第11章 搜 索 树

本章是关于树结构的最后一章，我们将给出一种适合于描述字典的树形结构。第 7章中的字典描述仅能提供比较好的平均性能，而在最坏情况下的性能很差。当用跳表来描述一个 n 元素的字典时，对其进行搜索、插入或者删除操作所需要的平均时间为 O(log$n$)，而最坏情况下的时间为$\Theta(n)$。当用散列来描述一个n 元素的字典时，对其进行搜索、插入或者删除操作所需要的平均和最坏时间分别为 Θ(l)和$\Theta(n)$。使用跳表很容易对字典元素进行高效的顺序访问（如按照升序搜索元素），而散列却做不到这一点。当用平衡搜索树来描述一个n 元素的字典时，对其进行搜索、插入或者删除所需要的平均时间和最坏时间均为 Θ(log$n$)，按元素排名进行的查找和删除操作所需要的时间为 O(log$n$)，并且所有字典元素能够在线性时间内按升序输出。正因为这样（无论是平衡还是非平衡搜索树），所以在搜索树中进行顺序访问时，搜索每个元素所需要的平均时间为Θ(l)。

实际上，如果所期望的操作为查找、插入和删除（均根据元素的关键值来进行），则可以借助于散列函数来实现平衡搜索树。当字典操作仅按关键值来进行时，可将平衡搜索用于那些对时间要求比较严格的应用，以确保任何字典操作所需要的时间都不会超过指定的时间量。平衡搜索树也可用于按排名来进行查找和删除操作的情形。对于那些不按精确的关键值匹配进行字典操作的应用（比如寻找关键值大于k 的最小元素），同样可使用平衡搜索树。

本章将首先介绍二叉搜索树。这种树提供了可与跳表相媲美的渐进复杂性。其搜索、插入和删除操作的平均时间复杂性为 O(log$n$)，最坏时间复杂性为 $\Theta(n)$。接下来将介绍两种大家比较熟悉的平衡树： AVL树和红 - 黑树。无论哪一种树，其搜索、插入和删除操作都能在对数时间内完成（平均和最坏情况）。两种结构的实际运行性能也很接近， AVL树一般稍微快一些。所有的平衡树结构都使用“ 旋转 ”来保持平衡。 AVL树在执行每个插入操作时最多需要一次旋转，执行每个删除操作时最多需要 O(log$n$)次旋转；而红 - 黑树对于每个插入和删除操作，都需要执行一次旋转。这种差别对于大多数仅需 Θ(l)时间进行一次旋转的应用来说无关紧要，但对于那些不能在常量时间内完成一次旋转的应用来说就非常重要了，比如平衡优先搜索树 McCreight 就是这样一种应用。平衡优先搜索树用于描述具有两个关键值的元素，此时，每个关键值是一对数(x,y)。它同时是一个关于y 的优先队列和关于x 的搜索树。在平衡优先搜索树中执行旋转时，每次旋转都需耗时 O(log$n$)。如果用红 - 黑树来描述平衡优先搜索树，由于每一次插入或删除后仅需执行一次旋转，因此插入或删除操作总的时间复杂性仍保持为 O(log$n$)；当使用AVL树时，删除操作的时间将变为O(log$n$)。

如果所描述的字典比较小（能够完全放入内存），AVL树和红 - 黑树均能提供比较高的性能，但对于很大的字典来说，它们就不适用了。当字典存储在磁盘上时，需要使用带有更高次数（因而有更小高度）的搜索树，本章也将介绍一个这样的搜索树—— B - 树。

本章的应用部分将给出三个搜索树的应用。第一个是直方图的计算，第二个是 10.5.1节所介绍的NP - 复杂问题——箱子装载，最后一个是关于在电子布线中所出现的交叉分布问题。在直方图的应用中，使用散列函数来取代搜索树，从而使性能得到提高。在最优匹配箱子装载应用中，由于搜索不是按精确匹配完成的，所以不能使用散列函数。在交叉分布问题中，操作是按排名完成的，因此也不能使用散列函数。

11.1 二叉搜索树

11.1.1 基本概念

7.1和7.4节介绍了抽象数据类型*Dictionary*，从中可以发现当用散列来描述一个字典时，字典操作（包括插入、搜索和删除）所需要的平均时间为$\Theta(1)$。而这些操作在最坏情况下的时间正比于字典中的元素个数n。如果扩充*Dictionary*的ADT描述，增加以下操作，那么散列将不能再提供比较好的平均性能：

1) 按关键值的升序输出字典元素。

2) 按升序找到第k个元素。

3) 删除第k个元素。

为了执行操作1)，需要从表中取出数据，将它们排序后输出。如果使用除数为D的链表，那么能在$\Theta(D+n)$的时间内取出元素，在$O(n\log n)$时间内完成排序和在$\Theta(n)$时间内输出，因此共需时间$O(D+n\log n)$。如果对散列使用线性开型寻址，则取出元素所需时间为$\Theta(b)$，b是桶的个数，这时所需时间为$O(b+n\log n)$。如果使用链表，操作2)和3)可以在$O(D+n)$的时间内完成，而如果使用线性开型寻址，它们可在$\Theta(b)$时间内完成。为了获得操作2)和3)的这种复杂性，必须采用一个线性时间算法来确定n元素集合中的第k个元素（参考14.5节）。

如果使用平衡搜索树，那么对字典的基本操作（搜索、插入和删除）能够在$O(\log n)$的时间内完成，操作1)能在$\Theta(n)$的时间内完成。通过使用带索引的平衡搜索树，也能够在$O(\log n)$的时间内完成操作2)和3)。11.3节将考察其他一些散列无法做到而平衡树可以有效解决的应用。

在学习平衡树之前，首先来看一种叫作二叉搜索树的简单结构。

定义 [**二叉搜索树**] 二叉搜索树（binary search tree）是一棵可能为空的二叉树，一棵非空的二叉搜索树满足以下特征：

1) 每个元素有一个关键值，并且没有任意两个元素有相同的关键值；因此，所有的关键值都是唯一的。

2) 根节点左子树的关键值（如果有的话）小于根节点的关键值。

3) 根节点右子树的关键值（如果有的话）大于根节点的关键值。

4) 根节点的左右子树也都是二叉搜索树。

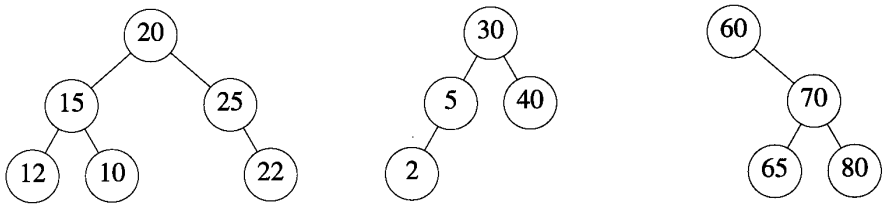
此定义中有一些冗余。特征2)、3)和4)在一起暗示了关键值必须是唯一的。因此，特征1)可以用这样的特征代替：根节点必须有关键值。然而，前一种定义比这种简化的定义要清楚明了。

图11-1给出了一些各元素含有不同关键值的二叉树。节点中的数字是元素的关键值。其中11-1a中的树尽管满足特征1)、2)和3)，但仍然不是二叉搜索树，因为它不满足特征4)，其中有一个子树的右子树的关键值（22）小于该子树根节点的关键值（25）。而图11-1b和c都是二叉搜索树。

我们可以放弃二叉搜索树中所有元素拥有不同关键值的要求，然后再用小于等于代替特征2)中的小于，用大于等于代替特征3)中的大于，这样，就得到了一棵有重复值的二叉搜索树（binary search tree with duplicates）。

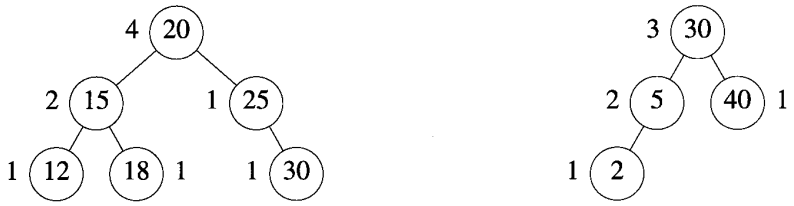
带索引的二叉搜索树（indexed binary search tree）源于普通的二叉搜索树，它只是在每个节点中添加一个LeftSize域。这个域的值是该节点左子树的元素个数加1。图11-2是两棵带索引

的二叉搜索树。节点里面的数字是元素的关键值，外面的是 LeftSize 的值。注意，LeftSize同时给出了一个元素在子树中排名。例如，在图11-2a 的树中，根为20的子树中的元素（已排序）分别为12，15，18，20，25和30，根节点的排名为4（即它在排序后的队列中是第4个元素），在根为25的子树中的元素（已排序）为25和30，因此25的排名为1且LeftSize 的值也为1。



a) b) c)

图11-1 二叉树



a) b)

图11-2 带索引的二叉搜索树

11.1.2 抽象数据类型BSTree 和IndexedBSTree

ADT 11-1给出了二叉搜索树的抽象数据类型描述。带索引的二叉搜索树支持所有的二叉搜索树操作。另外，它还支持按排名进行的查找和删除操作。 ADT 11-2给出了它的抽象数据类型描述。可以按照类似的方法来描述抽象数据类型 *DBSTree*(有重复值的二叉搜索树)和*DIndexedBSTree*。

ADT 11-1 二叉搜索树的抽象数据类型描述

抽象数据类型*BSTree* {

实例

二叉树，每一个节点中有一个元素，该元素有一个关键值域；所有元素的关键值各不相同；任何节点左子树的关键值小于该节点的关键值；任何节点右子树的关键值大于该节点的关键值。

操作

Create()：创建一个空的二叉搜索树

Search(k,e)：将关键值为*k* 的元素返回到*e* 中；如果操作失败则返回false，否则返回true

Insert(e)：将元素*e* 插入到搜索树中

Delete(k,e)：删除关键值为*k* 的元素并且将其返回到*e* 中

Ascend()：按照关键值的升序排列输出所有元素

}

ADT 11-2 带索引的二叉搜索树的抽象数据类型描述

抽象数据类型 *IndexedBSTree* {
实例
　除每一个节点有一个 LeftSize 域以外，其他与 BSTree 相同
操作
　Create()：产生一个空的带索引的二叉搜索树
　Search(k,e)：将关键值为 *k* 的元素返回到 *e* 中；如果操作失败返回 *false*，否则返回 *true*
　IndexSearch(k,e)：将第 *k* 个元素返回到 *e* 中
　Insert(e)：将元素 *e* 插入到搜索树
　Delete(k,e)：删除关键值为 *k* 的元素并且将其返回到 *e* 中
　IndexDelete(k,e)：删除第 *k* 个元素并将其返回到 *e* 中
　Ascend()：按照关键值的升序排列输出所有元素
}

11.1.3 类BSTree

因为在执行操作时，二叉搜索树中元素的数量和树的外形同时改变，所以可以用 8.4节中的链表来描述二叉搜索树。如果从类 BinaryTree（见程序8-7）中派生类BSTree，那么可以大大简化BSTree 类的设计，见程序 11-1。由于BSTree 是从BinaryTree 派生而来的，因此它继承了 BinaryTree 的所有成员。但是，它只能访问那些共享成员和保护成员。为了访问 BinaryTree 私有成员root，需要把BSTree 定义为BinaryTree 的友元。

程序11-1 二叉搜索树的类定义

```
template<class E, class K>
class BSTree : public BinaryTree<E> {
  public:
    bool Search(const K& k, E& e) const;
    BSTree<E,K>& Insert(const E& e);
    BSTree<E,K>& Delete(const K& k, E& e);
    void Ascend() {InOutput();}
};
```

IndexedBSTree 类也可以定义为BinaryTree 的一个派生类（见练习5）。可以通过调用8.9节所定义的中序输出函数——InOutput 将二叉搜索树按升序输出，该函数首先输出左子树中的元素（关键值较小的元素），然后输出根，最后输出右子树中的元素（关键值较大的元素）。对于有 n 个元素的树来说，该函数的时间复杂性为 $\Theta(n)$。

11.1.4 搜索

假设需要查找关键值为 k 的元素，那么先从根开始。如果根为空，那么搜索树不包含任何元素，查找失败，否则，将 k 与根的关键值相比较，如果 k 小于根节点的关键值，那么就不必搜索右子树中的元素，只要在左子树中搜索即可。如果 k 大于根节点的关键值，则正好相反，只需在右子树中搜索即可。如果 k 等于根节点的关键值，则查找成功，搜索终止。在

子树中的查找与此类似，程序 11-2给出了相应代码。该过程的时间复杂性为 O(*h*)，其中*h* 是树的高度。

程序11-2 在二叉搜索树中搜索元素

```
template<class E, class K>
bool BSTree<E,K>::Search(const K& k, E &e) const
{// 搜索与k匹配的元素
  // 指针 p 从树根开始进行查找
  BinaryTreeNode<E> *p = root;
  while (p) // 检查p->data
    if (k < p->data) p = p->LeftChild;
    else if (k > p->data) p = p->RightChild;
         else {// 找到元素
              e = p->data;
              return true;}
  return false;
}
```

可以用类似的方法在带索引的二叉搜索树中按索引进行查找。假设需要查找图 11-2a 中树的第三个元素，根节点的LeftSize为4，因此第三个元素在左子树中。左子树根节点的 LeftSize为2，因此第三个元素是左子树的右子树中的最小元素，而右子树根节点的 LeftSize是1，所以此根节点就是要找的元素。该操作时间复杂性也是O(*h*)。

11.1.5 插入

若在二叉搜索树中插入一个新元素e，首先要验证e 的关键值与树中已有元素的关键值是否相同，这可以通过用e 的关键值对二叉树进行搜索来实现。如果搜索不成功，那么新元素将被插入到搜索的中断点。例如，要将关键值为 80的元素插入到图11-1b 所示的树中去，首先对80进行搜索，由于搜索不成功而中断，最后检验的节点是关键值为 40的节点，新元素将被插入到该节点之下作为其右孩子。插入后的结果如图 11-3a 所示。图11-3b 给出了将关键值为35的元素插入到图11-3a 所示二叉树之后的结果。程序11-3实现了上述插入策略。

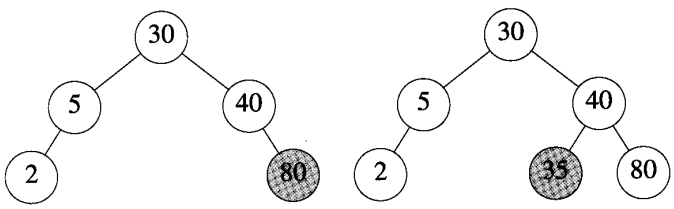


图11-3 将新元素插入到二叉搜索树中

程序11-3 将元素插入到二叉搜索树中

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::Insert(const E& e)
{// 如果不出现重复，则插入e
```

```
    BinaryTreeNode<E> *p = root,  // 搜索指针
                      *pp = 0;    // p的父节点指针
    // 寻找插入点
    while (p) {// 检查 p->data
      pp = p;
      // 将p移向孩子节点
      if (e < p->data) p = p->LeftChild;
      else if (e > p->data) p = p->RightChild;
           else throw BadInput(); // 出现重复
      }

    // 为e 建立一个节点，并将该节点连接至 to pp
    BinaryTreeNode<E> *r = new BinaryTreeNode<E> (e);
    if (root) {// 树非空
      if (e < pp->data) pp->LeftChild = r;
      else pp->RightChild = r;}
    else // 插入到空树中
       root = r;

    return *this;
  }
```

当将元素插入到带索引的二叉搜索树中时，可以用一个与程序 11-3相类似的过程，但此时需要更新从根节点到新插入节点路径上的所有节点的 LeftSize 值。插入过程仍然可以在O(*h*)时间内完成，其中*h* 是树的高度。

11.1.6 删除

对删除来说，我们考虑包含被删除元素的节点p 的三种情况：1) p 是树叶；2) p 只有一个非空子树；3) p 有两个非空子树。

情况1) 可以用丢弃树叶节点的方法来处理。要删除图 11-3b 所示树中的35，只要把其父节点的左孩子域置为零，然后删除该节点即可，删除后结果如图 11-3a 所示。要从树中删除 80，只要把节点40的右孩子域置为零并丢弃节点80，其结果如图11-1b 所示。

接下来考察情况2)。如果p 没有父节点（即p 是根节点），则将p 丢弃，p 的唯一子树的根节点成为新的搜索树的根节点。如果p 有父节点pp，则修改pp 的指针，使得pp 指向p 的唯一孩子，然后删除节点p。例如，如果希望从图11-3b 的树中删除关键值为5的元素，则修改该元素父节点的左孩子域，使其指向关键值为 2的节点。

最后，要删除一个左右子树都不为空的节点中的元素，只需将该元素替换为它的左子树中的最大元素或右子树中的最小元素。假设希望删除图 11-4a 中关键值为40的元素，那么既可以用它左子树中的最大元素 (35)，也可以用它右子树中的最小元素（60）来替换它。如果选择右子树中的最小元素，那么把关键值为60的元素移到40被删除的位置，再把原来的叶节点60删除即可。结果如图11-4b 所示。

假定用左子树中的最大元素来代替被删除的元素 40。左子树中的最大元素是35，且只有一个子女，把35移到40的节点中，将其左孩子指向原来节点35的唯一子女，结果如图11-4c 所示。

再来看另一个例子，删除图11-4c 中的节点30。既可以用5，也可以用31来替换节点30。如果选用5，而5是只有一个孩子的节点，那么只要把其左孩子域指向 5原来的唯一子女即可，结

果如图11-4d 所示。如果选用31替换30，而原来的31是树叶节点，那么只需删除该树叶节点。

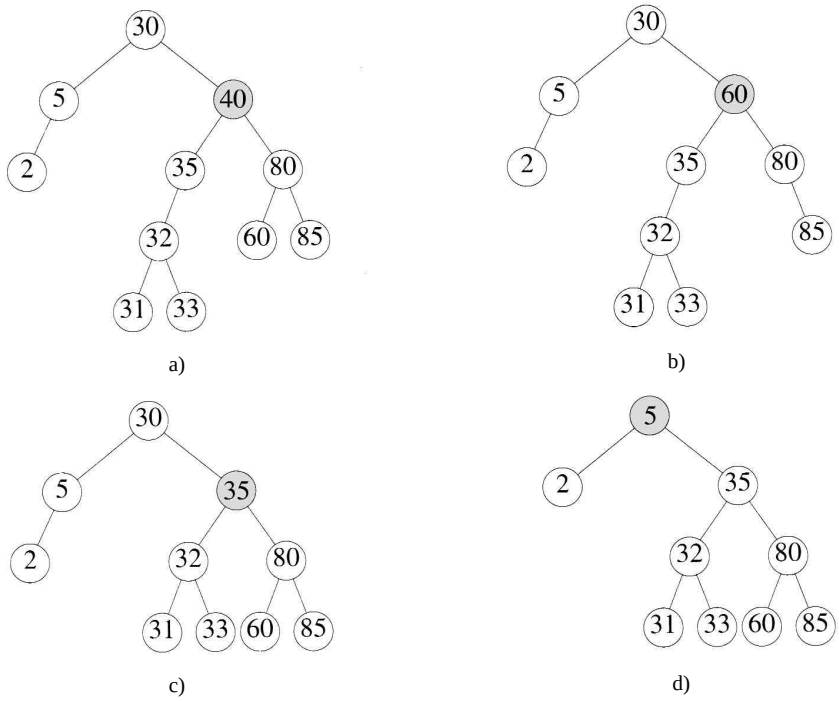


图11-4 二叉搜索树中元素的删除

注意，必须确保右子树中的最小元素以及左子树中的最大元素既不会在没有子树的节点中，也不会在只有一个子树的节点中。可以按下述方法来查找到左子树中的最大元素：首先移动到子树的根，然后沿着各节点的右孩子指针移动，直到右孩子指针为 0为止。类似地，也可以找到右子树中的最小元素：首先移动到子树的根，然后沿着各节点的左孩子指针移动，直到左孩子指针为0为止。

程序11-4给出了上述删除操作的算法。删除一个有两个非空子树的节点时，该程序一般使用左子树的最大元素来进行替换，其复杂性为 O(*h*)。用类似的过程，可以在相同的时间内完成带索引二叉搜索树的删除操作。首先按索引进行搜索，找到被删除元素，然后将其删除，如果需要的话，还需要修改从根至被删除元素路径上所有节点的 LeftSize 域。

程序11-4 二叉搜索树的删除

```
template<class E, class K>
BSTree<E,K>& BSTree<E,K>::Delete(const K& k, E& e)
{// 删除关键值为 k 的元素，并将其放入e

  // 将 p 指向关键值为 k的节点
  BinaryTreeNode<E> *p = root, // 搜索指针
                    *pp = 0;   // p的父节点指针
  while (p && p->data != k){// 移动到p的孩子
    pp = p;
    if (k < p->data) p = p->LeftChild;
```

```
    else p = p->RightChild;
    }
  if (!p) throw BadInput(); // 没有关键值为k的元素

  e = p->data;  // 保存欲删除的元素

  // 对树进行重构
  // 处理p有两个孩子的情形
  if (p->LeftChild && p->RightChild) {// 两个孩子
    // 转换成有0或1个孩子的情形
    // 在 p 的左子树中寻找最大元素
    BinaryTreeNode<E> *s = p->LeftChild, *ps = p;  // s的父节点
    while (s->RightChild) {// 移动到较大的元素
      ps = s;
      s = s->RightChild;}

    // 将最大元素从s移动到p
    p->data = s->data;
    p = s;
    pp = ps;}

  // p 最多有一个孩子
  // 在 c 中保存孩子指针
  BinaryTreeNode<E> *c;
  if (p->LeftChild) c = p->LeftChild;
  else c = p->RightChild;

  // 删除p
  if (p == root) root = c;
  else {// p 是 pp的左孩子还是pp的右孩子?
       if (p == pp->LeftChild)
          pp->LeftChild = c;
       else pp->RightChild = c;}
  delete p;

  return *this;
}
```

11.1.7 类DBSTree

若二叉搜索树中的不同元素可以包含相同的关键值，则称这种树为 DBSTree。在实现DBSTree类时，只需把BSTree::Insert的while循环（见程序11-3）改为程序11-5所示的while循环即可，其他代码无须改动。

程序11-5 对程序11-3的while 循环进行修改

```
while (p) {
  pp = p;
  if (e <= p->data) p = p->LeftChild;
```

```
    else p = p->RightChild;
    }
```

11.1.8 二叉搜索树的高度

一棵n 元素的二叉搜索树的高度可以与*n*一样大。例如，用程序11-3将一组关键值为[1,2,3,…,*n*]的元素按顺序插入到一棵空的二叉搜索树时，树的高度就会这样大，对树的搜索、插入和删除操作所需要的时间均为O(*n*)，这不比那些使用无序链表的操作好多少。但是可以证明，当用程序11-3和程序11-4进行随机插入和删除操作时，二叉搜索树的平均高度是 O(log*n*)。因此，每一个树操作的平均时间是O(log*n*)。

练习

1. 用跳表实现ADT11-1中的BSTree 操作需要多少时间（平均性能）？

2. 请描述抽象数据类型*DBSTree*（有重复值的二叉搜索树）。

3. 请描述抽象数据类型*DIndexedBSTree*。

4. 从BSTree类（见程序 11-1）中派生一个 C++ 类DBSTree，要求各函数的复杂性应与BSTree相同。测试程序的正确性。

5. 从BSTree 类（见程序11-1）中派生一个C++类IndexedBSTree。可以假设LeftSize 域是data域的一个子域。检验程序的正确性。根据元素的个数和（或）树的高度给出每个函数的复杂性。

6. 设计一个类IndexedBinaryTree，用于把线性表描述成一个二叉树（非二叉搜索树）。类必须支持程序3-1中所定义的所有线性表操作。除了Search函数以外，其他操作都必须在对数时间或更少的时间内完成。可以假设二叉树的平均高度是元素个数的对数。

7. 用DIndexedBSTree 替换IndexedBSTree 完成练习5，此时要求从DBSTree 类中派生DIndexedBSTree 类。

8. 首先产生一个从1到*n* 的随机排列，然后将关键值为1到*n* 的元素按照随机产生的排列顺序插入到一棵空的二叉树搜索树中。试测量二叉树的高度。重复以上实验，计算测量高度的平均值并与2[log2(*n*+1)]比较。*n* 的值可分别取100、500、1000、10 000、20 000、50 000。

9. 将程序11-2中while循环的第一次比较改为k==p->data，再用练习8中的方法随机产生不同大小的搜索树，比较修改前和修改后的程序在搜索树中元素时所需要的时间，从中能得出什么结论？

10. 二叉搜索树可用来对 n个元素进行排序。编写一个排序过程，首先将 n 个元素a[1:n]插入到一棵空的二叉搜索树中，然后对树进行中序遍历，并将元素按序放入数组 a 中。为简单起见，假设 a 中的数是互不相同的。将此过程的平均运行时间与插入排序和堆排序进行比较。

11. 编写一个从二叉搜索树中删除最大元素的函数，函数的时间复杂性必须是 O(*h*)，其中*h*是二叉搜索树的高度。

1) 用合适的测试数据测试代码的正确性。

2) 随机产生一个*n* 个元素的线性表和一个长度为*m* 的插入和最大删除操作序列。在所产生的操作序列中，插入操作的出现概率应近似为 0.5（同样，最大删除操作的概率也近似为 0.5）。使用第一个随机线性表中的*n* 个元素初始化一个最大堆和一棵二叉搜索树。检测用堆和二叉搜索树执行*m* 个操作的时间，用该时间除以*m* 就得到每一操作的平均时间。重复进行此实验，取

n=100, 500, 1000, 2000, …, 5000，假设m=5000。将所得结果用表格形式给出。

3) 指出这两种优先队列的相对优点和缺点。

*12. 扩充BinarySearchTree类，增加两个顺序访问函数Begin和Next。这两个函数分别返回指向字典第一个元素和下一个元素的指针。若没有第一个元素或没有下一个元素时，它们的返回值都是零。试证明这两个函数的平均复杂性均为Θ(l)。测试代码的正确性。

11.2 AVL树

11.2.1 基本概念

当确定搜索树的高度总是O(log*n*)时，能够保证每个搜索树操作所占用的时间为O(log*n*)。高度为O(log*n*)的树称为平衡树（balanced tree）。1962年，Adelson-Velskii 和Landis 提出了一种现在非常流行的平衡树——AVL树（AVL tree）。

定义 空二叉树是AVL树；如果T是一棵非空的二叉树，T_L和T_R分别是其左子树和右子树，那么当T满足以下条件时，T是一棵AVL树：1) T_L和T_R是AVL树；2) $|h_L-h_R|\leq 1$, h_L和h_R分别是左子树和右子树的高度。

AVL搜索树既是二叉搜索树，也是AVL树，图11-1a 和b 中的树都是AVL树，而c 不是。树a不是AVL搜索树，因为它不是二叉搜索树。树b 是AVL搜索树，图11-3中的树也是AVL搜索树。

带索引的AVL搜索树既是带索引的二叉搜索树，也是 AVL树。图11-2中的搜索树都是带索引的AVL搜索树，本节将不再具体介绍带索引的AVL搜索树。

如果用AVL树来描述字典并希望在对数时间内完成每一种字典操作，那么， AVL树必须具备下述特征：

1) n 个元素（节点）的AVL树的高度是O(log*n*)。

2) 对于每一个n（$n\geq 0$）值，都存在一棵AVL树。（否则，在插入完成后，一棵AVL树将不再是AVL树，因为对当前元素数来说不存在对应的AVL树）

3) 一棵n 元素的AVL 搜索树能在O(高度)=O(log$n$) 的时间内完成搜索。

4) 将一个新元素插入到一棵n 元素的AVL搜索树中，可得到一棵n+1元素的AVL树，这种插入过程可以在O(log*n*)时间内完成。

5) 从一棵n 元素的AVL搜索树中删除一个元素，可得到一棵n-1元素的AVL树，这种删除过程可以在O(log*n*)时间内完成。

特征4) 包含了特征2)，因此不需要明确说明特征2)，特征1)、3)、4) 和5) 将在以下小节中详细介绍。

11.2.2 AVL树的高度

我们能够获得一棵n 节点的AVL树的高度的范围。假设N_h 是一棵高度为h 的AVL树中最小的节点数。在最坏情况下，根节点的两个左右子树中一棵子树的高度是 h-1，另一棵子树的高度是h-2，而且两棵子树都是AVL树。因此有：

$$N_h=N_{h-1}+N_{h-2}+1,\quad N_0=0,\ N_1=1$$

可以看到N_h 的定义与斐波那契数列的定义非常相似：

$$F_n=F_{n-1}+F_{n-2},\quad F_0=0,\quad F_1=1$$

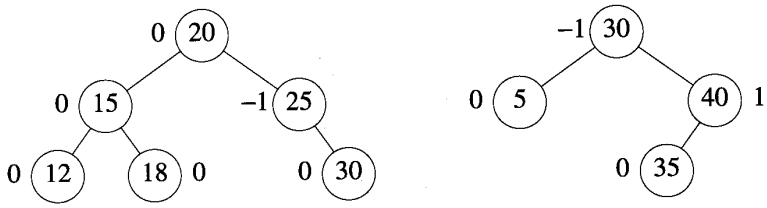
也可以这样来表示：$N_h = F_{h+2} - 1$，$h \geq 0$（见练习11）。由斐波那契定理可以知道$F_h \approx \phi^h / \sqrt{5}$，其中$\phi = (1+\sqrt{5})/2$，因此$N_h \approx \phi^{h+2} / \sqrt{5} - 1$。如果树中有$n$个节点，那么树的最大高度为：$\log_\phi (\sqrt{5}(n+1)) - 2 \sim 1.44 \log_2 (n+2) = O(\log n)$。

11.2.3 AVL树的描述

一般用链表方式来描述AVL树，但是，为简化插入和删除操作，我们为每个节点增加一个平衡因子bf。节点x 的平衡因子$bf(x)$ 定义为：

x 的左子树的高度 - x 的右子树的高度

从AVL树的定义可以知道，平衡因子的可能取值为 - 1，0和1。图11-5给出了两棵AVL搜索树和树中每个节点的平衡因子。



a) b)

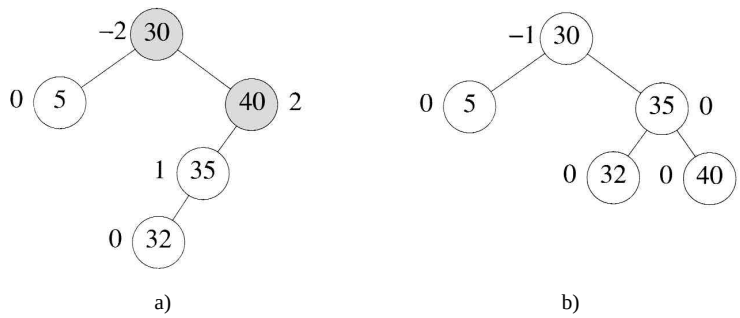
图11-5 AVL搜索树。每个节点外侧的数字是该节点的平衡因子

11.2.4 AVL搜索树的搜索

程序11-2可以不作任何修改就用于AVL搜索树的搜索。因为n元素AVL树的高度是O(log$n$)，所以搜索所需时间为O($n$log$n$)。

11.2.5 AVL搜索树的插入

如果用程序11-3的方法将元素插入到AVL搜索树中，得到的树可能不再是 AVL树。例如，如把一个关键值为32的元素插入到图11-5b 的AVL树中时，得到的新的搜索树如图 11-6a 所示。由于新树的节点中所包含的平衡因子不是 - 1，0和1，所以新树不是AVL树。当用程序 11-3的方法将一个新元素插入到AVL树中时，若得到的新树中有一个或多个节点的平衡因子的值不是 - 1，0或1，那么就说新树是不平衡的。可以通过移动不平衡树的子树来恢复树的平衡。



a) b)

图11-6 向AVL搜索树中插入元素

a) 刚插入时 b) 重新平衡之后

为了恢复平衡，在检查需移动的子树之前，先观察一下由插入操作导致产生不平衡树的几种现象：

1) 不平衡树中的平衡因子的值限于-2，-1，0，1和2。

2) 平衡因子为2的节点在插入前平衡因子为1，与此类似，平衡因子为-2的，插入前为-1。

3) 从根到新插入节点的路径上，只有经过的节点的平衡因子在插入后会改变。

4) 假设A是新插入节点最近的祖先，它的平衡因子是-2或2（图11-6a 的例子中，A是关键值为40的节点），那么，在插入前从A到新插入节点的路径上，所有节点的平衡因子都是0。

当我们从根节点往下移动寻找插入新元素的位置时，能够确定节点 A。从2) 中可以知道$bf(A)$在插入前的值既可以是-1，也可以是1。设X是最后一个具有这样平衡因子的节点，当把32插入到图11-5b 的AVL树中时，X是关键值为40的节点；当把22，28或50插入到图11-5a 的AVL 树中时，X是关键值为25的节点；当把10，14，16或19插入到图11-5a 的AVL 树中时，这样的节点X不存在。

如果节点X 不存在，那么从根节点至新插入节点途中经过的所有节点在插入前的平衡因子值都是0。由于插入操作只会使平衡因子增/减-1，0或1，并且只有从根节点至新插入节点途中经过的节点的平衡因子值才会被改变，所以插入后，树的平衡不会被破坏。因此，如果插入后的树是不平衡的，那么X就一定存在。如果插入后$bf(X)$=0，那么以X为根节点的子树的高度在插入前后是相同的。例如，如果插入前的高度是h，且$bf(X)$为1，那么，在插入前，X的左子树的高度X_L是$h-1$，右子树的高度X_R是$h-2$（如图11-7a所示）。由于平衡因子变为0，所以必须在X_R中作插入，得到高度为h-1的新子树X'_R（如图11-7b所示）。由于从X到新插入节点途中遇到的所有节点在插入前的平衡因子均为0，所以X'_R的高度必须增加到$h-1$。X的高度仍保持为h，X的祖先的平衡因子在插入前后保持相同，这样，所以树的平衡被保持住了。

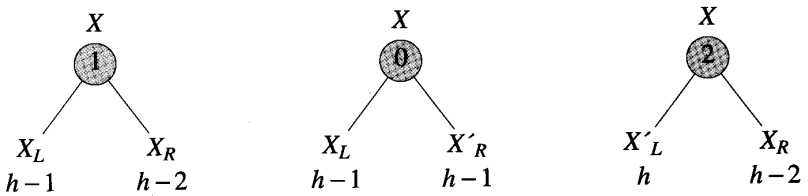
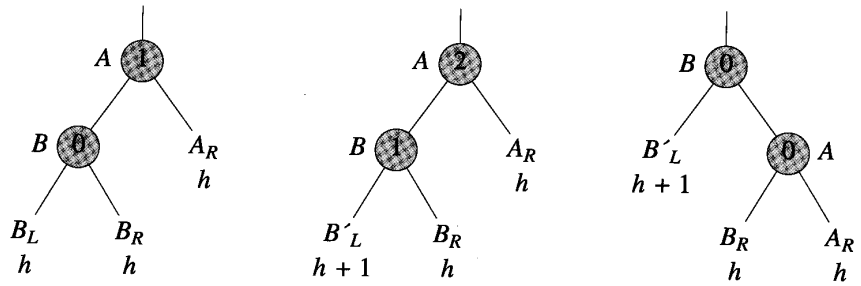


图11-7 向AVL搜索树中插入元素

a) 插入之前 b) 插入到X_R之后 c) 插入到X_L之后

使树的平衡遭到破坏的唯一一种情形是插入过程使得平衡因子$bf(X)$的值由-1变为-2，或者由1变为2。对于后一种情况来说，插入操作一定是在X的左子树X_L中进行（如图11-7c所示）的。现在要把X'_L的高度调整为h（因为在插入前所有从X到新插入节点途中的节点的平衡因子都为0），因此，X即4) 中所说的节点A。

当节点A已经被确定时，A的不平衡性可归类为L型不平衡（新插入节点在A的左子树中）或R型不平衡。通过确定A的哪一个孙节点在通往新插入节点的路径上，可以进一步细分不平衡类型。注意这种孙节点肯定存在，这是因为A节点的平衡因子是-2或2，所以节点A包含新插入节点的子树的高度至少必须是2。根据上述细分不平衡类型的方法，A节点的不平衡类型将是LL（新插入节点在A节点的左子树的左子树中），LR（新插入节点在A节点的左子树的右子树中），RR 和RL 四种类型中的一种。

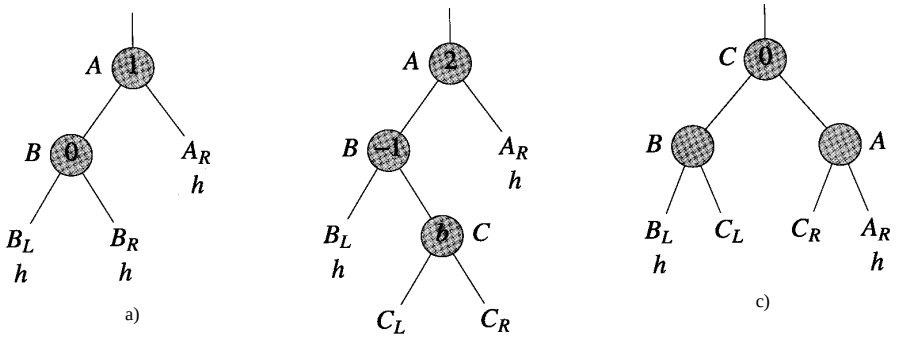


a) b) c)

图11-8 *LL*旋转。节点内为平衡因子，子树名称下面为子数的高度

a) 插入之前 b) 插入B_L之后 c) *LL*旋转之后

图11-8显示了一种普通的*LL*型不平衡。图11-8a 给出了插入前的条件，图11-8b 是在节点*B*的左子树*B′* 中插入一个元素后的情形，而恢复平衡所进行的子树移动如图 11-8c 所示。原来以*A* 为根节点的子树，现在以*B* 为根节点，B'_L 仍然是*B* 的左子树，*A* 变成*B* 的右子树，B_R 变成*A*的左子树，*A*的右子树不变。由于*A* 的平衡因子改变了，所以处于从 *B* 到新插入节点途中的B'的所有节点的平衡因子都将改变，其他节点的平衡因子与旋转前保持一致。图 11-8中a 和c 子树的高度是一样的，所以，子树的祖父节点的平衡因子与插入前是一样的。因此不再有平衡因子不是 - 1，0或1的节点。一个*LL* 旋转就已经使整个树重新获得平衡！可以验证重新平衡后的树确实是一棵二叉搜索树。



a) b) c)

若b=0，则bf(B)=bf(A)=0

若b=1，则bf(B)=0，bf(A)=-1

若b=-1，则bf(B)=1，bf(A)=0

图11-9 L_R 旋转

a) 插入之前 b) 插入B_R之后 c) *LR*旋转之后

图11-9给出了一种普通的LR 型不平衡。因为插入操作发生在 *B* 的右子树，这个子树在插入后不可能为空，因此*C* 是存在的。但是，它的子树 C_L 和C_R 有可能为空。为了恢复平衡，需要对子树进行重新整理，如图 11-9c 所示。重新整理后，*bf* (*B*) 和*bf* (*A*) 的值取决于*bf* (*C*) 在插入之后、重新整理之前的值 *b*。可以看到，重新整理后的子树仍是二叉搜索树。另外，由于图

11-9a 和c 中子树的高度是相同的，所以它们祖先（如果有的话）的平衡因子在插入前与在插入后也是相同的。因此，一个LR 旋转即可完成整个树的平衡。

RR 和RL 与上面所讨论的情形是对称的。我们把矫正 LL 和RR 型不平衡所作的转换称为单旋转（single rotation），而把矫正LR 和RL 型不平衡所作的转换称为双旋转（double rotation）。对LR 型不平衡的转换可以看作RR 旋转后的LL 旋转，而对RL 型不平衡的转换可以看作LL 旋转后的RR 旋转（练习15）。

根据上述讨论，可得到 AVL搜索树的插入算法，其步骤如图 11-10所示。这些步骤可以用C++代码重写，其复杂性为O(高度)=O(log$n$)。注意，如果插入引起了不平衡，使用单旋转就足以恢复平衡。

1) 沿着从根节点开始的路径对具有相同关键值的元素进行搜索，以找到插入新元素的位置。在此过程中，寻找最近的，平衡因子为 -1或1的节点，令其为A 节点。如果找到了相同关键值的元素，那么插入失败，以下步骤无需执行。
2) 如果没有这样的节点A，那么从根节点开始再遍历一次，并修改平衡因子，然后终止。
3) 如果$bf(A)$=1并且新节点插入到A的的右子树中，或者$bf(A)$=-1并且插入是在左子树中进行的，那么A 的新平衡因子是0。这种情况下，修改从A 到新节点途中的平衡因子，然后终止。
4) 确定A 的不平衡类型并执行相应的旋转，在从新子树根节点至新插入节点途中，根据旋转需要修改相应的平衡因子。

图11-10 AVL搜索树的插入步骤

11.2.6 AVL搜索树的删除

通过执行程序11-4，可从AVL搜索树中删除一个元素。设q 是被删除节点的父节点。如果要删除图11-5a 树中关键值为25的元素，那么删除包含该元素的节点，并且将根节点的右孩子指针指向被删除节点的唯一孩子。根节点是被删除节点的父节点，所以 q 就是根节点。如果被删除元素的关键值是15，那么关键值为12的元素将占用它的位置，而原来包含此元素的节点被删除。现在q 是原先包含15的节点（根的左孩子）。由于从根到q 途中的一些（全部）节点的平衡因子随着删除操作而改变了，所以再从q 沿原路返回根节点。

如果删除发生在q 的左子树，那么$bf(q)$ 减1，而如果删除发生在q 的右子树，那么$bf(q)$ 加1。可以看到如下现象：

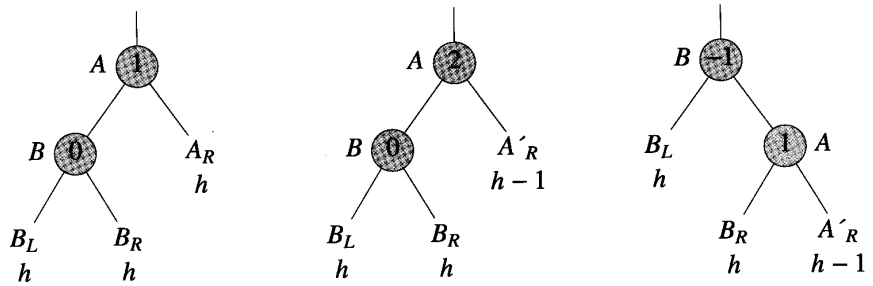
1) 如果q 新的平衡因子是0，那么它的高度减少了1，并且需要改变它的父节点（如果有的话）和其他某些祖先节点的平衡因子。

2) 如果q 新的平衡因子是－1或1，那么它的高度与删除前相同，并且无需改变其祖先的平衡因子值。

3) 如果q 新的平衡因子是－2或2，那么树在q 节点是不平衡的。

由于平衡因子可以沿从q 到根节点的路径改变（见2)），所以途中节点的平衡因子有可能为2或－2。设A 是第一个这样的节点，若要恢复A 节点的平衡，需要确定其不平衡的类型。如果删除发生在A 的左子树，那么不平衡是L 型；否则，不平衡就是R 型。如果删除后$bf(A)=2$，那么在删除之前$bf(A)$ 的值一定为1。因此，A 有一棵以B 为根的左子树。根据$bf(B)$ 的值，可以把一个R型不平衡细分为R0，R1和R－1类型。例如，R－1类型指的是这种情况：删除操作发生在A的右子树并且$bf(B)$=-1。类似的，L型不平衡也可以细分为L0，L1和L－1类型。

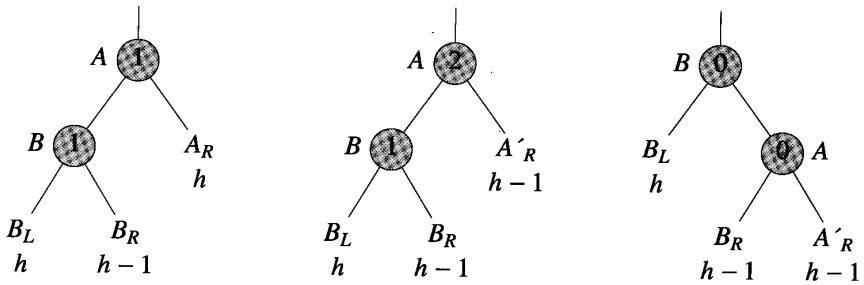
可通过旋转来矫正A点的R0型不平衡，如图11-11所示。注意图中子树的高度在删除前和删除后都是h+2，因此，到根节点途中的其他节点的平衡因子值没有改变。所以，整棵树重新获得了平衡。



a) b) c)

图11-11 R0类型的旋转（单旋转）

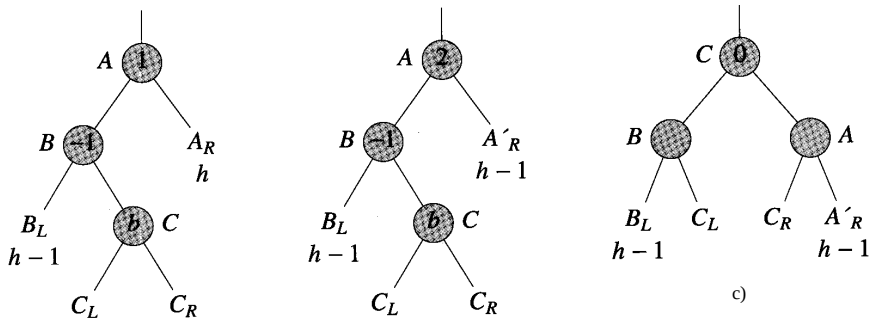
a) 删除之前 b) 以A_R中删除之后 c) R0旋转之后



a) b) c)

图11-12 R1类型的旋转（单旋转）

a) 删除之前 b) 以A_R中删除之后 c) R1旋转之后



a) b) c)

若b=0，则$bf(A)$=$bf(B)$=0

若b=1，则$bf(A)$=-1，$bf(B)$=0

若b=-1，则$bf(A)$=0，$bf(B)$=1

图11-13 R-1类型的旋转（双旋转）

a) 删除之前 b) 以A_R中删除之后 c) R-1旋转之后

图11-12给出了如何处理R1型不平衡。当指针的变化与R0型不平衡中的变化相同时，*A*和*B*的新平衡因子是不相同的并且旋转后子树的高度将是*h*+1，此高度比删除操作前减少了1。因此，如果*A*不是根节点，它的某些祖先的平衡因子将产生变化，可能需要进行旋转以保持平衡。R1旋转后，必须继续检查到达根节点路径上的节点。与插入情况不同，在删除操作之后，一次旋转可能还无法恢复平衡。所需要的旋转次数为O(log*n*)。

R-1类型的不平衡所需要的转换如图11-13所示。节点*A*和*B*旋转后的平衡因子取决于*B*的右孩子的平衡因子*b*，这次旋转得到了一棵高度为*h*+1的子树，而删除前子树的高度是*h*+2，因此，需要在到达根节点的路径上继续旋转。

LL与R1类型的旋转相同；LL与R0型旋转的区别仅在于*A*和*B*最后的平衡因子；而LR与R-1旋转也完全相同。

练习

13. 用数学归纳法证明：一棵高度为*h*的AVL树的最少节点数是

$$N_h = F_{h+2} - 1, \quad h \geq 0$$

14. 用程序11-3的方法证明11.2.5节中由插入操作导致不平衡树的现象1)～4)。

15. 为插入前$bf(X) = -1$这种情况画一个类似于图11-7的图。

16. 为RR和RL不平衡画一个类似于图11-8和11-9的图。

17. 从图11-9b所示的LR不平衡开始，画一个在*B*节点执行一个RR旋转的结果示意图。注意，对得到的树执行一次LL旋转即可得到图11-9b的树。

18. 为L0，L1和L-1不平衡情况分别画一个类似于图11-11，11-12和11-13的图。

*19. 设计一个C++的类AVLtree，它包含二叉搜索树的Search,Insert, Delete和Ascend函数，给出所有函数的代码并检验其正确性。前三种函数的时间复杂性应为O(log*n*)，最后一种函数的时间复杂性应为$\Theta(n)$。

*20. 针对如下情况完成练习19：二叉搜索树中有一些元素的关键值相同。新类的名称为DAVLtree。

*21. 设计一个C++类IndexedAVLTree，其中包含索引二叉搜索树的如下函数：Search，Insert，Delete，IndexSearch，IndexDelete和Ascend。给出函数的全部代码并检验其正确性。前5种函数的时间复杂性应为O(log*n*)，最后一种函数的时间复杂性应为$\Theta(n)$。

*22. 针对如下情况完成练习21：二叉搜索树中有一些元素的关键值相同。新类的名称为DIndexedAVLtree。

23. 解释一下如何用AVL树将5.5.3节中提到的火车车厢重排问题解决方法的渐进复杂性减少到O(*n*log*k*)。

*24. 设计一个类IndexedAVLList，将线性表描述为一棵二叉树，该二叉树与AVL树的区别仅在于它可能不是一棵二叉搜索树。类应支持程序3-1中所定义的所有线性表操作。除了Search函数，其他所有函数的运行时间不应超过对数时间。

11.3 红-黑树

11.3.1 基本概念

红-黑树（red-black tree）是这样的一棵二叉搜索树：树中的每一个节点的颜色是黑色或红色。红-黑树的其他特征可以用相应的扩充二叉树来说明。回忆一下9.4.1节，在一个规则的

二叉树中，用外部节点来替换每一个空指针，就得到了一棵扩充的二叉树。

RB1：根节点和所有外部节点的颜色是黑色。

RB2：根至外部节点途中没有连续两个节点的颜色是红色。

RB3：所有根至外部节点的路径上都有相同数目的黑色节点。

另一种等价的定义源于一个节点与其子女间指针的颜色。从父节点至黑色孩子的指针是黑色的，而至红色孩子的指针是红色的。

RB1'：从内部节点指向外部节点的指针是黑色的。

RB2'：从根至外部节点的途中没有两个连续的红色指针。

RB3'：所有根至外部节点的路径上都有相同数目的黑色指针。

注意，如果知道指针的颜色，就能够推断节点的颜色，反之亦然。在图11-14的红-黑树中，方块是外部节点，粗线是黑色指针，细线是红色指针。可以从指针的颜色和特征 RB1推断出节点的颜色。节点5，50，62和70是红色的，因为它们的父节点指向它们的指针是红色的，其余的节点是黑色的。注意，从根至外部节点的每条路径上都有两个黑色指针和三个黑色节点（包括根和外部节点）；不存在含有两个连续红色节点或指针的路径。

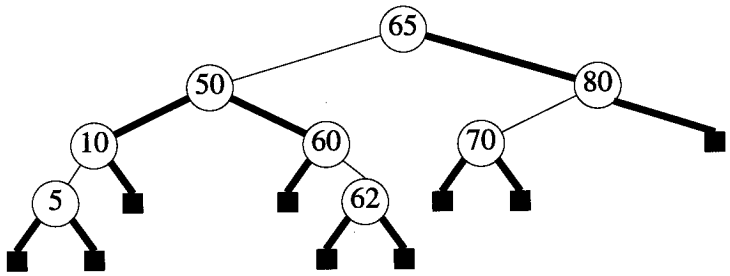


图11-14 红-黑树

设红-黑树中某节点的阶（rank）是从该节点到其子树中任意外部节点的任一条路径上的黑色指针的数量。因此，外部节点的阶是零，图 11-14中根节点的阶是2，其左孩子的阶是2，右孩子的阶是1。

定理11-1 设从根到外部节点的路径的长度（length）是该路径中指针的数量。如果P 和Q 是红-黑树中的两条从根至外部节点的路径，那么

$$length(P) \leq 2length(Q)$$

证明 考察任意一棵红-黑树。假设根节点的阶是r，由RB1' 知每条从根至外部节点的路径中最后一个指针为黑色，从 RB2' 知不存在包含连续两个红色指针的路径。因此，每个红色指针的后面都会跟着一个黑色指针，因而，每一条从根至外部节点的路径上都有 $r \sim 2r$ 个指针，故有$length(P) \leq 2length(Q)$。为了验证上限的可能性，可参考图11-14中的红-黑树。从根至5的左孩子的路径长度是4，而到80的右孩子的长度是2。

定理11-2 设h 是一棵红-黑树的高度（不包括外部节点），n 是树中内部节点的数量，而r 是根节点的阶，则

1) $h \leq 2r$。

2) $n \geq 2r-1$。

3) $h \leq 2\log_2(n+1)$。

证明 在定理11-1的证明中，我们知道从根至外部节点的路径的长度不会超过 $2r$，因此$h \leq 2r$（图11-14中除去外部节点的红-黑树的高度是$2r=4$）。

因为根节点的阶是r，所以从第1层至第r 层没有外部节点，因而在这些层中有 2^r-1个内部节点。也即内部节点的总数至少应为2^r-1（在图11-14的红-黑树中，第1和2层共有$2^2-1=3$个内部节点，而在第3和4层中还包含了其他内部节点）。

由2) 可以得到$r \leq \log_2(n+1)$，与1) 合起来即得到3)。

由于红-黑树的高度最多是$2\log_2(n+1)$，所以，搜索、插入和删除操作（在O(h)时间内完成）的复杂性为O(log$n$)。

注意，最坏情况下的红-黑树的高度大于最坏情况下具有相同（内部）节点数目的 AVL树的高度（近似于$1.44\log_2(n+2)$）。

11.3.2 红-黑树的描述

虽然在定义红-黑树时，将外部节点包括进来非常方便，但在执行过程中，我们仍然愿意用零指针或空指针，而不是物理节点来描述这些节点。进一步的说，由于指针的颜色与节点的颜色是紧密联系的，因此对于每个节点，需要储存的只是该节点的颜色或指向它的两个孩子的指针的颜色。存储每个节点的颜色只需要附加一个位，而存储每个指针的颜色则需要两位。既然两种方案需要的空间几乎相同，所以应该基于红-黑树算法的实际运行时间来做出选择。

在插入和删除操作的讨论中，只对节点颜色的改变做明确的说明，相应的指针颜色的变化可由推断得到。

11.3.3 红-黑树的搜索

可以使用对普通二叉搜索树进行搜索的代码（见程序 11-2）来完成对红-黑树的搜索。原代码的复杂性为O(h)，对于红-黑树则为O(log$n$)。由于用相同的代码来搜索普通二叉搜索树、AVL树和红-黑树，并且在最坏情况下 AVL树的高度是最小的，因此，在那些以搜索操作为主的应用中，在最坏情况下AVL树能获得最优的时间复杂性。

11.3.4 红-黑树的插入

可以利用普通二叉树的插入算法（见程序 11-3）将元素插入到红-黑树。当将新元素插入到红-黑树中时，需要为它上色，如果插入前是空树，那么新节点是根节点，颜色必须是黑色（参看特征RB1）。假设插入前树非空，如果新节点的颜色被赋予黑色，那么在从根到外部节点的路径中，将有一个特殊的黑色节点作为新节点的孩子。另一方面，如果新节点被赋予红色，那么可能出现两个连续的红色节点。把新节点赋为黑色将肯定导致违反 RB3，而把新节点赋为红色将可能违反，也可能不违反RB2，因此，应将新节点赋为红色。

如果将新节点赋为红色而导致违反特征 RB2，就说树的平衡被破坏了。通过检查新节点 u，其父节点pu 及祖父节点gu，可以确定不平衡的类型。考察违反RB2后的情况。现在有两个连续的红色节点，一个是u，另一个一定是它的父节点，因此pu 存在。因为pu 是红色的，它不可能是根（由特征RB1知根是黑色的），那么u 必然有一个祖父节点gu，并且它的颜色是黑色的（特征RB2）。当pu 是gu 的左孩子，u 是pu 的左孩子且gu 的另一个孩子是黑色时（这种情况包括gu的另一个孩子是外部节点的情况），不平衡是LLb 类型。其他的不平衡类型是 LLr（pu 是gu 的左孩子，u 是pu 的左孩子且gu 的另一个孩子是红色的），LRb（pu 是gu 的左孩子，u是pu 的

右孩子且gu的另一个孩子是黑色的），LLb，LRr，RRb，RRr，RLb和RLr。

XYr（X 和Y 既可以是L，也可以是R）类型的不平衡可以通过改变颜色来处理，而 XYb 类型则需要使用旋转。当改变一个节点的颜色时，树中可能有两层违反 RB2。这时需要对新层进行重新分类，将u 变为gu，然后再次进行转换。旋转结束后，不再违反 RB2，因此不需要再进行其他操作。

图11-15给出了LLr 和LRr 型不平衡的颜色变化，这些颜色的变化是一致的。黑色节点用深色阴影表示，而红色节点用浅色阴影表示，例如，在图 11-15中，gu 是黑色节点，u 和pu 是红色节点。从gu 到它的左右孩子的指针是红色的，gu_R 是gu 的右子树，pu_R 是pu 的右子树，LLr 和LRr 颜色的改变都需要我们将pu 的颜色和gu 右孩子的颜色由红色改为黑色。另外，如果gu 不是根，还要将gu 的颜色由黑色改为红色。由于gu 是根节点时其颜色没有改变，因此，若gu 是红-黑树的根节点，所有从根至外部节点路径上的黑色节点的数量都增加了一。

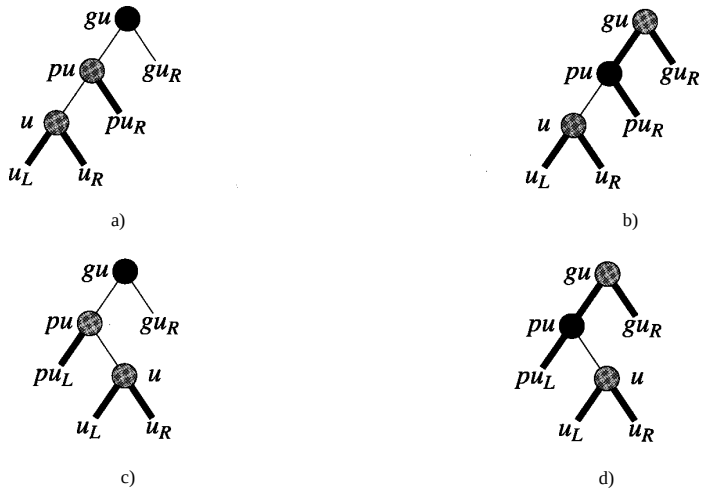


图11-15 LLr 和LRr 类型的颜色变化

a) LLr 型不平衡 b) LLr 颜色改变之后 c) LRr 型不平衡 d) LRr 颜色改变之后

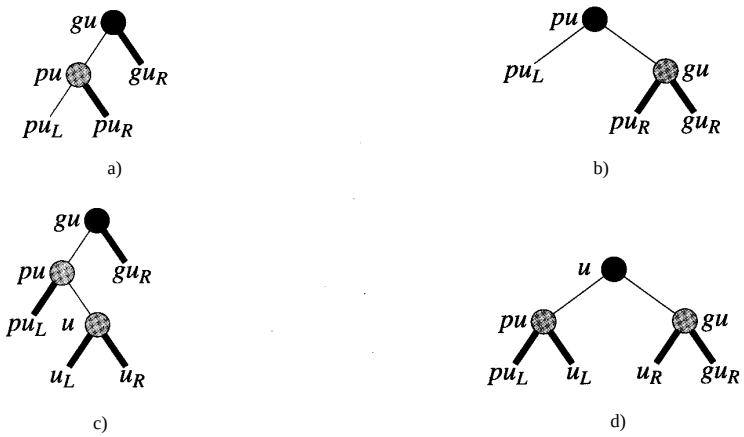


图11-16 红-黑树插入操作后的LLb和LRb旋转

a) LLb 型不平衡 b) LLb 旋转之后 c) LRb 型不平衡 d) LRb 旋转之后

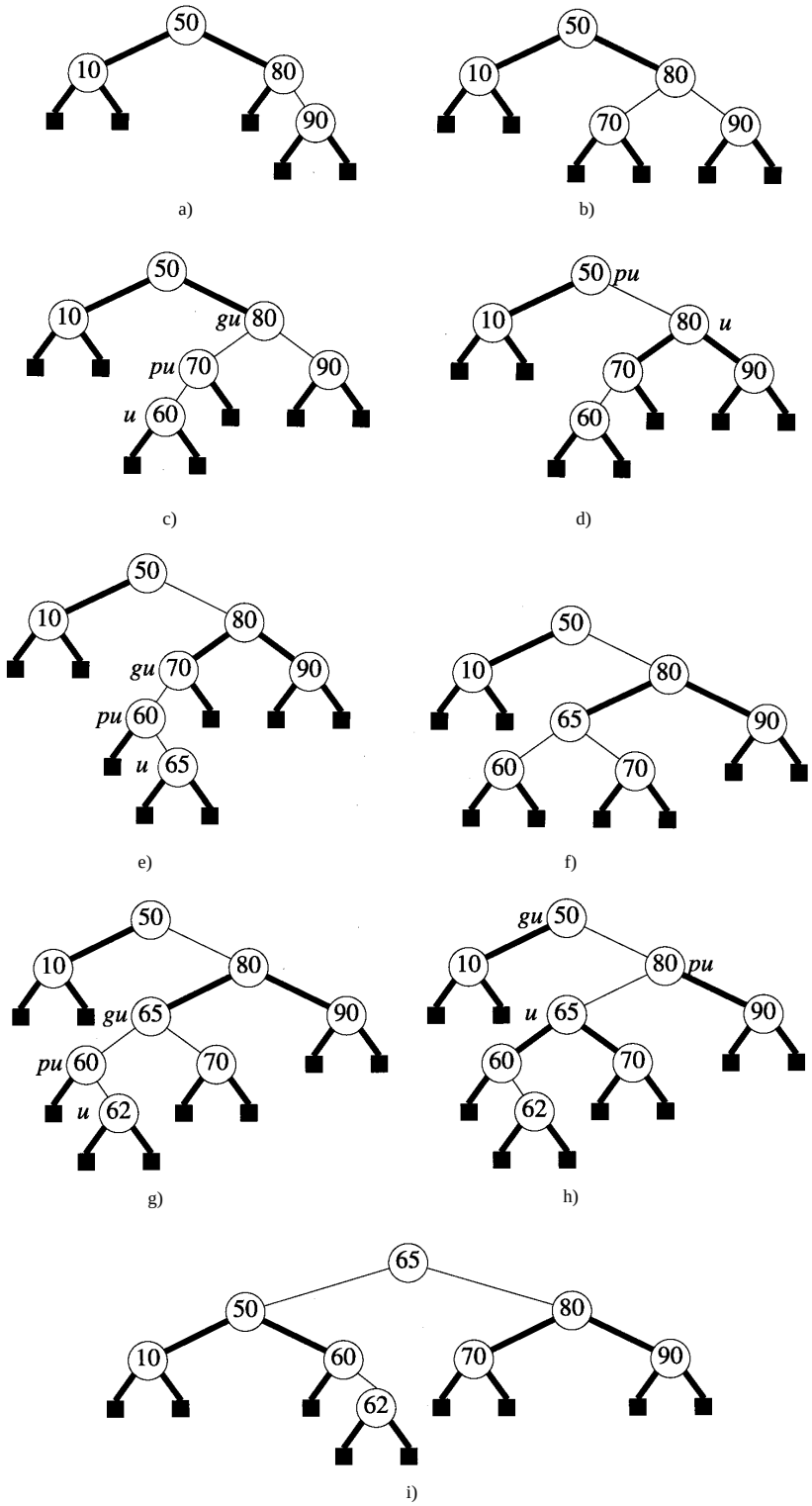


图11-17 红-黑树的插入

a) 初始状态 b) 插入70 c) 插入60 d) LLr 颜色 e) 插入65 f) LRb 旋转 g) 插入62 h) LRr 颜色 i) RLb 旋转

如果将*gu* 的颜色改为红色而引起了不平衡，那么 *gu* 就变成了新的*u* 节点，它的双亲就变成了新的*pu*，它的祖父节点就变成了新的*gu*，需要继续恢复平衡。如果*gu* 是根节点或者*gu* 节点的颜色改变没有违反规则RB2，那么工作就完成了。

图11-16给出了处理LLb 和LRb 不平衡的旋转操作。在图11-16a 和b 中，*pu* 是pu_L 的根。注意这些旋转与AVL 树的插入操作完成后，用来处理不平衡的 LL（如图11-8所示）和LR（如图11-9所示）旋转之间的相似之处。指针的改变是相同的，例如，在 LL和LR旋转中，由于指针改变了，故需要将*gu* 的颜色由黑色改为红色，而*pu* 的颜色由红色改为黑色。

检查图11-16中旋转之后的节点（或指针）颜色，会发现在所有从根至外部节点的路径上，黑色节点（指针）的数量是不变的，进一步说，相关子树的根（旋转前的 *gu* 和旋转后的*pu*）在旋转后是黑色的，因此，在从根节点至新的 *pu* 的路径上，连续两个红色节点是不存在的，不需要再作平衡。插入后，一次旋转（O(log*n*)次颜色改变）已经足够保持平衡了。

例11-1 考察图11-17a 所示的红-黑树，该图只给出了指针颜色，节点颜色可以从指针颜色以及根节点一般是黑色等常识中推断。为了方便理解，给出了外部节点。实际上，指向外部节点的指针只是简单的空和零，并且外部节点不必专门描述。注意，所有从根至外部节点的路径中都有两个黑色指针。

现在，采用程序11-3的算法，将70插入到红-黑树中。新节点作为80的左孩子插入到树中，由于插入在一棵非空的树中进行，新节点被赋予红色，因此，从父节点（ 80）指向它的指针也是红色。这次插入操作不会导致违反RB2，所以不需要矫正。

接下来，把60插入到图11-17b 的树中，程序11-3的算法将把新节点作为70的左孩子，如图11-17c。新节点为红色，指向它的指针也为红色。新节点是 *u* 节点，其父节点（70）是*pu*，祖父节点（80）是*gu*，由于*pu* 和*u* 都是红色，这里就存在一个不平衡，这个不平衡是*LLr* 类型的不平衡（*pu* 是*gu* 的左孩子，*u* 是*pu* 的左孩子，*gu* 的另一个孩子是红色）。执行图11-15a 和b 的颜色改变，得到图11-17d。现在，*u*，*pu* 和*gu* 节点都上升了两层，节点80是新的*u*节点，根节点是*pu*，*gu* 是零。由于没有*gu* 节点，所以这里不会产生违反RB2类型的不平衡，所有从根到外部节点的路径实际上都有两个黑色指针。

现在将65插入到图11-17d 的树中，结果如图11-17e 所示。新节点是*u* 节点，它的父节点和祖父节点分别是*pu* 和*gu* 节点。这里产生了一个LRb 类型的不平衡，需要执行图11-16c 和d 的旋转，结果如图11-17f 所示。

最后，将62插入到树中，得到图11-7g，产生一个LRr 类型的不平衡，需要进行颜色的改变。图11-17h 给出了所得到的树和新的*u*，*pu* 和*gu* 节点。颜色的改变引起了RLb 类型的不平衡，必需执行RLb 旋转，旋转后的结果如图11-17i 所示。旋转后，不需要再作任何工作，红-黑树的插入操作完成。

11.3.5 红-黑树的删除

对于删除操作，首先使用普通二叉搜索树的删除算法（程序 11-4），然后进行颜色的矫正，如果需要的话，还要作一次单旋转。考察图 11-18a 中的红-黑树，如果用程序11-4删除70，将得到图11-18b 所示的树（如果给出了指针颜色，还需要改变 90的左指针的颜色）。当从a 中删除90时，得到树c（如果使用了指针颜色，那么65的右指针的颜色也需要改变）。从树a 中删除65得到树d（重复强调一次，如果使用了指针颜色，那么指针颜色也要改变）。设*y* 是替代被删

除节点的节点，如图11-18所示。在图11-18b 的情况中，90的左孩子被删除了，它的新的左孩子是外部节点*y*。

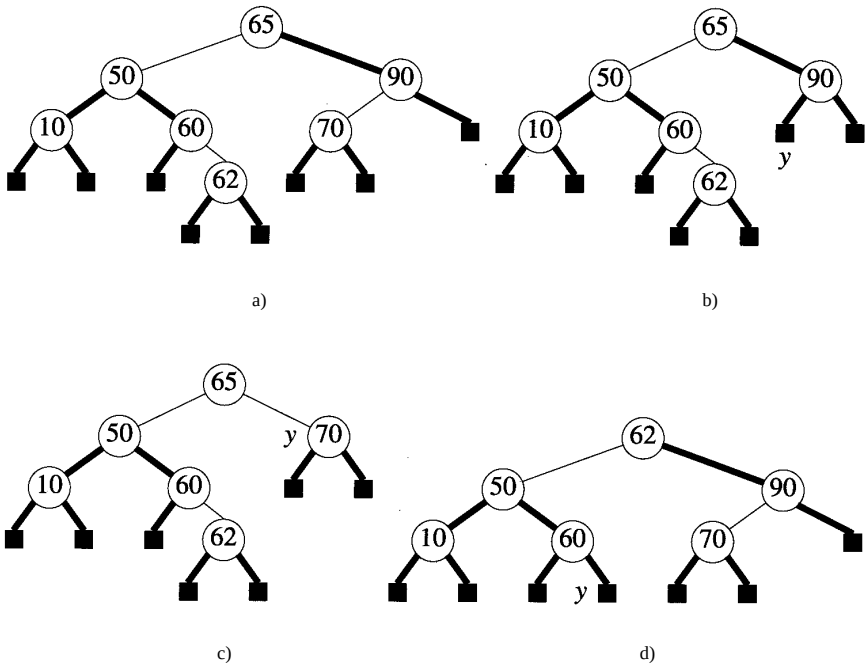


图11-18 红-黑树的删除

a) 初始状态 b) 删除70 c) 删除90 d) 删除65

在树b 的情况中，被删除节点（树a 中的70）是红色的，删除它不会影响从根至外部节点的路径中黑色节点的数量，因此不需要作任何矫正工作。在树c 中，被删除节点（树a 中的90）是黑色的，那么，从根至外部节点路径中黑色节点（和指针）的数量比删除前少了一个。由于*y* 不是新的根，因此违反了RB3，而在树d 中被删除节点是红色的，所以不会出现违反RB3的情况。仅当被删除节点是黑色的且*y* 不是所得树的根时，会出现违反RB3的情况。用程序11-4执行删除操作后不可能出现违反其他红-黑树特征的情况。

当违反RB3的情况发生时，以*y* 为根节点的子树缺少一个黑色节点（或一个黑色指针），因此，在从根至*y* 子树的外部节点的路径上的黑色节点数量比从根至其他外部节点路径上的黑色节点数量少一个，这时树是不平衡的。通过识别*y* 的父节点*py* 和同胞节点*v* 来区分不平衡的性质，当*y* 是*py* 的右孩子时，不平衡是R 类型的，否则不平衡是L 类型的。可以看到，如果*y* 缺少一个黑色节点，那么*v* 就肯定不是一个外部节点。如果*v* 是一个黑色节点，那么不平衡是Lb或Rb 类型的；而当*v* 是红色节点时，不平衡是Lr 或Rr 类型的。

首先考察Rb 类型的不平衡。Lb 型不平衡的处理与之相似。根据*v* 的红色子女的数量，把Rb 型不平衡分为三种子情况：Rb0，Rb1 和Rb2。

当不平衡类型是Rb0 时，需要执行颜色的改变（如图11-19所示）。图11-19给出了*py* 颜色的两种可能改变。如果*py* 是黑色的，那么颜色的改变将导致以*py* 为根的子树缺少一个黑色节点，并且，在图11-19b 中，从根至*v* 的外部节点路径上的黑色节点数量也减少了一个，因此，颜色改变后，无论路径是到*v* 的外部节点还是到*y* 中的外部节点都会缺少一个黑色节点。如果

py 是整棵红-黑树的根，那么就不需要再作其他工作，否则， *py* 就成为新的*y*，*y* 的不平衡需要重新划分，并且在这个新的*y* 点再进行合适的矫正工作。

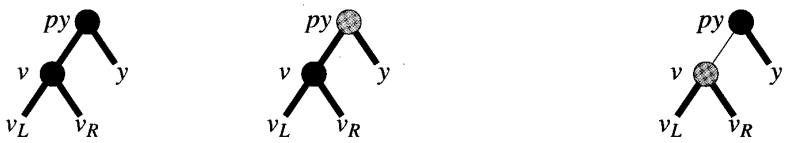


图11-19 红-黑树删除操作的Rb0 颜色改变

a) Rb0 型不平衡 b) Rb0 颜色改变

若改变颜色前*py* 为红色，则从*y* 到外部节点路径上的黑色节点数量增加了一个，而*v* 中的没有改变，整棵树就达到了平衡。

不平衡类型是Rb1 和Rb2 时，需要进行旋转，如图11-20所示。图中的非阴影节点表示那些既可能是红色，也可能是黑色的节点。这种节点的颜色在旋转后不会发生变化，因此，图11-20b 中所示子树的根在旋转前和旋转后，颜色保持不变。b 中*v* 的颜色与a 中*py* 的颜色是一样的。你应当能够发现在旋转后，*y* 中从根至外部节点路径上的黑色节点（指针）数量增加了一个，而从根至其他外部节点路径上的黑色节点的数量没有变化，旋转使树恢复了平衡。

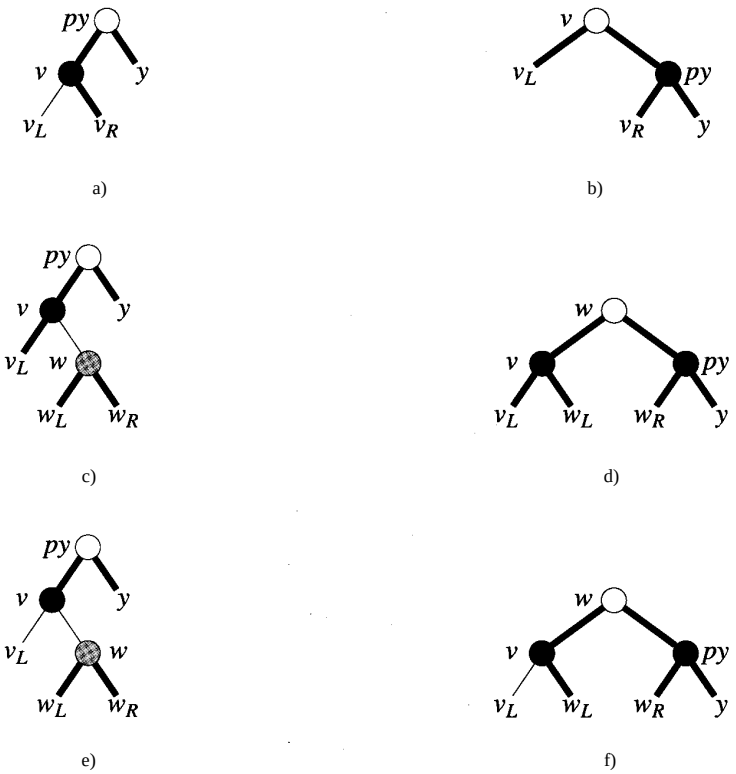


图11-20 红-黑树删除操作中Rb1 和Rb2 型不平衡的旋转

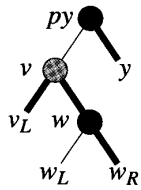
a) Rb(i) 型不平衡 b) Rb(i) 旋转之后 c) Rb1(ii) 型不平衡 d) Rb1(ii) 旋转之后 e) Rb2 不平衡 f) Rb2 旋转之后

接下来考察Rr 类型的不平衡，*Lr* 型不平衡与它是对称的。由于 *y* 中缺少一个黑色节点并且*v* 是红色的，v_L 和v_R 中至少有一个黑色节点不是外部节点，因此，*v* 的孩子都是内部节点。根据*v* 的右孩子中红色孩子（0，1或2）的数量，可以进一步把Rr 类型的不平衡划分为三种情况，这三种情况都可以用旋转来处理。如图 11-21和11-22所示，可以验证其中的旋转使整棵树恢复了平衡。

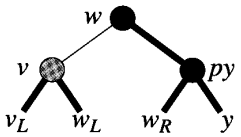
a) b)

图11-21 红-黑树删除操作的Rr0 旋转

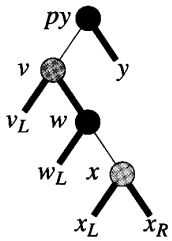
a) Rr0 型不平衡 b) Rr0 旋转



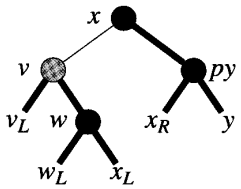
a)



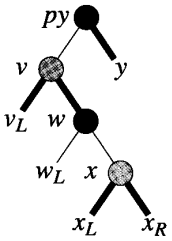
b)



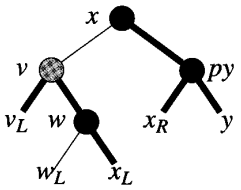
c)



d)



e)



f)

图11-22 红-黑树删除操作的Rr1 和Rr2 旋转

a) Rrl(i) 型不平衡 b) Rrl(i) 旋转之后 c) Rrl(ii) 型不平衡 d) Rrl(ii) 旋转之后 e) Rr2 型不平衡 f) Rr2 旋转之后

例11-2 如果从图11-17i 的红-黑树中删除90，就得到图11-23所示的树。由于被删除节点不是

根节点且是黑色的，因此产生了Rb0 型不平衡。执行颜色的改变后得到了图11-23b 中的树。由于py 原来是红色的，因此改变颜色后使树重新恢复了平衡。

如果现在从树b 中将80删除，就得到了树c。由于删除的是红色节点，删除后树仍然是平衡的。从树c 中将70删除，得到树d，这次删除的是非根黑色节点，树的平衡被破坏了，不平衡类型是Rr1（ii）类型（*v* 的右孩子*w* 的右孩子指针是红色的）。执行Rr1（ii）型旋转后得到树e，树的平衡得到了恢复。

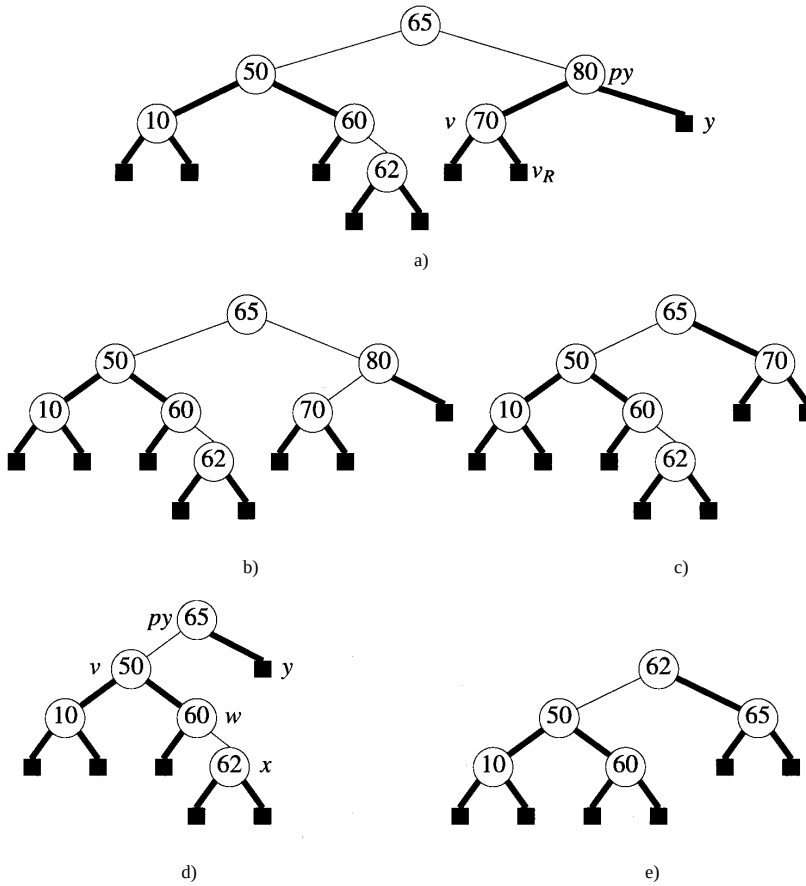


图11-23 红-黑树的删除

a) 删除90 b) Rb0 颜色改变之后 c) 删除80 d) 删除70 e) Rrl(ii) 旋转之后

11.3.6 实现细节的考虑及复杂性分析

在插入或删除操作后，为了重新恢复红-黑树的平衡而采取的矫正方法需要我们回到从根节点至插入或删除节点的路径上来。如果除数据、左孩子、右孩子和颜色域外每个节点还有一个双亲域，那么这种回溯很容易实现。还有一种方法是将从根节点至插入 /删除节点路径上所遇到的每个节点的指针保存到一个堆栈内，通过从堆栈中删除这些指针，就可以返回到根节点。对于一个n元素的红-黑树，增加双亲域使得对空间的需求增加了 $\Theta(n)$，而使用堆栈则使空间的需求增加了 $\Theta(\log n)$。虽然堆栈方法在空间上更节省，但双亲指针方法运行得更快一些。

插入或删除后，由于颜色的改变是沿着向根节点的方向进行的，故需要时间 O(log*n*)。另一方面，旋转能够保证树的重新平衡。每次插入 /删除操作最多需要一次旋转。每次颜色改变或旋转操作需要的时间是Θ(l)，因此插入/删除操作需要的总时间是O(log*n*)。

练习

25. 画出对应于图11-15中LLr和LRr类型的RRr和RLr类型的颜色改变。
26. 画出对应于图11-16中LLb和LRb类型的RRb和RLb类型的颜色改变。
27. 画出对应于图11-19中Rb0类型的Lb0类型的颜色改变。
28. 画出Lb1和Lb2类型的旋转示意图，相对应的Rb1和Lb2类型的旋转如图11-20所示。
29. 画出Lr0、Lr1和Lr2类型的旋转示意图，相对应的Rr0、Rr1和Rr2类型的旋转如图11-21和图11-22所示。

*30. 设计一个C++类RedBlackTree，它包括二叉搜索树的函数 Search、Insert、Delete和Ascend。编写所有函数并检验其正确性。证明前三种操作的复杂性是 O(log*n*)，最后一种操作的复杂性是Θ(l)。Insert 和Delete 函数的实现必须采用本节所讨论的方法。

11.4 B-树

11.4.1 索引顺序访问方法

当字典足够小，可以驻留在内存中时， AVL树和红-黑树都能够保证获得很好的性能。对于较大的字典（外部字典或文件），它们必须存储在磁盘上，可以通过采用度数高的搜索树来改善字典操作的性能。在研究高度数搜索树之前，先看一下用于外部字典的索引顺序访问方法（indexed sequential access method, ISAM），这种方法提供了很好的顺序和随机访问。

在ISAM方法中，可用的磁盘空间被划分为很多块，块是磁盘空间的最小单位，被用来作为输入和输出。块一般具有与磁道同样的长度，且可以用单个搜索和很小的延迟进行输入输出。字典元素以升序存储在块中。

在顺序访问时，依次输入各个块，在每个块中按升序搜索元素。如果每个块包含 *m*个元素，则搜索每个元素所需要的磁盘访问次数为1/*m*。

要支持随机访问，索引是不可缺少的。索引中包括每个块中的最大关键值。由于索引中所包含的关键值数量仅与块数相同，并且每个块一般都能贮存很多元素（ *m*值通常较大），因此索引足以驻留在内存中。对关键值为 *k*的元素作一次随机访问，首先只要寻找包含相应元素的块的索引，然后将相应的块从磁盘中取出并在其中寻找需要的元素。这样，执行一次随机访问只需要一次磁盘访问就足够了。

这种技术可以扩充到更大的字典，这种字典能跨越几个磁盘。现在，元素按升序被分配到各个磁盘以及每个磁盘的不同块中。每个磁盘都有一个块索引，其中保留了该磁盘每个块中的最大关键值。另外，有一个磁盘索引保存每个磁盘中的最大关键值，这个索引一般驻留在内存中。

在上述情况下如果要执行一个随机访问，首先需在磁盘索引中进行搜索以判断所需要的记录可能存储在哪个磁盘上，找到磁盘后，取出相应磁盘的块索引，并在其中搜索所需要的的块，块被取出后，在其内部搜索所需要的记录。

由于ISAM方法本质上是一种公式化描述方法，当执行插入和删除时它就会陷入困境。通

过在每个块中留一些空间可以部分减轻这种困难，这样在执行少量的插入时可以不必在块之间移动元素。类似地，在删除操作后可把空间保留下来，以避免在块之间进行代价昂贵的元素移动。

11.4.2 m 叉搜索树

定义 [m 叉搜索树] m 叉搜索树（m-way search tree）可以是一棵空树，如果非空，它必须满足以下特征：

1) 在相应的扩充搜索树中（用外部节点替换零指针），每个内部节点最多可以有 m 个子女及1～m-1个元素（外部节点不含元素和子女）。

2) 每个含 p 个元素的节点，有 p+1个子女。

3) 考察含 p 个元素的任意节点。设 k_1, ..., k_p 是这些元素的关键值。这些元素顺序排列，即有 $k_1 < k_2 < ... < k_p$。设 c_0, c_1, ..., c_p 是节点的 p+1个孩子。以 c_0 为根的子树中的元素关键值小于 k_1，而以 c_p 为根的子树中的元素关键值大于 k_p，并且以 c_i 为根的子树中的元素关键值会大于 k_i 而小于 k_{i+1}，其中1≤i≤p。

虽然在定义 m 叉搜索树时把外部节点包括进来很有用，但在实际实现中不需要专门描述外部节点，只需用零指针或空指针来表示外部节点。

图11-24给出了一棵七叉搜索树，图中的黑色方块代表外部节点，所有其他节点都是内部节点。根包含两个元素（关键值是10和80）和三个子女。中间的子女有6个元素和7个孩子，其中6个孩子是外部节点。

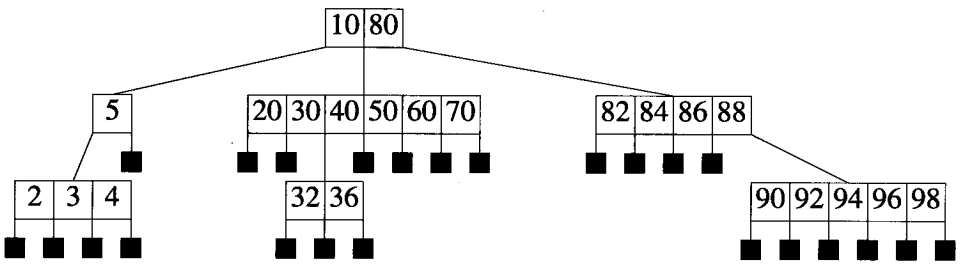


图11-24 七叉树

1. 在m叉搜索树中进行搜索

要从图11-24中的七叉搜索树中将关键值为31的元素搜索出来，可先从根节点开始。31位于10和80之间，沿中间的指针往下找。（由定义知，第一棵子树中所有元素的关键值＜ 10，而第三棵子树中所有元素的关键值＞ 80）。中间子树的根被搜索。由于 k_2＜31＜k_3，故移到该节点的第三棵子树中。此时可以发现 31＜k_1，所以移动到第一棵子树中。这次移动使我们从树中“掉”了下来，即到达了外部节点。因此可以得出结论，搜索树中不包含关键值为 31的元素。

2. 向m叉搜索树中插入元素

如果希望将关键值为31的元素插入到树中，则先要按以上步骤搜索 31，在节点[32，36]处搜索失败。由于该节点可以容纳六个元素（七叉搜索树的每个节点最多可以容纳六个元素），新元素可以插到该节点的第一个位置。

假定要将65插到树中，则先对65进行搜索，在节点[20,30,40,50,60,70]的第六棵子树中搜索失败，由于此节点不能再容纳额外的元素，所以必须产生一个新节点。新元素放入新节点中，新节点成为节点[20,30,40,50,60,70] 的第6个孩子。

3. 从m叉搜索树中删除元素

从图11-24中的搜索树中删除关键值为 20的元素，首先要进行搜索，该元素是根节点中间孩子的第一个元素。由于$k_1=20$并且$c_0=c_1=0$，故可以简单地将它从节点中删除，根节点新的中间子女变成[30,40,50,60,70]。类似地，如要删除关键值为84的元素，首先对它定位，它是根节点第三个子女的第二个元素，由于$c_1=c_2=0$，故可以从节点中直接删除该元素，原节点变为[82，86，88]。

当删除关键值为5的元素时，需要多作一些工作。由于被删除元素是节点中的第一个元素并且它的相邻子女（这里是 c_0 和c_1）中至少有1个是非零的，因此，需要从非空相邻子树中找一个元素来替换被删除元素。从左子树（c_0）中，可以将具有最大关键值的元素移上来(这里是关键值为4的元素）

从图11-24中的根节点中删除关键值为10的元素时，既可以用c_0 中的最大元素，也可以用c_1 中的最小元素进行替换。如果用 c_0 中的最大元素进行替换，那么元素 5将被移上来，而且要对它在原节点中的位置作一次替换，元素4就被移到元素5原来的位置。

4. m叉搜索树的高度

一棵高度为h 的m 叉搜索树最少可以有h 个元素（每层一个节点，每个节点含一个元素），最多可以有m^h-1个元素。此上限取自这种情况：从 1到$h-1$层的每个节点都含有m 个孩子并且第h层的节点没有孩子。这样一棵树共有 $\sum_{i=0}^{h-1} m^i=(m^h-1)/(m-1)$个节点。由于每个节点可以有$m-1$元素，所以元素的总数为$m^h-1$。

由于高度为h 的m 叉搜索树中元素的个数在h 到m^h-1 之间，所以一棵n 元素的m 叉搜索树的高度在$\log_m(n+1)$到n 之间。

例如，一棵高度为5的200叉搜索树能够容纳$32*10^{10}-1$个元素，但也可以只容纳五个元素。同样，一棵含有$32*10^{10}-1$个元素的200叉搜索树的高度可以是 5，也可以是$32*10^{10}-1$。当搜索树存储在磁盘上时，搜索、插入和删除时间取决于磁盘的访问次数（假设每个节点不大于一个磁盘块）。由于搜索、插入和删除操作需要的磁盘访问次数是 O(h)，其中h 是树的高度，因此，必须确保高度值接近于$\log_m(n+1)$。这种保证可由m 叉平衡搜索树提供。

11.4.3 m 序B-树

定义 [m序B-树] m序B-树（B-Tree of order m）是一棵m 叉搜索树，如果B-树非空，那么相应的扩充树满足下列特征：

1) 根节点至少有2个孩子。

2) 除了根节点以外，所有内部节点至少有「m/2ù个孩子。

3)所有外部节点位于同一层上。

图11-24中的七叉搜索树不是一棵七序 B-树，因为它的外部节点不在同一层上。即使它所有的外部节点在同一层上，它也不会是一棵七序 B-树，因为它的非根内部节点 [5]有2个孩子，内部节点[32，36]有3个孩子，而七序B-树的非根节点必须至少有「7/2ù=4个孩子。图11-25是一棵七序B-树，所有外部节点均位于第三层，根节点有 3个孩子，且剩下的所有内部节点至少有4个孩子，此外，它也是一个七叉搜索树。

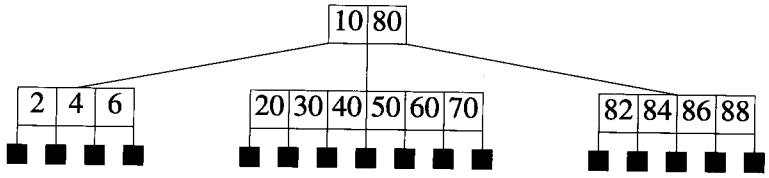


图11-25 七序B-树

在一棵二序B-树中，没有哪个内部节点会有2个以上的孩子。由于在二序B-树中每个内部节点必须至少有2个子女，所以一棵二序B-树的所有内部节点都恰好有2个孩子。这一点以及要求所有外部节点必须在同一层上暗示了二序B-树是一棵满二叉树。正因如此，这些树存在的条件是元素的个数为2^h-1，其中h是一个整数。

在一棵三序B-树中，内部节点既可以有2个也可以有3个孩子，因此也把三序B树称作2-3树。由于四序B-树的内部节点必须有2个、3个或4个孩子，这种树也叫作2-3-4树（或简称2,4树）。图11-26中给出了一棵2-3树的例子，虽然该树中没有含4个孩子的内部节点，但只要把关键值为14和16的元素加入到20的左孩子中，就可以得到一棵至少有一个含4个孩子的节点的2-3-4树了。

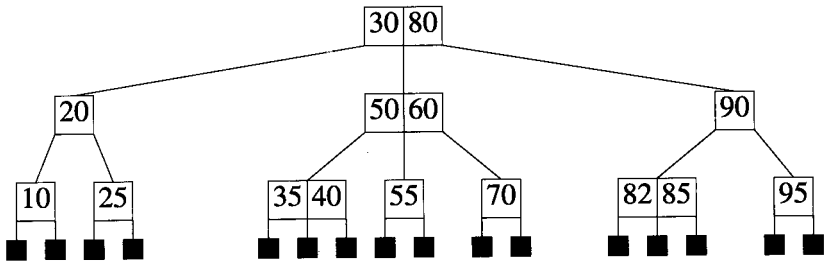


图11-26 2-3树或三序B-树

11.4.4 B-树的高度

定理11-3 设T是一棵高度为h的m序B-树，$d=[m/2]$且n是T中的元素个数，则

1) $2d^{h-1}-1\leq n\leq m^h-1$。

2) $\log_m(n+1)\leq h\leq \log_d(\frac{n+1}{2})+1$。

证明 n的上限源于T是一棵m叉搜索树，这在前面已经证明过了。对于下限，注意相应的扩充B-树的外部节点都在$h+1$层，而1，2，3，4，...，$h+1$层的节点最小数目是1，2，$2d$，$2d^2$，...，$2d^{h-1}$，因此B-树中外部节点的最小数是$2d^{h-1}$。由于外部节点的数量比元素的个数多1，因此

$$n\geq 2d^{h-1}-1$$

从1)直接可以得到2)

由定理11-3可知，一棵高度为3的200序B-树中至少有19 999个元素，而高度为5的200序B-树中至少有$2*10^8-1$个元素。因此，如果使用200序或更高序B-树，即使元素数量再多，树的高度也可以很小。实际上，B-树的序取决于磁盘块的大小和单个元素的大小。节点小于磁盘块的大小并无好处，这是因为每次磁盘访问只读或写一个块。节点大于磁盘块的大小会带

来多重磁盘访问，每次磁盘访问都伴随一次搜索和时间延迟，因此节点大于磁盘块的大小也是不可取的。

虽然在实际应用中，B-树的序很大，但在我们的例子中所用的m值很小，因为一棵两层的m序B-树至少有$2d-1$个元素，当m的值为200，d为100时，一棵两层的200序B-树至少有199个元素，处理一棵如此多元素的树非常麻烦。

11.4.5 B-树的搜索

B-树的搜索算法与m叉搜索树的搜索算法相同。在搜索过程中，从根至外部节点路径上的所有内部节点都有可能被搜索到，因此，磁盘访问次数最多是h（h是B-树的高度）。

11.4.6 B-树的插入

将一个元素插入B-树中时，首先要检查具有相同关键值的元素是否存在，如果找到了这个元素，那么插入失败，因为不允许重复值存在。当搜索不成功时，便可以将元素插入到搜索路径中所遇到的最后一个内部节点处。例如，当将关键值为3的元素插入到图11-25的B-树中时，首先检查根节点及其左孩子，在左孩子的第二个外部节点处搜索失败。由于左孩子可以容纳六个元素而目前只有三个，因此新元素可以直接插入到这个节点中。结果如图 11-27a 所示。对根节点及左孩子有两次磁盘读操作，而对左孩子另有一次磁盘写操作。

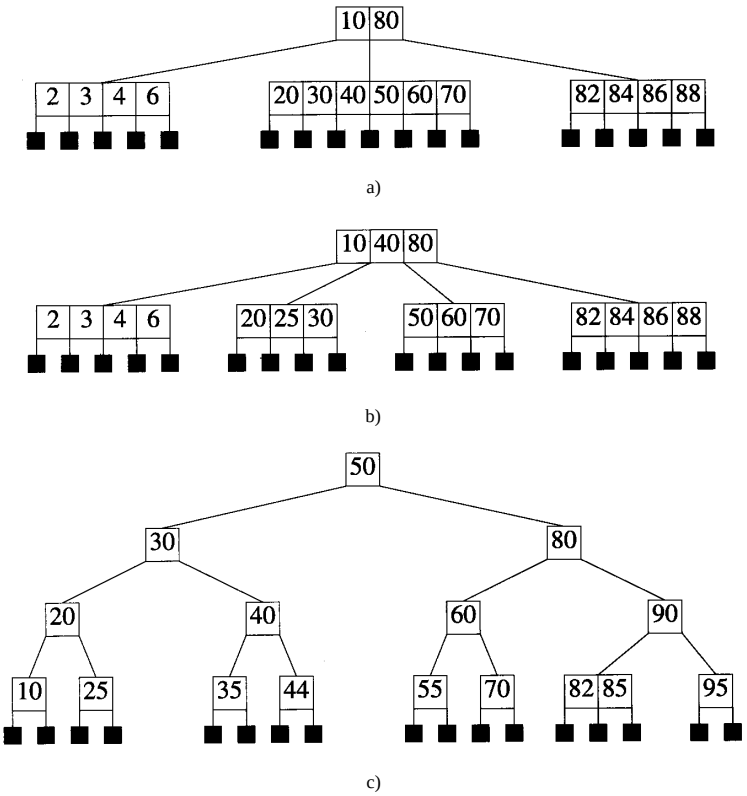


图11-27 B-树的插入

a) 向图11-25中插入30 b) 向a) 中插入25 c) 向图11-26中插入44

下面，将关键值为 25的元素插入到图 11-27a 的B - 树中，这个元素将被插入到节点[20,30,40,50,60,70]中，但该节点已经饱和了。当新元素需要插入到饱和节点中时，饱和节点需要被分开。设P是饱和节点，现将带有空指针的新元素 e 插入到P中，得到一个有 m 个元素和m+1个孩子的溢出节点。用下面的序列表示溢出节点：

m，c_0，(e_1, c_1)，...，(e_m, c_m)

其中e_i 是元素，c_i 是孩子指针。从e_d 处分开此节点，其中 d=「m /2ù。左边的元素保留在 P中，右边的元素移到新节点Q中，(e_d, Q) 被插入到P 的父节点中。新的P 和Q 的格式为：

P：d-1，c_0，(e_1, c_1)，...，(e_{d-1}, c_{d-1})

Q：m-d，c_d，(e_{d+1}, c_{d+1})，...，(e_m, c_m)

注意P和Q的孩子数量至少是d。

在本例中，溢出节点是

7，0，(20,0)，(25,0)，(30,0)，(40,0)，(50,0)，(60,0)，(70,0)

且d=4。从e_4 处分开后的两个节点是：

P：3，0，(20,0)，(25,0)，(30,0)

Q：3，0，(50,0)，(60,0)，(70,0)

当把(40,Q)插入到P的父节点中时，得到图11-27b 所示的B - 树。

将25插入到图11-27a，需要从磁盘中得到根节点及其中间孩子，然后将分开的两个节点和修改后的根节点写回到磁盘中，磁盘访问次数一共是 5次。

来看最后一个例子。考察将关键值为 44的元素插入到图11-26的2-3树中，此元素将插入到节点[35,40]中，由于该节点是饱和节点，故得到溢出节点：

3，0，(35,0)，(40,0)，(44,0)

从e_d=e_2处分开得到2个节点：

P：1，0，(35,0)

Q：1，0，(44,0)

当把(40,Q)插入到P的父节点A中时，发现该节点也是饱和的，插入后，又得到溢出节点：

A：3，P，(40,Q)，(50,C)，(60,D)

其中C 和D 是指向节点[55] 和[70] 的指针。溢出节点A 被分开，产生节点B。新节点A 和B 如下：

A：1，P，(40,Q)

B：1，C，(60,D)

现在需要将(50,B)插入到根节点中，在此之前根节点的结构是；

R：2，S，(30,A)，(80,T)

其中S和T是分别指向根节点第一和第三棵子树的指针。插入完成后，得到溢出节点：

R：3，S，(30,A)，(50,B)，(80,T)

将此节点从关键值为50的元素处分开，并产生一个新节点R和一个新节点U，如下所示：

R：1，S，(30,A)

U：1，B，(80,T)

(50,U)一般应插入到R的父节点中，但是R 没有父节点，因此，产生一个新的根节点如下：

1，R，(50,U)

得到的2-3树如图11-27c 所示。

读取节点[30,80]、[50,60]和[35,40]时执行了3次磁盘访问。对每次节点分裂，将修改的节

点和新产生的节点写回磁盘需执行2次磁盘访问，由于有3个节点被分开，因此需执行6次写操作。最后产生一个新的根节点并写回磁盘，又需占用1次额外的磁盘访问，因此磁盘访问的总次数为10。

当插入操作引起了 s 个节点的分裂时，磁盘访问的次数为 h (读取搜索路径上的节点)+2s (回写两个分裂出的新节点)+1（回写新的根节点或插入后没有导致分裂的节点）。因此，所需要的磁盘访问次数是h+2s+1，最多可达到3h+1。

11.4.7 B-树的删除

删除分为两种情况： 1) 被删除元素位于其孩子均为外部节点的节点中（即元素在树叶中）；2) 被删除元素在非树叶节点中。既可以用左相邻子树中的最大元素，也可以用右相邻子树中的最小元素来替换被删除元素，这样2) 就转化为1)。替换元素必须确保在树叶中。

考察从图11-27a 的B-树中删除关键值为80的元素。由于元素不在树叶中，需要找一个合适的替换。关键值为70（左相邻子树中的最大元素）和82（右相邻子树中的最小元素）成为候选对象。当选用70时，还存在将此元素从树叶中删除的问题。

如果要从图11-27c 的2-3树中将关键值为80的元素删除，既可以用70，也可以用82来替换此元素。如果选择82，那么还有从树叶[82,85]中删除82的问题。

由于2) 转化为1) 非常容易，故只讨论1)。从一个包含多于最少数目元素（如果树叶同时是根节点，那么最少元素数目是 1，如果不是根节点，则为「m/2ù-1）的树叶中删除一个元素，只需要将修改后的节点写回。（如果该节点是根节点，则B-树就成为空树）。从图11-27a 的B-树中删除50，需将修改后的节点[20,30,40,60,70]写回磁盘。而从图11-27c 的2-3树中删除85，需将节点[82]写回磁盘。两种情况下，在沿搜索路径到树叶的过程中都需要 h 次磁盘访问，将包含被删除元素的修改后的树叶写回磁盘还需要一次额外的磁盘访问。

当被删除元素在一个非根节点中且该节点中的元素数量为最小值时，可用其最相邻的左或右兄弟中的元素来替换它。注意到除了根节点以外的每个节点都会有一个最相邻的左兄弟或一个最相邻的右兄弟，或二者都有。例如，假设希望从图 11-27b 的B-树中删除元素25，此次删除留下了一个节点[20,30]，它恰好有两个元素，但是，由于这个节点是七序 B-树的一个非根节点，它必须至少有三个元素，而它的最相邻的左兄弟 [2,3,4,6]中多一个元素，因此可把该节点中的最大元素移到其父节点中，所牵涉的元素（关键值为 10）被向下移动，从而产生图11-28a 所示的B-树。磁盘访问次数是2（从根到包含25的树叶）+1（读取该树叶的最相邻左兄弟）+3（写回修改后的树叶、兄弟和父节点）=6。

假如没有检查[20,30]的最相邻左兄弟，而是检查它的最相邻右兄弟 [50,60,70]。由于此节点只含有三个元素，所以不能从中删除元素。（若此节点有四个或更多的元素，则可以把其中最小元素移到它的父节点中，并且将这两个相邻兄弟之间的父节点中的元素移动到缺少一个元素的树叶中）。现在，可以检查[20,30]的最相邻左兄弟。执行检查需要一次额外的磁盘访问，并且不能肯定在这个最相邻兄弟中有这样一个额外元素。为保持低次数的磁盘访问，只检查缺少一个元素的最相邻兄弟之中的一个。

当最相邻兄弟中不含额外的元素时，将两个兄弟与父节点中介于两个兄弟之间的元素合并成一个节点。由于两兄弟分别有d-2和d-1个元素，合并后节点共有2d-2个元素。当m 是奇数时，2d-2等于m-1；而当m 是偶数时，2d-2等于m-2。节点中有足够的空间来容纳这么多元素。

在本例中，两兄弟 [20,30]和[50,60,70]以及关键值为 40的元素被合并成一个节点[20,30,40,50,60,70]。得到的B-树如图11-27a 所示。删除过程中，到达节点[20,25,30]需要2次磁

盘访问，读取最相邻右兄弟需要1次，将两个修改后的节点写回还需要2次，因此磁盘访问次数一共是5次。

由于合并减少了父节点中的元素个数，父节点有可能会缺少一个元素，如果这样，需要检查父节点的最相邻兄弟，要么从中取一个元素，要么与它合并。如果从最相邻右（左）兄弟中取一个元素，那么此兄弟节点的最左（最右）子树也将被读取到。如果进行合并，那么祖父节点也可能会缺少一个元素，此过程又需要在祖父节点中重复应用。最坏情况下，这种过程会一直回溯到根节点。当根节点缺少一个元素时，它变成空节点，将被抛弃，树的高度减1。

假设需要从图11-26的2-3树中删除10。删除留下了一个不含元素的树叶节点，它的最相邻右兄弟没有额外的元素。因此，两个兄弟树叶及父节点(10)中的元素被合并到一个节点中，新的树结构如图11-28b所示。现在第二层中有一个节点缺少一个元素，它的最相邻右兄弟中有一个额外的元素，最左边元素（关键值为50）移到父节点中，并将关键值为30的元素移下来，得到的2-3树如图11-28c所示。注意到[50,60]的左子树也被移动。到达包含被删除元素的树叶时执行了3次读访问，到达第二和三层的最相邻右兄弟节点时执行了2次读访问，将第一，二和三层的4个修改后的节点写回磁盘需要4次写访问。因此总的磁盘访问次数是9次。

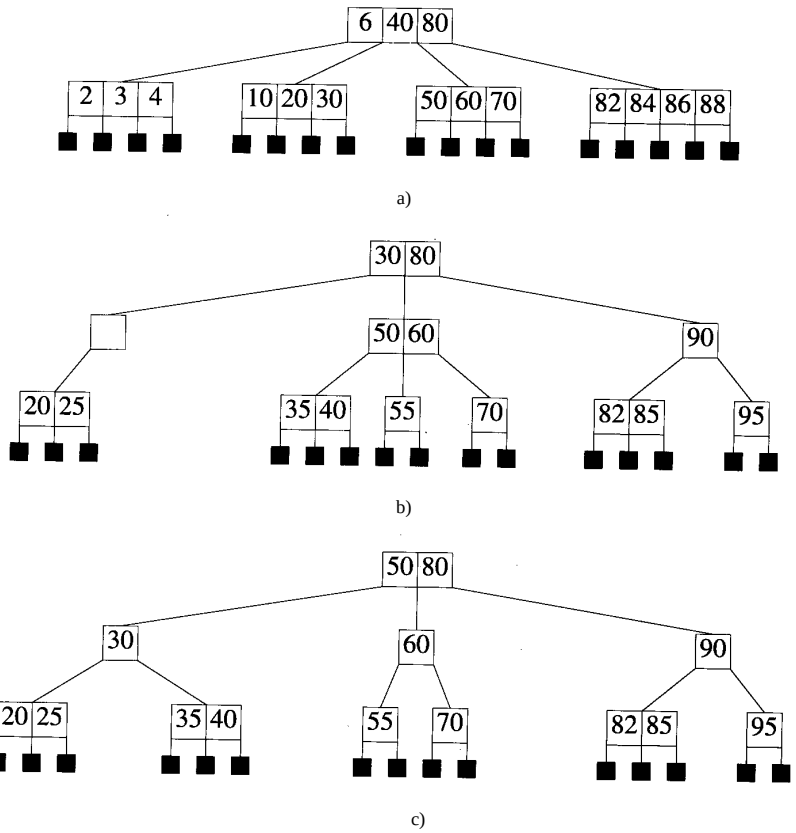
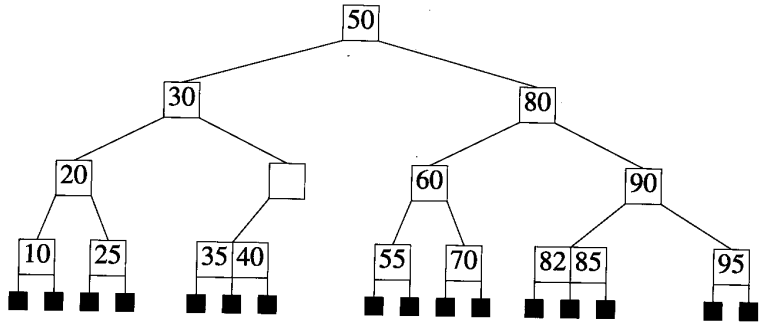


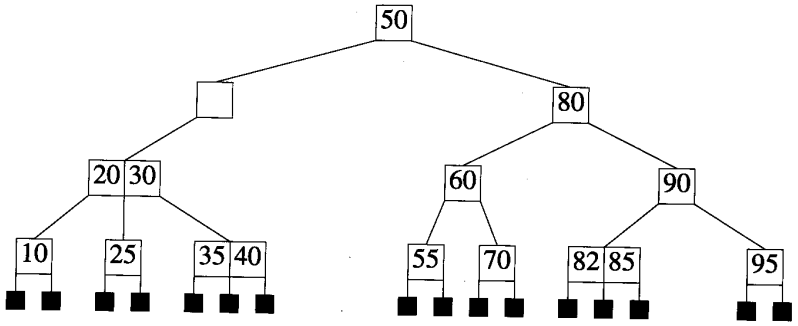
图11-28 B-树的删除

a) 从图11-27b中删除25 b) 树叶层合并之后 c) 从图11-26中删除10

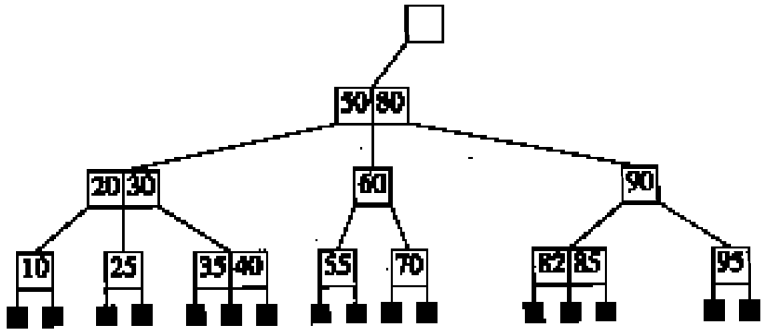
最后一个例子，考察删除图11-27c的2-3树中的44。当将44从树叶中删除时，树叶中将缺少一个元素，它的最相邻左兄弟没有额外的元素，因此两兄弟与父节点中的元素被合并，得到



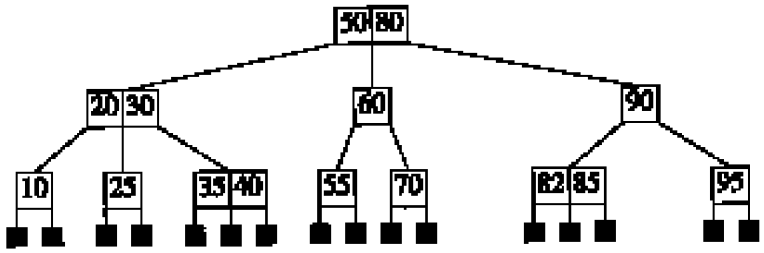
a)



b)



c)



d)

图11-29 从图11-27c 中的2-3树中删除44

a) 树叶层合并之后 b) 第3层合并之后 c) 第2层合并之后 d) 删除根节点之后

图11-29a 所示的树。现在第三层有一个节点，缺少一个元素，它的最相邻左兄弟中不含有额外的元素，因此两兄弟与父节点的元素合并，得到的树如图 11-29b 所示，现在第二层中的一个节点缺少一个元素，它的最相邻右兄弟不含有额外的元素，执行合并后得到图 11-29c 中的树。此时根节点缺少一个元素，由于只有根节点为空时它才会缺少一个元素，因此将根节点抛弃。最后的2-3树如图11-29d 所示，根节点被抛弃后，树的高度减少了一层。

找到含有被删除元素的树叶需要4次磁盘访问，对最相邻兄弟有3次访问，另有3次写访问，因此总的访问次数是10次。

对于高度为h 的B-树的删除操作，最坏情况出现在当合并发生在 h,h-1,...,3,2层时，需要从最相邻兄弟中获取一个元素。最坏情况下磁盘访问次数是 $3h$＝（找到包含被删除元素需要h 次读访问）+（获取第2至h 层的最相邻兄弟需要h-1次读访问）+（在第3至h层的合并需要h-2次写访问）+（对修改过的根节点和第2层的两个节点进行3次写访问）。

11.4.8 节点结构

在以上的讨论中假定节点的结构如下：

s，c_0，(e_1, c_1)，(e_2, c_2)，...，(e_s, c_s)

其中s 是节点中元素的个数，e_i 是按关键值升序排列的元素，c_i 是孩子指针。当元素的大小比关键值的大小更大时，可采用以下的节点结构：

s，c_0，(k_1, c_1, p_1)，(k_2, c_2, p_2)，...，(k_s, c_s, p_s)

其中k_i 是元素的关键值，p_i 是相应元素在磁盘中的位置。利用这种结构，可以得到更高序的B-树，这种更高序的B-树称作B'-树。如果非树叶节点中不含有 p_i 指针并且在树叶中用 p_i 指针替换了空孩子指针，就可能产生更高序的B-树。

另一种可能是用平衡的二叉搜索树描述每个节点的内容。利用平衡的二叉搜索树可以减小B-树的序值，因为对于每个元素都需要一个左-右孩子指针以及一个平衡因子或颜色域。但是将元素插入到节点中或从节点中删除一个元素所花费的 CPU时间减少了。这种方法能否导致性能的提高取决于具体的应用。在某些情况中，较小的 m 可能能增加B-树的高度，导致每一个搜索／插入／删除操作需要更多的磁盘访问。

练习

31. 如果每个节点占用2个磁盘块并且需要2次磁盘访问才能搜索出来，那么在一棵2m序B-树的搜索过程中需要的最大磁盘访问次数是多少？将该次数与节点大小占用 1个磁盘块的m 序B-树的磁盘访问次数相比较，并论述节点大小大于磁盘块大小时的优点。

32. 删除一个m 序B-树中非树叶节点的元素需要的最大磁盘访问次数是多少？

33. 假设按如下方法修改从B-树中删除元素的方式：如果一个节点既有最相邻左兄弟也有最相邻右兄弟，那么在合并前对两个兄弟都要作检查。从一棵高度为 h 的B-树中删除元素时需要的最大磁盘访问次数是多少？

*34. 一棵2-3-4树可以描述为一棵二叉树，其中每一个节点要么是红色的，要么是黑色的。在2-3-4树中只含有一个元素的节点被描述为黑色节点；含有 2个元素的节点被描述为有1个红色孩子的黑色节点（红色孩子既可以是黑色节点的左孩子，也可以是右孩子）；含有3个元素的节点被描述为有2个红色孩子的黑色节点。

1) 画一棵2-3-4树，其中至少包含一个2元素节点和一个3元素节点，用上述方法将其画成

带颜色节点的二叉树。

2) 验证该二叉树是一棵红-黑树。

3) 证明当用上述方法把任意二叉树描述成一棵带颜色的二叉树时，所得到的二叉树是一棵红-黑树。

4) 证明可以用相反的映射方式把任意一棵红-黑树描述成一棵2-3-4树。

5) 验证下列事实：对于红-黑树的插入操作，11.4.4节给出的改变颜色和旋转的方法，也可以从采用4) 中映射模式的B-树插入方法中得到。

6) 对从红-黑树中删除元素的情况重做5)。

*35. 设计一个类TwoThree，实现一棵2-3树。类中应包括搜索、插入和删除操作。测试代码是否正确。

*36. 设计一个类TwoFour，实现一棵2-3-4树。类中应包括搜索、插入和删除操作。测试代码是否正确。

11.5 应用

11.5.1 直方图

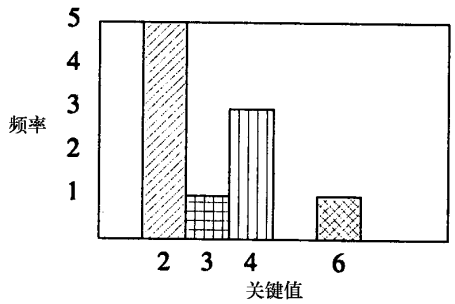
在直方图问题中，从一个具有 n 个关键值的集合开始，要求输出不同关键值的列表以及每个关键值在集合中出现的次数（频率）。图11-30给出了一个含有10个关键值的例子。图11-30a给出了直方图的输入，直方图的表格形式如图 11-30b 所示，直方图的图形形式如图 11-30c 所示。直方图一般用来确定数据的分布，例如，考试的分数、图象中的灰色比例、在生产商注册的汽车和居住在洛杉矶的人所获得的最高学位都可以用直方图来表示。当关键值为从 0到r 范围内的整数且r 的值足够小时，可以在线性时间内，用一个相当简单的过程（见程序 11-6）产生直方图。在该过程中用数组元素$h[i]$代表关键值i 的频率。可以使用程序11-6把其他关键值类型映射到这个范围中。例如，如果关键值是小写字母，则可以用映射 [a,b,c,...,z]=[0,1,...,25]。

$n = 10$; 关键值 $= [2, 4, 2, 2, 3, 4, 2, 6, 4, 2]$

a)

关键值	频率
2	5
3	1
4	3
6	1

b)



c)

图11-30 直方图举例

a) 输入 b) 表格形直方图 c) 直方图的图形形式

程序11-6 简单的直方图程序

```
void main(void)
```

```
{// 非负整数值的直方图
  int n,  // 元素数
      r;  // 整数值位于 0 到 r之间
  cout << "Enter number of elements and range" << endl;
  cin >> n >> r;

  // 创建数组 h
  int *h;
  try {h = new int[r+1];}
  catch (NoMem)
    {cout << "range is too large" << endl;
     exit(1);}

  // 将数组h初始化为0
  for (int i = 0; i <= r; i++)
    h[i] = 0;

  // 输入数据并计算直方图
  for (i = 1; i <= n; i++) {
    int key;  // input value
    cout << "Enter element " << i << endl;
    cin >> key;
    h[key]++;
    }

  // 输出直方图
  cout << "Distinct elements and frequencies are" << endl;
  for (i = 0; i <= r; i++)
    if (h[i]) cout << i << "   " << h[i] << endl;
}
```

当关键值类型不是整型（如关键值类型是实数）或关键值范围变化很大时，程序 11-6不可用。假设要确定一个文本中不同词语出现的频率，与文本中实际出现的数量相比，可能的不同词语的数量是非常大的，在这种情况下，可以将关键值排序，然后用一个简单的自左至右的扫描方法确定每一个不同关键值的数量。搜索可在 O($n$log$n$)时间内完成（例如，用HeapSort（见程序9-12）堆排序），从左至右扫描需要 $\Theta(n)$；因此总的复杂性是O($n$log$n$)。当与n 相比，不同关键值的数量m 非常小时，可以进一步改进这种方法。通过使用 AVL和红-黑树之类的二叉搜索树，可以在O($n$log$m$)时间内解决直方图问题。另外，采用平衡的搜索树只需把不同的关键值存储在内存中，因此，当n 的值非常大，没有足够的内存来容纳所有的关键值（当然，对于不同关键值来说，内存是足够的）时，这种方法是适用的。

上述方法使用的是二叉搜索树，其平均复杂性为O($n$log$m$)。通过用平衡的搜索树来取代二叉搜索树，可以得到所要求的复杂性。在二叉搜索树方案中，我们扩充了类 BSTree，增加了以下共享成员：

```
BSTree <E,K>& InsertVisit
          (const E&e, void(*Visit) (E& u));
```

若树中不存在关键值等于 e.key 的元素时，该函数将元素 e 插入到搜索树中。如果存在这样的元素u，则调用函数Visit。为了获得InsertVisit 的代码，可把Insert（见程序11-3）中的如下语句：

```
else throw BadInput( );
```

用下面的语句来替换：

```
else {Visit(p→data);
      return  *this;};
```

程序11-7给出了新的直方图程序代码。在访问一个元素的过程中，该元素的频率增加了 1。

程序11-7　使用搜索树的直方图程序

```
class eType {
  friend void main(void);
  friend void Add1(eType&);
  friend ostream& operator <<(ostream&, eType);
  public:
    operator int() const {return key;}
  private:
    int key,    // 元素值
        count;  // 频率
};

ostream& operator<<(ostream& out, eType x)
  {out << x.key << " " << x.count << "   "; return out;}

void Add1(eType& e) {e.count++;}

void main(void)
{// 使用搜索数的直方图程序
  BSTree<eType,int> T;
  int n;  // 元素数目
  cout << "Enter number of elements" << endl;
  cin >> n;

  // 输入元素并插入树中
  for (int i = 1; i <= n; i++) {
    eType e;  // 输入元素
    cout << "Enter element " << i << endl;
    cin >> e.key;
    e.count = 1;
    // 将e 插入树中，除非已存在同值元素
    // 在后一种情况下，将count 增1
    try {T.InsertVisit(e, Add1);}
    catch (NoMem)
      {cout << "Out of memory" << endl;
       exit(1);}
    }

  // 输出所有相异的元素以及它们的计数
```

```
    cout << "Distinct elements and frequencies are"
         << endl;
    T.Ascend();
}
```

11.5.2 用最优匹配法求解箱子装载问题

将n 个物品装入到容量为c 的箱子中的最优匹配方法，已在10.5.1节中介绍过。通过使用平衡的搜索树，能够在O($n$log$n$) 时间内完成箱子装载过程。搜索树的每一个元素代表一个正在使用的并且还能继续存放物品的箱子。假设当物品 i 被装载时，已使用的九个箱子中还有一些剩余空间，设这些箱子的剩余容量分别是 1,3,12,6,8,1,20,6和5。可以用一棵二叉搜索树来存储这九个箱子，每个箱子的剩余容量作为节点的关键值，因此这棵树应是允许有重复值的二叉搜索树（即为DBSTree或DAVLtree）。

图11-31给出了一棵存储上述九个箱子的二叉搜索树。节点内部是箱子的剩余容量，节点外侧是箱子的名称。这棵树也是一棵 AVL树。如果需要装载的物品 i 需要s[i]=4个空间单元，那么可以从根节点开始搜索，直至找到最优匹配的箱子。由根节点可知，箱子 h 的剩余容量是6，由于物体i 可以放入该箱中，因此箱子h 成为一个候选。由于根节点右子树中所有箱子的剩余容量至少是6，故不需要再从右子树中寻找合适的箱子，寻找只需要在左子树中进行。箱子 b 的剩余容量不能容纳该物品，因此搜索转移到了箱子 b 的右子树中，右子树的根节点箱子 i 可以容纳该物品，所以箱子i 成为适合的候选。从这里，搜寻转移到箱子i 的左子树，由于左子树为空，因此不再有更好的候选，所以箱子i 即要找的箱子。

再看另一个例子，假设s[i]=7，从根节点开始搜寻。根节点的箱子 h 不能装载物品i，因此转移到右子树中，箱子c 可以容纳物品i，因此成为新的候选箱子。从这里再向下搜寻，节点d没有足够的剩余容量容纳此物品，因此继续查找 d 的右子树，箱子 e 可以容纳物品 i，因此 e成为新的候选，然后转移到e 的左子树，左子树为空，搜索终止。

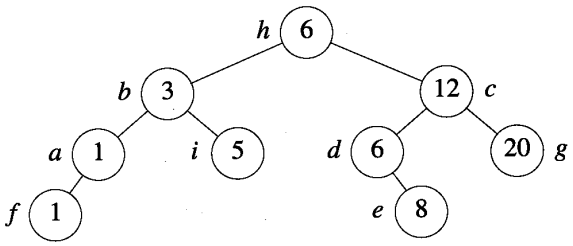


图11-31 含有重复值的AVL树

当找到最合适的箱子后，可以将它从搜索树中删除，将其剩余容量减去 s[i]，再将它重新插入到树中（除非它的剩余容量为零）。如果没有找到合适的箱子，则可以用一个新的箱子来装载物品i。

为了实现上述思想，既可以采用类DBSTree（可以得到平均性能O($n$log$n$)），也可以采用类DAVLtree（在各种情况中都能得到平均性能 O($n$log$n$)）。无论哪一种方法，都需要扩充类的定义，增加共享成员FindGE(k, Kout)，该函数可以找到剩余容量Kout≥k 的具有最小剩余容量的箱子。FindGE 的代码见程序11-8，它的复杂性是O(*height*)。类AVLtree中FindGE的代码可以与程序11-8完全相同。

程序11-8 寻找大于等于K的最小关键值

```
template<class E, class K>
```

```
bool DBSTree<E,K>::FindGE(const K& k, K& Kout) const
{// 寻找值 ≥ k 的最小元素
  BinaryTreeNode<E> *p = root, // 搜索指针
                    *s = 0;    // 指向迄今所找到的 >= k 的最小元素
  // 对树进行搜索
  while (p) {
    // p是一个候选吗?
    if (k <= p->data) {// 是的
       s = p; // p 是比 s更好的候选
       // 较小的元素仅会在左子树中
       p = p->LeftChild;}
    else // 不是, p->data太小, 试一试右子树
       p = p->RightChild;
    }

  if (!s) return false; // 没找到
  Kout = s->data;
  return true;
}
```

程序11-9使用最优匹配方法将n 个物品装入到箱子中，它使用了与 FirstFit（见程序10-7）相同的接口。这样，只要用＃ include 语句将类DBSTree 的文件加载到程序中，就可以用程序10-6来调用程序11-9。

程序11-9 用最优匹配法求解箱子装载问题的算法

```
class BinNode {
  friend void BestFitPack(int *, int, int);
  friend ostream& operator<<(ostream&, BinNode);
  public:
    operator int() const {return avail;}
  private:
    int ID,    // 箱子标号
        avail; // 可用容量
};

ostream& operator<<(ostream& out, BinNode x)
  {out << "Bin " << x.ID << " " << x.avail;
   return out;}

void BestFitPack(int s[], int n, int c)
{
  int b = 0;              // 所使用的箱子数
  DBSTree<BinNode, int> T; // 由箱子容量构成的树

  // 依次装载每个物品
  for (int i = 1; i <= n; i++) {// 装载物品 i
    int k;     // 最优匹配的箱子
    BinNode e; // 对应的节点
```

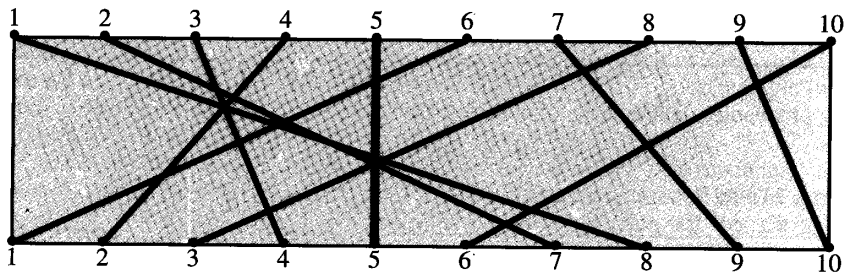
```
        if (T.FindGE(s[i], k)) // 寻找最优箱子
            T.Delete(k, e);     // 从树中删除最优箱子
        else {//没有足够大的箱子
              // 从一个新箱子开始
              e = *(new BinNode);
              e.ID = ++b;
              e.avail = c;}

        cout << "Pack object " << i << " in bin " << e.ID << endl;

        // 更新可用容量并将箱子插入树中，除非可用容量为 0
        e.avail -= s[i];
        if (e.avail) T.Insert(e);
        }
    }
```

11.5.3 交叉分布

在交叉分布问题中，从一个布线通道开始，通道的顶部和底部各有 n 个针脚。图11-32给出了n=10的情况。布线区域是图中带阴影的长方形区域，针脚的序号从 1到n，在通道的顶部和底部从左至右分布。另外，有 [1,2,3,...,n]的一个排列C。必须用一根电线将顶部的针脚 i与底部的针脚C_i 连接起来。在图11-32的例子中，C=[8,7,4,2,5,1,9,3,10,6]。需要连接的n 根线被编号为1到n，第i 根线连接顶部的i 和底部的C_i 。当且仅当$i<j$ 时，连线i 在连线j 的左边。



C = [8, 7, 4, 2, 5, 1, 9, 3, 10, 6]

图11-32 布线举例

图11-32显示了在布线区域中无论线路 9和10如何设置，它们一定会在某一点相交。我们不希望交叉的出现，以免出现短路，为此可以在交叉点放置绝缘物或将两条线路布设在不同层。因此，必须寻找最小的交叉点数目。可以验证，当线路是按图 11-32中的直线布设时，交叉点的数量是最少的。

每个交叉点用（i, j）表示，i 和j 是通过该交叉点的两条线路。识别所有交叉点的一种方法是检查线路的每一个（i, j）并且检验其中两条直线是否相交。为避免对同一交叉点检验两次，可以要求$i<j$（即交叉点（10，9）与（9，10）是一样的）。设k_i 是（i, j）的数量，$i<j$。在图11-32的例子中，k_9 =1，k_{10} =0。在图11-33中，列出了图11-32中所有的交叉及k_i 的值。该表的第i 行首先给出了k 的值，然后给出了j 的值，$i<j$，其中i 与j 是相交的，相交的总数K 由所有k_i 的和确定。在本例中K=22。由于k_i 只记录了线路i 与其右边线路（即$i<j$）相交的数量，因此k_i 给

出的是线路i右侧相交的数量。

为了降低复杂性，要求在通道的每一半区域中都含有大约相同的交叉点（一部分含 ëk/2û 个交叉点，另一部分含「k/2ù 个交叉点）。图11-34给出了图11-32中的另一种布线方法，在这种方法中，在通道的上、下部分各有11个交叉点。

上半部分的连接由排列 A=[1,4,6,3,7,2,9,5,10,8]给出，即顶部的针脚i与中间的针脚A_i相连。下半部分的连接由排列 B=[8,1,2,7,3,4,5,6,9,10]给出，中间的针脚i与底部的针脚B_i相连。可以看出$C_i = B_{A_i}$，$1 \le i \le n$。要完成C给出的连接，这个等式是必不可少的。

i	k_i	交叉
1	7	2 3 4 5 6 8 10
2	6	3 4 5 6 8 10
3	3	4 6 8
4	1	6
5	2	6 7
6	0	
7	2	8 10
8	0	
9	1	10
10	0	

图11-33 相交表

通过检查（i, j），在$\Theta(n^2)$的时间内可以计算出相交数量k_i及总的相交数量K。可采用程序11-10中的线性表从C计算出A和B。

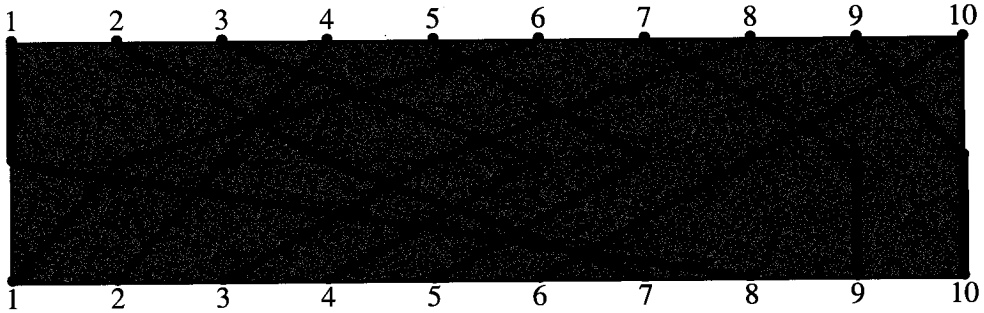


图11-34 分割交叉

程序11-10 使用线性表的交叉分布程序

```
LinearList<int> L(n);
int r = K/2; // 上半部分需要的交叉数

// 从右至左扫描线路
int w = n;   // w 是当前线路
while (r) {  // 上半部分需要更多的交叉
  if (k[w] < r) {// 使用w的所有交叉 L.Insert(k[w], w);
                 r -= k[w];}
  else {// 仅使用w的r个交叉
        L.Insert(r, w);
      r = 0;}
  w--;}

// 在中线上确定线路排列
// 前w条线的排列次序不变
for (int i = 1; i <= w; i++)
  X[i] = i;
```

```
// 其余线路的排列次序来自 L
for (i = w+1; i <= n; i++)
  L.Delete(1, X[i]);

// 计算上半部分的排列
for (i = 1; i <= n; i++)
  A[X[i]] = i;

// 计算下半部分的排列
for (i = 1; i <= n; i++)
  B[i] = C[X[i]];
```

在while 循环中，从右至左扫描这些线路，确定它们在布线通道中央的相关次序。目标是在通道中部产生一个布线次序，使得在布线通道上半部分的中间正好有 r=k/2个交叉。

线性表L用于记录通道中部当前所得到的次序。当考察线路 w 时，把w 右侧与w相交的线路中的最多k[w]个交叉分配到上半部分。第一个交点是与 L中的第一条线路相交产生的，第二个交点是与L 中第二条线路相交产生的，如此往复。如果将与 w 相交的k[w]条线路中的c 条分配到上半部分，那么这条线路必定与L中的前c 条线路相交。另外从w 到n 的次序也可以通过将 w 插入到L 中的第c 条线之后而得到。当在程序11-10的while 循环中考察线路w 时，线路w+1至n 已经在L 中。而且，由于k[w]不会超过线路w 右边的线路数，因此在考察 w 时，L中至少已有k[w]条线路。

当在程序11-10中的while循环中考察线路 w时，除非上半部分所需要的相交数 r比k[w]小，否则将所有右边相交点都分配到上半部分。

当while 循环终止时，布线通道中间的线路次序就建立好了。线路 1到w在上半部分没有相交，因此在这半部分中不必改变它们的相对次序。因而， X[1:w]=[1,2,...,w]，余下的线路排序由线性表L给出，程序11-10中的前两个for循环构造了X。

下面根据图11-32的例子来构造X。先将线路10添加到L中，得L=(10)。没有产生交叉，所以接下来把线路9添加到L中，得L=(10,9)。这时在上半部分产生了1个相交。之后将8加入到第 k_8 个元素后边，得L=(8,10,9)。此时上半部分的右边相交总数仍然是1。下面将7加入到L的第二个元素之后，上半部分有 2个相交，L变为(8,10,7,9)，所需的相交次数r降到了8。当6加入后，得到L=(6,8,10,7,9)，r=8。将线路5加入后又产生了2个相交，L=(6,8,5,10,7,9)，r=6。当4被加到L的第一个元素之后时，产生了一个相交，得到 L=(6,4,8,5,10,7,9)，r=5。下面将3加入，得到L=(6,4,8,3,5,10,7,9)，r=2。最后，考察2时，我们发现，虽然它能产生 k_2 =6个相交，但只能将其中 2 个分配到通道的上半部分，因此它被插入到L第二个元素的后边，得到L=(6,4,2,8,3,5,10,7,9)。剩下的线路保持它们的相对次序。

现在完成了上半部分的布线，然后计算线路的排列，通过在序列（ 1,2,...,w）上附加L得到X=[1,6,4,2,8,3,5,10,7,9]。

排列A与X关系密切。A[j]表明线路j应该连接到中间的哪个针脚上，而X[j]表明哪一条线路连接到中间的针脚j上。程序11-10中的第三个for循环就是用这种信息来计算 A的。在第四个for循环中，利用X和C计算出B。

由于将元素插入到大小为 s 的线性表中需要的时间是O(s)，所以程序11-10中的while循环需占用$O(n^2)$时间，第二个for 循环也需要这么多时间，其他代码需要 $\Theta(n)$时间，因此程序11-10的全部代码复杂性是$O(n^2)$。将程序11-10需要的时间与计算K 和k[i] 所需要的时间联系在一起，

可知在使用线性表解决交叉分布问题时，所需要的全部时间是$O(n^2)$。

通过使用平衡的搜索树来代替线性表，可以把解决方案的复杂性降低到$O(n\log n)$。为了得到平均复杂性$O(n\log n)$，可以使用带索引的二叉搜索树，而不使用带索引的平衡搜索树。这两种情况在技术上是相同的，下面将用带索引的二叉搜索树来阐述此技术。

首先，来看一下如何计算相交数量k_i，$1\leq i\leq n$。假设检查的线路次序是n, n-1, ..., 1并且检查线路i 时，将C_i 插入到带索引的搜索树中。对于图11-32中的例子，首先从空树开始。检查线路10并将$C_{10}=6$插入到空树中，得到图11-35a 所示的树。节点外侧的数字是其LeftSize的值，节点内部是其关键值（或C的值）。注意到k_n 总是为零，因此设$k_n=0$。然后检查线路9并将$C_9=10$插入到树中，得到图11-35b 所示的树。为了实现这次插入，越过了根节点（其LeftSize=1）。根据这个LeftSize值，可知线路9的底部节点正好在目前所看到的一条线路的右边，因此$k_9=1$。下面检查线路8，将$C_8=3$插入到树中，得到图11-35c 所示的树。由于C_8 是树中最小的入口，因此没有线路相交且$k_8=0$。对于线路7，$C_7=9$被插入后得到图11-35d 中的树，C_7 成为树中的第三个最小入口。通过对进入其右子树的各节点的LeftSize值连续求和，可以确定C_7 是第三个最小入口。C_7 被插入时，这个和是2，因此，新元素是当前第三个最小的。由此可以得出结论，它的底部节点位于树中其他2个节点的右边，因此，$k_7=2$。按此方法进行下去，当检查线路 6至2时，产

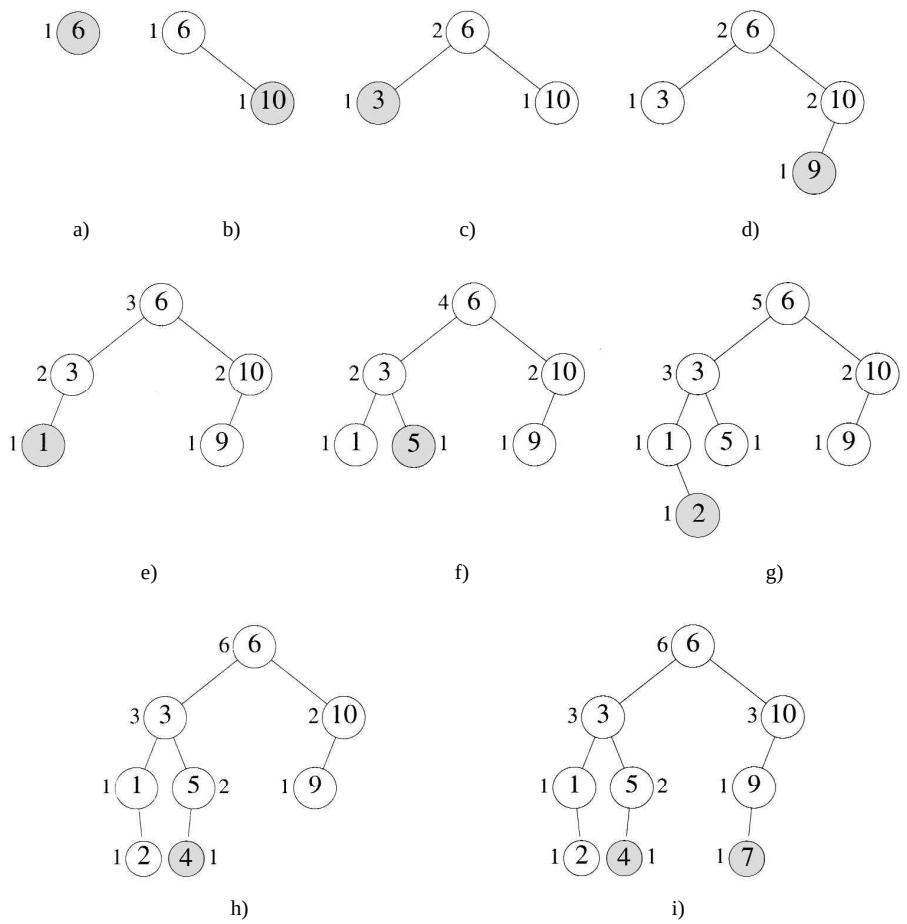


图11-35 计算交叉的数量

生了图11-35e 至图11-35i 的树。最后，检查线路1时，将C_1=8插入到树中，作为关键值为7的节点的右孩子。进入其右子树的各节点的LeftSize值之和是6+1=7，线路1的底部节点位于树中7条线路的右边，因此k_1=7。

检查线路i 和计算k_i 需要的时间是O(h)，其中h 是当前带索引的搜索树的高度。因此，用带索引的二叉搜索树可在O($n$log$n$)平均时间内或用带索引的平衡搜索树在 O($n$log$n$) 平均时间内计算出所有的k_i。

为计算A，可以用一个修改的带索引的搜索树来实现程序 11-10的代码。在这种修改中，元素没有关键值，每个元素只有一个位置或阶。 Insert(j, e) 用来插入元素e 使得e 成为阶为j+1的元素。为了按阶的次序排列元素，可以进行一次中序遍历。执行程序 11-10所需要的时间为O($n$log$n$)。

得到排列A的另一种方法是首先计算 $r=\sum_{i=1}^{n}k_i$ 12和s=最小的i，使得 $\sum_{l=i}^{n}k_l \le r$。对于上面的例子有r=11和s=3。程序11-10实现了线路 n，n-1，…，s 的所有相交，其中上半部分包含线路 s-1的 $r-\sum_{l=s}^{n}k_l$ 个相交，下半部分包含余下的相交。为了得到上半部分的相交点，对插入 C_s 后的树进行检查。在本例中，检查图 11-35h 中的树。对树 h 的中序遍历可产生序列（ 1,2,3,4,5,9,10）。用相应的线路编号来替换这些底部节点，可得到序列（ 6,4,8,3,5,10,7,9），它给出了按图 11-35h 中所描述的9个相交线路所得到的排列。对于另外 2个相交，将线路s=2插入到该序列中第二条线路之后，得到新的线路序列（ 6,4,2,8,3,5,10,7,9）。剩下的线路 1到s-1加到序列前部，可得(1,6,4,2,8,3,5,10,7,9)，它就是程序11-10所计算出的排列X。为了用这种方法得到 X，需要重新运行用来计算 k_i 的部分代码，执行一个中序遍历，插入线路s 并在序列之前增加少量线路。所有步骤需要的时间是 O($n$log$n$)。程序 11-10的最后两个 for循环可在线性时间内从 X中得到A和B。

练习

37. 写一个直方图程序，首先将n个关键值输入到一个数组中，然后对数组进行排序，最后对数组从左到右进行扫描，输出不同的关键值和每个关键值出现的次数。

38. 写一个直方图程序，使用链表散列而不是程序 11-7中的二叉搜索树来储存不相同的关键值和它们的频率。将程序的运行时间与程序 11-7相比较。

39. 1) 扩充类DBSTree，增加共享成员DeleteGE(k,e)。该函数用来删除最小关键值不小于k的元素，被删除元素在e中返回，如果删除失败DeleteGE可以引发异常。

2) 用DeleteGE(而不是FindGE)设计一个BestFit的新版本。

3) 哪一种运行得更快？为什么？

40. 1) 用一个带索引的AVL搜索树为交叉分布问题设计一个性能为O($n$log$n$)的算法。

2) 检验代码是否正确。

3) 根据实际运行时间，把该算法与本节（见程序 11-10）所介绍的$\Theta(n^2)$算法进行比较。可使用随机产生的排列C和n=1000, 10 000和50 000来进行实验。

11.6 参考及推荐读物

AVL树是由G.Adelson-Velskii和E.Landis在1962年发明的，在D.Knuth. *The Art of Computer Programming*: *Sorting and Searching*. Addison-Wesley, 1973. 一书中可以找到更多的关于这些树的材料。

红-黑树是由R.Bayer在1972年发明的，但是Bayer将这种树叫作“对称的平衡B-树”，红-黑树这一术语是Guibas和Sedgewick在更详细地研究了这种树之后在1978年提出的。这方面的先期论文有R.Bayer. Symmetric Binary B-Tress: Data Structures and Maintenance Algorithms. *Acta Informatica*,1,1972,290~306 和L.Guibas, R.Sedgewick. A Dichromatic Framework for Balanced Trees, *Proceedings of the 10th IEEE Symposium on Foundations of Computer Science*, 1978，8~21。

用红-黑树来实现优先搜索树方面的应用可参考 E.McCreight. Priority Search Trees. *SIAM Journal on Computing*, 14，2，1985，257~276。

交叉分布问题的解决算法由S.Cho和S.Sahni给出（未发表）。

具有相同渐进复杂性的各种不同搜索树结构可参考 E.Horowitz, S.Sahni, D.Mehta. *Fundamentals of Data Structures in C++*. W.H.Freeman, 1994。这本书还介绍了B'-树和其他各种变化的B*-树。