兄弟要么为空，要么太“瘦”
// 情况2：向右兄弟借关键码
   if ( p->child.size() - 1 > r ) { //若v不是p的最后一个孩子，则
      BTNodePosi(T) rs = p->child[r + 1]; //右兄弟必存在
      if ( ( _order + 1 ) / 2 < rs->child.size() ) { //若该兄弟足够“胖”，则
         v->key.insert ( v->key.size(), p->key[r] ); //p借出一个关键码给v（作为最大关键码）
         p->key[r] = rs->key.remove ( 0 ); //ls的最小关键码转入p
         v->child.insert ( v->child.size(), rs->child.remove ( 0 ) );
         //同时rs的最左侧孩子过继给v
         if ( v->child[v->child.size() - 1] ) //作为v的最右侧孩子
            v->child[v->child.size() - 1]->parent = v;
         return; //至此，通过左旋已完成当前层（以及所有层）的下溢处理
      }
   } //至此，右兄弟要么为空，要么太“瘦”
// 情况3：左、右兄弟要么为空（但不可能同时），要么都太“瘦”——合并
   if ( 0 < r ) { //与左兄弟合并
      BTNodePosi(T) ls = p->child[r - 1]; //左兄弟必存在
      ls->key.insert ( ls->key.size(), p->key.remove ( r - 1 ) ); p->child.remove ( r );
      //p的第r - 1个关键码转入ls，v不再是p的第r个孩子
      ls->child.insert ( ls->child.size(), v->child.remove ( 0 ) );
      if ( ls->child[ls->child.size() - 1] ) //v的最左侧孩子过继给ls做最右侧孩子
         ls->child[ls->child.size() - 1]->parent = ls;
      while ( !v->key.empty() ) { //v剩余的关键码和孩子，依次转入ls
         ls->key.insert ( ls->key.size(), v->key.remove ( 0 ) );
         ls->child.insert ( ls->child.size(), v->child.remove ( 0 ) );
         if ( ls->child[ls->child.size() - 1] ) ls->child[ls->child.size() - 1]->parent = ls;
      }
      release ( v ); //释放v
   } else { //与右兄弟合并
      BTNodePosi(T) rs = p->child[r + 1]; //右兄度必存在
      rs->key.insert ( 0, p->key.remove ( r ) ); p->child.remove ( r );
      //p的第r个关键码转入rs，v不再是p的第r个孩子
```

```
      rs->child.insert ( 0, v->child.remove ( v->child.size() - 1 ) );
      if ( rs->child[0] ) rs->child[0]->parent = rs; //v的最左侧孩子过继给ls做最右侧孩子
      while ( !v->key.empty() ) { //v剩余的关键码和孩子，依次转入rs
         rs->key.insert ( 0, v->key.remove ( v->key.size() - 1 ) );
         rs->child.insert ( 0, v->child.remove ( v->child.size() - 1 ) );
         if ( rs->child[0] ) rs->child[0]->parent = rs;
      }
      release ( v ); //释放v
   }
   solveUnderflow ( p ); //上升一层，如有必要则继续分裂——至多递归O(logn)层
   return;
}
```

代码8.12 B-树节点的下溢处理

如前所述，若下溢现象持续传播至树根，且树根当时仅含一个关键码。于是，在其仅有的两个孩子被合并、仅有的一个关键码被借出之后，原树根将退化为单分支节点。对这一特殊情况，需删除该树根，并以刚合并而成的节点作为新的树根——整树高度也随之降低一层。

■ 实例

考查如图8.20(a)所示的3阶B-树。

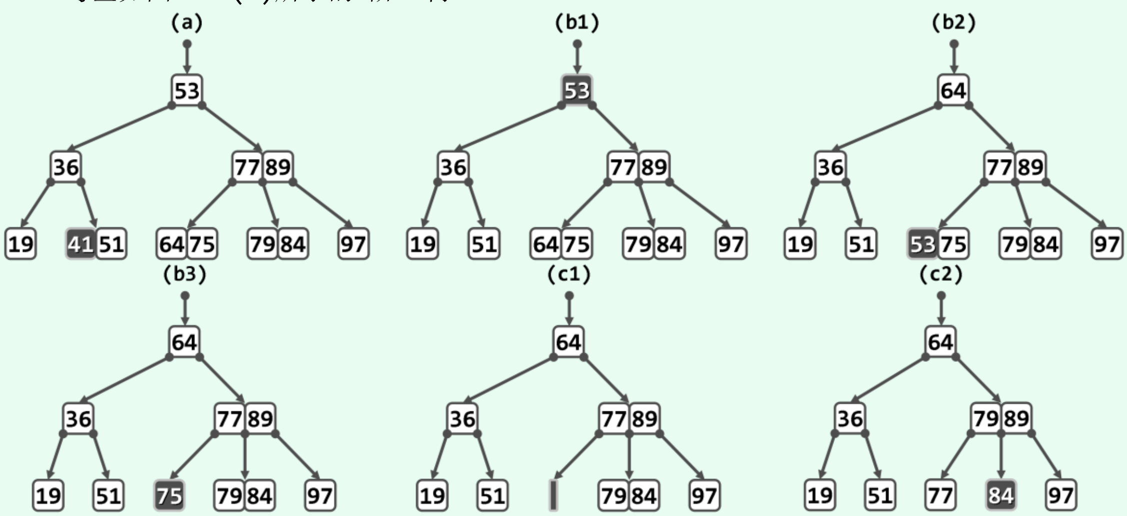

图8.20 3阶B-树删除操作实例（I）

首先执行remove(41)：因关键码41来自底层叶节点，且从中删除该关键码后未发生下溢，故无需修复，结果如图(b1)所示。接下来执行remove(53)：因关键码53并非来自底层叶节点，故在将该关键码与其直接后继64交换位置之后，如图(b2)所示关键码，53必属于某底层叶节点；在删除该关键码之后，其所属节点并未发生下溢，故亦无需修复，结果如图(b3)所示。

然后执行remove(75)：关键码75来自底层叶节点，故被直接删除后其所属节点如图(c1)所示发生下溢；在经父节点中转，从右侧兄弟间接借得一个关键码之后，结果如图(c2)所示。

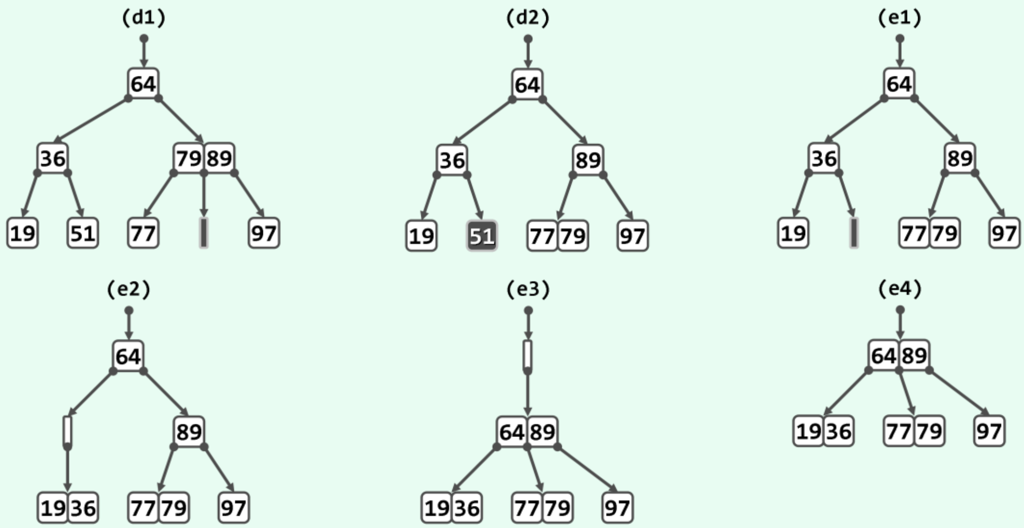

图8.21 3阶B-树删除操作实例（II）

继续执行remove(84)：同样地，删除关键码84后，其原属底层叶节点如以图8.21(d1)所示发生下溢；此时左、右兄弟均无法借出关键码，故在从父节点借得关键码79后，该下溢节点可与其左侧兄弟合并；父节点借出一个关键码之后尚未下溢，故结果如图(d2)所示。

最后执行remove(51)：删除关键码51后，其原属底层叶节点如图(e1)所示发生下溢；从父节点借得关键码36后，该节点可与左侧兄弟合并，但父节点如图(e2)所示因此发生下溢；从祖父（根）节点借得关键码64后，父节点可与其右侧兄弟合并，但祖父节点如图(e3)所示因此发生下溢。此时已抵达树根，故直接删除空的根节点，如图(e4)所示全树高度降低一层。

■ 复杂度

与插入操作同理，在存有N个关键码的m阶B-树中的每次关键码删除操作，都可以在$\mathcal{O}(\log_m N)$时间内完成。另外同样地，因某一关键码的删除而导致$\Omega(\log_m N)$次合并操作的情况也极为罕见，单次删除操作过程中平均只需做常数次节点的合并。

§8.3 *红黑树

平衡二叉搜索树的形式多样，且各具特色。比如，8.1节的伸展树实现简便、无需修改节点结构、分摊复杂度低，但可惜最坏情况下的单次操作需要$\Omega(n)$时间，故难以适用于核电站、医院等对可靠性和稳定性要求极高的场合。反之，7.4节的AVL树尽管可以保证最坏情况下的单次操作速度，但需在节点中嵌入平衡因子等标识；更重要的是，删除操作之后的重平衡可能需做多达$\Omega(\log n)$次旋转，从而频繁地导致全树整体拓扑结构的大幅度变化。

红黑树即是针对后一不足的改进。通过为节点指定颜色，并巧妙地动态调整，红黑树可保证：在每次插入或删除操作之后的重平衡过程中，全树拓扑结构的更新仅涉及常数个节点。尽管最坏情况下需对多达$\Omega(\log n)$个节点重染色，但就分摊意义而言仅为$\mathcal{O}(1)$个（习题[8-14]）。

当然，为此首先需在AVL树“适度平衡”标准的基础上，进一步放宽条件。实际上，红黑树所采用的“适度平衡”标准，可大致表述为：任一节点左、右子树的高度，相差不得超过两倍。

8.3.1 概述

■ 定义与条件

为便于对红黑树的理解、实现与分析，这里不妨仿照8.2.1节中B-树的做法，如图8.22所示统一地引入n + 1个外部节点，以保证原树中每一节点（现称作内部节点，白色八角形）的左、右孩子均非空——尽管有可能其中之一甚至二者同时是外部节点。当然，这些外部节点的引入只是假想式的，在具体实现时并不一定需要兑现为真实的节点。如此扩展之后的便利之处在于，我们的考查范围只需覆盖真二叉树。

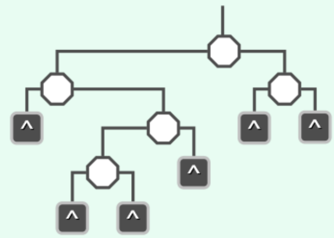

图8.22 通过假想式地引入外部节点（黑色正方形），将二叉树扩展为真二叉树

由红、黑两色节点组成的二叉搜索树若满足以下条件，即为红黑树⑤（red-black tree）：

(1) 树根始终为黑色
(2) 外部节点均为黑色
(3) 其余节点若为红色，则其孩子节点必为黑色
(4) 从任一外部节点到根节点的沿途，黑节点的数目相等

其中，条件(1)和(2)意味着红节点均为内部节点，且其父节点及左、右孩子必然存在。另外，条件(3)意味着红节点之父必为黑色，因此树中任一通路都不含相邻的红节点。

由此可知，在从根节点通往任一节点的沿途，黑节点都不少于红节点。除去根节点本身，沿途所经黑节点的总数称作该节点的黑深度（black depth）——根节点的黑深度为0，其余依此类推。故条件(4)亦可等效地理解和描述为“所有外部节点的黑深度统一”。

由条件(4)可进一步推知，在从任一节点通往其任一后代外部节点的沿途，黑节点的总数亦必相等。除去（黑色）外部节点，沿途所经黑节点的总数称作该节点的黑高度（black height）。如此，所有外部节点的黑高度均为0，其余依此类推。

特别地，根节点的黑高度亦称作全树的黑高度，在数值上与外部节点的黑深度相等。

■ (2,4)-树

红黑树的上述定义，不免令人困惑和费解。幸运的是，借助此前已掌握的概念，我们完全可以清晰地理解和把握红黑树的定义，及其运转过程。为此，需注意到如下有趣的事实：在红黑树与8.2节的4阶B-树之间，存在极其密切的联系；经适当转换之后，二者相互等价！

具体地，自顶而下逐层考查红黑树各节点。每遇到一个红节点，都将对应的子树整体提升一层，从而与其父节点（必黑）水平对齐，二者之间的联边则相应地调整为横向。

如此转换之后，横向边或向左或向右，但由红黑树的条件(3)，同向者彼此不会相邻；即便不考虑联边的左右方向，沿水平方向相邻的边至多两条（向左、右各一条），涉及的节点至多三个（一个黑节点加上零到两个红节点）。此时，若将原红黑树的节点视作关键码，沿水平方向相邻的每一组（父子至多三个）节点即恰好构成4阶B-树的一个节点。

⑤ 其雏形由R. Bayer于1972年发明[44]，命名为对称二叉B-树（symmetric binary B-tree）后由L. J. Guibas与R. Sedgewick于1978年做过改进[45]，并定名为红黑树（red-black tree）

图8.23针对所有可能的四种情况，分别给出了具体的转换过程。可见，按照上述对应关系，每棵红黑树都等价于一棵(2,4)-树；前者的每一节点都对应于后者的一个关键码。通往黑节点的边，对黑高度有贡献，并在(2,4)-树中得以保留；通往红节点的边对黑高度没有贡献，在(2,4)-树中对应于节点内部一对相邻的关键码。在本节的插图中，这两类边将分别以实线、虚线示意。

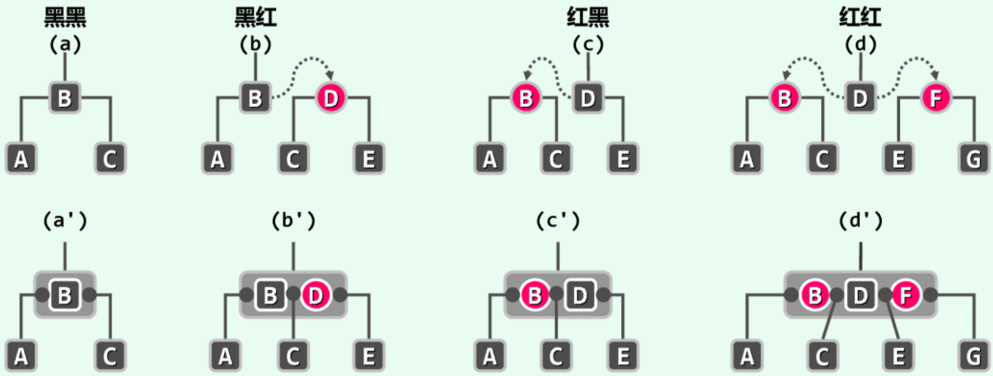

图8.23 红黑树到4阶B-树的等价转换（在完全彩色版尚未出版之前本书约定，分别以圆形、正方形和八角形表示红黑树的红节点、黑节点和颜色未定节点，以长方形表示B-树节点）

为使讲解简洁，在不致引起歧义的前提下，以下将不再严格区分红黑树中的节点及其在(2,4)-树中对应的关键码。当然，照此理解，此时的关键码也被赋予了对应的颜色。对照红黑树的条件，(2,4)-树中的每个节点应包含且仅包含一个黑关键码，同时红关键码不得超过两个。而且，若某个节点果真包含两个红关键码，则黑关键码的位置必然居中。

■ 平衡性

与所有二叉搜索树一样，红黑树的性能首先取决于其平衡性。那么，红黑树的高度可以在多大范围之内变化呢？实际上，即便计入扩充的外部节点，包含n个内部节点的红黑树T的高度h也不致超过$\mathcal{O}(\log n)$。更严格地有：

$$\log_2(n+1) \quad \leq \quad h \quad \leq \quad 2\cdot\log_2(n+1)$$

左侧的“≤”显然成立，故以下只需证明右侧“≤”也成立。

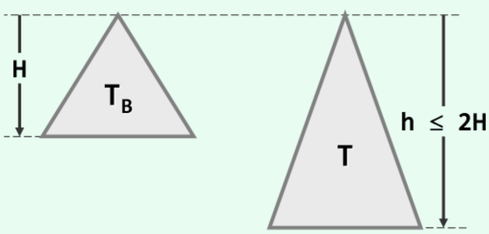

图8.24 红黑树的黑高度不低于高度的一半；反之，高度不超过黑高度的两倍

如图8.24所示，若将T的黑高度记作H，则H也是T所对应(2,4)-树T_B的高度，故由8.2.4节关于B-树高度与所含关键码总数关系的结论，有：

$$H \quad \leq \quad \log_{\lceil 4/2 \rceil}\left\lfloor\frac{n+1}{2}\right\rfloor + 1 \quad \leq \quad \log_2\left\lfloor\frac{n+1}{2}\right\rfloor + 1 \quad \leq \quad \log_2(n+1)$$

另一方面，既然任一通路都不含相邻的红节点，故必有：

$$h \quad \leq \quad 2H \quad \leq \quad 2\cdot\log_2(n+1) \quad = \quad \mathcal{O}(\log n)$$

也就是说，尽管红黑树不能如完全树那样可做到理想平衡，也不如AVL树那样可做到较严格的适度平衡，但其高度仍控制在最小高度的两倍以内（习题[8-11]），从渐进的角度看仍是$\mathcal{O}(\log n)$，依然保证了适度平衡——这正是红黑树可高效率支持各种操作的基础。

8.3.2 红黑树接口定义

基于185页代码7.2中的BST模板类，可派生出RedBlack模板类如代码8.13所示。

```
#include "../BST/BST.h" //基于BST实现RedBlack
template <typename T> class RedBlack : public BST<T> { //RedBlack树模板类
protected:
   void solveDoubleRed ( BinNodePosi(T) x ); //双红修正
   void solveDoubleBlack ( BinNodePosi(T) x ); //双黑修正
   int updateHeight ( BinNodePosi(T) x ); //更新节点x的高度
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8.13 基于BST定义的红黑树接口

可见，这里直接沿用了二叉搜索树标准的查找算法search()，并根据红黑树的重平衡规则与算法，重写了insert()和remove()接口；新加的两个内部功能接口solveDoubleRed()和solveDoubleBlack()，分别用于在节点插入或删除之后恢复全树平衡。其具体实现稍后介绍。

另外，这里还需使用此前二叉树节点模板类BinNode（117页代码5.1）中预留的两个成员变量height和color。如代码8.14所示，仿照AVL树的实现方式，可借助辅助宏来检查节点的颜色以及判定是否需要更新（黑）高度记录，如此可大大简化相关算法的描述。

```
#define IsBlack(p) ( ! (p) || ( RB_BLACK == (p)->color ) ) //外部节点也视作黑节点
#define IsRed(p) ( ! IsBlack(p) ) //非黑即红
#define BlackHeightUpdated(x) ( /*RedBlack高度更新条件*/ \
   ( stature( (x).lc ) == stature( (x).rc ) ) && \
   ( (x).height == ( IsRed(& x) ? stature( (x).lc ) : stature( (x).lc ) + 1 ) ) \
)
```

代码8.14 用以简化红黑树算法描述的宏

可见，这里的确并未真正地实现图8.22中所引入的外部节点，而是将它们统一地直接判定为黑“节点”——尽管它们实际上只不过是NULL。其余节点，则一概视作红节点。

```
template <typename T> int RedBlack<T>::updateHeight ( BinNodePosi(T) x ) { //更新节点高度
   x->height = max ( stature ( x->lc ), stature ( x->rc ) ); //孩子一般黑高度相等，除非出现双黑
   return IsBlack ( x ) ? x->height++ : x->height; //若当前节点为黑，则计入黑深度
} //因统一定义stature(NULL) = -1，故height比黑高度少一，好在不致影响到各种算法中的比较判断
```

代码8.15 红黑树节点的黑高度更新

此处的height已不再是指常规的树高，而是红黑树的黑高度。故如代码8.15所示，节点黑高度需要更新的情况共分三种：或者左、右孩子的黑高度不等；或者作为红节点，黑高度与其孩子不相等；或者作为黑节点，黑高度不等于孩子的黑高度加一。

8.3.3 节点插入算法

■ 节点插入与双红现象

如代码8.16所示，不妨假定经调用接口search(e)做查找之后，确认目标节点尚不存在。于是，在查找终止的位置x处创建节点，并随即将其染成红色（除非此时全树仅含一个节点）。现在，对照红黑树的四项条件，唯有(3)未必满足——亦即，此时x的父亲也可能是红色。

```
template <typename T> BinNodePosi(T) RedBlack<T>::insert ( const T& e ) { //将e插入红黑树
   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, NULL, NULL, -1 ); _size++; //创建红节点x：以_hot为父，黑高度-1
   solveDoubleRed ( x ); return x ? x : _hot->parent; //经双红修正后，即可返回
} //无论e是否存在于原树中，返回时总有x->data == e
```

代码8.16 红黑树insert()接口

因新节点的引入，而导致父子节点同为红色的此类情况，称作“双红”（double red）。为修正双红缺陷，可调用solveDoubleRed(x)接口。每引入一个关键码，该接口都可能迭代地调用多次。在此过程中，当前节点x的兄弟及两个孩子（初始时都是外部节点），始终均为黑色。

将x的父亲与祖父分别记作p和g。既然此前的红黑树合法，故作为红节点p的父亲，g必然存在且为黑色。g作为内部节点，其另一孩子（即p的兄弟、x的叔父）也必然存在，将其记作u。以下，视节点u的颜色不同，分两类情况分别处置。

■ 双红修正（RR-1）

首先，考查u为黑色的情况。此时，x的兄弟、两个孩子的黑高度，均与u相等。图8.25(a)和(b)即为此类情况的两种可能（另两种对称情况，请读者独立补充）。

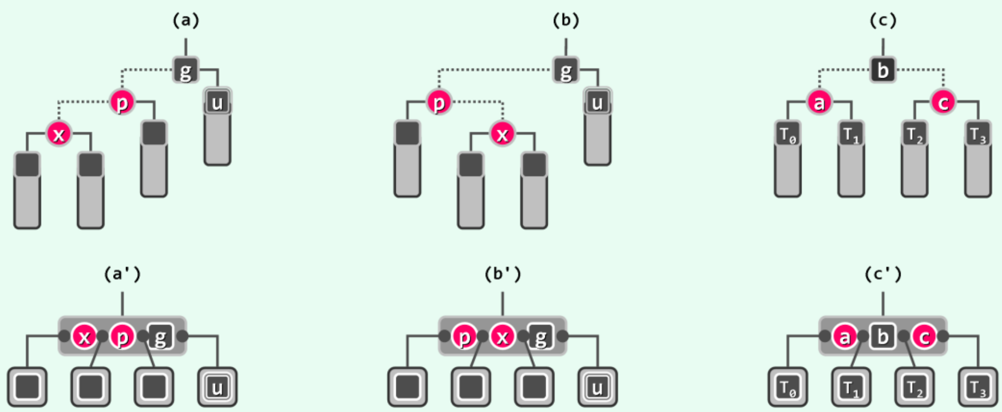

图8.25 双红修正第一种情况（RR-1）及其调整方法（上方、下方分别为红黑树及其对应B-树的局部）

此时红黑树条件(3)的违反，从B-树角度等效地看，即同一节点不应包含紧邻的红色关键码。故如图8.25(c')所示，只需令黑色关键码与紧邻的红色关键码互换颜色。从图(c)红黑树的角度看，这等效于按中序遍历次序，对节点x、p和g及其四棵子树，做一次局部“3 + 4”重构。

不难验证，如此调整之后，局部子树的黑高度将复原，这意味着全树的平衡也必然得以恢复。同时，新子树的根节点b为黑色，也不致引发新的双红现象。至此，整个插入操作遂告完成。

■ 双红修正（RR-2）

再考查节点u为红色的情况。此时，u的左、右孩子非空且均为黑色，其黑高度必与x的兄弟以及两个孩子相等。图8.26(a)和(b)给出了两种可能的此类情况（另两种对称情况，请读者独立补充）。此时红黑树条件(3)的违反，从B-树角度等效地看，即该节点因超过4度而发生上溢。

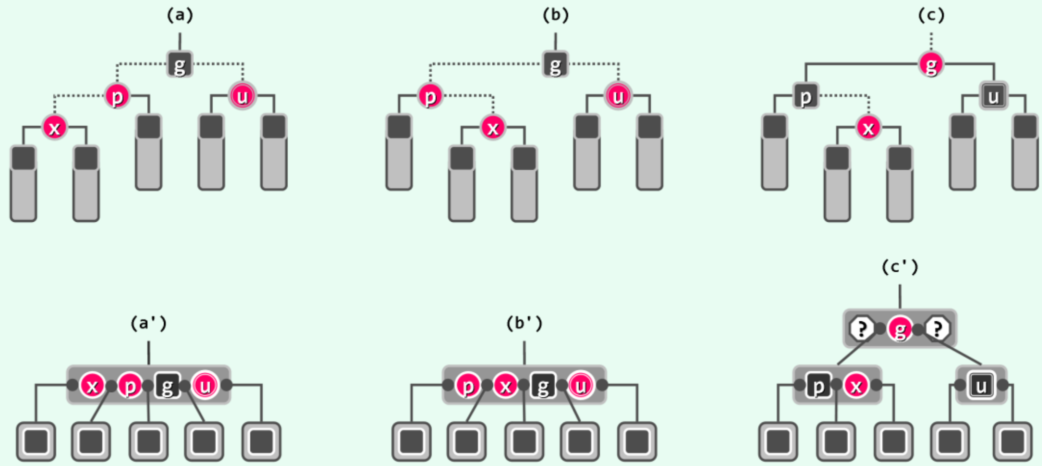

图8.26 双红修正第二种情况（RR-2）及其调整方法（带问号的关键码可能存在）

以图8.26(b)为例。从图(c)红黑树的角度来看，只需将红节点p和u转为黑色，黑节点g转为红色，x保持红色。从图(c')B-树的角度来看，等效于上溢节点的一次分裂。

不难验证，如此调整之后局部子树的黑高度复原。然而，子树根节点g转为红色之后，有可能在更高层再次引发双红现象。从图8.26(c')B-树的角度来看，对应于在关键码g被移出并归入上层节点之后，进而导致上层节点的上溢——即上溢的向上传播。

若果真如此，可以等效地将g视作新插入的节点，同样地分以上两类情况如法处置。请注意，每经过一次这样的迭代，节点g都将在B-树中（作为关键码）上升一层，而在红黑树中存在双红缺陷的位置也将相应地上升两层，故累计至多迭代$\mathcal{O}(\log n)$次。

特别地，若最后一步迭代之后导致原树根的分裂，并由g独立地构成新的树根节点，则应遵照红黑树条件(1)的要求，强行将其转为黑色——如此，全树的黑高度随即增加一层。

■ 双红修正的复杂度

以上情况的处理流程可归纳为图8.27。其中的重构、染色等局部操作均只需常数时间，故只需统计这些操作在修正过程中被调用的总次数。

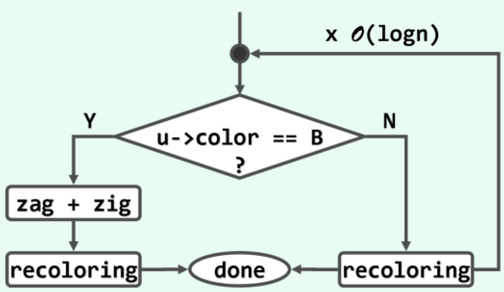

图8.27 双红修正流程图

表8.1 双红修正算法所涉及局部操作的统计

情　况		#旋转	#染色	单轮修正之后
RR-1	u为黑	1~2	2	调整随即完成
RR-2	u为红	0	3	或再次双红 但必上升两层

具体统计，可归纳为表8.1。可见，对于前一种情况，只需做一轮修正；后一种情况虽有可能需要反复修正，但由于修正位置的高度会严格单调上升，故总共也不过$O(\log n)$轮。另外从该表也可看出，每一轮修正只涉及到常数次的节点旋转或染色操作。

因此，节点插入之后的双红修正，累计耗时不会超过$O(\log n)$。即便计入此前的关键码查找以及节点接入等操作，红黑树的每次节点插入操作，都可在$O(\log n)$时间内完成。

需要特别指出的是，只有在RR-1修复时才需做1~2次旋转；而且一旦旋转后，修复过程必然随即完成。故就全树拓扑结构而言，每次插入后仅涉及常数次调整；而且稍后将会看到，红黑树的节点删除操作亦是如此——回顾7.4节的AVL树，却只能保证前一点。

■ 双红修正算法的实现

以上针对双红缺陷的各种修正方法，可以概括并实现如代码8.17所示。

```
/******************************************************************************************
 * RedBlack双红调整算法：解决节点x与其父均为红色的问题。分为两大类情况：
 *    RR-1：2次颜色翻转，2次黑高度更新，1~2次旋转，不再递归
 *    RR-2：3次颜色翻转，3次黑高度更新，0次旋转，需要递归
 ******************************************************************************************/
template <typename T> void RedBlack<T>::solveDoubleRed ( BinNodePosi(T) x ) { //x当前必为红
   if ( IsRoot ( *x ) ) //若已（递归）转至树根，则将其转黑，整树黑高度也随之递增
      {  _root->color = RB_BLACK; _root->height++; return;  } //否则，x的父亲p必存在
   BinNodePosi(T) p = x->parent; if ( IsBlack ( p ) ) return; //若p为黑，则可终止调整。否则
   BinNodePosi(T) g = p->parent; //既然p为红，则x的祖父必存在，且必为黑色
   BinNodePosi(T) u = uncle ( x ); //以下，视x叔父u的颜色分别处理
   if ( IsBlack ( u ) ) { //u为黑色（含NULL）时
      if ( IsLChild ( *x ) == IsLChild ( *p ) ) //若x与p同侧（即zIg-zIg或zAg-zAg），则
         p->color = RB_BLACK; //p由红转黑，x保持红
      else //若x与p异侧（即zIg-zAg或zAg-zIg），则
         x->color = RB_BLACK; //x由红转黑，p保持红
      g->color = RB_RED; //g必定由黑转红
///// 以上虽保证总共两次染色，但因增加了判断而得不偿失
///// 在旋转后将根置黑、孩子置红，虽需三次染色但效率更高
      BinNodePosi(T) gg = g->parent; //曾祖父（great-grand parent）
      BinNodePosi(T) r = FromParentTo ( *g ) = rotateAt ( x ); //调整后的子树根节点
      r->parent = gg; //与原曾祖父联接
   } else { //若u为红色
      p->color = RB_BLACK; p->height++; //p由红转黑
      u->color = RB_BLACK; u->height++; //u由红转黑
      if ( !IsRoot ( *g ) ) g->color = RB_RED; //g若非根，则转红
      solveDoubleRed ( g ); //继续调整g（类似于尾递归，可优化为迭代形式）
   }
}
```

代码8.17 双红修正solveDoubleRed()

8.3.4 节点删除算法

■ 节点删除与双黑现象

```
template <typename T> bool RedBlack<T>::remove ( const T& e ) { //从红黑树中删除关键码e
   BinNodePosi(T) & x = search ( e ); if ( !x ) return false; //确认目标存在（留意_hot的设置）
   BinNodePosi(T) r = removeAt ( x, _hot ); if ( ! ( --_size ) ) return true; //实施删除
// assert: _hot某一孩子刚被删除，且被r所指节点（可能是NULL）接替。以下检查是否失衡，并做必要调整
   if ( ! _hot ) //若刚被删除的是根节点，则将其置黑，并更新黑高度
      { _root->color = RB_BLACK; updateHeight ( _root ); return true; }
// assert: 以下，原x（现r）必非根，_hot必非空
   if ( BlackHeightUpdated ( *_hot ) ) return true; //若所有祖先的黑深度依然平衡，则无需调整
   if ( IsRed ( r ) ) //否则，若r为红，则只需令其转黑
      { r->color = RB_BLACK; r->height++; return true; }
// assert: 以下，原x（现r）均为黑色
   solveDoubleBlack ( r ); return true; //经双黑调整后返回
} //若目标节点存在且被删除，返回true；否则返回false
```

代码8.18 红黑树remove()接口

如代码8.18所示，为删除关键码e，首先调用标准接口BST::search(e)，查找目标节点x。若查找成功，则调用内部接口removeAt(x)实施删除。按照7.2.6节对该接口所做的语义约定，其间无论是否做过一次节点交换，均以r指向实际被删除节点x的接替者，p = _hot为其父亲。

不难验证，此时红黑树的前两个条件继续满足，但后两个条件却未必依然满足。

如图8.28所示，除了其接替者r，x还应有另一个孩子w。既然x是实际被删除者，故w必为（黑色的）外部节点NULL。

如图(a)和(a')所示，若x为红色，则在删除x并代之以r后，条件(3~4)依然满足；反之，若x为黑色，则要看其替代者r的颜色。

如图(b)和(b')所示，若r为红色，则只需将其翻转为黑色，即可使条件(3~4)重新满足。

然而如图(c)和(c')所示，若x和r均为黑色，则为使条件(3~4)重新成立，还需要做略微复杂一些的处理。

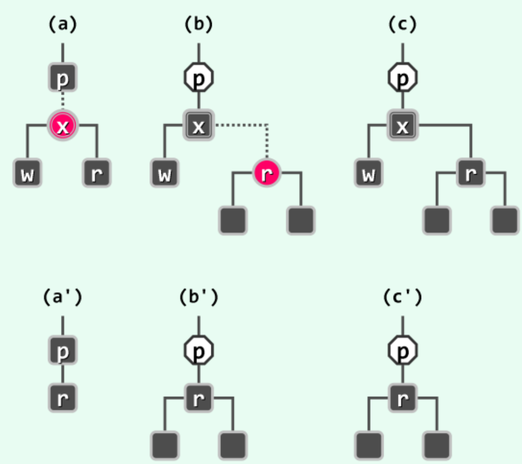

图8.28 删除节点x之后，红黑树条件(4)：(a)或依然满足，(b)或经重染色后重新满足，(c)或不再满足

因某一无红色孩子的黑节点被删除，而导致的此类复杂情况，称作“双黑”（double black）现象。此时，需从r出发调用solveDoubleBlack(r)算法予以修正。

自然，原黑节点x的兄弟必然非空，将其记作s；x的父亲记作p，其颜色不确定（故在图中以八角形示意）。以下视s和p颜色的不同组合，按四种情况分别处置。

双黑修正（BB-1）

既然节点x的另一孩子w = NULL，故从B-树角度（图8.29(a')）看节点x被删除之后的情况，可以等效地理解为：关键码x原所属的节点发生下溢；此时，t和s必然属于B-树的同一节点，且该节点就是下溢节点的兄弟。故可参照B-树的调整方法，下溢节点从父节点借出一个关键码(p)，然后父节点从向下溢节点的兄弟节点借出一个关键码（s），调整后的效果如图(b')。

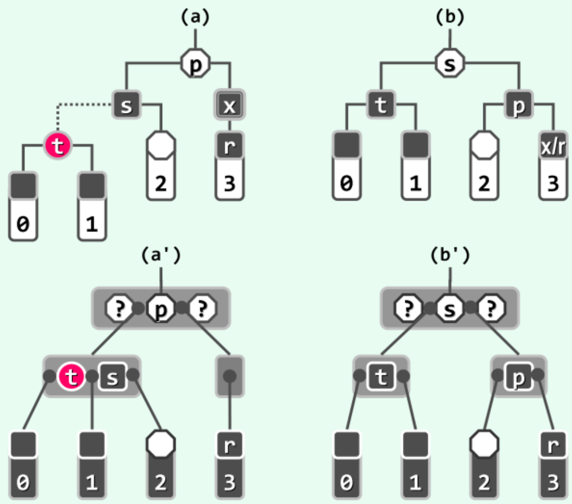

图8.29 双黑修正（情况BB-1）
（带问号的关键码可能存在，且颜色不定）

从红黑树的角度（图(b)）来看，上述调整过程等效于，对节点t、s和p实施“3 + 4”重构。

此外，根据红黑树与B-树的对应关系不难理解，若这三个节点按中序遍历次序重命名为a、b和c，则还需将a和c染成黑色，b则继承p此前的颜色。就图8.29的具体实例而言，也就是将t和p染成黑色，s继承p此前的颜色。注意，整个过程中节点r保持黑色不变。

由图8.29(b)（及其对称情况）不难验证，经以上处理之后，红黑树的所有条件，都在这一局部以及全局得到恢复，故删除操作遂告完成。

双黑修正（BB-2-R）

节点s及其两个孩子均为黑色时，视节点p颜色的不同，又可进一步分为两种情况。

先考虑p为红色的情况。如图8.30(a)所示，即为一种典型的此类情况（与之对称的情况，请读者独立补充）。

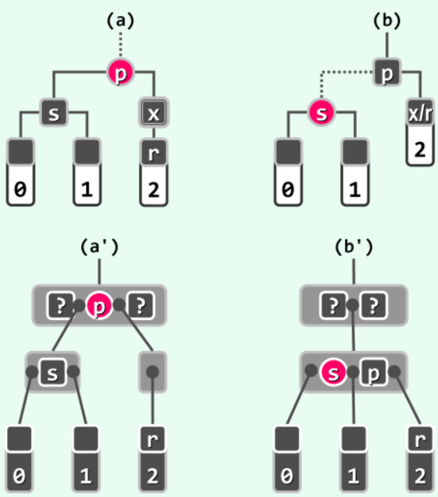

图8.30 双黑修正（情况BB-2-R）
（带问号的黑关键码可能但不会同时存在）

与BB-1类似，在对应的B-树中，关键码x的删除导致其所属的节点下溢。但因此时关键码s所在节点只有两个分支，故下溢节点无法从父节点借出关键码（p）。

按照8.2.8节的B-树平衡算法，此时应如图(b')所示，将关键码p取出并下降一层，然后以之为“粘合剂”，将原左、右孩子合并为一个节点。从红黑树角度看，这一过程可如图(b)所示等效地理解为：s和p颜色互换。

由图8.30(b)（及其对称情况）可知，经过以上处理，红黑树的所有条件都在此局部得以恢复。另外，由于关键码p原为红色，故如图8.30(a')所示，在关键码p所属节点中，其左或右必然还有一个黑色关键码（当然，不可能左、右兼有）——这意味着，在关键码p从其中取出之后，不致引发新的下溢。至此，红黑树条件亦必在全局得以恢复，删除操作即告完成。

■ 双黑修正（BB-2-B）

接下来，再考虑节点s、s的两个孩子以及节点p均为黑色的情况。

如图8.31(a)所示，即为一种典型的此类情况（与之对称的情况，请读者独立补充）。此时与BB-2-R类似，在对应的B-树中，因关键码x的删除，导致其所属节点发生下溢。

因此可如图(b')所示，将下溢节点与其兄弟合并。从红黑树的角度来看，这一过程可如图(b)所示等效地理解为：节点s由黑转红。

由图8.31(b)（及其对称情况）可知，经以上处理，红黑树所有条件都在此局部得到恢复。

然而，因s和x在此之前均为黑色，故如图8.31(a')所示，p原所属的B-树节点必然仅含p这一个关键码。于是在p被借出之后，该节点必将继而发生下溢，故有待于后续进一步修正。

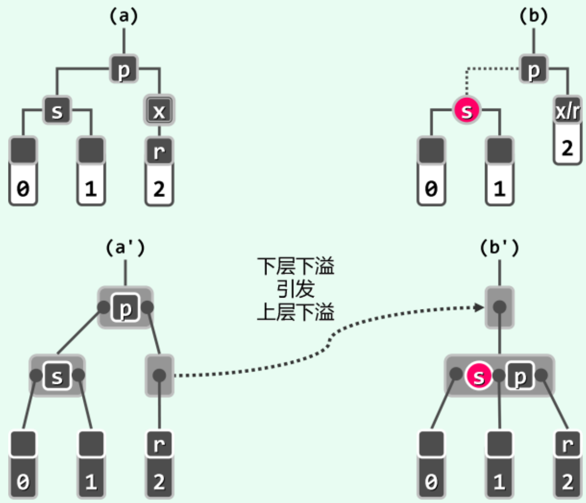

图8.31 双黑修正（情况BB-2-B）

从红黑树的角度来看，此时的状态可等效地理解为：节点p的父节点刚被删除。当然，可以按照本节所介绍的算法，视具体的情况继续调整。

实际上稍后总结时将会看出，这也是双黑修正过程中，需要再次迭代的唯一可能。幸运的是，尽管此类情况可能持续发生，下溢的位置也必然不断上升，故至多迭代$\mathcal{O}(\log n)$次后必然终止。

■ 双黑修正（BB-3）

最后，考虑节点s为红色的情况。如图8.32(a)所示，即为一种典型的此类情况（与之对称的情况，请读者独立补充）。此时，作为红节点s的父亲，节点p必为黑色；同时，s的两个孩子也应均为黑色。

于是从B-树的角度来看，只需如图(b')所示，令关键码s与p互换颜色，即可得到一棵与之完全等价的B-树。而从红黑树的角度来看，这一转换对应于以节点p为轴做一次旋转，并交换节点s与p的颜色。

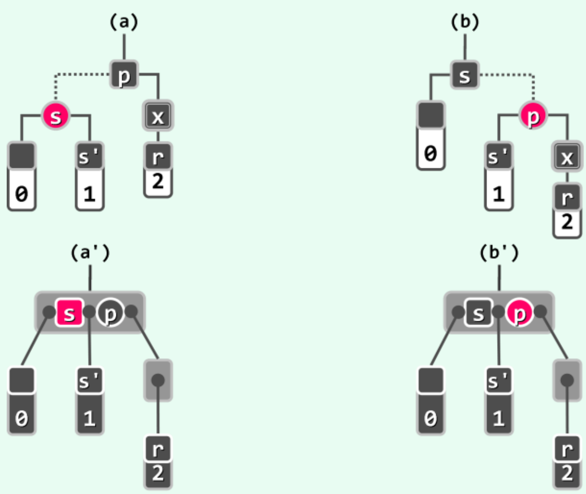

图8.32 双黑修正（情况BB-3）

读者可能会发现，经过如此处理之后，双黑缺陷依然存在，而且缺陷位置的高度也并未上升。既然如此，这一步调整的意义又何在呢？

实际上，经过这一转换之后，情况已经发生了微妙而本质的变化。仔细观察图(b)不难发现，在转换之后的红黑树中，被删除节点x（及其替代者节点r）有了一个新的兄弟s'——与此前的兄弟s不同，s'必然是黑的！这就意味着，接下来可以套用此前所介绍其它情况的处置方法，继续并最终完成双黑修正。

还有一处本质的变化，同样需要注意：现在的节点p，也已经黑色转为红色。因此接下来即便需要继续调整，必然既不可能转换回此前的情况BB-3，也不可能转入可能需要反复迭代的情况BB-2-B。实际上反过来，此后只可能转入更早讨论过的两类情况——BB-1或BB-2-R。这就意味着，接下来至多再做一步迭代调整，整个双黑修正的任务即可大功告成。

■ 双黑修正的复杂度

以上各种情况的处理流程，可以归纳为图8.33。

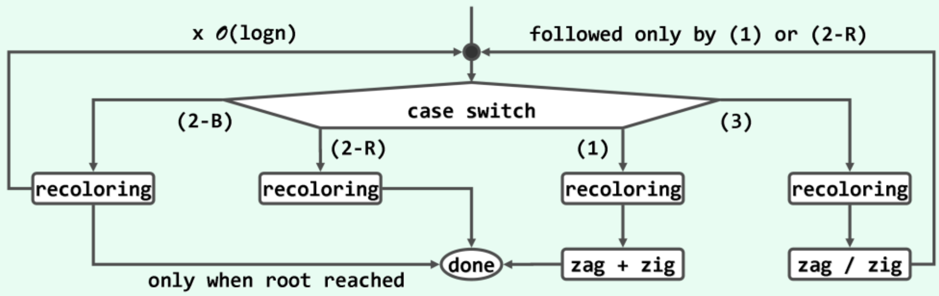

图8.33 双黑修正流程图

其中涉及的重构、染色等局部操作，均可在常数时间内完成，故为了估计整个双黑修正过程的时间复杂度，也只需统计这些操作各自的累计执行次数。具体统计可归纳为表8.2。

表8.2 双黑修正算法所涉及局部操作的统计

情　况		#旋转	#染色	单轮修正之后
BB-1	黑s有红子t	1~2	3	调整随即完成
BB-2-R	黑s无红子，p红	0	2	调整随即完成
BB-2-B	黑s无红子，p黑	0	1	必然再次双黑，但将上升一层
BB-3	红s	1	2	转为(BB-1)或(BB-2-R)

可见，前两种情况各自只需做一轮修正，最后一种情况亦不过两轮。

情况BB-2-B虽可能需要反复修正，但由于待修正位置的高度严格单调上升，累计也不致过$O(\log n)$轮，故双黑修正过程总共耗时不超过$O(\log n)$。即便计入此前的关键码查找和节点摘除操作，红黑树的节点删除操作总是可在$O(\log n)$时间内完成。

纵览各种情况，不难确认：一旦在某步迭代中做过节点的旋转调整，整个修复过程便会随即完成。因此与双红修正一样，双黑修正的整个过程，也仅涉及常数次的拓扑结构调整操作。

这一有趣的特性同时也意味着，在每此插入操作之后，拓扑联接关系有所变化的节点绝不会超过常数个——这一点与AVL树（的删除操作）完全不同，也是二者之间最本质的一项差异。

双黑修正算法的实现

以上针对双黑缺陷的各种修正方法，可以概括并实现如代码8.19所示。

```
/******************************************************************************************
 * RedBlack双黑调整算法：解决节点x与被其替代的节点均为黑色的问题
 * 分为三大类共四种情况：
 *    BB-1 ：2次颜色翻转，2次黑高度更新，1~2次旋转，不再递归
 *    BB-2R：2次颜色翻转，2次黑高度更新，0次旋转，不再递归
 *    BB-2B：1次颜色翻转，1次黑高度更新，0次旋转，需要递归
 *    BB-3 ：2次颜色翻转，2次黑高度更新，1次旋转，转为BB-1或BB2R
 ******************************************************************************************/
template <typename T> void RedBlack<T>::solveDoubleBlack ( BinNodePosi(T) r ) {
   BinNodePosi(T) p = r ? r->parent : _hot; if ( !p ) return; //r的父亲
   BinNodePosi(T) s = ( r == p->lc ) ? p->rc : p->lc; //r的兄弟
   if ( IsBlack ( s ) ) { //兄弟s为黑
      BinNodePosi(T) t = NULL; //s的红孩子（若左、右孩子皆红，左者优先；皆黑时为NULL）
      if      ( HasLChild ( *s ) && IsRed ( s->lc ) ) t = s->lc;
      else if ( HasRChild ( *s ) && IsRed ( s->rc ) ) t = s->rc;
      if ( t ) { //黑s有红孩子：BB-1
         RBColor oldColor = p->color; //备份原子树根节点p颜色，并对t及其父亲、祖父
         BinNodePosi(T) b = FromParentTo ( *p ) = rotateAt ( t ); //重平衡，并将新子树的左、右孩子染黑
         if ( HasLChild ( *b ) ) b->lc->color = RB_BLACK; updateHeight ( b->lc ); //左孩子
         if ( HasRChild ( *b ) ) b->rc->color = RB_BLACK; updateHeight ( b->rc ); //右孩子
         b->color = oldColor; updateHeight ( b ); //新子树根节点继承原根节点的颜色
      } else { //黑s无红孩子
         s->color = RB_RED; s->height--; //s转红
         if ( IsRed ( p ) ) { //BB-2R
            p->color = RB_BLACK; //p转黑，但黑高度不变
         } else { //BB-2B
            p->height--; //p保持黑，但黑高度下降
            solveDoubleBlack ( p );
         }
      }
   } else { //兄弟s为红：BB-3
      s->color = RB_BLACK; p->color = RB_RED; //s转黑，p转红
      BinNodePosi(T) t = IsLChild ( *s ) ? s->lc : s->rc; //取t与其父s同侧
      _hot = p; FromParentTo ( *p ) = rotateAt ( t ); //对t及其父亲、祖父做平衡调整
      solveDoubleBlack ( r ); //继续修正r处双黑——此时的p已转红，故后续只能是BB-1或BB-2R
   }
}
```

代码8.19 双黑修正solveDoubleBlack()

§8.4 *kd-树

8.4.1 范围查询

■ 一维范围查询

如图8.34所示，许多实际应用问题，都可归结为如下形式的查询：给定直线L上的点集P = { p_0, ..., p_{n-1} }，对于任一区间R = [x_1, x_2]，P中的哪些点落在其中？

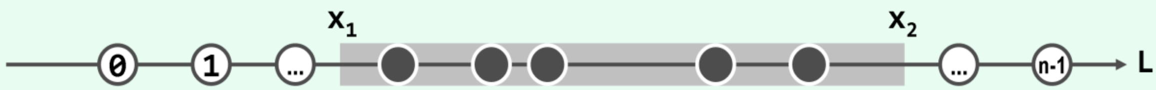

图8.34 一维范围查询

比如，在校友数据库中查询1970至2000级的学生，或者查询IP介于166.111.68.1至166.111.68.255之间的在线节点等，此类问题统称为一维范围查询（range query）。

■ 蛮力算法

表面看来，一维范围查询问题并不难解决。比如，只需遍历点集P，并逐个地花费$\mathcal{O}(1)$时间判断各点是否落在区间R内——如此总体运行时间为$\Theta(n)$。这一效率甚至看起来似乎还不差——毕竟在最坏情况下，的确可能有多达$\Omega(n)$个点命中，而直接打印报告也至少需要$\Omega(n)$时间。

然而，当我们试图套用以上策略来处理更大规模的输入点集时，就会发现这种方法显得力不从心。实际上，蛮力算法的效率还有很大的提升空间，这一点可从以下角度看出。

首先，当输入点集的规模大到需要借助外部存储器时，遍历整个点集必然引发大量I/O操作。正如8.2.1节所指出的，此类操作往往是制约算法实际效率提升的最大瓶颈，应尽量予以避免。

另外，当数据点的坐标分布范围较大时，通常的查询所命中的点，在整个输入点集中仅占较低甚至极低的比例。此时，“查询结果的输出需要$\Omega(n)$时间”的借口，已难以令人信服。

■ 预处理

在典型的范围查询应用中，输入点集数据与查询区域的特点迥异。一方面，输入点集P通常会在相当长的时间内保持相对固定——数据的这种给出及处理方式，称作批处理（batch）或离线（offline）方式。同时，对于同一输入点集，往往需要针对大量的随机定义的区间R，反复地进行查询——数据的这种给出及处理方式，称作在线（online）方式。

因此，只要通过适当的预处理，将输入点集P提前整理和组织为某种适当的数据结构，就有可能进一步提高此后各次查询操作的效率。

■ 有序向量

最为简便易行的预处理方法，就是在$\mathcal{O}(n\log n)$时间内，将点集P组织为一个有序向量。

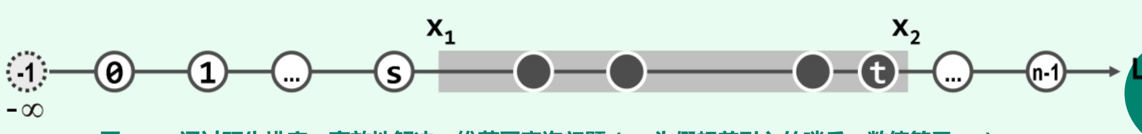

图8.35 通过预先排序，高效地解决一维范围查询问题（p_{-1}为假想着引入的哨兵，数值等于-∞）

如图8.35所示，此后对于任何R = [x_1, x_2]，首先利用有序向量的查找算法（代码2.20），在$\mathcal{O}(\log n)$时间内找到不大于x_2的最大点p_t。然后从p_t出发，自右向左地遍历向量中的各点，直至第一个离开查询区间的点p_s。其间经过的所有点，既然均属于区间范围，故可直接输出。

如此，在每一次查询中，p_t的定位需要$O(\log n)$时间。若接下来的遍历总共报告出r个点，则总体的查询时间成本为$O(r + \log n)$。

请注意，此处估计时间复杂度的方法，不免有点特别。这里，需要同时根据问题的输入规模和输出规模进行估计。一般地，时间复杂度可以这种形式给出的算法，也称作输出敏感的（output sensitive）算法。从以上实例可以看出，与此前较为粗略的最坏情况估计法相比，这种估计方法可以更加准确和客观地反映算法的实际效率。

二维范围查询

接下来的难点和挑战在于，在实际应用中，往往还需要同时对多个维度做范围查找。以人事数据库为例，诸如“年龄介于某个区间，而且工资介于某个区间”之类的组合查询十分普遍。

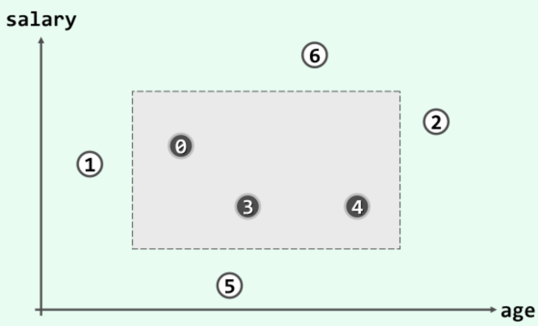

图8.36 平面范围查询（planar range query）

如图8.36所示，若将年龄与工资分别表示为两个正交维度，则人事数据库中的记录，将对应于二维平面上（第一象限内）的点。于是相应地，这类查询都可以抽象为在二维平面上，针对某一相对固定的点集的范围查询，其查询范围可描述为矩形$R = [x_1, x_2] \times [y_1, y_2]$。

很遗憾，上述基于二分查找的方法并不能直接推广至二维情况，更不用说更高维的情况了，因此必须另辟蹊径，尝试其它策略。

平衡二叉搜索树

我们还是回到该问题的一维版本，并尝试其它可以推广至二维甚至更高维版本的方法。比如，不妨在$O(n\log n)$时间内，将输出点集组织并转化为如图8.37所示的一棵平衡二叉搜索树。

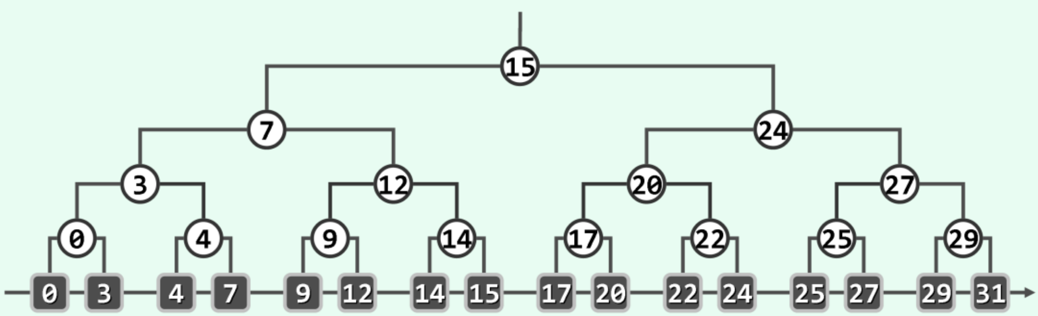

图8.37 平衡二叉搜索树：叶节点存放输入点，内部节点等于左子树中的最大者

请注意，其中各节点的关键码可能重复。不过，如此并不致于增加渐进的空间和时间复杂度：每个关键码至多重复一次，总体依然只需$O(n)$空间；尽管相对于常规二叉搜索树仅多出一层，但树高依然是$O(\log n)$。

如此在空间上所做的些许牺牲，可以换来足够大的收益：查找的过程中，在每一节点处，至多只需做一次（而不是两次）关键码的比较。当然另一方面，无论成功与否，每次查找因此都必然终止于叶节点——不小于目标关键码的最小叶节点。

不难验证，就接口和功能而言，此类形式二叉搜索树，完全对应于和等价于2.6.8节所介绍二分查找算法的版本C（代码2.24）。

■ 查询算法

借助上述形式的平衡二叉搜索树，如何高效地解决一维范围查询问题呢？

仍然继续上例，如图8.38所示，设查询区间为[1, 23]。

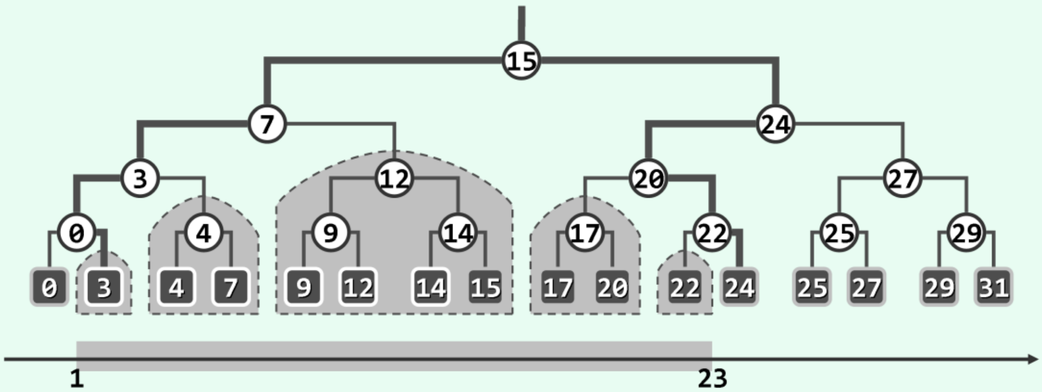

图8.38 借助平衡二叉搜索树解决一维范围查询问题（针对区间端点的两条查找路径加粗示意）

首先，在树中分别查找这一区间的左、右端点1和23，并分别终止于叶节点3和24。

接下来，考查这两个叶节点共同祖先中的最低者，即所谓的最低共同祖先（lowest common ancestor, LCA），具体地亦即

lca(3, 24)　=　15

然后，从这一共同祖先节点出发，分别重走一遍通往节点3和24的路径（分别记作path(3)和path(24)）。在沿着path(3)/path(24)下行的过程中，忽略所有的右转/左转；而对于每一次左转/右转，都需要遍历对应的右子树/左子树（图中以阴影示意），并将其中的叶节点悉数报告出来。就本例而言，沿path(3)被报告出来的叶节点子集，依次为：

{ 9, 12, 14, 15 }、{ 4, 7 }、{ 3 }

沿path(24)被报告出来的叶节点子集，依次为：

{ 17, 20 }、{ 22 }

■ 正确性

不难看出，如此分批报告出来的各组节点，都属于查询输出结果的一部分，且它们相互没有重叠。另一方面，除了右侧路径的终点24需要单独地判断一次，其余的各点都必然落在查询范围以外。因此，该算法所报告的所有点，恰好就是所需的查询结果。

■ 效率

在每一次查询过程中，针对左、右端点的两次查找及其路径的重走，各自不过$O(\log n)$时间（实际上，这些操作还可进一步合并精简）。

在树中的每一层次上，两条路径各自至多报告一棵子树，故累计不过$O(\log n)$棵。幸运的是，根据习题[5-11]的结论，为枚举出这些子树中的点，对它们的遍历累计不超过$O(r)$的时间，其中r为实际报告的点数。

综合以上分析，每次查询都可在$O(r + \log n)$时间内完成。该查询算法的运行时间也与输出规模相关，故同样属于输出敏感的算法。

新算法的效率尽管并不高于基于有序向量的算法，却可以便捷地推广至二维甚至更高维。

8.4.2 kd-树

循着上一节采用平衡二叉搜索树实现一维查询的构思，可以将待查询的二维点集组织为所谓的kd-树（kd-tree）[⑥]结构。在任何的维度下，kd-树都是一棵递归定义的平衡二叉搜索树。

以下不妨以二维情况为例，就2d-树的原理以及构造和查询算法做一介绍。

■ 节点及其矩形区域

具体地，2d-树中的每个节点，都对应于二维平面上的某一矩形区域，且其边界都与坐标轴平行。当然，有些矩形的面积可能无限。

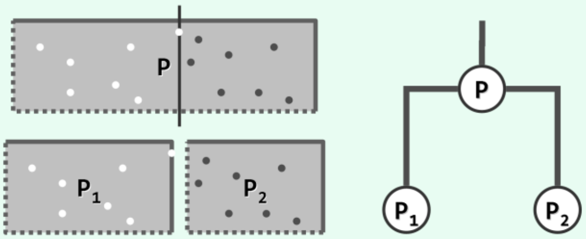

图8.39 2d-树中各节点对应的区域，逐层递归地按所包含的输入点数均衡切分

后面将会看到，同层节点各自对应的矩形区域，经合并之后恰好能够覆盖整个平面，同时其间又不得有任何交叠。因此，不妨如图8.39所示统一约定，每个矩形区域的左边和底边开放，右边和顶边封闭。

■ 构造算法

作为以上条件的特例，树根自然对应于整个平面。一般地如图8.39所示，若P为输入点集与树中当前节点所对应矩形区域的交集（即落在其中的所有点），则可递归地将该矩形区域切分为两个子矩形区域，且各包含P中的一半点。

若当前节点深度为偶（奇）数，则沿垂直（水平）方向切分，所得子区域随同包含的输入点分别构成左、右孩子。如此不断，直至子区域仅含单个输入点。每次切分都在中位点（median point）——按对应的坐标排序居中者——处进行，以保证全树高度不超过$\mathcal{O}(\log n)$。

具体地，2d-树的整个构造过程，可形式化地递归描述如算法8.1所示。

```
KdTree* buildKdTree(P, d) { //在深度为d的层次，构造一棵对应于（子）集合P的（子）2d-树
   if (P == {p}) return CreateLeaf(p); //递归基
   root = CreateKdNode(); //创建（子）树根
   root->splitDirection = Even(d) ? VERTICAL : HORIZONTAL; //确定划分方向
   root->splitLine = FindMedian(root->splitDirection, P); //确定中位点
   (P1, P2) = Divide(P, root->splitDirection, root->splitLine); //子集划分
   root->lc = buildKdTree(P1, d + 1); //在深度为d + 1的层次，递归构造左子树
   root->rc = buildKdTree(P2, d + 1); //在深度为d + 1的层次，递归构造右子树
   return root; //返回（子）树根
}
```

算法8.1 构造2d-树

⑥ 由J. L. Bentley于1975年发明[46]，其名字来源于“k-dimensional tree”的缩写
适用于任意指定维度的欧氏空间，并视具体的维度，相应地分别称作2d-树、3d-树、...，等
故上节所介绍的一维平衡二叉搜索树，也可称作1d-树

■ 实例

以图8.40(a)为例，首先创建树根节点，并指派以整个平面以及全部7个输入点。

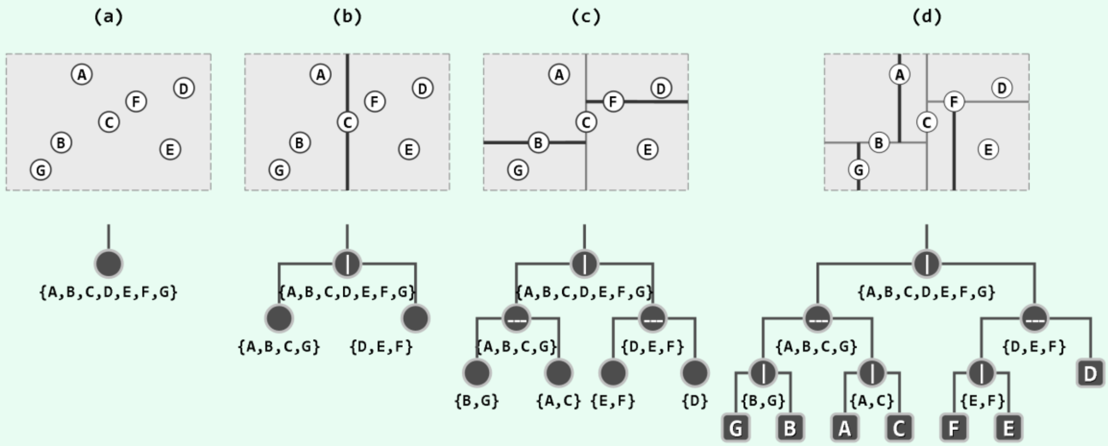

图8.40 2d-树的构造过程，就是对平面递归划分的过程

第一轮切分如图(b)所示。以水平方向的中位点C为界，将整个平面分作左、右两半，点集P也相应地被划分为子集{ A, B, C, G }和{ D, E, F }，它们随同对应的半平面，被分别指派给深度为1的两个节点。

第二轮切分如图(c)所示。对于左半平面及其对应的子集{ A, B, C, G }，以垂直方向的中位点B为界，将其分为上、下两半，并分别随同子集{B, G}和{A, C}，指派给深度为2的一对节点；对于右半平面及其对应的子集{ D, E, F }，以垂直方向的中位点F为界，将其分为上、下两半，并分别随同子集{ E, F }和{ D }，指派给深度为2的另一对节点。

最后一轮切分如图(d)所示。对树中仍含有至少两个输入点的三个深度为2的节点，分别沿其各自水平方向的中位点，将它们分为左、右两半，并随同对应的子集分配给三对深度为3的节点。至此，所有叶节点均只包含单个输入点，对平面的整个划分过程遂告完成，同时与原输入点集P对应的一棵2d-树也构造完毕。

8.4.3 基于2d-树的范围查询

■ 过程

经过如上预处理，将待查询点集P转化为一棵2d-树之后，对于任一矩形查询区域R，范围查询的过程均从树根节点出发，按如下方式递归进行。因为不致歧义，以下叙述将不再严格区分2d-树节点及其对应的矩形子区域和输入点子集。

在任一节点v处，若子树v仅含单个节点，则意味着矩形区域v中仅覆盖单个输入点，此时可直接判断该点是否落在R内。否则，不妨假定矩形区域v中包含多个输入点。

此时，视矩形区域v与查询区域R的相对位置，无非三种情况：

情况A：若矩形区域v完全包含于R内，则其中所有的输入点亦均落在R内，于是只需遍历一趟子树v，即可报告这部分输入点。

情况B：若二者相交，则有必要分别深入到v的左、右子树中，继续递归地查询。

情况C：若二者彼此分离，则子集v中的点不可能落在R内，对应的递归分支至此即可终止。

■ 算法

以上查询过程，可递归地描述如算法8.2所示。

```
kdSearch(v, R) { //在以v为根节点的(子)2d-树中，针对矩形区域R做范围查询
   if (isLeaf(v)) //若抵达叶节点，则
      { if (inside(v, R)) report(v); return; } //直接判断，并终止递归

   if (region(v->lc) ⊆ R) //情况A：若左子树完全包含于R内，则直接遍历
      reportSubtree(v->lc);
   else if (region(v->lc) ∩ R ≠ ∅) //情况B：若左子树对应的矩形与R相交，则递归查询
      kdSearch(v->lc, R);

   if (region(v->rc) ⊆ R) //情况A：若右子树完全包含于R内，则直接遍历
      reportSubtree(v->rc);
   else if (region(v->rc) ∩ R ≠ ∅) //情况B：若右子树对应的矩形与R相交，则递归查询
      kdSearch(v->rc, R);
}
```

算法8.2 基于2d-树的平面范围查询

可见，递归只发生于情况B；对于其余两种情况，递归都会随即终止。特别地，情况C只需直接返回，故在算法中并无与之对应的显式语句。

■ 实例

考查图8.41中的2d-树，设采用kdSearch()算法，对阴影区域进行查询。

不难验证，递归调用仅发生于黑色节点（情况B）；而在灰色节点处，并未发生递归调用（情况C或父节点属情况A）。

命中的节点共分两组：{ C }作为叶节点经直接判断后确定；{ F, H }则因其所对应区域完全包含于查询区域内部（情况A），经遍历悉数输出（习题[8-17]）。

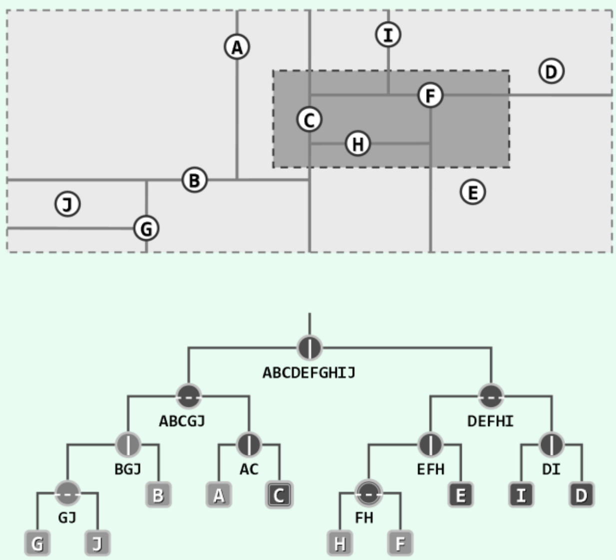

图8.41 基于2d-树的平面范围查询

（A～J共计10个输入点；命中子树的根节点，以双线圆圈示意）

■ 正确性

由上可见，凡被忽略的子树，其对应的矩形区域均完全落在查询区域之外，故该算法不致漏报。反之，凡被报告的子树，其对应的矩形区域均完全包含在查询区域以内（且互不相交），故亦不致误报。

■ 复杂度

平面范围查询与一维情况不同，在同一深度上可能递归两次以上，并报告出多于两棵子树。但更精细的分析（习题[8-16]）表明，被报告的子树总共不超过$O(\sqrt{n})$棵，累计耗时$O(\sqrt{n})$。