5

关联式容器

associative containers

容器，置物之所也，第 4 章一开始曾对此做了一些描述。所谓 STL 容器，即是将最常被运用的一些数据结构（data structures）实现出来，其涵盖种类有可能在每五年召开一次的 C++ 标准委员会议中不断增订。

根据“数据在容器中的排列”特性，容器可概分为序列式（sequence）和关联式（associative）两种，如图 5-1。第 4 章已经探讨过序列式容器，本章将探讨关联式容器。

标准的 STL 关联式容器分为 set（集合）和 map（映射表）两大类，以及这两大类的衍生体 multiset（多键集合）和 multimap（多键映射表）。这些容器的底层机制均以 RB-tree（红黑树）完成。RB-tree 也是一个独立容器，但并不开放给外界使用。

此外，SGI STL 还提供了一个不在标准规格之列的关联式容器：hash table（散列表）[1]，以及以此 hash table 为底层机制而完成的 hash_set（散列集合）、hash_map（散列映射表）、hash_multiset（散列多键集合）、hash_multimap（散列多键映射表）。

[1] hash table（散列表）及其衍生容器相当重要。它们未被纳入 C++标准的原因是，提案太迟了。下一代 C++ 标准程序库很有可能会纳入它们。

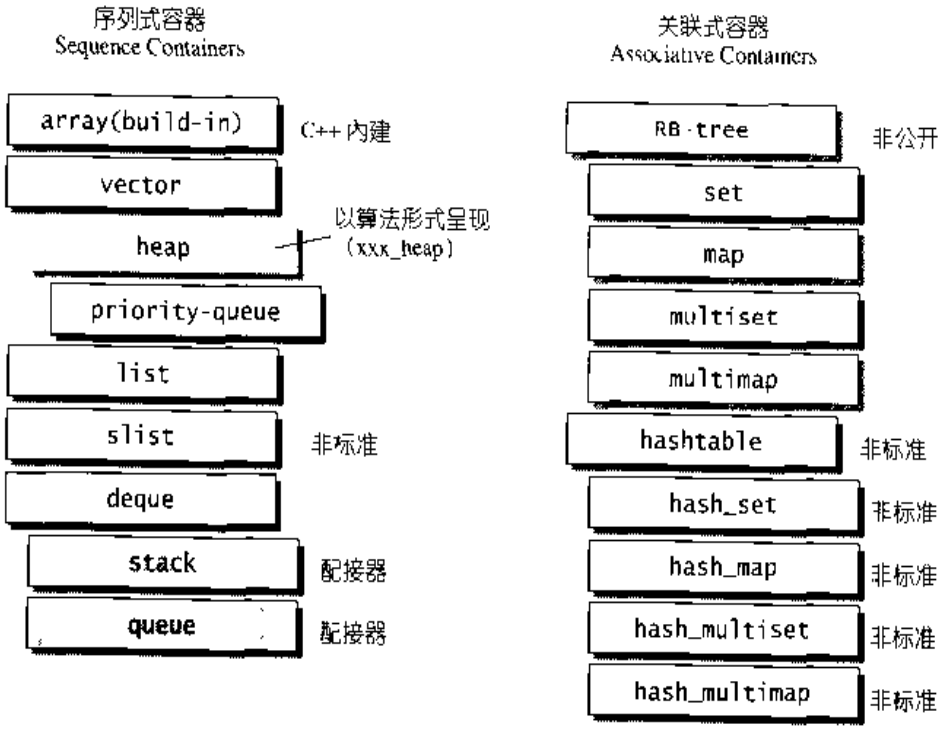


图 **5-1** SGI STL 的各种容器。本图以内缩方式来表达基层与衍生层的关系。这里所谓的衍生，并非继承（inheritance）关系，而是内含（containment）关系。例如 heap 内含一个 vector，priority-queue 内含一个 heap，stack 和 queue 都内含一个 deque，set/map/multiset/multimap 都内含一个 RB-tree，hast_x 都内含一个 hashtable。

关联式容器（associative containers）

所谓关联式容器，观念上类似关联式数据库（实际上则简单许多）：每笔数据（每个元素）都有一个键值（*key*）和一个实值（*value*）[2]。当元素被插入到关联式容器中时，容器内部结构（可能是 RB-tree，也可能是 hash-table）便依照其键值大小，以某种特定规则将这个元素放置于适当位置。关联式容器没有所谓头尾（只有最大元素和最小元素），所以不会有所谓 `push_back()`、`push_front()`、`pop_back()`、`pop_front()`、`begin()`、`end()` 这样的操作行为。

一般而言，关联式容器的内部结构是一个 balanced binary tree（平衡二叉树），以便获得良好的搜寻效率。balanced binary tree 有许多种类型，包括 AVL-tree、

[2] set 的键值就是实值，map 的键值可以和实值分开，并形成一种映射关系，所以 map 被称为映射表，或称为字典（dictionary，取“字典之英文单字为键值索引”之象征）。

RB-tree、AA-tree，其中最被广泛运用于 STL 的是 RB-tree（红黑树）。为了探讨 STL 的关联式容器，我必须先探讨 RB-tree。

进入 RB-tree 主题之前，让我们先对 tree 的来龙去脉有个概念。以下讨论都和最终目标 RB-tree 有密切关联。这些讨论都只是提纲挈领，如果你需要更全面的知识，请阅读数据结构和算法方面的专著。

5.1 树的导览

树（tree），在计算机科学里，是一种十分基础的数据结构。几乎所有操作系统都将文件存放在树状结构里；几乎所有的编译器都需要实现一个表达式树（expression tree）；文件压缩所用的哈夫曼算法（Huffman's Algorithm）需要用到树状结构；数据库所使用的 B-tree 则是一种相当复杂的树状结构。

本章所要介绍的 RB-tree（红黑树）是一种被广泛运用、可提供良好搜寻效率的树状结构。

树由节点（nodes）和边（edges）构成，如图 5-2 所示。整棵树有一个最上端节点，称为根节点（root）。每个节点可以拥有具方向性的边（directed edges），用来和其它节点相连。相连节点之中，在上者称为父节点（parent），在下者称为子节点（child）。无子节点者称为叶节点（leaf）。子节点可以存在多个，如果最多只允许两个子节点，即所谓二叉树（binary tree）。不同的节点如果拥有相同的父节点，则彼此互为兄弟节点（siblings）。根节点至任何节点之间有唯一路径（path），路径所经过的边数，称为路径长度（length）。根节点至任一节点的路径长度，即所谓该节点的深度（depth）。根节点的深度永远是 0。某节点至其最深子节点（叶节点）的路径长度，称为该节点的高度（height）。整棵树的高度，便以根节点的高度来代表。节点 A → B 之间如果存在（唯一）一条路径，那么 A 称为 B 的祖代（ancestor），B 称为 A 的子代（descendant）。任何节点的大小（size）是指其所有子代（包括自己）的节点总数。

图 5-2 对这些术语做了一个总整理。

```
size of E : 3
height of E : 1
depth of E : 2
length of E :2

size of C : 4
height of C : 2
depth of C : 1
length of C : 1

size of A : 12
height of A : 3
depth of A : 0
length of A : 0
```

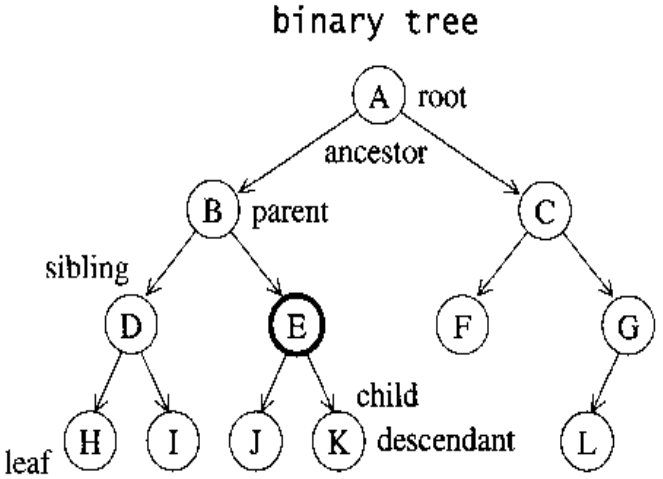


图 **5-2** 树状结构的相关术语整理。图中浅灰色术语是相对于节点 E 而言。

5.1.1 二叉搜索树（binary search tree）

所谓二叉树（binary tree），其意义是：“任何节点最多只允许两个子节点”。这两个子节点称为左子节点和右子节点。如果以递归方式来定义二叉树，我们可以说：“一个二叉树如果不为空，便是由一个根节点和左右两子树构成；左右子树都可能为空”。二叉树的运用极广，先前提到的编译器表达式树（expression tree）和哈夫曼编码树（Huffman coding tree）都是二叉树，如图 5-3 所示。

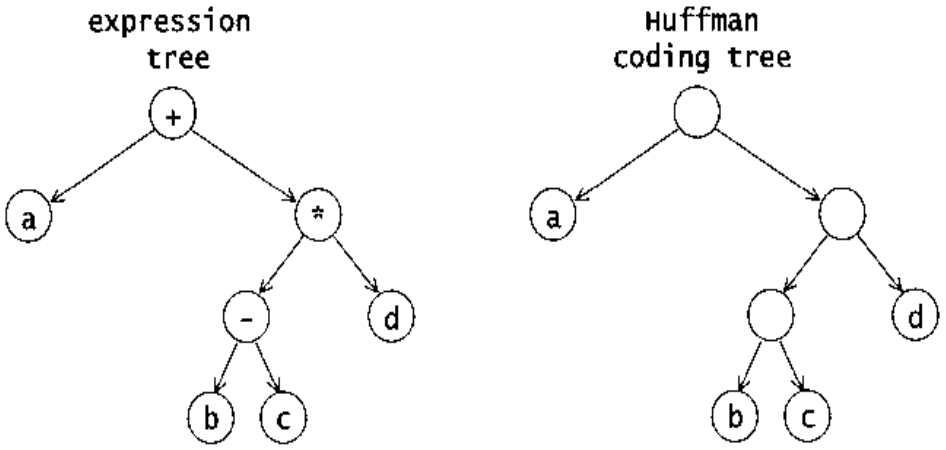


图 **5-3** 二叉树的应用

所谓二叉搜索树（binary search tree），可提供对数时间（logarithmic time）[3]的元素插入和访问。二叉搜索树的节点放置规则是：任何节点的键值一定大于其左子树中的每一个节点的键值，并小于其右子树中的每一个节点的键值[4]。因此，从根节点一直往左走，直至无左路可走，即得最小元素；从根节点一直往右走，直至无右路可走，即得最大元素。图 5-4 所示的就是一棵二叉搜索树。

要在一棵二叉搜索树中找出最大元素或最小元素，是一件极简单的事：就像上述所言，一直往左走或一直往右走即是。比较麻烦的是元素的插入和移除。图 5-5 是二叉搜索树的元素插入操作图解。插入新元素时，可从根节点开始，遇键值较大者就向左，遇键值较小者就向右，一直到尾端，即为插入点。

图 5-6 是二叉搜索树的元素移除操作图解。欲删除旧节点 A，情况可分两种。如果 A 只有一个子节点，我们就直接将 A 的子节点连至 A 的父节点，并将 A 删除。如果 A 有两个子节点，我们就以右子树内的最小节点取代 A。注意，右子树的最小节点极易获得：从右子节点开始（视为右子树的根节点），一直向左走至底即是。

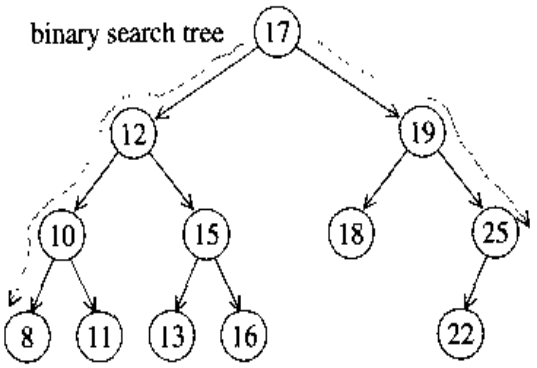


图 **5-4** 这是一棵二叉搜索树。任何节点的键值（*key*）一定大于其左子树中的每一个节点的键值，并小于其右子树中的每一个节点的键值。图中节点内的数值代表键值。

[3] 对数时间（logarithmic time）用来表示复杂度。详见第 6 章。

[4] 注意，键值（key）可能和实值（value）相同，也可能和实值不同。

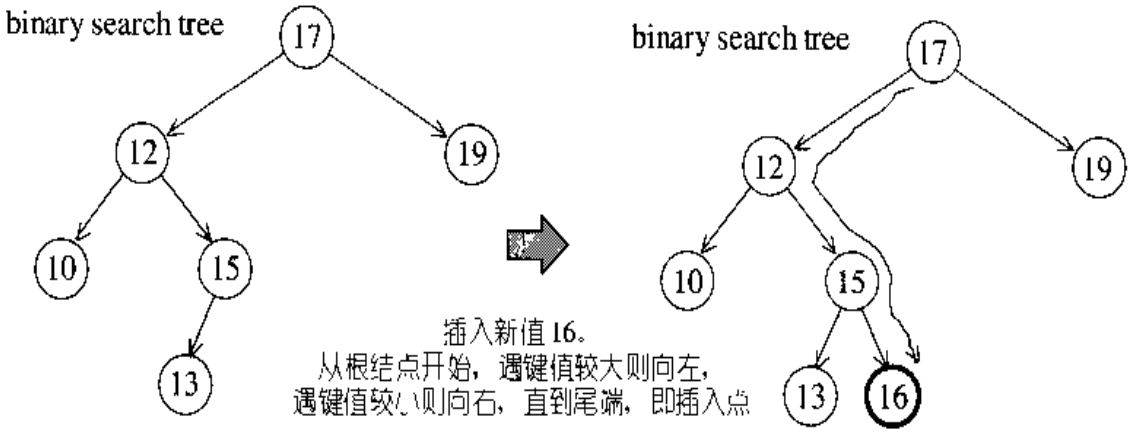


图 **5-5** 二叉搜索树的节点插入操作

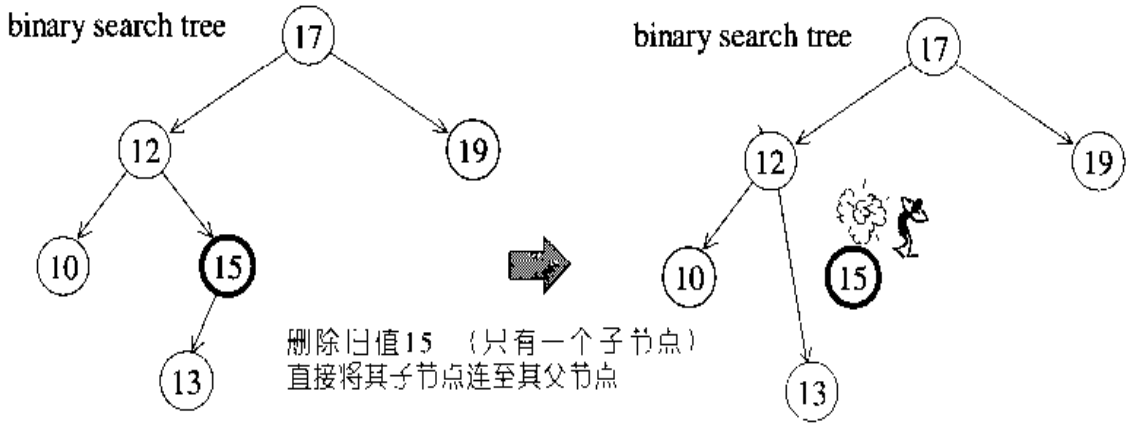


图 **5-6a** 二叉搜索树的节点删除操作之一（目标节点只有一个子节点）

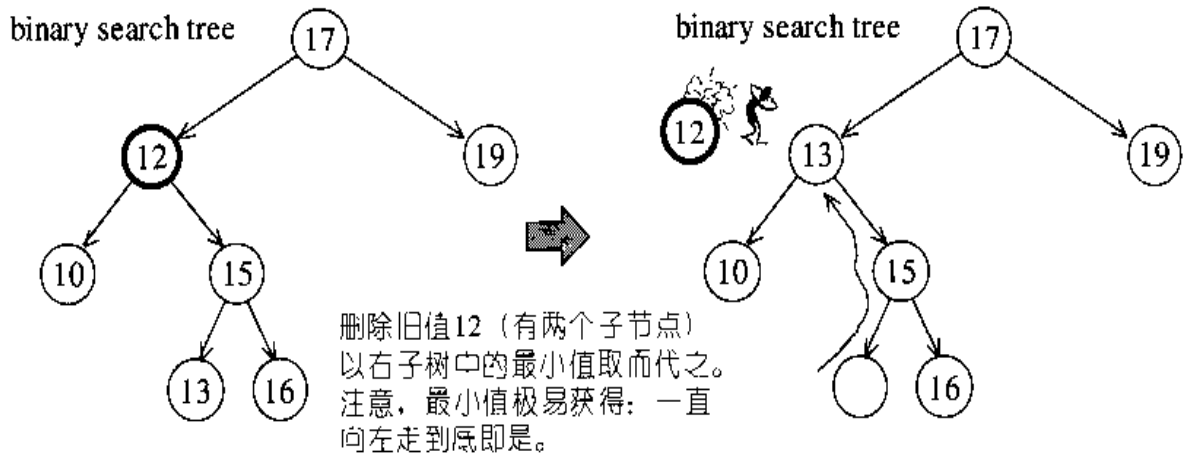


图 **5-6b** 二叉搜索树的节点删除操作之二（目标节点有两个子节点）

5.1.2 平衡二叉搜索树（balanced binary search tree）

也许因为输入值不够随机，也许因为经过某些插入或删除操作，二叉搜索树可能会失去平衡，造成搜寻效率低落的情况，如图 5-7 所示。

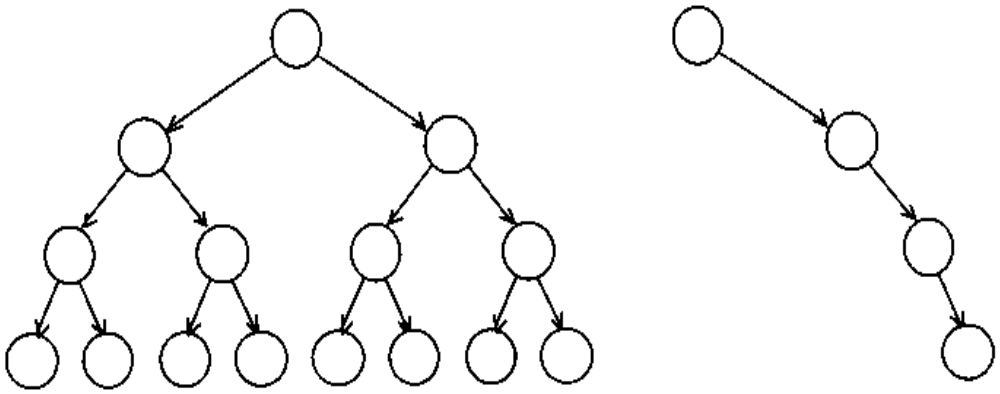


图 **5-7** 树形的极度平衡与极度不平衡

所谓树形平衡与否，并没有一个绝对的测量标准。“平衡”的大致意义是：没有任何一个节点过深（深度过大）。不同的平衡条件，造就出不同的效率表现，以及不同的实现复杂度。有数种特殊结构如 AVL-tree、RB-tree、AA-tree，均可实现出平衡二叉搜索树，它们都比一般的（无法绝对维持平衡的）二叉搜索树复杂，因此，插入节点和删除节点的平均时间也比较长，但是它们可以避免极难应付的最坏（高度不平衡）情况，而且由于它们总是保持某种程度的平衡，所以元素的访问（搜寻）时间平均而言也就比较少。一般而言其搜寻时间可节省 25% 左右。

5.1.3 AVL tree（Adelson-Velskii-Landis tree）

AVL tree 是一个“加上了额外平衡条件”的二叉搜索树。其平衡条件的建立是为了确保整棵树的深度为 O(logN)。直观上的最佳平衡条件是每个节点的左右子树有着相同的高度，但这未免太过严苛，我们很难插入新元素而又保持这样的平衡条件。AVL tree 于是退而求其次，要求任何节点的左右子树高度相差最多 1。这是一个较弱的条件，但仍能够保证“对数深度（logarithmic depth）”平衡状态。

图 5-8 左侧所示的是一个 AVL tree，插入了节点 11 之后（图右），灰色节点违反 ATL tree 的平衡条件。由于只有“插入点至根节点”路径上的各节点可能改变平衡状态，因此，只要调整其中最深的那个节点，便可使整棵树重新获得平衡。

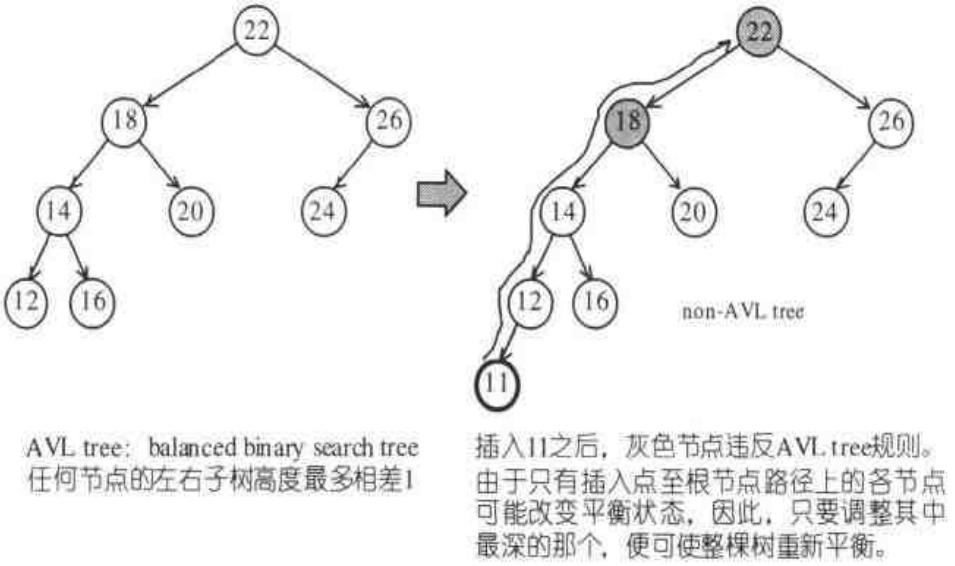


图 **5-8** 图左是 AVL tree。插入节点 11 后，图右灰色节点违反 ATL tree 条件

前面说过，只要调整“插入点至根节点”路径上，平衡状态被破坏之各节点中最深的那一个，便可使整棵树重新获得平衡。假设该最深节点为 X，由于节点最多拥有两个子节点，而所谓“平衡被破坏”意味着 X 的左右两棵子树的高度相差 2，因此我们可以轻易将情况分为四种（图 5-9）：

1. 插入点位于 X 的左子节点的左子树——左左。
2. 插入点位于 X 的左子节点的右子树——左右。
3. 插入点位于 X 的右子节点的左子树——右左。
4. 插入点位于 X 的右子节点的右子树——右右。

情况 1, 4 彼此对称，称为外侧（outside）插入，可以采用单旋转操作（single rotation）调整解决。情况 2, 3 彼此对称，称为内侧（inside）插入，可以采用双旋转操作（double rotation）调整解决。

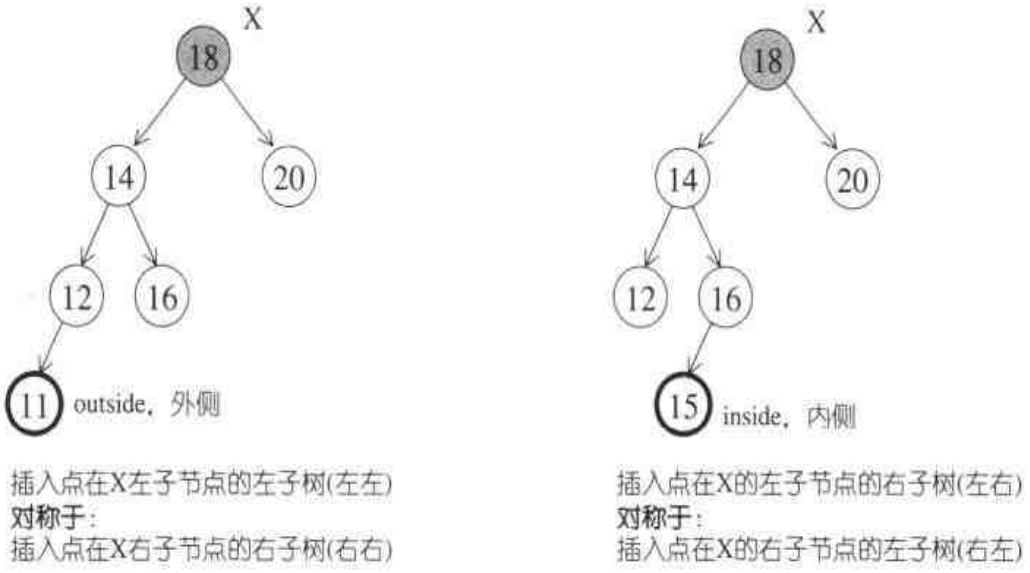


图 **5-9** AVL-tree 的四种 “平衡破坏” 情况

5.1.4 单旋转（Single Rotation）

在外侧插入状态中，k2 “插入前平衡，插入后不平衡” 的唯一情况如图 5-10 左侧所示。A 子树成长了一层，致使它比 C 子树的深度多 2。B 子树不可能和 A 子树位于同一层，否则 k2 在插入前就处于不平衡状态了。B 子树也不可能和 C 子树位于同一层，否则第一个违反平衡条件的将是 k1 而不是 k2。

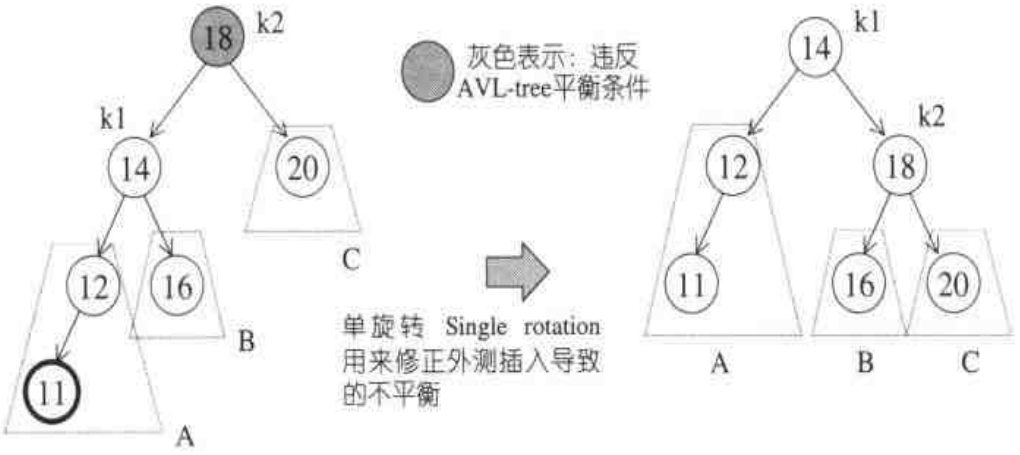


图 **5-10** 延续图 5-8 的状况，以 “单旋转” 修正外侧插入所导致的不平衡

为了调整平衡状态，我们希望将 A 子树提高一层，并将 C 子树下降一层 — 这已经比 AVL-tree 所要求的平衡条件更进一步了。图 5-10 右侧即是调整后的情况。我们可以这么想象，把 k1 向上提起，使 k2 自然下滑，并将 B 子树挂到 k2 的左侧。这么做是因为，二叉搜索树的规则使我们知道，k2 > k1，所以 k2 必须成为新树形中的 k1 的右子节点。二叉搜索树的规则也告诉我们，B 子树的所有节点的键值都在 k1 和 k2 之间，所以新树形中的 B 子树必须落在 k2 的左侧。

以上所有调整操作都只需要将指针稍做搬移，就可迅速达成。完成后的新树形符合 AVL-tree 的平衡条件，不需再做调整。

图 5-10 所显示的是 “左左”外侧插入。至于 “右右”外侧插入，情况如出一辙。

5.1.5 双旋转（Double Rotation）

图 5-11 左侧为内侧插入所造成的不平衡状态。单旋转无法解决这种情况。第一，我们不能再以 k3 为根节点，其次，我们不能将 k3 和 k1 做一次单旋转，因为旋转之后还是不平衡（你不妨自行画图试试）。唯一的可能是以 k2 为新的根节点，这使得（根据二叉搜索树的规则）k1 必须成为 k2 的左子节点，k3 必须成为 k2 的右子节点，而这么一来也就完全决定了四个子树的位置。新的树形满足 AVL-tree 的平衡条件，并且，就像单旋转的情况一样，它恢复了节点插入之前的高度，因此保证不再需要任何调整。

为什么称这种调整为双旋转呢？因为它可以利用两次单旋转完成，见图 5-12。

以上所有调整操作都只需要将指针稍做搬移，就可迅速达成。完成之后的新树形符合 AVL-tree 的平衡条件，不需再做调整。

图 5-11 显示的是 “左右”内侧插入。至于 “右左”内侧插入，情况如出一辙。

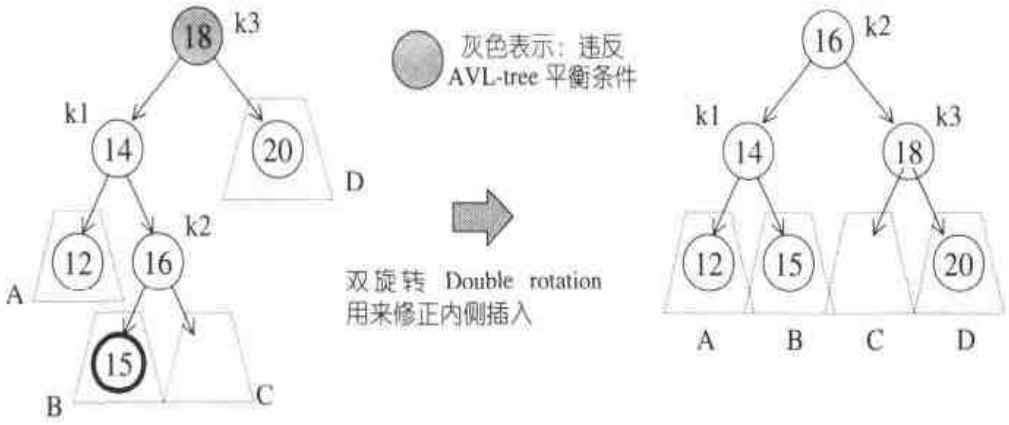


图 **5-11** 延续图 5-8 的状况，以双旋转修正因内侧插入而导致的不平衡。

本图显示的是 “左右”内侧插入。至于 “右左”内侧插入，情况如出一辙。

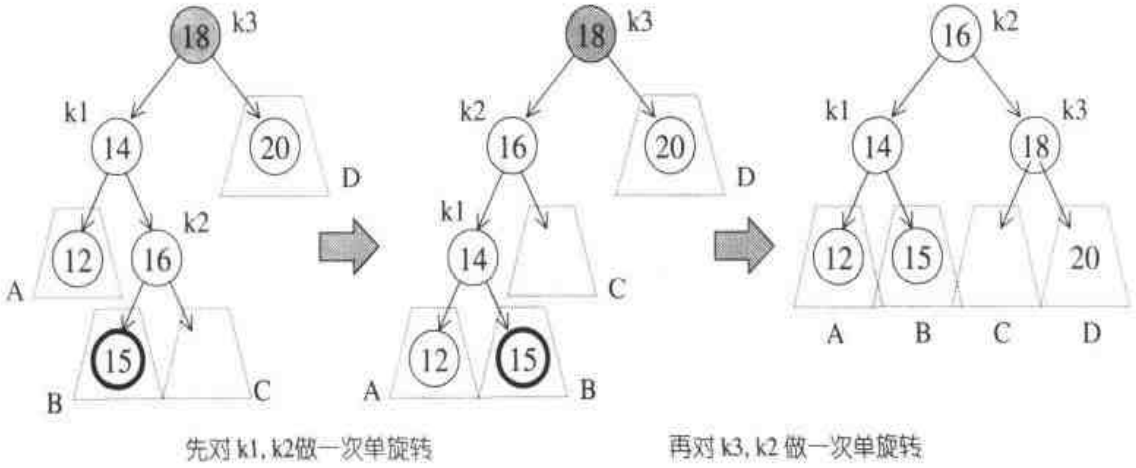


图 **5-12** 双旋转（如图 5-11）可由两次单旋转合并而成。这对编程带来不少方便。

RB-tree 是另一个被广泛使用的平衡二叉搜索树，也是 SGI STL 唯一实现的一种搜寻树，作为关联式容器（associated containers）的底部机制之用。RB-tree 的平衡条件虽然不同于 AVL-tree，但同样运用了单旋转和双旋转修正操作。下一节我将详细介绍 RB-tree。

5.2 RB-tree（红黑树）

AVL-tree 之外，另一个颇具历史并被广泛运用的平衡二叉搜索树是 RB-tree（红黑树）。所谓 RB-tree，不仅是一个二叉搜索树，而且必须满足以下规则：

1. 每个节点不是红色就是黑色（图中深色底纹代表黑色，浅色底纹代表红色，下同）。
2. 根节点为黑色。
3. 如果节点为红，其子节点必须为黑。
4. 任一节点至 NULL（树尾端）的任何路径，所含之黑节点数必须相同。

根据规则 4，新增节点必须为红；根据规则 3，新增节点之父节点必须为黑。当新节点根据二叉搜索树的规则到达其插入点，却未能符合上述条件时，就必须调整颜色并旋转树形。见图 5-13 说明。

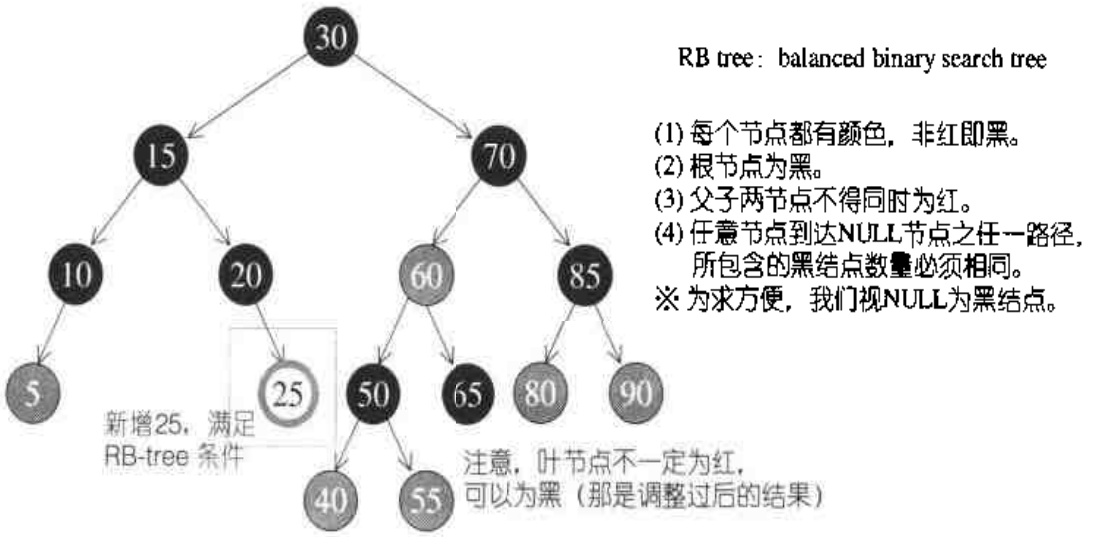


图 **5-13** RB-tree 的条件与实例

5.2.1 插入节点

现在让我们延续图 5-13 的状态，插入一些新节点，看看会产生什么变化。我要举出四种不同的典型。

假设我为图 5-13 的 RB-tree 分别插入 3, 8, 35, 75，根据二叉搜索树的规则，这四个新节点的落脚处应该如图 5-14 所示。啊，是的，它们都破坏了 RB-tree 的规则，因此我们必须调整树形，也就是旋转树形并改变节点颜色。

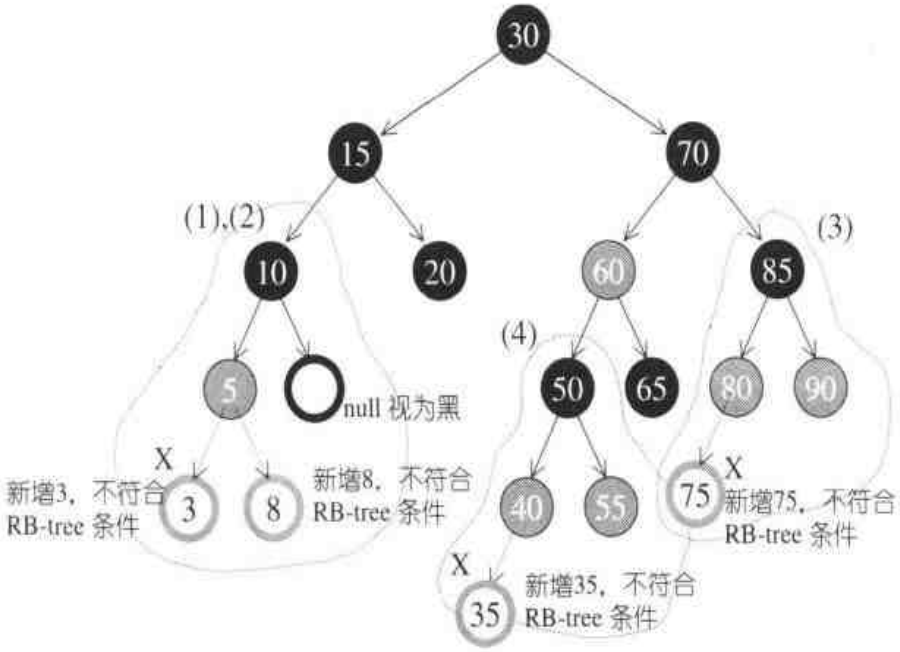


图 **5-14** 为 RB-tree 插入四个新节点：3, 8, 35, 75。新增节点必为红色，暂以空心粗框表示。不论插入 3, 8, 35, 75 之中的哪一个节点，都会破坏 RB-tree 的规则，致使我们必须旋转树形并调整节点的颜色。

为了方便讨论，让我先为某些特殊节点定义一些代名。以下讨论都将沿用这些代名。假设新节点为 X，其父节点为 P，祖父节点为 G，伯父节点（父节点之兄弟节点）为 S，曾祖父节点为 GG。现在，根据二叉搜索树的规则，新节点 X 必为叶节点。根据红黑树规则 4，X 必为红。若 P 亦为红（这就违反了规则 3，必须调整树形），则 G 必为黑（因为原为 RB-tree，必须遵循规则 3）。于是，根据 X 的插入位置及外围节点（S 和 GG）的颜色，有了以下四种考虑。

- 状况 1：S 为黑且 X 为外侧插入。对此情况，我们先对 P,G 做一次单旋转，再更改 P,G 颜色，即可重新满足红黑树的规则 3。见图 5-15a。

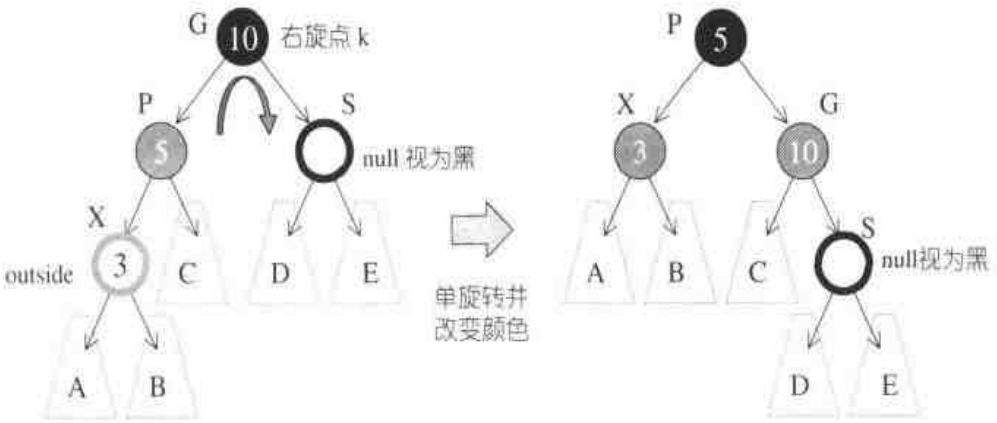


图 **5-15a** S 为黑且 X 为外侧插入。先对 P,G 做一次单旋转，再更改 P,G 颜色，即可重新满足红黑树规则 3。

注意，此时可能产生不平衡状态（高度相差 1 以上）。例如图中的 A 和 B 为 null，D 或 E 不为 null。这倒没关系，因为 RB-tree 的平衡性本来就比 AVL-tree 弱。然而 RB-tree 通常能够导致良好的平衡状态，是的，经验告诉我们，RB-tree 的搜寻平均效率和 AVL-tree 几乎相等。

- 状况 2：S 为黑且 X 为内侧插入。对此情况，我们必须先对 P, X 做一次单旋转并更改 G, X 颜色，再将结果对 G 做一次单旋转，即可再次满足红黑树规则 3。见图 5-15b。

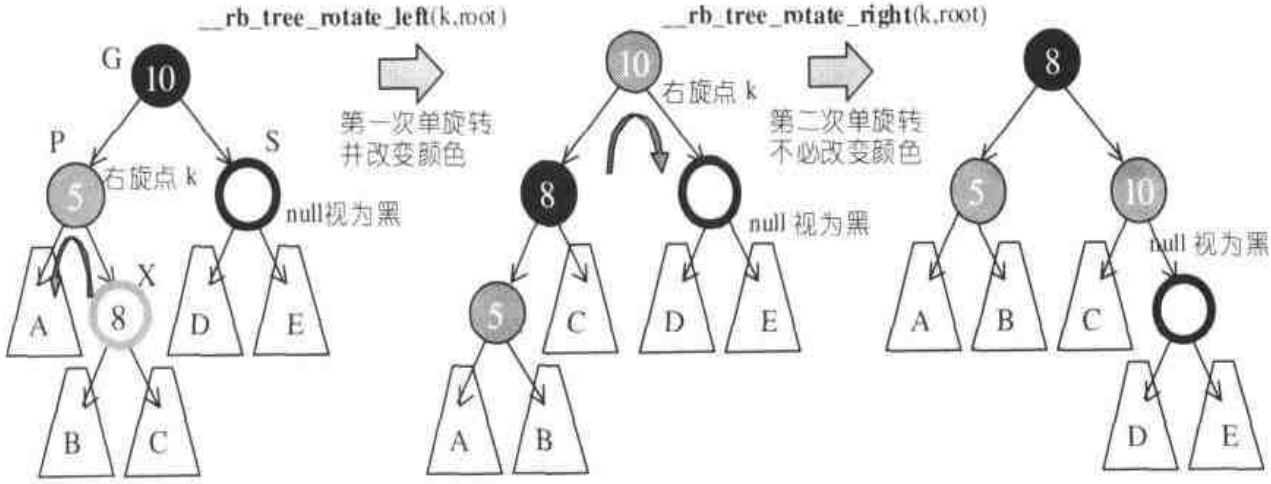


图 **5-15b** 最上方所示为 SGI <stl_tree.h> 所提供的函数，用于左旋或右旋。

- 状况 3：S 为红且 X 为外侧插入。对此情况，先对 P 和 G 做一次单旋转，并改变 X 的颜色。此时如果 GG 为黑，一切搞定，如图 5-15c。但如果 GG 为红，则问题就比较大些，咱…见状况 4。

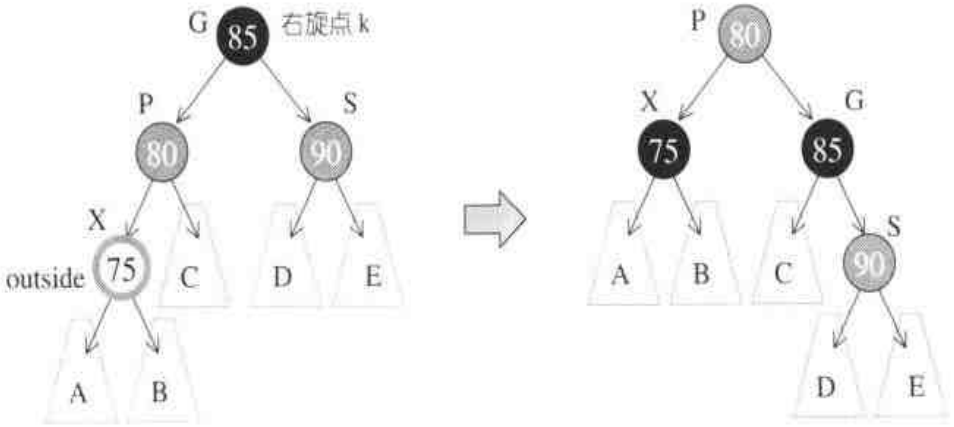


图 **5-15c** RB-tree 元素插入状况 3

- 状况 4：S 为红且 X 为外侧插入。对此情况，先对 P 和 G 做一次单旋转，并改变 X 的颜色。此时如果 GG 亦为红，还得持续往上做，直到不再有父子连续为红的情况。

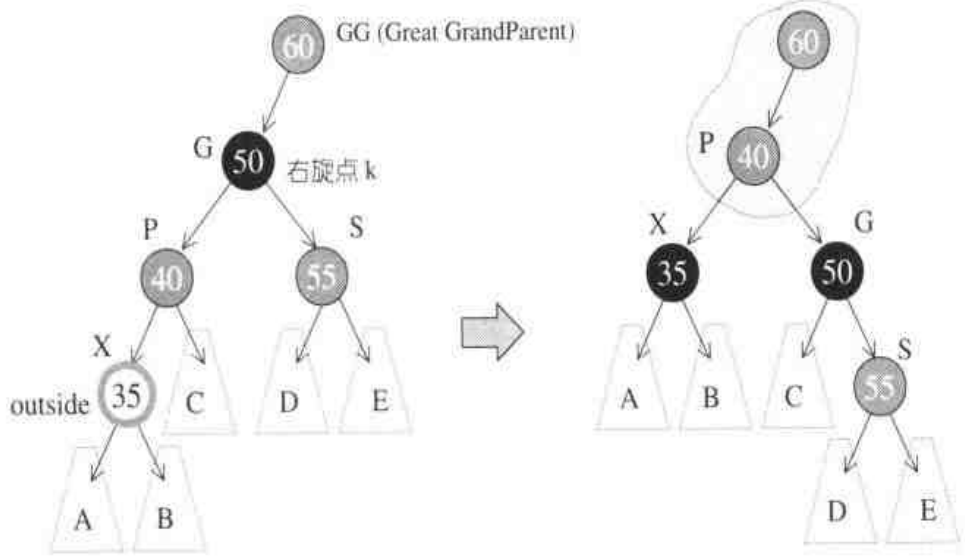


图 **5-15d** RB-tree 元素插入状况 4

5.2.2 一个由上而下的程序

为了避免状况 4 “父子节点皆为红色”的情况持续向 RB-tree 的上层结构发展，形成处理时效上的瓶颈，我们可以施行一个由上而下的程序（top-down procedure）：假设新增节点为 A，那么就延着 A 的路径，只要看到有某节点 X 的两个子节点皆为红色，就把 X 改为红色，并把两个子节点改为黑色，如图 5-15e 所示。

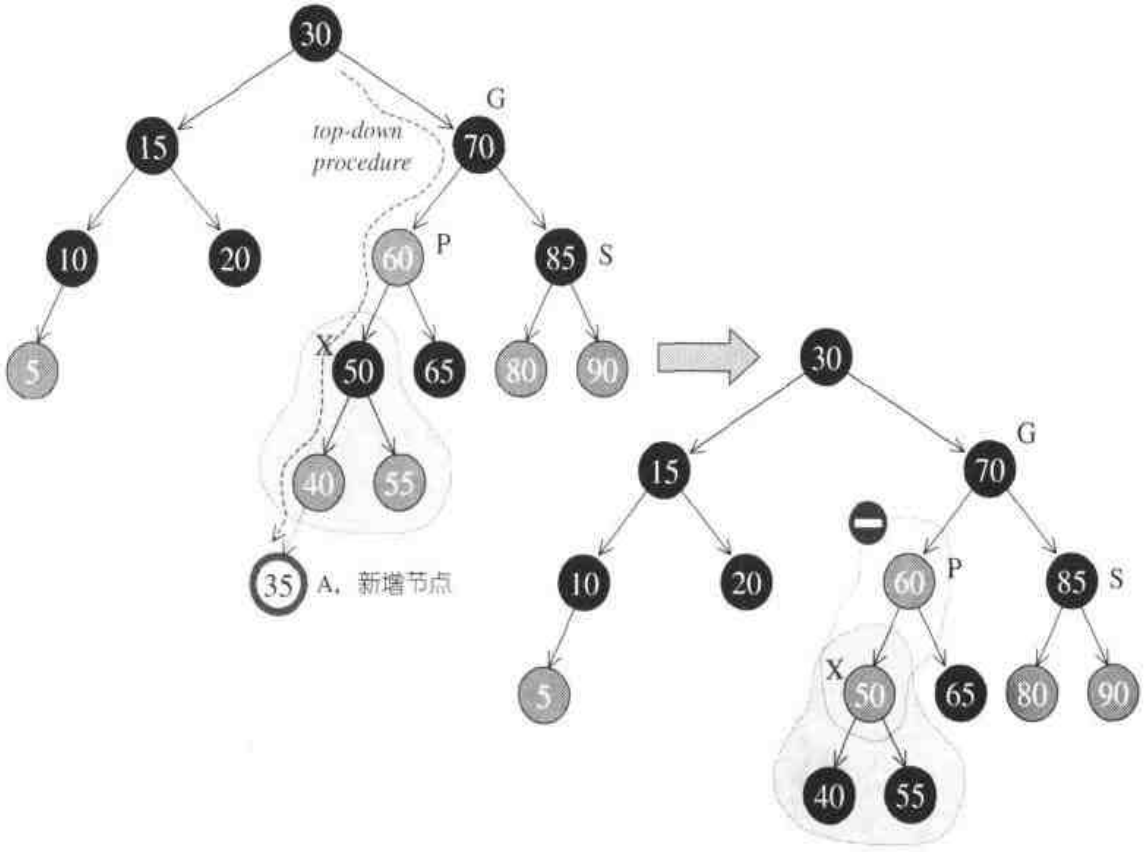


图 **5-15e** 沿着 X 的路径，由上而下修正节点颜色。

但是如果 A 的父节点 P 亦为红色（注意，此时 S 绝不可能为红），就得像状况 1 一样地做一次单旋转并改变颜色，或是像状况 2 一样地做一次双旋转并改变颜色。

在此之后，节点 35 的插入就很单纯了：要么直接插入，要么插入后再一次单旋转即可，如图 5-15f 所示。

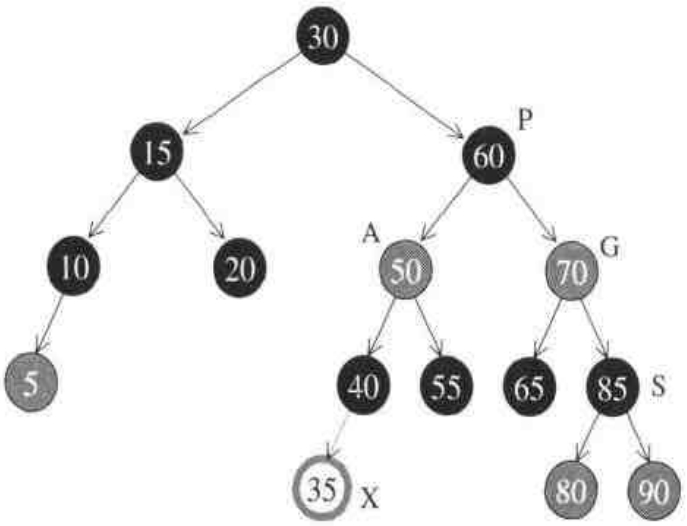


图 **5-15f** 延续图 5-15e 右侧状态，对 G, P 做一次右旋转，并改变颜色。

5.2.3 RB-tree 的节点设计

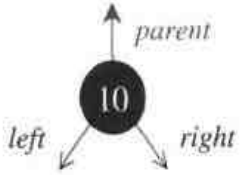
RB-tree 有红黑二色，并且拥有左右子节点，我们很容易就可以勾勒出其结构风貌。下面是 SGI STL 的实现源代码。为了有更大的弹性，节点分为两层，稍后图 5-17 将显示节点双层结构和迭代器双层结构的关系。

从以下的 minimum() 和 maximum() 函数可清楚看出，**RB-tree** 作为一个二叉搜索树，其极值是多么容易找到。由于 **RB-tree** 的各种操作时常需要上溯其父节点，所以特别在数据结构中安排了一个 parent 指针。

```
typedef bool __rb_tree_color_type;
const __rb_tree_color_type __rb_tree_red = false;  // 红色为 0
const __rb_tree_color_type __rb_tree_black = true; // 黑色为 1

struct __rb_tree_node_base
{
  typedef __rb_tree_color_type color_type;
  typedef __rb_tree_node_base* base_ptr;

  color_type color;    // 节点颜色，非红即黑
  base_ptr parent;     // RB 树的许多操作，必须知道父节点
  base_ptr left;       // 指向左节点
  base_ptr right;      // 指向右节点

  static base_ptr minimum(base_ptr x)
  {
```

```
    while (x->left != 0) x = x->left;      // 一直向左走，就会找到最小值，
    return x;                              // 这是二叉搜索树的特性
  }

  static base_ptr maximum(base_ptr x)
  {
    while (x->right != 0) x = x->right;    // 一直向右走，就会找到最大值，
    return x;                              // 这是二叉搜索树的特性
  }
};

template <class Value>
struct __rb_tree_node : public __rb_tree_node_base
{
  typedef __rb_tree_node<Value>* link_type;
  Value value_field;  // 节点值
};
```

下面是RB-tree的节点图标，其中将 __rb_tree_node::value_field填为10：



5.2.4 RB-tree 的迭代器

要成功地将 RB-tree 实现为一个泛型容器，迭代器的设计是一个关键。首先我们要考虑它的类别（category），然后要考虑它的前进（increment）、后退（decrement）、提领（dereference）、成员访问（member access）等操作。

为了更大的弹性，SGI 将 RB-tree 迭代器实现为两层，这种设计理念和 4.9 节的 slist 类似。图 5-16 所示的便是双层节点结构和双层迭代器结构之间的关系，其中主要意义是：__rb_tree_node 继承自 __rb_tree_node_base，__rb_tree_iterator 继承自 __rb_tree_base_iterator。有了这样的认识，我们就可以将迭代器稍做转型，然后解开 RB-tree 的所有奥秘[5]，追踪其一切状态。

[5] 因为不论是 rb-tree 的节点或迭代器，都是以 struct 完成，而 struct 的所有成员都是 public，可被外界自由取用。

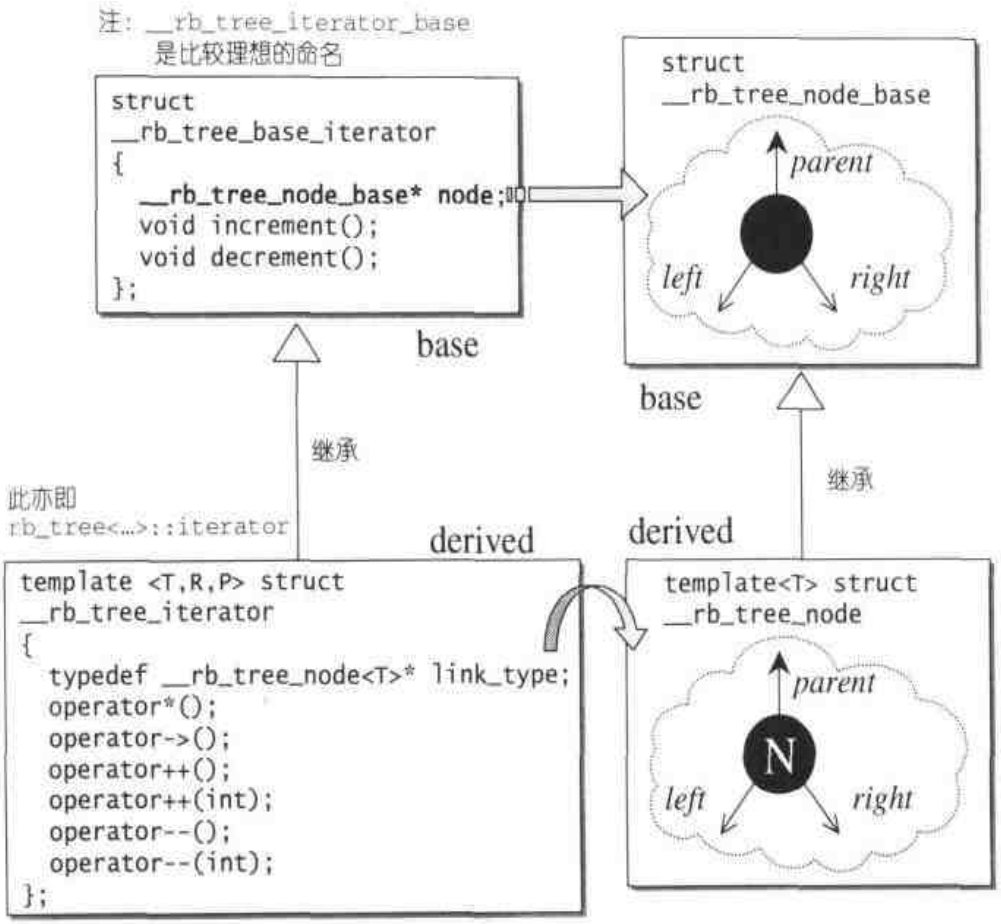


图 5-16 RB-tree 的节点和迭代器之间的关系。

这种双层架构和 4.9 节的 slist 极相似，请参考图 4-25。

RB-tree 迭代器属于双向迭代器，但不具备随机定位能力，其提领操作和成员访问操作与 list 十分近似，较为特殊的是其前进和后退操作。注意，RB-tree 迭代器的前进操作 operator++() 调用了基层迭代器的 increment()，RB-tree 迭代器的后退操作 operator--() 则调用了基层迭代器的 decrement()。前进或后退的举止行为完全依据二叉搜索树的节点排列法则，再加上实现上的某些特殊技巧。我加注于这两个函数内的说明，适足以说明其操作原则。至于实现上的特殊技巧（针对根节点），稍后另有说明。

```
// 基层迭代器
struct __rb_tree_base_iterator
{
  typedef __rb_tree_node_base::base_ptr base_ptr;
  typedef bidirectional_iterator_tag iterator_category;
```

```
typedef ptrdiff_t difference_type;

base_ptr node;  // 它用来与容器之间产生一个连结关系(make a reference)

// 以下其实可实现于 operator++ 内，因为再无他处会调用此函数了
void increment()
{
  if (node->right != 0) {       // 如果有右子节点。状况(1)
    node = node->right;         // 就向右走
    while (node->left != 0)     // 然后一直往左子树走到底
      node = node->left;        // 即是解答
  }
  else {                        // 没有右子节点。状况(2)
    base_ptr y = node->parent; // 找出父节点
    while (node == y->right) { // 如果现行节点本身是个右子节点，
      node = y;                 // 就一直上溯，直到“不为右子节点”止
      y = y->parent;
    }
    if (node->right != y)       // 若此时的右子节点不等于此时的父节点
      node = y;                 // 状况(3) 此时的父节点即为解答
                                // 否则此时的 node 为解答。状况(4)
  }
  // 注意，以上判断“若此时的右子节点不等于此时的父节点”，是为了应付一种
  // 特殊情况：我们欲寻找根节点的下一节点，而恰巧根节点无右子节点
  // 当然，以上特殊做法必须配合 RB-tree 根节点与特殊之 header 之间的
  // 特殊关系
}

// 以下其实可实现于 operator-- 内，因为再无他处会调用此函数了
void decrement()
{
  if (node->color == __rb_tree_red &&   // 如果是红节点，且
      node->parent->parent == node)     // 父节点的父节点等于自己，
    node = node->right;                 // 状况(1) 右子节点即为解答
  // 以上情况发生于 node 为 header 时（亦即 node 为 end() 时）
  // 注意，header 之右子节点即 mostright，指向整棵树的 max 节点
  else if (node->left != 0) {           // 如果有左子节点。状况(2)
    base_ptr y = node->left;            // 令 y 指向左子节点
    while (y->right != 0)               // 当 y 有右子节点时
      y = y->right;                     // 一直往右子节点走到底
    node = y;                           // 最后即为答案
  }
  else {                                // 既非根节点，亦无左子节点
    base_ptr y = node->parent;          // 状况(3) 找出父节点
    while (node == y->left) {           // 当现行节点身为左子节点
      node = y;                         // 一直交替往上走，直到现行节点
      y = y->parent;                    // 不为左子节点
    }
    node = y;                           // 此时之父节点即为答案
```

```
    }
  }
};

// RB-tree 的正规迭代器
template <class Value, class Ref, class Ptr>
struct __rb_tree_iterator : public __rb_tree_base_iterator
{
  typedef Value value_type;
  typedef Ref reference;
  typedef Ptr pointer;
  typedef __rb_tree_iterator<Value, Value&, Value*>    iterator;
  typedef __rb_tree_iterator<Value, const Value&, const Value*> const_iterator;
  typedef __rb_tree_iterator<Value, Ref, Ptr>  self;
  typedef __rb_tree_node<Value>* link_type;

  __rb_tree_iterator() {}
  __rb_tree_iterator(link_type x) { node = x; }
  __rb_tree_iterator(const iterator& it) { node = it.node; }

  reference operator*() const { return link_type(node)->value_field; }
#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
#endif /* __SGI_STL_NO_ARROW_OPERATOR */

  self& operator++() { increment(); return *this; }
  self operator++(int) {
    self tmp = *this;
    increment();
    return tmp;
  }

  self& operator--() { decrement(); return *this; }
  self operator--(int) {
    self tmp = *this;
    decrement();
    return tmp;
  }
};
```

在__rb_tree_iterator_base 的 increment() 和 decrement() 两函数中，较令人费解的是前者的状况 4 和后者的状况 1（见源代码注释标示），它们分别发生于图 5-17 所展示的状态下。

当迭代器指向根节点而后者无右子节点时，若对迭代器进行++操作，会进入 __rb_tree_base_iterator::increment() 的状况 (2),(4)。

当迭代器为 end()时，若对它进行 操作，会进入 __rb_tree_base_iterator::decrement() 的状况 (1)。

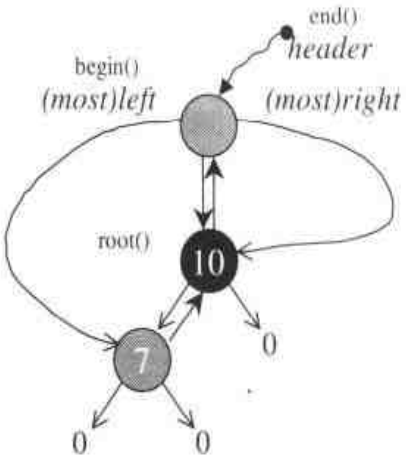


图 **5-17** increment() 和 decrement() 两函数中较令人费解的状况 4 和状况 1，其中的 header 是实现上的特殊技巧，见稍后说明。

5.2.5 RB-tree 的数据结构

下面是 rb-tree 的定义。你可以看到其中定义有专属的空间配置器，每次用来配置一个节点大小，也可以看到各种型别定义，用来维护整棵 RB-tree 的三笔数据（其中有个仿函数，functor，用来表现节点的大小比较方式），以及一些 member functions 的定义或声明。

```
template <class Key, class Value, class KeyOfValue, class Compare,
          class Alloc = alloc>
class rb_tree {
protected:
  typedef void* void_pointer;
  typedef __rb_tree_node_base* base_ptr;
  typedef __rb_tree_node<Value> rb_tree_node;
  typedef simple_alloc<rb_tree_node, Alloc> rb_tree_node_allocator;
  typedef __rb_tree_color_type color_type;
public:
  // 注意，没有定义 iterator（不，定义在后面！）
  typedef Key key_type;
  typedef Value value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef rb_tree_node* link_type;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
protected:
```

```
  link_type get_node() { return rb_tree_node_allocator::allocate(); }
  void put_node(link_type p) { rb_tree_node_allocator::deallocate(p); }

  link_type create_node(const value_type& x) {
    link_type tmp = get_node();              // 配置空间
    __STL_TRY {
      construct(&tmp->value_field, x);       // 构造内容
    }
    __STL_UNWIND(put_node(tmp));
    return tmp;
  }

  link_type clone_node(link_type x) {        // 复制一个节点（的值和色）
    link_type tmp = create_node(x->value_field);
    tmp->color = x->color;
    tmp->left = 0;
    tmp->right = 0;
    return tmp;
  }

  void destroy_node(link_type p) {
    destroy(&p->value_field);    // 析构内容
    put_node(p);                 // 释放内存
  }

protected:
  // RB-tree 只以三笔数据表现
  size_type node_count; // 追踪记录树的大小（节点数量）
  link_type header;     // 这是实现上的一个技巧
  Compare key_compare; // 节点间的键值大小比较准则。应该会是个 function object

  // 以下三个函数用来方便取得 header 的成员
  link_type& root() const { return (link_type&) header->parent; }
  link_type& leftmost() const { return (link_type&) header->left; }
  link_type& rightmost() const { return (link_type&) header->right; }

  // 以下六个函数用来方便取得节点 x 的成员
  static link_type& left(link_type x)
    { return (link_type&)(x->left); }
  static link_type& right(link_type x)
    { return (link_type&)(x->right); }
  static link_type& parent(link_type x)
    { return (link_type&)(x->parent); }
  static reference value(link_type x)
    { return x->value_field; }
  static const Key& key(link_type x)
    { return KeyOfValue()(value(x)); }
  static color_type& color(link_type x)
    { return (color_type&)(x->color); }
```

```
  // 以下六个函数用来方便取得节点 x 的成员
  static link_type& left(base_ptr x)
    { return (link_type&)(x->left); }
  static link_type& right(base_ptr x)
    { return (link_type&)(x->right); }
  static link_type& parent(base_ptr x)
    { return (link_type&)(x->parent); }
  static reference value(base_ptr x)
    { return ((link_type)x)->value_field; }
  static const Key& key(base_ptr x)
    { return KeyOfValue()(value(link_type(x)));}
  static color_type& color(base_ptr x)
    { return (color_type&)(link_type(x)->color); }

  // 求取极大值和极小值。node class 有实现此功能，交给它们完成即可
  static link_type minimum(link_type x) {
    return (link_type)  __rb_tree_node_base::minimum(x);
  }
  static link_type maximum(link_type x) {
    return (link_type) __rb_tree_node_base::maximum(x);
  }

public:
  typedef __rb_tree_iterator<value_type, reference, pointer> iterator;

private:
  iterator __insert(base_ptr x, base_ptr y, const value_type& v);
  link_type __copy(link_type x, link_type p);
  void __erase(link_type x);
  void init() {
    header = get_node();    // 产生一个节点空间，令 header 指向它
    color(header) = __rb_tree_red; // 令 header 为红色，用来区分 header
                                   // 和 root，在 iterator.operator++ 之中
    root() = 0;
    leftmost() = header;    // 令 header 的左子节点为自己
    rightmost() = header;   // 令 header 的右子节点为自己
  }

public:
                            // allocation/deallocation
  rb_tree(const Compare& comp = Compare())
    : node_count(0), key_compare(comp) { init(); }

  ~rb_tree() {
    clear();
    put_node(header);
  }
  rb_tree<Key, Value, KeyOfValue, Compare, Alloc>&
```

```
  operator=(const rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& x);

public:
                              // accessors:
  Compare key_comp() const { return key_compare; }
  iterator begin() { return leftmost(); }  // RB 树的起头为最左（最小）节点处
  iterator end() { return header; }  // RB 树的终点为 header 所指处
  bool empty() const { return node_count == 0; }
  size_type size() const { return node_count; }
  size_type max_size() const { return size_type(-1); }

public:
                              // insert/erase
  // 将 x 插入到 RB-tree 中（保持节点值独一无二）
  pair<iterator,bool> insert_unique(const value_type& x);
  // 将 x 插入到 RB-tree 中（允许节点值重复）。
  iterator insert_equal(const value_type& x);
...
};
```

5.2.6 RB-tree 的构造与内存管理

下面是 RB-tree 所定义的专属空间配置器 rb_tree_node_allocator，每次可恰恰配置一个节点。它所使用的 simple_alloc<> 定义于第二章：

```
template <class Key, class Value, class KeyOfValue, class Compare,
          class Alloc = alloc>
class rb_tree {
protected:
  typedef __rb_tree_node<Value> rb_tree_node;
  typedef simple_alloc<rb_tree_node, Alloc> rb_tree_node_allocator;
...
};
```

先前所列的程序片段也显示了数个节点相关函数，如 get_node()，put_node()，create_node()，clone_node()，destroy_node()。

RB-tree 的构造方式有两种，一种是以现有的 RB-tree 复制一个新的 RB-tree，另一种是产生一棵空空如也的树，如下所示：

```
rb_tree<int, int, identity<int>, less<int> > itree;
```
[6]

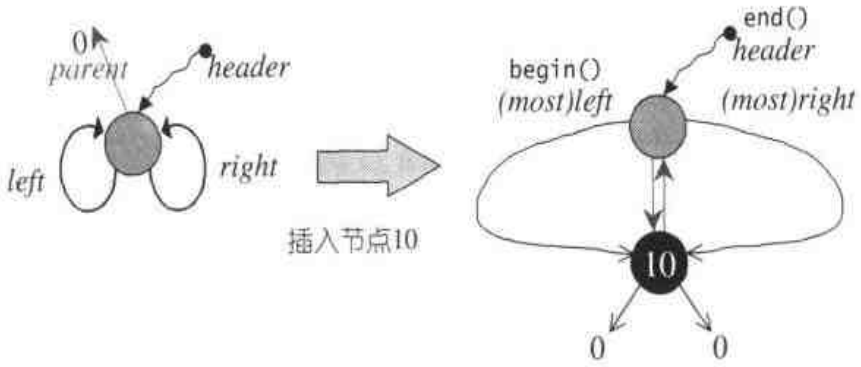
这行程序代码分别指定了节点的键值、实值、大小比较标准… 然后调用 RB-tree 的 default constructor：

```
rb_tree(const Compare& comp = Compare())
  : node_count(0), key_compare(comp) { init(); }
```

其中的 init() 是实现技巧上的一个关键点：

```
private:
  void init() {
    header = get_node();    // 产生一个节点空间，令 header 指向它
    color(header) = __rb_tree_red; // 令 header 为红色，用来区分 header
                                   // 和 root（在 iterator.operator++ 中）
    root() = 0;
    leftmost() = header;    // 令 header 的左子节点为自己
    rightmost() = header;   // 令 header 的右子节点为自己
  }
```

我们知道，树状结构的各种操作，最需注意的就是边界情况的发生，也就是走到根节点时要有特殊的处理。为了简化处理，SGI STL 特别为根节点再设计一个父节点，名为 header，并令其初始状态如图 5-18 所示。



注意，header和 root 互为对方的父节点，这是一种实现技巧

图 **5-18** 图左是 RB-tree 的初始状态，图右为加入第一个节点后的状态。

[6] 注意，RB-tree 并未明列于 STL 标准规格之中，我们能够这么用，是因为我们现在已经相当了解 SGI STL。

接下来，每当插入新节点时，不但要依照 RB-tree 的规则来调整，并且维护 header 的正确性，使其父节点指向根节点，左子节点指向最小节点，右子节点指向最大节点。节点的插入所带来的影响，是下一小节的描述重点。

5.2.7 RB-tree 的元素操作

本节主要只谈元素（节点）的插入和搜寻。RB-tree 提供两种插入操作：insert_unique() 和 insert_equal()，前者表示被插入节点的键值（*key*）在整棵树中必须独一无二（因此，如果树中已存在相同的键值，插入操作就不会真正进行），后者表示被插入节点的键值在整棵树中可以重复，因此，无论如何插入都会成功（除非空间不足导致配置失败）。这两个函数都有数个版本，以下以最简单的版本（单一参数，用以表现将被插入的节点实值（*value*））作为说明对象。注意，虽然只指定实值，但 RB-tree 一开始即要求用户必须明确设定所谓的 KeyOfValue 仿函数，因此，从实值（*value*）中取出键值（*key*）是毫无问题的。

元素插入操作 insert_equal()

```
// 插入新值：节点键值允许重复
// 注意，返回值是一个 RB-tree 迭代器，指向新增节点
template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::insert_equal(const Value& v)
{
  link_type y = header;
  link_type x = root();     // 从根节点开始
  while (x != 0) {          // 从根节点开始，往下寻找适当的插入点
    y = x;
    x = key_compare(KeyOfValue()(v), key(x)) ? left(x) : right(x);
    // 以上，遇 "大" 则往左，遇 "小于或等于" 则往右
  }
  return __insert(x, y, v);
  // 以上，x 为新值插入点，y 为插入点之父节点，v 为新值
}
```

元素插入操作 insert_unique()

```
// 插入新值：节点键值不允许重复，若重复则插入无效
// 注意，返回值是个pair，第一元素是个 RB-tree 迭代器，指向新增节点，
// 第二元素表示插入成功与否
template <class Key, class Value, class KeyOfValue,
          class Compare, class Alloc>
pair<typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator,
     bool>
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::
    insert_unique(const Value& v)
{
  link_type y = header;
  link_type x = root();     // 从根节点开始
  bool comp = true;
  while (x != 0) {          // 从根节点开始，往下寻找适当的插入点
    y = x;
    comp = key_compare(KeyOfValue()(v), key(x)); // v 键值小于目前节点之键值?
    x = comp ? left(x) : right(x);   // 遇 "大"则往左，遇 "小于或等于"则往右
  }
  // 离开 while 循环之后，y 所指即插入点之父节点（此时的它必为叶节点）

  iterator j = iterator(y);   // 令迭代器 j 指向插入点之父节点 y
  if (comp)  // 如果离开 while 循环时 comp 为真（表示遇 "大"，将插入于左侧）
    if (j == begin())   // 如果插入点之父节点为最左节点
      return pair<iterator,bool>(__insert(x, y, v), true);
      // 以上，x 为插入点，y 为插入点之父节点，v 为新值
    else // 否则（插入点之父节点不为最左节点）
      --j;    // 调整 j，回头准备测试...
  if (key_compare(key(j.node), KeyOfValue()(v)))
    // 小于新值（表示遇 "小"，将插入于右侧）
    return pair<iterator,bool>(__insert(x, y, v), true);
    // 以上，x 为新值插入点，y 为插入点之父节点，v 为新值

  // 进行至此，表示新值一定与树中键值重复，那么就不该插入新值
  return pair<iterator,bool>(j, false);
}
```

真正的插入执行程序 __insert()

```
template <class Key, class Value, class KeyOfValue,
          class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::
    __insert(base_ptr x_, base_ptr y_, const Value& v) {
// 参数 x_ 为新值插入点，参数 y_ 为插入点之父节点，参数 v 为新值
  link_type x = (link_type) x_;
  link_type y = (link_type) y_;
```

```
  link_type z;

  // key_compare 是键值大小比较准则。应该会是个 function object
  if (y == header || x != 0 || key_compare(KeyOfValue()(v), key(y))) {
    z = create_node(v);  // 产生一个新节点
    left(y) = z;         // 这使得当 y 即为 header 时，leftmost() = z
    if (y == header) {
      root() = z;
      rightmost() = z;
    }
    else if (y == leftmost())    // 如果 y 为最左节点
      leftmost() = z;            // 维护 leftmost()，使它永远指向最左节点
  }
  else {
    z = create_node(v);          // 产生一个新节点
    right(y) = z;                // 令新节点成为插入点之父节点 y 的右子节点
    if (y == rightmost())
      rightmost() = z;           // 维护 rightmost()，使它永远指向最右节点
  }
  parent(z) = y;          // 设定新节点的父节点
  left(z) = 0;            // 设定新节点的左子节点
  right(z) = 0;           // 设定新节点的右子节点
                          // 新节点的颜色将在 __rb_tree_rebalance() 设定（并调整）
  __rb_tree_rebalance(z, header->parent);// 参数一为新增节点，参数二为 root
  ++node_count;           // 节点数累加
  return iterator(z);  // 返回一个迭代器，指向新增节点
}
```

调整 RB-tree（旋转及改变颜色）

任何插入操作，于节点插入完毕后，都要做一次调整操作，将树的状态调整到符合RB-tree的要求。__rb_tree_rebalance() 是具备如此能力的一个全局函数：

```
// 全局函数
// 重新令树形平衡（改变颜色及旋转树形）
// 参数一为新增节点，参数二为 root
inline void
__rb_tree_rebalance(__rb_tree_node_base* x, __rb_tree_node_base*& root)
{
  x->color = __rb_tree_red;     // 新节点必为红
  while (x != root && x->parent->color == __rb_tree_red) { // 父节点为红
    if (x->parent == x->parent->parent->left) { // 父节点为祖父节点之左子节点
      __rb_tree_node_base* y = x->parent->parent->right;// 令 y 为伯父节点
      if (y && y->color == __rb_tree_red) {          // 伯父节点存在，且为红
        x->parent->color = __rb_tree_black;          // 更改父节点为黑
        y->color = __rb_tree_black;                  // 更改伯父节点为黑
        x->parent->parent->color = __rb_tree_red;    // 更改祖父节点为红
        x = x->parent->parent;
```

```
    }
    else { // 无伯父节点，或伯父节点为黑
      if (x == x->parent->right) { // 如果新节点为父节点之右子节点
        x = x->parent;
        __rb_tree_rotate_left(x, root); // 第一参数为左旋点
      }
      x->parent->color = __rb_tree_black;     // 改变颜色
      x->parent->parent->color = __rb_tree_red;
      __rb_tree_rotate_right(x->parent->parent, root); // 第二参数为右旋点
    }
  }
  else {   // 父节点为祖父节点之右子节点
    __rb_tree_node_base* y = x->parent->parent->left; // 令y为伯父节点
    if (y && y->color ==   rb_tree_red) {          // 有伯父节点，且为红
      x->parent->color = __rb_tree_black;          // 更改父节点为黑
      y->color = __rb_tree_black;                  // 更改伯父节点为黑
      x->parent->parent->color = __rb_tree_red;    // 更改祖父节点为红
      x = x->parent->parent;   // 准备继续往上层检查
    }
    else { // 无伯父节点，或伯父节点为黑
      if (x == x->parent->left) {   // 如果新节点为父节点之左子节点
        x = x->parent;
        __rb_tree_rotate_right(x, root);      // 第一参数为右旋点
      }
      x->parent->color = __rb_tree_black;     // 改变颜色
      x->parent->parent->color = __rb_tree_red;
      __rb_tree_rotate_left(x->parent->parent, root); // 第二参数为左旋点
    }
  }
 } // while 结束
 root->color = __rb_tree_black;     // 根节点永远为黑
}
```

这个树形调整操作，就是5.2.2节所说的那个“由上而下的程序”。从源代码清楚可见，某些时候只需调整节点颜色，某些时候要做单旋转，某些时候要做双旋转（两次单旋转）；某些时候要左旋，某些时候要右旋。下面是左旋函数和右旋函数：

```
// 全局函数
// 新节点必为红节点。如果插入处之父节点亦为红节点，就违反红黑树规则，此时必须
// 做树形旋转（及颜色改变，在程序它处）
inline void
__rb_tree_rotate_left(__rb_tree_node_base* x,
                      __rb_tree_node_base*& root)
{
  // x 为旋转点
  __rb_tree_node_base* y = x->right; // 令y为旋转点的右子节点
  x->right = y->left;
  if (y->left !=0)
```

```
    y->left->parent = x;          // 别忘了回马枪设定父节点
  y->parent = x->parent;

  // 令 y 完全顶替 x 的地位（必须将 x 对其父节点的关系完全接收过来）
  if (x == root)                          // x 为根节点
    root = y;
  else if (x == x->parent->left)          // x 为其父节点的左子节点
    x->parent->left = y;
  else                                    // x 为其父节点的右子节点
    x->parent->right = y;
  y->left = x;
  x->parent = y;
}

// 全局函数
// 新节点必为红节点。如果插入处之父节点亦为红节点，就违反红黑树规则，此时必须
// 做树形旋转（及颜色改变，在程序其它处）
inline void
__rb_tree_rotate_right(  rb_tree_node_base* x,
                         __rb_tree_node_base*& root)
{
  // x 为旋转点
  __rb_tree_node_base* y = x->left;  // y 为旋转点的左子节点
  x->left = y->right;
  if (y->right != 0)
    y->right->parent = x;   // 别忘了回马枪设定父节点
  y >parent = x->parent;

  // 令 y 完全顶替 x 的地位（必须将 x 对其父节点的关系完全接收过来）
  if (x == root)                          // x 为根节点
    root = y;
  else if (x == x->parent->right)         // x 为其父节点的右子节点
    x->parent->right = y;
  else                                    // x 为其父节点的左子节点
    x->parent->left = y;
  y->right = x;
  x->parent = y;
}
```

下面是客户端程序连续插入数个元素到 RB-tree 中并加以测试的过程：

```
// file: 5rbtree-test.cpp
rb_tree<int, int, identity<int>, less<int> > itree;
  cout << itree.size() << endl;       // 0
```

```
// 以下注释中所标示的函数名称，是我修改 <stl_tree.h>[7]，令三个函数
// 打印出函数名称而后得
itree.insert_unique(10);      // __rb_tree_rebalance
itree.insert_unique(7);       // __rb_tree_rebalance
itree.insert_unique(8);       // __rb_tree_rebalance
                                  //   __rb_tree_rotate_left
                                  //   __rb_tree_rotate_right
itree.insert_unique(15);      // __rb_tree_rebalance
itree.insert_unique(5);       // __rb_tree_rebalance
itree.insert_unique(6);       // __rb_tree_rebalance
                                  //   __rb_tree_rotate_left
                                  //   __rb_tree_rotate_right
itree.insert_unique(11);      // __rb_tree_rebalance
                                  //   __rb_tree_rotate_right
                                  //   __rb_tree_rotate_left
itree.insert_unique(13);      // __rb_tree_rebalance
itree.insert_unique(12);

cout << itree.size() << endl;  // 9
for(; ite1 != ite2; ++ite1)
   cout << *ite1 << ' '; // 5 6 7 8 10 11 12 13 15
cout << endl;

// 测试颜色和 operator++ (亦即  rb_tree_iterator_base::increment)
rb_tree<int, int, identity<int>, less<int> >::iterator
  ite1=itree.begin();
rb_tree<int, int, identity<int>, less<int> >::iterator
  ite2=itree.end();
__rb_tree_base_iterator rbtite;[8]

for(; ite1 != ite2; ++ite1) {
   rbtite = __rb_tree_base_iterator(ite1);
   // 以上，向上转型 up-casting，永远没问题。见《多型与虚拟 2/e》第三章
   cout << *ite1 << '(' << rbtite.node->color << ") ";
}
cout << endl;
// 结果：5(0) 6(1) 7(0) 8(1) 10(1) 11(0) 12(0) 13(1) 15(0)
```

图 5-19 是上述程序操作的完整图标，一步一步展现 RB-tree 的成长与调整。

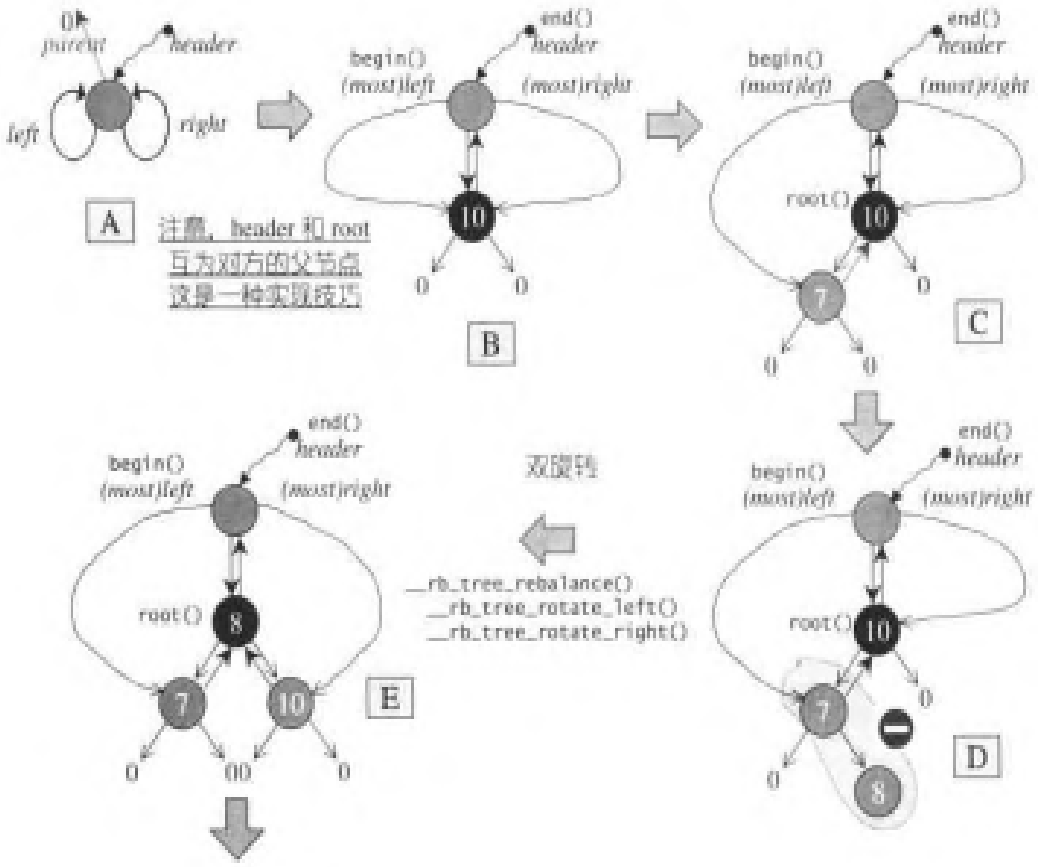
[7] 注意，如果你要像我一样，修改 SGI STL 源代码，请注意备份并谨慎行事。

[8] __rb_tree_base_iterator 是 SGI STL 内部使用的东西。此处为了测试（为了直接取得节点颜色），所以在程序中取用之。当然这是完全合法的，不需修改任何 STL 源代码。

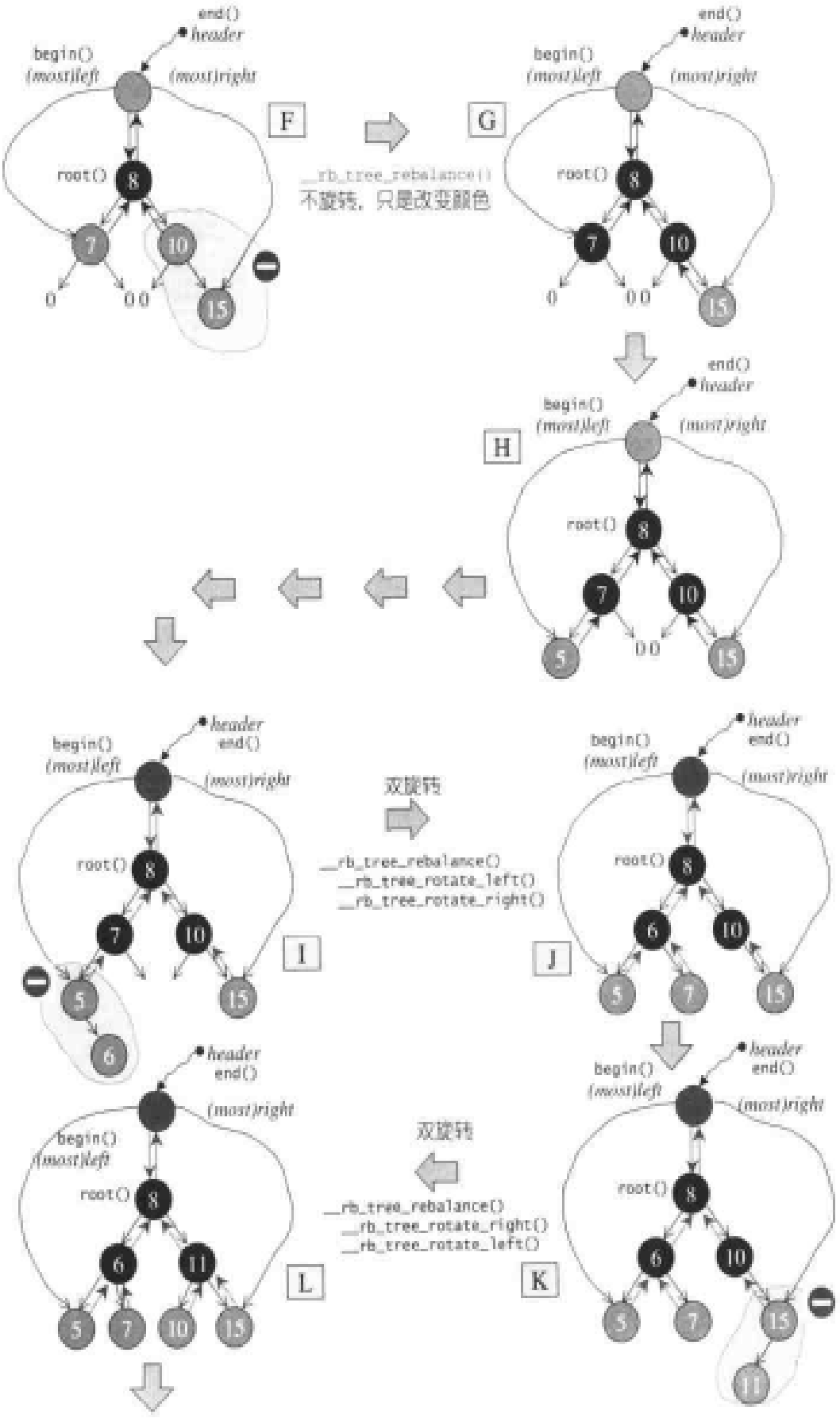
元素的搜寻

RB-tree 是一个二叉搜索树，元素的搜寻正是其拿手项目。以下是 RB-tree 提供的 find 函数：

```
// 寻找 RB 树中是否有键值为 k 的节点
template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::find(const Key& k) {
  link_type y = header;       // Last node which is not less than k.
  link_type x = root();       // Current node.

  while (x != 0)
    // 以下，key_compare 是节点键值大小比较准则。应该会是个 function object。
    if (!key_compare(key(x), k))
      // 进行到这里，表示 x 键值大于 k。遇到大值就向左走
      y = x, x = left(x);   // 注意语法！
    else
      // 进行到这里，表示 x 键值小于 k。遇到小值就向右走
      x = right(x);

  iterator j = iterator(y);
  return (j == end() || key_compare(k, key(j.node))) ? end() : j;
}
```


0
parent
header
left
right
A
注意，header 和 root
互为对方的父节点
这是一种实现技巧
begin()
(most)left
end()
header
(most)right
10
0
0
B
root()
7
C
8
D
E
双旋转
__rb_tree_rebalance()
__rb_tree_rotate_left()
__rb_tree_rotate_right()

续下页


end()
header
begin()
(most)left
(most)right
root()
F
G
__rb_tree_rebalance()
不旋转，只是改变颜色
H
I
双旋转
__rb_tree_rebalance()
__rb_tree_rotate_left()
__rb_tree_rotate_right()
J
K
双旋转
__rb_tree_rebalance()
__rb_tree_rotate_right()
__rb_tree_rotate_left()
L

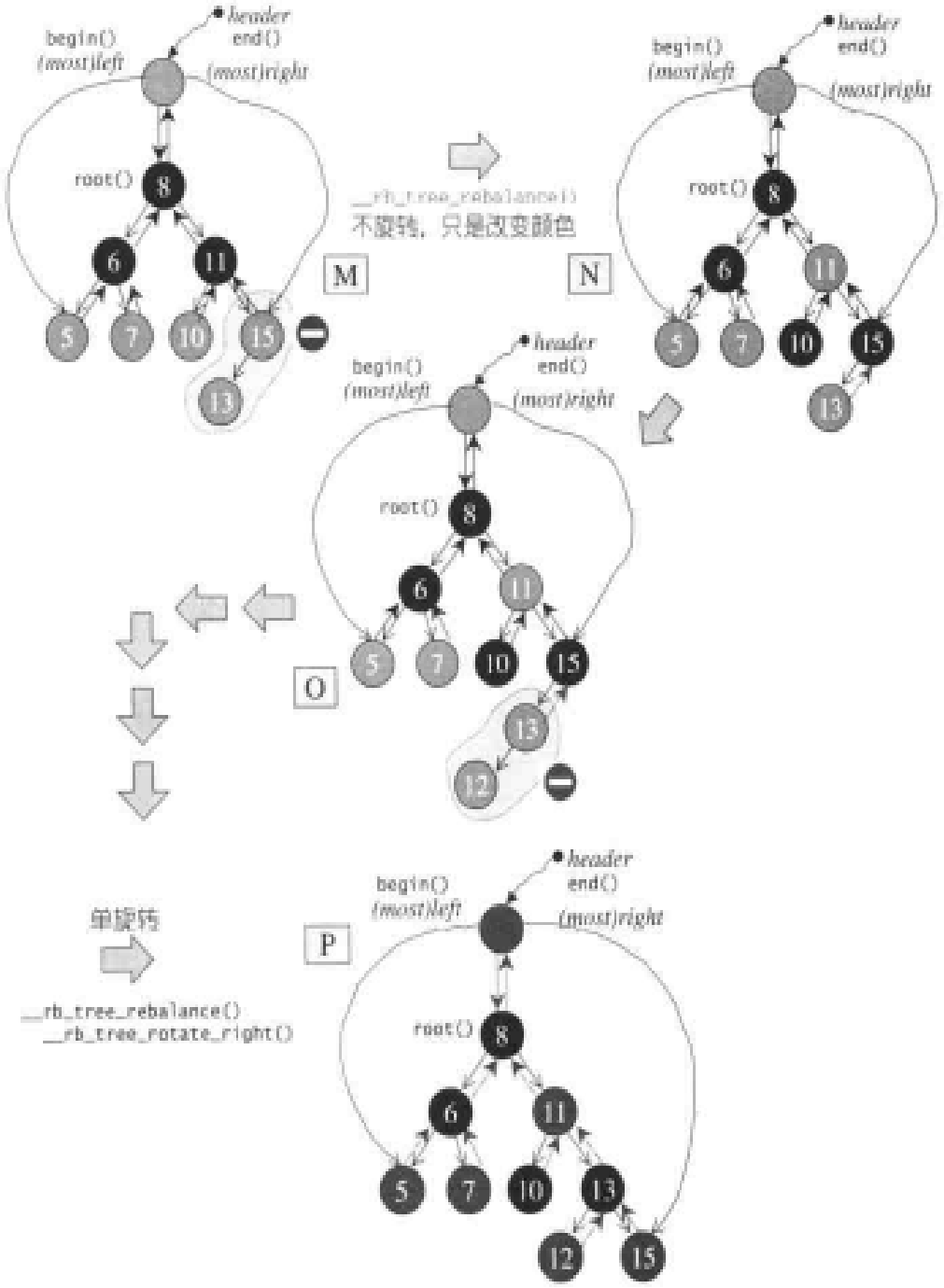


图 **5-19** 依序将 10,7,8,15,5,6,11,13,12 插入至 RB-tree，一步一步的成长与调整。此图根据前述之 5rbtree-test.cpp 实例绘制而成。

5.3 set

set 的特性是，所有元素都会根据元素的键值自动被排序。set 的元素不像 map 那样可以同时拥有实值（*value*）和键值（*key*），set 元素的键值就是实值，实值就是键值。set 不允许两个元素有相同的键值。

我们可以通过 set的迭代器改变 set 的元素值吗？不行，因为 set 元素值就是其键值，关系到 set 元素的排列规则。如果任意改变 set 元素值，会严重破坏 set 组织。稍后你会在 set 源代码之中看到，`set<T>::iterator` 被定义为底层 RB-tree 的 `const_iterator`，杜绝写入操作。换句话说，set iterators 是一种 constant iterators（相对于 mutable iterators）。

set 拥有与 list 相同的某些性质：当客户端对它进行元素新增操作（insert）或删除操作（erase）时，操作之前的所有迭代器，在操作完成之后都依然有效。当然，被删除的那个元素的迭代器必然是个例外。

STL 特别提供了一组 set/multiset 相关算法，包括交集 `set_intersection`，联集 `set_union`、差集 `set_difference`，对称差集 `set_symmetric_difference`，详见 6.5 节。

由于 RB-tree 是一种平衡二叉搜索树，自动排序的效果很不错，所以标准的 STL set 即以 RB-tree 为底层机制[9]。又由于 set 所开放的各种操作接口，RB-tree 也都提供了，所以几乎所有的 set 操作行为，都只是转调用 RB-tree 的操作行为而已。

下面是 set 的源代码摘录，其中的注释几乎说明了一切，本节不再另做文字解释。

```
template <class Key,
          class Compare = less<Key>,      // 缺省情况下采用递增排序
          class Alloc = alloc>
class set {
public:
  // typedefs:
```

[9] SGI 另提供一种以 hash-table 为底层机制的 set，称为 hash_set，详见 5.8 节。

```
  typedef Key key_type;
  typedef Key value_type;
  // 注意，以下 key_compare 和 value_compare 使用同一个比较函数
  typedef Compare key_compare;
  typedef Compare value_compare;
private:
  // 注意，以下的 identity 定义于 <stl_function.h>，参见第 7 章，其定义为：
  /*
    template <class T>
    struct identity : public unary_function<T, T> {
      const T& operator()(const T& x) const { return x; }
    };
  */
  typedef rb_tree<key_type, value_type,
                  identity<value_type>, key_compare, Alloc> rep_type;
  rep_type t;  // 采用红黑树 (RB-tree) 来表现 set
public:
  typedef typename rep_type::const_pointer pointer;
  typedef typename rep_type::const_pointer const_pointer;
  typedef typename rep_type::const_reference reference;
  typedef typename rep_type::const_reference const_reference;
  typedef typename rep_type::const_iterator iterator;
  // 注意上一行，iterator 定义为 RB-tree 的 const_iterator，这表示 set 的
  // 迭代器无法执行写入操作，这是因为 set 的元素有一定次序安排
  // 不允许用户在任意处进行写入操作
  typedef typename rep_type::const_iterator const_iterator;
  typedef typename rep_type::const_reverse_iterator reverse_iterator;
  typedef typename rep_type::const_reverse_iterator const_reverse_iterator;
  typedef typename rep_type::size_type size_type;
  typedef typename rep_type::difference_type difference_type;

  // allocation/deallocation
  // 注意， set 一定使用 RB-tree 的 insert_unique() 而非 insert_equal()
  // multiset 才使用 RB-tree 的 insert_equal()
  // 因为 set 不允许相同键值存在，multiset 才允许相同键值存在
  set() : t(Compare()) {}
  explicit set(const Compare& comp) : t(comp) {}

  template <class InputIterator>
  set(InputIterator first, InputIterator last)
    : t(Compare()) { t.insert_unique(first, last); }

  template <class InputIterator>
  set(InputIterator first, InputIterator last, const Compare& comp)
    : t(comp) { t.insert_unique(first, last); }

  set(const set<Key, Compare, Alloc>& x) : t(x.t) {}
  set<Key, Compare, Alloc>& operator=(const set<Key, Compare, Alloc>& x) {
    t = x.t;
```

```
  return *this;
}

// 以下所有的 set 操作行为，RB-tree 都已提供，所以 set 只要传递调用即可

// accessors:
key_compare key_comp() const { return t.key_comp(); }
// 以下注意，set 的 value_comp() 事实上为 RB-tree 的 key_comp()
value_compare value_comp() const { return t.key_comp(); }
iterator begin() const { return t.begin(); }
iterator end() const { return t.end(); }
reverse_iterator rbegin() const { return t.rbegin(); }
reverse_iterator rend() const { return t.rend(); }
bool empty() const { return t.empty(); }
size_type size() const { return t.size(); }
size_type max_size() const { return t.max_size(); }
void swap(set<Key, Compare, Alloc>& x) { t.swap(x.t); }

// insert/erase
typedef  pair<iterator, bool> pair_iterator_bool;
pair<iterator,bool> insert(const value_type& x) {
  pair<typename rep_type::iterator, bool> p = t.insert_unique(x);
  return pair<iterator, bool>(p.first, p.second);
}
iterator insert(iterator position, const value_type& x) {
  typedef typename rep_type::iterator rep_iterator;
  return t.insert_unique((rep_iterator&)position, x);
}
template <class InputIterator>
void insert(InputIterator first, InputIterator last) {
  t.insert_unique(first, last);
}
void erase(iterator position) {
  typedef typename rep_type::iterator rep_iterator;
  t.erase((rep_iterator&)position);
}
size_type erase(const key_type& x) {
  return t.erase(x);
}
void erase(iterator first, iterator last) {
  typedef typename rep_type::iterator rep_iterator;
  t.erase((rep_iterator&)first, (rep_iterator&)last);
}
void clear() { t.clear(); }

// set operations:
iterator find(const key_type& x) const { return t.find(x); }
size_type count(const key_type& x) const { return t.count(x); }
iterator lower_bound(const key_type& x) const {
```

```
    return t.lower_bound(x);
  }
  iterator upper_bound(const key_type& x) const {
    return t.upper_bound(x);
  }
  pair<iterator,iterator> equal_range(const key_type& x) const {
    return t.equal_range(x);
  }
  // 以下的__STL_NULL_TMPL_ARGS 被定义为 <>，详见 2.9.1 节
  friend bool operator== __STL_NULL_TMPL_ARGS (const set&, const set&);
  friend bool operator< __STL_NULL_TMPL_ARGS (const set&, const set&);
};

template <class Key, class Compare, class Alloc>
inline bool operator==(const set<Key, Compare, Alloc>& x,
                       const set<Key, Compare, Alloc>& y) {
  return x.t == y.t;
}

template <class Key, class Compare, class Alloc>
inline bool operator<(const set<Key, Compare, Alloc>& x,
                      const set<Key, Compare, Alloc>& y) {
  return x.t < y.t;
}
```

下面是一个小小的 set 测试程序：

```
// file: 5set-test.cpp
#include <set>
#include <iostream>
using namespace std;

int main()
{
  int i;
  int ia[5] = { 0,1,2,3,4};
  set<int> iset(ia, ia+5);

  cout << "size=" << iset.size() << endl;          // size=5
  cout << "3 count=" << iset.count(3) << endl;     // 3 count=1
  iset.insert(3);
  cout << "size=" << iset.size() << endl;          // size=5
  cout << "3 count=" << iset.count(3) << endl;     // 3 count=1
  iset.insert(5);
  cout << "size=" << iset.size() << endl;          // size=6
  cout << "3 count=" << iset.count(3) << endl;     // 3 count=1
  iset.erase(1);
  cout << "size=" << iset.size() << endl;          // size=5
```

```
    cout << "3 count=" << iset.count(3) << endl;     // 3 count=1
    cout << "1 count=" << iset.count(1) << endl;     // 1 count=0

    set<int>::iterator ite1=iset.begin();
    set<int>::iterator ite2=iset.end();
    for(; ite1 != ite2; ++ite1)
       cout << *ite1;
    cout << endl;                                    // 0 2 3 4 5


    // 使用 STL 算法 find() 来搜寻元素，可以有效运作，但不是好办法
    ite1 = find(iset.begin(), iset.end(), 3);
    if (ite1 != iset.end())
       cout << "3 found" << endl;                    // 3 found

    ite1 = find(iset.begin(), iset.end(), 1);
    if (ite1 == iset.end())
       cout << "1 not found" << endl;                // 1 not found

    // 面对关联式容器，应该使用其所提供的 find 函数来搜寻元素，会比
    // 使用 STL 算法 find() 更有效率。因为 STL 算法 find() 只是循序搜寻
    ite1 = iset.find(3);
    if (ite1 != iset.end())
       cout << "3 found" << endl;                    // 3 found

    ite1 = iset.find(1);
    if (ite1 == iset.end())
       cout << "1 not found" << endl;                // 1 not found

    // 企图通过迭代器来改变 set 元素，是不被允许的
    *ite1 = 9;   // error, assignment of read-only location
}
```

5.4 map

map 的特性是，所有元素都会根据元素的键值自动被排序。map 的所有元素都是 pair，同时拥有实值（*value*）和键值（*key*）。pair 的第一元素被视为键值，第二元素被视为实值。map 不允许两个元素拥有相同的键值。下面是<stl_pair.h>中的 pair 定义：

```
template <class T1, class T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;        // 注意，它是 public
  T2 second;       // 注意，它是 public
```

```
  pair() : first(T1()), second(T2()) {}
  pair(const T1& a, const T2& b) : first(a), second(b) {}
};
```

我们可以通过 map 的迭代器改变 map 的元素内容吗？如果想要修正元素的键值，答案是不行，因为 map 元素的键值关系到 map 元素的排列规则。任意改变 map 元素键值将会严重破坏 map 组织。但如果想要修正元素的实值，答案是可以，因为 map 元素的实值并不影响 map 元素的排列规则。因此，map iterators 既不是一种 constant iterators，也不是一种 mutable iterators。

map 拥有和 list 相同的某些性质：当客户端对它进行元素新增操作（insert）或删除操作（erase）时，操作之前的所有迭代器，在操作完成之后都依然有效 当然，被删除的那个元素的迭代器必然是个例外。

由于 RB-tree 是一种平衡二叉搜索树，自动排序的效果很不错，所以标准的 STL map 即以 RB-tree 为底层机制[10]，又由于 map 所开放的各种操作接口，RB-tree 也都提供了，所以几乎所有的 map 操作行为，都只是转调用 RB-tree 的操作行为而已。

图 5-20 说明 map 的架构。下页是 map 源代码摘录，其中注释说明了一切。

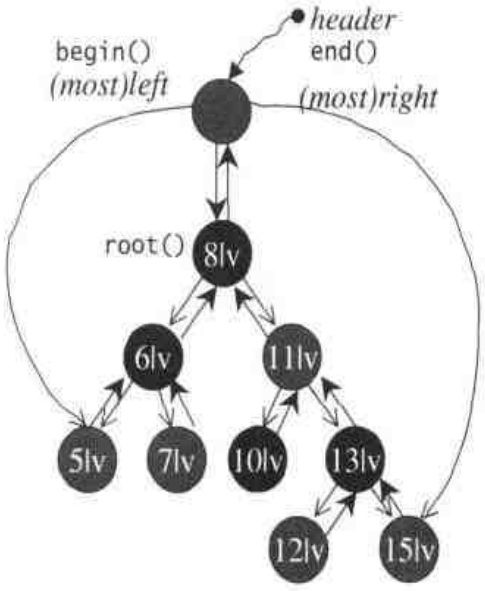


图 5-20 SGI STL map 以红黑树为底层机制，每个节点的内容是一个 pair pair 的第一元素被视为键值（*key*），第二元素被视为实值（*value*）

[10] SGI 另提供一种以 hash table 为底层机制的 map，称为 hash_map，详见 5.9 节。

```
// 注意，以下 Key 为键值（key）型别，T 为实值（value）型别

template <class Key, class T,
          class Compare = less<Key>,     // 缺省采用递增排序
          class Alloc = alloc>
class map {
public:

  // typedefs:
  typedef Key key_type;     // 键值型别
  typedef T data_type;      // 数据（实值）型别
  typedef T mapped_type;    //
  typedef pair<const Key, T> value_type; // 元素型别（键值/实值）
  typedef Compare key_compare;  // 键值比较函数

  // 以下定义一个 functor，其作用就是调用 “元素比较函数”
  class value_compare
    : public binary_function<value_type, value_type, bool> {
  friend class map<Key, T, Compare, Alloc>;
  protected :
    Compare comp;
    value_compare(Compare c) : comp(c) {}
  public:
    bool operator()(const value_type& x, const value_type& y) const {
      return comp(x.first, y.first);
    }
  };

private:
  // 以下定义表述型别（representation type），以 map 元素型别（一个 pair）
  // 的第一型别，作为 RB-tree 节点的键值型别
  typedef rb_tree<key_type, value_type,
                  select1st<value_type>, key_compare, Alloc> rep_type;
  rep_type t;  // 以红黑树（RB-tree）表现 map
public:
  typedef typename rep_type::pointer pointer;
  typedef typename rep_type::const_pointer const_pointer;
  typedef typename rep_type::reference reference;
  typedef typename rep_type::const_reference const_reference;
  typedef typename rep_type::iterator iterator;
  // 注意上一行，map 并不像 set 一样将 iterator 定义为 RB-tree 的
  // const_iterator，因为它允许用户通过其迭代器修改元素的实值（value）
  typedef typename rep_type::const_iterator const_iterator;
  typedef typename rep_type::reverse_iterator reverse_iterator;
  typedef typename rep_type::const_reverse_iterator const_reverse_iterator;
  typedef typename rep_type::size_type size_type;
  typedef typename rep_type::difference_type difference_type;

  // allocation/deallocation
```

```
// 注意， map 一定使用底层 RB-tree 的 insert_unique() 而非 insert_equal()
// multimap 才使用 insert_equal()
// 因为 map 不允许相同键值存在，multimap 才允许相同键值存在

map() : t(Compare()) {}
explicit map(const Compare& comp) : t(comp) {}

template <class InputIterator>
map(InputIterator first, InputIterator last)
  : t(Compare()) { t.insert_unique(first, last); }

template <class InputIterator>
map(InputIterator first, InputIterator last, const Compare& comp)
  : t(comp) { t.insert_unique(first, last); }

map(const map<Key, T, Compare, Alloc>& x) : t(x.t) {}
map<Key, T, Compare, Alloc>& operator=(const map<Key, T, Compare, Alloc>& x)
{
  t = x.t;
  return *this;
}

// accessors:
// 以下所有的 map 操作行为，RB-tree 都已提供，map 只要转调用即可

key_compare key_comp() const { return t.key_comp(); }
value_compare value_comp() const { return value_compare(t.key_comp()); }
iterator begin() { return t.begin(); }
const_iterator begin() const { return t.begin(); }
iterator end() { return t.end(); }
const_iterator end() const { return t.end(); }
reverse_iterator rbegin() { return t.rbegin(); }
const_reverse_iterator rbegin() const { return t.rbegin(); }
reverse_iterator rend() { return t.rend(); }
const_reverse_iterator rend() const { return t.rend(); }
bool empty() const { return t.empty(); }
size_type size() const { return t.size(); }
size_type max_size() const { return t.max_size(); }
// 注意以下 下标 (subscript) 操作符
T& operator[](const key_type& k) {
  return (*((insert(value_type(k, T()))).first)).second;
}
void swap(map<Key, T, Compare, Alloc>& x) { t.swap(x.t); }

// insert/erase

// 注意以下 insert 操作返回的型别
pair<iterator,bool> insert(const value_type& x) {
  return t.insert_unique(x); }
```

```
  iterator insert(iterator position, const value_type& x) {
    return t.insert_unique(position, x);
  }
  template <class InputIterator>
  void insert(InputIterator first, InputIterator last) {
    t.insert_unique(first, last);
  }

  void erase(iterator position) { t.erase(position); }
  size_type erase(const key_type& x) { return t.erase(x); }
  void erase(iterator first, iterator last) { t.erase(first, last); }
  void clear() { t.clear(); }

  // map operations:

  iterator find(const key_type& x) { return t.find(x); }
  const_iterator find(const key_type& x) const { return t.find(x); }
  size_type count(const key_type& x) const { return t.count(x); }
  iterator lower_bound(const key_type& x) {return t.lower_bound(x); }
  const_iterator lower_bound(const key_type& x) const {
    return t.lower_bound(x);
  }
  iterator upper_bound(const key_type& x) {return t.upper_bound(x); }
  const_iterator upper_bound(const key_type& x) const {
    return t.upper_bound(x);
  }

  pair<iterator,iterator> equal_range(const key_type& x) {
    return t.equal_range(x);
  }
  pair<const_iterator,const_iterator> equal_range(const key_type& x) const {
    return t.equal_range(x);
  }
  friend bool operator== __STL_NULL_TMPL_ARGS (const map&, const map&);
  friend bool operator< __STL_NULL_TMPL_ARGS (const map&, const map&);
};

template <class Key, class T, class Compare, class Alloc>
inline bool operator==(const map<Key, T, Compare, Alloc>& x,
                       const map<Key, T, Compare, Alloc>& y) {
  return x.t == y.t;
}

template <class Key, class T, class Compare, class Alloc>
inline bool operator<(const map<Key, T, Compare, Alloc>& x,
                      const map<Key, T, Compare, Alloc>& y) {
  return x.t < y.t;
}
```

下面是一个小小的测试程序：

```
// file: 5map-test.cpp
#include <map>
#include <iostream>
#include <string>
using namespace std;

int main()
{
  map<string, int> simap;        // 以 string 为键值，以 int 为实值
  simap[string("jjhou")]  = 1;   // 第 1 对内容是 ("jjhou", 1)
  simap[string("jerry")]  = 2;   // 第 2 对内容是 ("jerry", 2)
  simap[string("jason")]  = 3;   // 第 3 对内容是 ("jason", 3)
  simap[string("jimmy")]  = 4;   // 第 4 对内容是 ("jimmy", 4)

  pair<string, int> value(string("david"),5);
  simap.insert(value);

  map<string, int>::iterator simap_iter = simap.begin();
  for (; simap_iter != simap.end(); ++simap_iter)
    cout << simap_iter->first << ' '
         << simap_iter->second << endl;
                                    // david 5
                                    // jason 3
                                    // jerry 2
                                    // jimmy 4
                                    // jjhou 1

  int number = simap[string("jjhou")];
  cout << number << endl;           // 1

  map<string, int>::iterator ite1;

  // 面对关联式容器，应该使用其所提供的 find 函数来搜寻元素，会比
  // 使用 STL 算法 find() 更有效率。因为 STL 算法 find() 只是循序搜寻
  ite1 = simap.find(string("mchen"));
  if (ite1 == simap.end())
     cout << "mchen not found" << endl;     // mchen not found

  ite1 = simap.find(string("jerry"));
  if (ite1 != simap.end())
     cout << "jerry found" << endl;         // jerry found

  ite1->second = 9;   // 可以通过 map 迭代器修改 "value"(not key)
  int number2 = simap[string("jerry")];
  cout << number2 << endl;              // 9
}
```

我想针对其中使用的 insert() 函数及 subscript（下标）操作符做一些说明。首先是 insert() 函数：

```
// 注意以下 insert 操作返回的型别
pair<iterator,bool> insert(const value_type& x)
  { return t.insert_unique(x); }
```

此式将工作直接转给底层机制 RB-tree 的 insert_unique() 去执行，原也不必多说。要注意的是其返回值型别是一个 pair，由一个迭代器和一个 bool 值组成，后者表示插入成功与否，成功的话前者即指向被插入的那个元素。

至于 subscript（下标）操作符，用法有两种，可能作为左值运用（内容可被修改），也可能作为右值运用（内容不可被修改），例如：

```
map<string, int> simap;          // 以 string 为键值，以 int 为实值
simap[string("jjnou")] = 1;      // 左值运用
...
int number = simap[string("jjhou")]; // 右值运用
```

左值或右值都适用的关键在于，返回值采用 by reference 传递形式[11]。

无论如何，subscript 操作符的工作，都得先根据键值找出其实值，再做打算。下面是其实际操作：

```
template <class Key, class T,
          class Compare = less<Key>,
          class Alloc = alloc>
class map {
public:
// typedefs:
  typedef Key key_type;     // 键值型别
  typedef pair<const Key, T> value_type; // 元素型别（键值/实值）
  ...
public:
  T& operator[](const key_type& k) {
    return (*((insert(value_type(k, T()))).first)).second;   // (A)
  }
  ...
};
```

上述 (A) 式真是复杂，让我细说分明。首先，根据键值和实值做出一个元素，

11 可参考 [Meyers96] 条款 30: *proxy classes*.

由于实值未知，所以产生一个与实值型别相同的暂时对象[12] 替代：

```
value_type(k, T())
```

再将该元素插入到 map 里面去：

```
insert(value_type(k, T()))
```

插入操作返回一个 pair，其第一元素是个迭代器，指向插入妥当的新元素，或指向插入失败点（键值重复）的旧元素。注意，如果下标操作符作为左值运用（通常表示要填加新元素），我们正好以此 “实值待填”的元素将位置卡好；如果下标操作符作为右值运用（通常表示要根据键值取其实值），此时的插入操作所返回的 pair 的第一元素（是个迭代器）恰指向键值符合的旧元素。

现在我们取插入操作所返回的 pair 的第一元素：

```
(insert(value_type(k, T()))).first
```

这第一元素是个迭代器，指向被插入的元素。现在，提领该迭代器：

```
*((insert(value_type(k, T()))).first)
```

获得一个 map 元素，是一个由键值和实值组成的 pair。取其第二元素，即为实值：

```
(*((insert(value_type(k, T()))).first)).second;
```

注意，这个实值以 by reference 方式传递，所以它作为左值或右值都可以。这便是 (A) 式的最后形式。

---

12 这种语法不常见到，但在 STL 的运用中（尤其是 functor）却很频繁。详见第 7 章。

5.5 multiset

multiset 的特性以及用法和 set 完全相同，唯一的差别在于它允许键值重复，因此它的插入操作采用的是底层机制 RB-tree 的 insert_equal() 而非 insert_unique()。下面是 multiset的源代码提要，只列出了与 set 不同之处：

```
template <class Key, class Compare = less<Key>, class Alloc = alloc>
class multiset {
public:
  // typedefs:
  ... (与 set 相同)

  // allocation/deallocation
  // 注意，multiset 一定使用 insert_equal() 而不使用 insert_unique()
  // set 才使用 insert_unique()

  template <class InputIterator>
  multiset(InputIterator first, InputIterator last)
    : t(Compare()) { t.insert_equal(first, last); }
  template <class InputIterator>
  multiset(InputIterator first, InputIterator last, const Compare& comp)
    : t(comp) { t.insert_equal(first, last); }
  ... (其它与 set 相同)

  // insert/erase
  iterator insert(const value_type& x) {
    return t.insert_equal(x);
  }
  iterator insert(iterator position, const value_type& x) {
    typedef typename rep_type::iterator rep_iterator;
    return t.insert_equal((rep_iterator&)position, x);
  }

  template <class InputIterator>
  void insert(InputIterator first, InputIterator last) {
    t.insert_equal(first, last);
  }
  ... (其它与 set 相同)
```

5.6 multimap

multimap 的特性以及用法与 map 完全相同，唯一的差别在于它允许键值重复，因此它的插入操作采用的是底层机制 RB-tree 的 insert_equal() 而非 insert_unique()。下面是 multimap 的源代码提要，只列出了与 map 不同之处：

```
template <class Key, class T, class Compare = less<Key>, class Alloc = alloc>
class multimap {
public:
// typedefs:
  ... (与 set 相同)

  // allocation/deallocation
  // 注意， multimap 一定使用 insert_equal() 而不使用 insert_unique()
  // map 才使用 insert_unique()

  template <class InputIterator>
  multimap(InputIterator first, InputIterator last)
    : t(Compare()) { t.insert_equal(first, last); }

  template <class InputIterator>
  multimap(InputIterator first, InputIterator last, const Compare& comp)
    : t(comp) { t.insert_equal(first, last); }
  ... (其它与 map 相同)

  // insert/erase

  iterator insert(const value_type& x) { return t.insert_equal(x); }
  iterator insert(iterator position, const value_type& x) {
    return t.insert_equal(position, x);
  }
  template <class InputIterator>
  void insert(InputIterator first, InputIterator last) {
    t.insert_equal(first, last);
  }
  ... (其它与 map 相同)
```

5.7 hashtable

5.1.1 节介绍了所谓的二叉搜索树，5.1.2 节介绍了所谓的平衡二叉搜索树，5.2 节则是十分详细地介绍了一种被广泛运用的平衡二叉搜索树：RB-tree（红黑树）。RB-tree 不仅在树形的平衡上表现不错，在效率表现和实现复杂度上也保持相当的“平衡”☺，所以运用甚广，也因此成为 STL set 和 map 的标准底层机制。

二叉搜索树具有对数平均时间（logarithmic average time）的表现，但这样的表现构造在一个假设上：输入数据有足够的随机性。这一节要介绍一种名为 hash table（散列表）的数据结构，这种结构在插入、删除、搜寻等操作上也具有“常数平均时间”的表现，而且这种表现是以统计为基础，不需仰赖输入元素的随机性。

5.7.1 hashtable 概述

hash table 可提供对任何有名项（named item）的存取操作和删除操作。由于操作对象是有名项，所以 hashtable 也可被视为一种字典结构（dictionary）。这种结构的用意在于提供常数时间之基本操作，就像 stack 或 queue 那样。乍听之下这几乎是不可能的任务，因为约束制条件如此之少，而元素个数增加，搜寻操作必定耗费更多时间。

倒也不尽然。

举个例子，如果所有的元素都是 16-bits 且不带正负号的整数，范围 0~65535，那么简单地运用一个 array 就可以满足上述期望。首先配置一个 array A，拥有 65536 个元素，索引号码 0~65535，初值全部为 0，如图 5-21。每一个元素值代表相应元素的出现次数。如果插入元素 i，我们就执行 A[i]++，如果删除元素 i，我们就执行 A[i]--，如果搜寻元素 i，我们就检查 A[i] 是否为 0。以上的每一个操作都是常数时间。这种解法的额外负担（overhead）是 array 的空间和初始化时间。

hashing 5
hashing 8
hashing 3
hashing 8
hashing 58
hashing 65535
hashing 65534

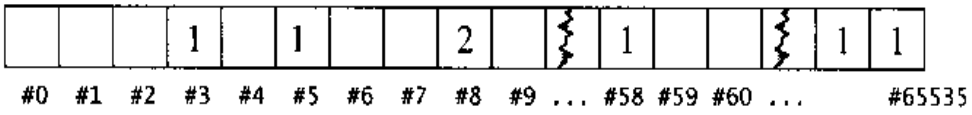


图 **5-21** 如果所有的元素都是 16-bits 且不带正负号的整数，我们可以一个拥有 65536 个元素的 array A，初值全部为 0，每个元素值代表相应元素的出现次数。于是，不论是插入、删除、搜寻，每个操作都在常数时间内完成。

这个解法存在两个现实问题。第一，如果元素是 32-bits 而非 16-bits，我们所准备的 array A 的大小就必须是 2^{32} = 4GB，这就大得不切实际了。第二，如果元素型态是字符串（或其它）而非整数，将无法被拿来作为 array 的索引。

第二个问题（关于索引）不难解决。就像数值 1234 是由阿拉伯数字 1,2,3,4 构成一样，字符串 "jjhou" 是由字符 'j','j','h','o','u' 构成。那么，既然数值 1234 是 $1*10^3+2*10^2+3*10^1+4*10^0$，我们也可以把字符编码，每个字符以一个 7-bits 数值来表示（也就是 ASCII 编码），从而将字符串 "jjhou" 表现为：

$$'j'*128^4 + 'j'*128^3 + 'h'*128^2 + 'o'*128^1 + 'u'*128^0$$

于是先前的 array 实现法就可适用于“元素型别为字符串”的情况了。但这并不实用，因为这会产生出非常巨大的数值。"jjhou" 的索引值将是：

$$106*128^4 + 106*128^3 + 104*128^2 + 111*128^1 + 117*128^0 = 28678174709$$

这太不切实际了。更长的字符串会导致更大的索引值！这就回归到第一个问题：array 的大小。

如何避免使用一个大得荒谬的 array 呢？办法之一就是使用某种映射函数，将大数映射为小数。负责将某一元素映射为一个“大小可接受之索引”，这样的

函数称为 hash function(散列函数)。例如,假设 x 是任意整数,TableSize 是 array 大小,则 x%TableSize 会得到一个整数,范围在 0 ~ TableSize-1 之间,恰可作为表格(也就是 array)的索引。

使用 hash function 会带来一个问题:可能有不同的元素被映射到相同的位置(亦即有相同的索引)。这无法避免,因为元素个数大于 array 容量。这便是所谓的"碰撞(collision)"问题。解决碰撞问题的方法有许多种,包括线性探测(linear probing)、二次探测(quadratic probing)、开链(separate chaining)…等做法。每种方法都很容易,导出的效率各不相同——与 array 的填满程度有很大的关连。

线性探测(linear probing)

首先让我们认识一个名词:负载系数(**loading factor**),意指元素个数除以表格大小。负载系数永远在 0~1 之间——除非采用开链(separate chaining)策略,后述。

当 hash function 计算出某个元素的插入位置,而该位置上的空间已不再可用时,我们应该怎么办?最简单的办法就是循序往下一一寻找(如果到达尾端,就绕到头部继续寻找),直到找到一个可用空间为止。只要表格(亦即 array)足够大,总是能够找到一个安身立命的空间,但是要花多少时间就很难说了。进行元素搜寻操作时,道理也相同,如果 hash function 计算出来的位置上的元素值与我们的搜寻目标不符,就循序往下一一寻找,直到找到吻合者,或直到遇上空格元素。至于元素的删除,必须采用惰性删除(lazy deletion)[13],也就是只标记删除记号,实际删除操作则待表格重新整理(rehashing)时再进行——这是因为 hash table 中的每一个元素不仅表述它自己,也关系到其它元素的排列。

图 5-22 是线性探测的一个实例。

[13] 请参考 [meyers96] item17: *Consider using lazy evaluation.*

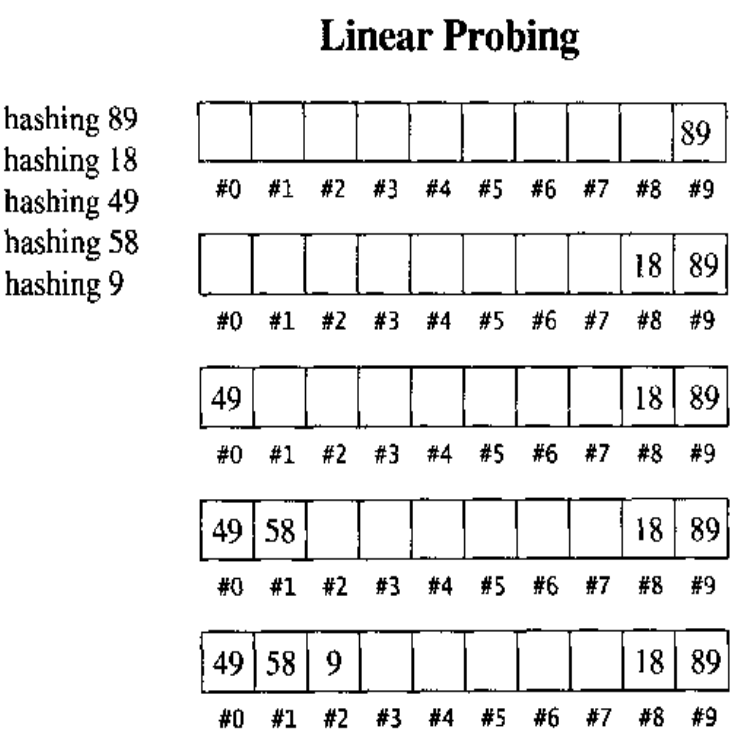


图 **5-22** 线性探测（linear probing）。依序插入 5 个元素，array 的变化。

欲分析线性探测的表现，需要两个假设：(1) 表格足够大；(2) 每个元素都够独立。在此假设之下，最坏的情况是线性巡访整个表格，平均情况则是巡访一半表格，这已经和我们所期望的常数时间天差地远了，而实际情况犹更糟糕。问题出在上述第二个假设太过于天真。

拿实际例子来说，接续图 5-22 的最后状态，除非新元素经过 hash function 的计算之后直接落在位置 #4 ~ #7，否则位置 #4 ~ #7 永远不可能被运用，因为位置 #3 永远是第一考虑。换句话说，新元素不论是 8,9,0,1,2,3 中的哪一个，都会落在位置 #3 上。新元素如果是 4 或 5，或 6，或 7，才会各自落在位置 #4，或 #5，或 #6，或 #7 上。这很清楚地突显了一个问题：平均插入成本的成长幅度，远高于负载系数的成长幅度。这样的现象在 hashing 过程中称为主集团（**primary clustering**）。此时的我们手上有的是一大团已被用过的方格，插入操作极有可能在主集团所形成的泥泞中奋力爬行，不断解决碰撞问题，最后才射门得分，但是却又助长了主集团的泥泞面积。

二次探测（quadratic probing）

二次探测主要用来解决主集团（primary clustering）的问题。其命名由来是因为解决碰撞问题的方程式 `F(i) = i^2` 是个二次方程式。更明确地说，如果 hash function 计算出新元素的位置为 `H`，而该位置实际上已被使用，那么我们就依序尝试 $H+1^2$, $H+2^2$, $H+3^2$, $H+4^2$, …, $H+i^2$，而不是像线性探测那样依序尝试 `H+1`, `H+2`, `H+3`, `H+4`, …, `H+i`。图 5-23 所示的是二次探测的一个实例。

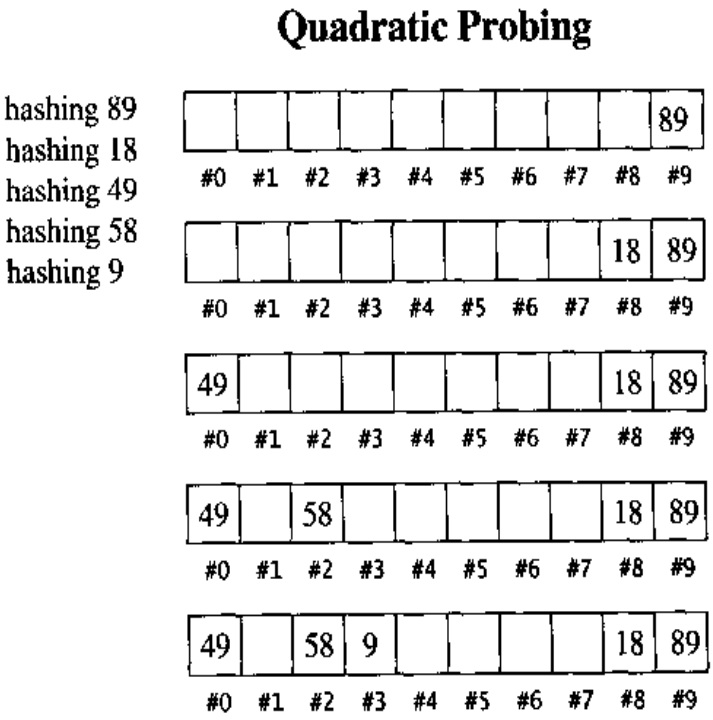


图 **5-23** 二次探测（linear probing）。依序插入 5 个元素，array 的变化。

二次探测带来一些疑问：

- 线性探测法每次探测的都必然是一个不同的位置，二次探测法是否能够保证如此？二次探测法是否能够保证如果表格之中没有 X，那么我们插入 X 一定能够成功？
- 线性探测法的运算过程极其简单，二次探测法则显然复杂得多。这是否会在执行效率上带来太多的负面影响？
- 不论线性探测或二次探测，当负载系数过高时，表格是否能够动态成长？

幸运的是，如果我们假设表格大小为质数（prime），而且永远保持负载系数在 0.5 以下（也就是说超过 0.5 就重新配置并重新整理表格），那么就可以确定每插入一个新元素所需要的探测次数不多于 2。

至于实现复杂度的问题，一般总是这样考虑：赚的比花的多，才值得去做。我们受累增加了探测次数，所获得的利益好歹总得多过二次函数计算所多花的时间，不能吃肥走瘦[14]，得不偿失。线性探测所需要的是一个加法（加 1）、一个测试（看是否需要绕转回头），以及一个偶需为之的减法（用以绕转回头）。二次探测需要的则是一个加法（从 i-1 到 i）、一个乘法（计算 i^2），另一个加法，以及一个 mod 运算。看起来很有得不偿失之嫌。然而这中间却有一些小技巧，可以去除耗时的乘法和除法：

$$H_i = H_2 + i^2 (\mathrm{mod}\ M)$$
$$H_{i-1} = H_2 + (i-1)^2 (\mathrm{mod}\ M)$$

整理可得：

$$H_i - H_{i-1} = i^2 - (i-1)^2 (\mathrm{mod}\ M)$$
$$H_i = H_{i-1} + 2i - 1 (\mathrm{mod}\ M)$$

因此，如果我们能够以前一个 H 值来计算下一个 H 值，就不需要执行二次方所需要的乘法了。虽然还是需要一个乘法，但那是乘以 2，可以位移位（bit shift）快速完成。至于 mod 运算，也可证明并非真有需要（本处略）。

最后一个问题是 array 的成长。欲扩充表格，首先必须找出下一个新的而且够大（大约两倍）的质数，然后必须考虑表格重建（*rehashing*）的成本——是的，不可能只是原封不动地拷贝而已，我们必须检验旧表格中的每一个元素，计算其在新表格中的位置，然后再插入到新表格中。

二次探测可以消除主集团（primary clustering），却可能造成次集团（**secondary clustering**）：两个元素经 hash function 计算出来的位置若相同，则插入时所探测的位置也相同，形成某种浪费。消除次集团的办法当然也有，例如复式散列（**double hashing**）。

---

14 吃肥走瘦是台湾俚语，意指走大老远路去吃一顿，所吃的还不够走路消耗的哩。

虽然目前还没有对二次探测有数学上的分析，不过，以上所有考虑加加减减起来，总体而言，二次探测仍然值得投资。

开链（separate chaining）

另一种与二次探测法分庭抗礼的，是所谓的开链（**separate chaining**）法。这种做法是在每一个表格元素中维护一个 list；hash function 为我们分配某一个 list，然后我们在那个 list 身上执行元素的插入、搜寻、删除等操作。虽然针对 list 而进行的搜寻只能是一种线性操作，但如果 list 够短，速度还是够快。

使用开链手法，表格的负载系数将大于 1。SGI STL 的 hash table 便是采用这种做法，稍后便有详细的实现介绍。

5.7.2 hashtable 的桶子（buckets）与节点（nodes）

图 5-24 是以开链法（separate chaining）完成 hash table 的图形表述。为了解说 SGI STL 源代码，我遵循 SGI 的命名，称 hash table 表格内的元素为桶子（bucket），此名称的大约意义是，表格内的每个单元，涵盖的不只是个节点（元素），甚且可能是一“桶”节点。

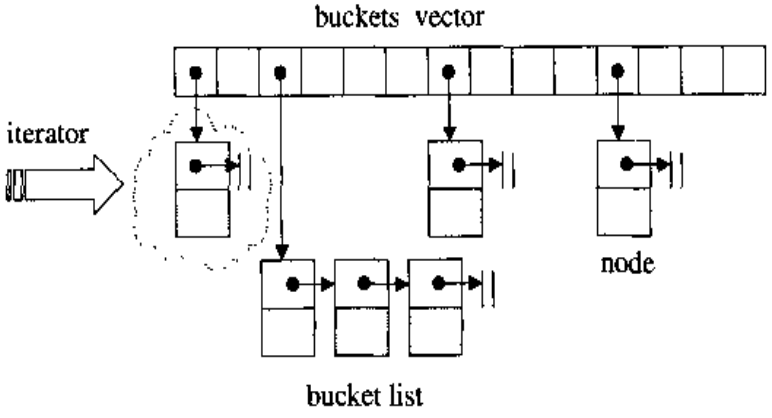
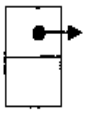


图 **5-24** 以开链（separate chaining）法完成的 hash table。SGI 即采此法。

下面是 hash table 的节点定义：

```
template <class Value>
struct __hashtable_node
{
```

```
  __hashtable_node* next;
  Value val;
};
```



这是一个 __hashtable_node object

注意，bucket 所维护的 linked list，并不采用 STL 的 list 或 slist，而是自行维护上述的 hash table node。至于 buckets 聚合体，则以 vector（4.2 节）完成，以便有动态扩充能力。稍后在 hash table 的定义式中我们可以清楚看到这一点。

5.7.3 hashtable 的迭代器

以下是 hash table 迭代器的定义：

```
template <class Value, class Key, class HashFcn,
          class ExtractKey, class EqualKey, class Alloc>
struct __hashtable_iterator {
  typedef hashtable<Value, Key, HashFcn, ExtractKey, EqualKey, Alloc>
        hashtable;
  typedef __hashtable_iterator<Value, Key, HashFcn,
                               ExtractKey, EqualKey, Alloc>
        iterator;
  typedef __hashtable_const_iterator<Value, Key, HashFcn,
                                     ExtractKey, EqualKey, Alloc>
        const_iterator;
  typedef __hashtable_node<Value> node;

  typedef forward_iterator_tag iterator_category;
  typedef Value value_type;
  typedef ptrdiff_t difference_type;
  typedef size_t size_type;
  typedef Value& reference;
  typedef Value* pointer;

  node* cur;      // 迭代器目前所指之节点
  hashtable* ht;  // 保持对容器的连结关系（因为可能需要从 bucket 跳到 bucket）

  __hashtable_iterator(node* n, hashtable* tab) : cur(n), ht(tab) {}
  __hashtable_iterator() {}
  reference operator*() const { return cur->val; }
  pointer operator->() const { return &(operator*()); }
  iterator& operator++();
  iterator operator++(int);
```

```
  bool operator==(const iterator& it) const { return cur == it.cur; }
  bool operator!=(const iterator& it) const { return cur != it.cur; }
};
```

注意，hasttable 迭代器必须永远维系着与整个 “buckets vector” 的关系，并记录目前所指的节点。其前进操作是首先尝试从目前所指的节点出发，前进一个位置（节点），由于节点被安置于 list 内，所以利用节点的 next 指针即可轻易达成前进操作。如果目前节点正巧是 list 的尾端，就跳至下一个 bucket 身上，那正是指向下一个 list 的头部节点。

```
template <class V, class K, class HF, class ExK, class EqK, class A>
__hashtable_iterator<V, K, HF, ExK, EqK, A>&
__hashtable_iterator<V, K, HF, ExK, EqK, A>::operator++()
{
  const node* old = cur;
  cur = cur->next;     // 如果存在，就是它。否则进入以下 if 流程
  if (!cur) {
    // 根据元素值，定位出下一个 bucket，其起头处就是我们的目的地
    size_type bucket = ht->bkt_num(old->val);
    while (!cur && ++bucket < ht->buckets.size())  // 注意，operator++
      cur = ht->buckets[bucket];
  }
  return *this;
}

template <class V, class K, class HF, class ExK, class EqK, class A>
inline __hashtable_iterator<V, K, HF, ExK, EqK, A>
__hashtable_iterator<V, K, HF, ExK, EqK, A>::operator++(int)
{
  iterator tmp = *this;
  ++*this;   // 调用 operator++()
  return tmp;
}
```

请注意，hashtable 的迭代器没有后退操作（operator--()），hashtable 也没有定义所谓的逆向迭代器（reverse iterator）。

5.7.4 hashtable 的数据结构

下面是 hashtable 的定义摘要，其中可见 buckets 聚合体以 vector 完成，以利动态扩充：

```
template <class Value, class Key, class HashFcn,
          class ExtractKey, class EqualKey, class Alloc = alloc>
class hashtable;

// ...

template <class Value, class Key, class HashFcn,
        class ExtractKey, class EqualKey,
        class Alloc>  // 先前声明时，已给予 Alloc 默认值 alloc
class hashtable {
public:
  typedef HashFcn hasher;       // 为 template 型别参数重新定义一个名称
  typedef EqualKey key_equal;   // 为 template 型别参数重新定义一个名称
  typedef size_t    size_type;

private:
  // 以下三者都是 function objects，<stl_hash_fun.h> 中定义有数个
  // 标准型别（如 int,c-style string 等）的 hasher
  hasher hash;
  key_equal equals;
  ExtractKey get_key;

  typedef __hashtable_node<Value> node;
  typedef simple_alloc<node, Alloc> node_allocator;

  vector<node*,Alloc> buckets;  // 以 vector 完成
  size_type num_elements;

public:
  // bucket 个数即 buckets vector 的大小
  size_type bucket_count() const { return buckets.size(); }
...
};
```

hashtable 的模板参数相当多，包括：

- Value：节点的实值型别。
- Key：节点的键值型别。
- HashFcn：hash function 的函数型别。
- ExtractKey：从节点中取出键值的方法（函数或仿函数）。

- EqualKey：判断键值相同与否的方法（函数或仿函数）。
- Alloc：空间配置器，缺省使用 std::alloc。

如果对 STL 的运用缺乏深厚的功力，不太容易给出正确的参数。稍后我会以一个小小的测试程序告诉大家如何在客户端应用程序中直接使用 hashtable。5.7.7 节的 <stl_hash_fun.h> 定义有数个现成的 hash functions，全都是仿函数（仿函数请见第 7 章）。注意，先前谈及概念时，我们说 hash function 是计算元素位置（如今应该说是 bucket 位置）的函数，SGI 将这项任务赋予 bkt_num()，由它调用 hash function 取得一个可以执行 modulus（取模）运算的值。稍后执行这项“计算元素位置”的任务时，我会再提醒你一次。

虽然开链法（separate chaining）并不要求表格大小必须为质数，但 SGI STL 仍然以质数来设计表格大小，并且先将 28 个质数（逐渐呈现大约两倍的关系）计算好，以备随时访问，同时提供一个函数，用来查询在这 28 个质数之中，“最接近某数并大于某数”的质数：

```
// 注意：假设 long 至少有 32 bits
static const int __stl_num_primes = 28;
static const unsigned long __stl_prime_list[__stl_num_primes] =
{
  53,         97,         193,         389,       769,
  1543,       3079,       6151,        12289,     24593,
  49157,      98317,      196613,      393241,    786433,
  1572869,    3145739,    6291469,     12582917,  25165843,
  50331653,   100663319,  201326611,   402653189, 805306457,
  1610612741, 3221225473ul, 4294967291ul
};

// 以下找出上述 28 个质数之中，最接近并大于 n 的那个质数
inline unsigned long __stl_next_prime(unsigned long n)
{
  const unsigned long* first = __stl_prime_list;
  const unsigned long* last = __stl_prime_list + __stl_num_primes;
  const unsigned long* pos = lower_bound(first, last, n);
  // 以上，lower_bound() 是泛型算法，见第 6 章
  // 使用 lower_bound()，序列需先排序。没问题，上述数组已排序
  return pos == last ? *(last - 1) : *pos;
}

  // 总共可以有多少 buckets。以下是 hast_table 的一个 member function
  size_type max_bucket_count() const
    { return __stl_prime_list[__stl_num_primes - 1]; }
      // 其值将为 4294967291
```

5.7.5 hashtable 的构造与内存管理

上一节 hashtable 定义式中展现了一个专属的节点配置器：

```
typedef simple_alloc<node, Alloc> node_allocator;
```

下面是节点配置函数与节点释放函数：

```
node* new_node(const value_type& obj)
{
  node* n = node_allocator::allocate();
  n->next = 0;
  __STL_TRY {
    construct(&n->val, obj);
    return n;
  }
  __STL_UNWIND(node_allocator::deallocate(n));
}

void delete_node(node* n)
{
  destroy(&n->val);
  node_allocator::deallocate(n);
}
```

当我们初始构造一个拥有 50 个节点的 hash table 如下：

```
// <value, key, hash-func, extract-key, equal-key, allocator>
// 注意：hash table 没有供应 default constructor
hashtable<int, int, hash<int>, identity<int>, equal_to<int>, alloc>
  iht(50, hash<int>(), equal_to<int>());

cout<< iht.size() << endl;              // 0
cout<< iht.bucket_count() << endl;      // 53，STL 提供的第一个质数
```

上述定义调用以下构造函数：

```
hashtable(size_type n,
          const HashFcn&    hf,
          const EqualKey&   eql)
  : hash(hf), equals(eql), get_key(ExtractKey()), num_elements(0)
{
  initialize_buckets(n);
}

void initialize_buckets(size_type n)
{
  const size_type n_buckets = next_size(n);
  // 举例：传入 50，返回 53。以下首先保留 53 个元素空间，然后将其全部填 0
```

```
    buckets.reserve(n_buckets);
    buckets.insert(buckets.end(), n_buckets, (node*) 0);
    num_elements = 0;
  }
```

其中的 next_size() 返回最接近 n 并大于 n 的质数：

```
size_type next_size(size_type n) const { return __stl_next_prime(n); }
```

然后为 buckets vector 保留空间，设定所有 buckets 的初值为 0（null 指针）。

插入操作（insert）与表格重整（resize）

当客户端开始插入元素（节点）时：

```
iht.insert_unique(59);
iht.insert_unique(63);
iht.insert_unique(108);
```

hash table 内将会进行以下操作：

```
// 插入元素，不允许重复
pair<iterator, bool> insert_unique(const value_type& obj)
{
  resize(num_elements + 1);   // 判断是否需要重建表格，如需要就扩充
  return insert_unique_noresize(obj);
}

// 以下函数判断是否需要重建表格，如果不需要，立刻回返。如果需要，就动手…
template <class V, class K, class HF, class Ex, class Eq, class A>
void hashtable<V, K, HF, Ex, Eq, A>::resize(size_type num_elements_hint)
{
  // 以下，“表格重建与否”的判断原则颇为奇特，是拿元素个数（把新增元素计入后）和
  // bucket vector 的大小来比。如果前者大于后者，就重建表格
  // 由此可判知，每个 bucket (list) 的最大容量和 buckets vector 的大小相同
  const size_type old_n = buckets.size();
  if (num_elements_hint > old_n) {   // 确定真的需要重新配置
    const size_type n = next_size(num_elements_hint);   // 找出下一个质数
    if (n > old_n) {
      vector<node*, A> tmp(n, (node*) 0); // 设立新的 buckets
      __STL_TRY {
        // 以下处理每一个旧的 bucket
        for (size_type bucket = 0; bucket < old_n; ++bucket) {
          node* first = buckets[bucket]; // 指向节点所对应之串行的起始节点
          // 以下处理每一个旧 bucket 所含（串行）的每一个节点
          while (first) {    // 串行还没结束时
            // 以下找出节点落在哪一个新 bucket 内
            size_type new_bucket = bkt_num(first->val, n);
            // 以下四个操作颇为微妙
```

```
            // (1) 令旧 bucket 指向其所对应之串行的下一个节点（以便迭代处理）
            buckets[bucket] = first->next;
            // (2)(3) 将当前节点插入到新 bucket 内，成为其对应串行的第一个节点
            first->next = tmp[new_bucket];
            tmp[new_bucket] = first;
            // (4) 回到旧 bucket 所指的待处理串行，准备处理下一个节点
            first = buckets[bucket];
          }
        }
        buckets.swap(tmp);  // vector::swap。新旧两个 buckets 对调
        // 注意，对调两方如果大小不同，大的会变小，小的会变大
        // 离开时释放 local tmp 的内存
      }
    }
  }
}

// 在不需重建表格的情况下插入新节点，键值不允许重复
template <class V, class K, class HF, class Ex, class Eq, class A>
pair<typename hashtable<V, K, HF, Ex, Eq, A>::iterator, bool>
hashtable<V, K, HF, Ex, Eq, A>::insert_unique_noresize(const value_type& obj)
{
  const size_type n = bkt_num(obj); // 决定 obj 应位于 #n bucket
  node* first = buckets[n]; // 令 first 指向 bucket 对应之串行头部

  // 如果 buckets[n] 已被占用，此时 first 将不为 0，于是进入以下循环，
  // 走过 bucket 所对应的整个链表
  for (node* cur = first; cur; cur = cur->next)
    if (equals(get_key(cur->val), get_key(obj)))
      // 如果发现与链表中的某键值相同，就不插入，立刻返回
      return pair<iterator, bool>(iterator(cur, this), false);

  // 离开以上循环（或根本未进入循环）时，first 指向 bucket 所指链表的头部节点
  node* tmp = new_node(obj);     // 产生新节点
  tmp->next = first;
  buckets[n] = tmp;              // 令新节点成为链表的第一个节点
  ++num_elements;                // 节点个数累加 1
  return pair<iterator, bool>(iterator(tmp, this), true);
}
```

前述的 resize() 函数中，如有必要，就得做表格重建工作。操作分解如下，并示于图 5-25 中。

```
// (1) 令旧 bucket 指向其所对应之链表的下一个节点（以便迭代处理）
buckets[bucket] = first->next;
// (2)(3) 将当前节点插入到新 bucket 内，成为其对应链表的第一个节点
first->next = tmp[new_bucket];
tmp[new_bucket] = first;
// (4) 回到旧 bucket 所指的待处理链表，准备处理下一个节点
```

```
first = buckets[bucket];
```

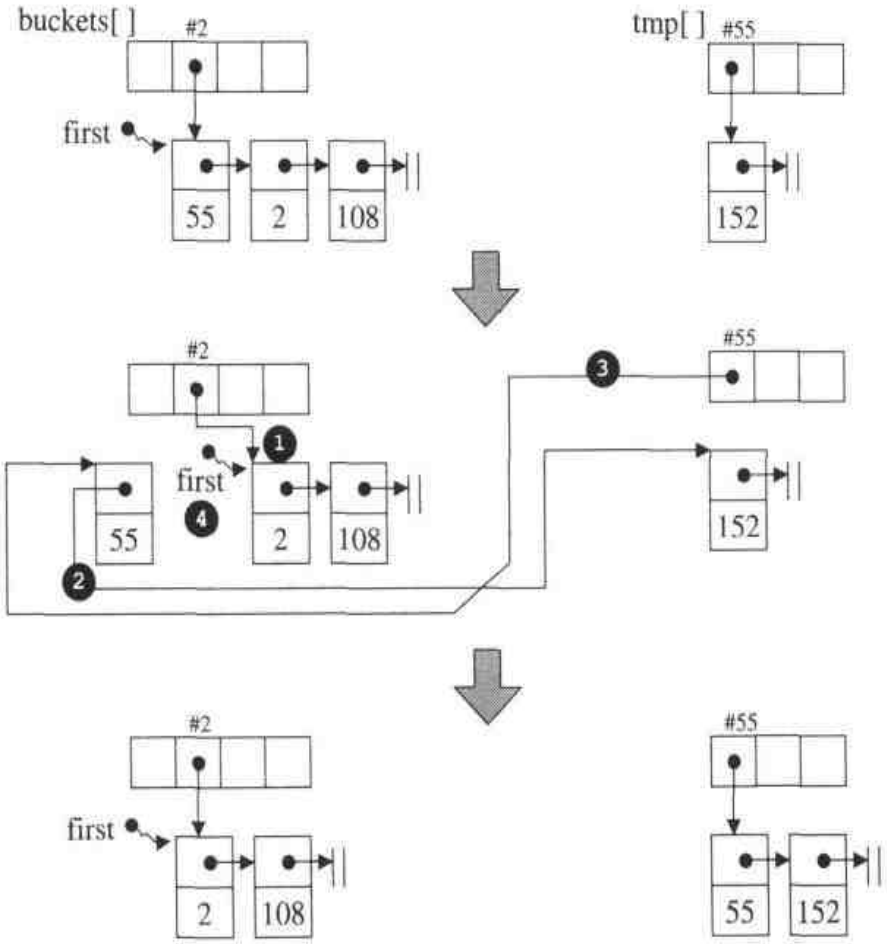


图 5-25 表格重建操作分解。本图所表现的是 hashtable<T>::resize() 内的行为，图左的 buckets[] 是旧有的 buckets，图右的 tmp[] 是新建的 buckets。最后会将新旧两个 buckets 对调，使 buckets[] 有了新风貌而 tmp[] 还诸系统。

如果客户端执行的是另一种节点插入行为（不再是 insert_unique，而是 insert_equal）：

```
iht.insert_equal(59);
iht.insert_equal(59);
```

进行的操作如下：

```
// 插入元素，允许重复
iterator insert_equal(const value_type& obj)
{
  resize(num_elements + 1);    // 判断是否需要重建表格，如需要就扩充
  return insert_equal_noresize(obj);
}
```

```
// 在不需重建表格的情况下插入新节点。键值允许重复
template <class V, class K, class HF, class Ex, class Eq, class A>
typename hashtable<V, K, HF, Ex, Eq, A>::iterator
hashtable<V, K, HF, Ex, Eq, A>::insert_equal_noresize(const value_type& obj)
{
  const size_type n = bkt_num(obj); // 决定obj应位于 #n bucket
  node* first = buckets[n]; // 令 first 指向 bucket 对应之链表头部

  // 如果 buckets[n] 已被占用，此时 first 将不为0，于是进入以下循环，
  // 走过 bucket 所对应的整个链表
  for (node* cur = first; cur; cur = cur->next)
    if (equals(get_key(cur->val), get_key(obj))) {
      // 如果发现与链表中的某键值相同，就马上插入，然后返回
      node* tmp = new_node(obj);      // 产生新节点
      tmp->next = cur->next;          // 将新节点插入于目前位置
      cur->next = tmp;
      ++num_elements;                 // 节点个数累加1
      return iterator(tmp, this);     // 返回一个迭代器，指向新增节点
    }

  // 进行至此，表示没有发现重复的键值
  node* tmp = new_node(obj);      // 产生新节点
  tmp->next = first;              // 将新节点插入于链表头部
  buckets[n] = tmp;
  ++num_elements;                 // 节点个数累加1
  return iterator(tmp, this);     // 返回一个迭代器，指向新增节点
}
```

判知元素的落脚处（bkt_num）

本节程序代码在许多地方都需要知道某个元素值落脚于哪一个 bucket 之内。这本来是 hash function 的责任，SGI 把这个任务包装了一层，先交给 bkt_num() 函数，再由此函数调用 hash function，取得一个可以执行 modulus（取模）运算的数值。为什么要这么做？因为有些元素型别无法直接拿来对 hashtable 的大小进行模运算，例如字符字符串 const char*，这时候我们需要做一些转换。下面是 bkt_num() 函数，5.7.7 列有 SGI 内建的所有 hash functions。

```
// 版本1：接受实值（value）和buckets 个数
size_type bkt_num(const value_type& obj, size_t n) const
{
  return bkt_num_key(get_key(obj), n); // 调用版本4
}

// 版本2：只接受实值（value）
```

```
size_type bkt_num(const value_type& obj) const
{
  return bkt_num_key(get_key(obj));     // 调用版本 3
}

// 版本 3：只接受键值
size_type bkt_num_key(const key_type& key) const
{
  return bkt_num_key(key, buckets.size());  // 调用版本 4
}

// 版本 4：接受键值和 buckets 个数
size_type bkt_num_key(const key_type& key, size_t n) const
{
  return hash(key) % n;  // SGI 的所有内建的 hash() 列于 5.7.7 节
}
```

复制（copy_from）和整体删除（clear）

由于整个 hash table 由 vector 和 linked-list 组合而成，因此，复制和整体删除，都需要特别注意内存的释放问题。下面是 hashtable 提供的两个相关函数：

```
template <class V, class K, class HF, class Ex, class Eq, class A>
void hashtable<V, K, HF, Ex, Eq, A>::clear()
{
  // 针对每一个 bucket.
  for (size_type i = 0; i < buckets.size(); ++i) {
    node* cur = buckets[i];
    // 将 bucket list 中的每一个节点删除掉
    while (cur != 0) {
      node* next = cur->next;
      delete_node(cur);
      cur = next;
    }
    buckets[i] = 0;    // 令 bucket 内容为 null 指针
  }
  num_elements = 0;    // 令总节点个数为 0

  // 注意，buckets vector 并未释放掉空间，仍保有原来大小
}

template <class V, class K, class HF, class Ex, class Eq, class A>
void hashtable<V, K, HF, Ex, Eq, A>::copy_from(const hashtable& ht)
{
  // 先清除己方的 buckets vector. 这操作是调用 vector::clear，造成所有元素为 0
  buckets.clear();
  // 为己方的 buckets vector 保留空间，使与对方相同
  // 如果己方空间大于对方，就不动，如果己方空间小于对方，就会增大
```

```
  buckets.reserve(ht.buckets.size());
  // 从己方的 buckets vector 尾端开始，插入 n 个元素，其值为 null 指针
  // 注意，此时 buckets vector 为空，所以所谓尾端，就是起头处
  buckets.insert(buckets.end(), ht.buckets.size(), (node*) 0);
  __STL_TRY {
    // 针对 buckets vector
    for (size_type i = 0; i < ht.buckets.size(); ++i) {
      // 复制 vector 的每一个元素（是个指针，指向 hashtable 节点）
      if (const node* cur = ht.buckets[i]) {
        node* copy = new_node(cur->val);
        buckets[i] = copy;

        // 针对同一个 bucket list，复制每一个节点
        for (node* next = cur->next; next; cur = next, next = cur->next) {
          copy->next = new_node(next->val);
          copy = copy->next;
        }
      }
    }
    num_elements = ht.num_elements;  // 重新登录节点个数（hashtable 的大小）
  }
  __STL_UNWIND(clear());
}
```

5.7.6 hashtable 运用实例

先前说明 hash table 的实现源代码时，已经零零散散地示范了一些客户端程序。下面是一个完整的实例。

```
// file: 5hashtable-test.cpp
//注意：客户端程序不能直接含入 <stl_hashtable.h>，应该含入有用到 hashtable
// 的容器头文件，例如 <hash_set.h> 或 <hash_map.h>
#include <hash_set>  // for hashtable
#include <iostream>
using namespace std;

int main()
{
  // hash-table
  // <value, key, hash-func, extract-key, equal-key, allocator>
  // note: hash-table has no default ctor
  hashtable<int,
            int,
            hash<int>,
            identity<int>,
            equal_to<int>,
            alloc>
```

```
 iht(50,hash<int>(),equal_to<int>()); // 指定保留 50 个buckets

cout<< iht.size() << endl;                  // 0
cout<< iht.bucket_count() << endl;          // 53. 这是 STL 供应的第一个质数
cout<< iht.max_bucket_count() << endl; // 4294967291
                                            // 这是 STL 供应的最后一个质数
iht.insert_unique(59);
iht.insert_unique(63);
iht.insert_unique(108);
iht.insert_unique(2);
iht.insert_unique(53);
iht.insert_unique(55);
cout<< iht.size() << endl;    // 6.  此即 hashtable<T>::num_elements

// 以下声明一个 hashtable 迭代器
hashtable<int,
          int,
          hash<int>,
          identity<int>,
          equal_to<int>,
          alloc>
    ::iterator ite = iht.begin();

// 以迭代器遍历 hashtable，将所有节点的值打印出来
for(int i=0; i< iht.size(); ++i, ++ite)
   cout << *ite << ' ';                  // 53 55 2 108 59 63
cout << endl;

// 遍历所有 buckets，如果其节点个数不为 0，就打印出节点个数
for(int i=0; i< iht.bucket_count(); ++i) {
  int n = iht.elems_in_bucket(i);
  if (n != 0)
     cout << "bucket[" << i << "] has " << n << " elems." << endl;
}
// bucket[0] has 1 elems.
// bucket[2] has 3 elems.
// bucket[6] has 1 elems.
// bucket[10] has 1 elems.

// 为了验证 “bucket(list) 的容量就是 buckets vector 的大小”（这是从
// hashtable<T>::resize() 得知的结果），我刻意将元素加到 54 个，
// 看看是否发生 “表格重建 (re-hashing)”
for(int i=0; i<=47; i++)
    iht.insert_equal(i);
cout<< iht.size() << endl;                  // 54. 元素（节点）个数
cout<< iht.bucket_count() << endl;          // 97. buckets 个数
// 遍历所有 buckets，如果其节点个数不为 0，就打印出节点个数
for(int i=0; i< iht.bucket_count(); ++i) {
  int n = iht.elems_in_bucket(i);
```

```
    if (n != 0)
      cout << "bucket[" << i << "] has " << n << " elems." << endl;
  }
  // 打印结果：bucket[2] 和 bucket[11] 的节点个数为2，
  // 其余的bucket[0]~bucket[47] 的节点个数均为1
  // 此外，bucket[53],[55],[59],[63] 的节点个数均为1

  // 以迭代器遍历hashtable，将所有节点的值打印出来
  ite = iht.begin();
  for(int i=0; i< iht.size(); ++i, ++ite)
    cout << *ite << ' ';   //
  cout << endl;
  // 0 1 2 2 3 4 5 6 7 8 9 10 11 108 12 13 14 15 16 17 18 19 20 21
  // 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
  // 43 44 45 46 47 53 55 59 63

  cout << *(iht.find(2)) << endl;  // 2
  cout << iht.count(2) << endl;    // 2
}
```

这个程序详细测试出 hash table 的节点排列状态与表格重整结果。一开始我保留 50 个节点，由于最接近的 STL 质数为 53，所以 buckets vector 保留的是 53 个 buckets，每个 buckets（指针，指向一个 hash table 节点）的初值为 0。接下来，循序加入 6 个元素：53, 55, 2, 108, 59, 63，于是 hash table 变成图 5-26 所示的样子。

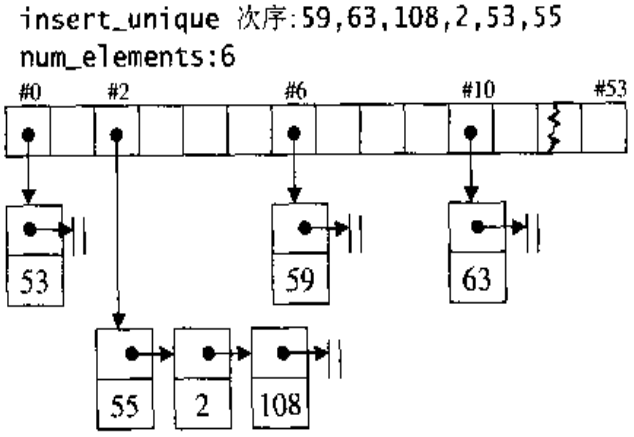


图 **5-26** 插入 6 个元素后，hash table 的状态

接下来，我再插入 48 个元素，使总元素达到 54 个，超过当时的 buckets vector 的大小，符合表格重建条件（这是从 `hashtable<T>::resize()` 函数中得知的），于是 hash table 变成了图 5-27 所示的模样。

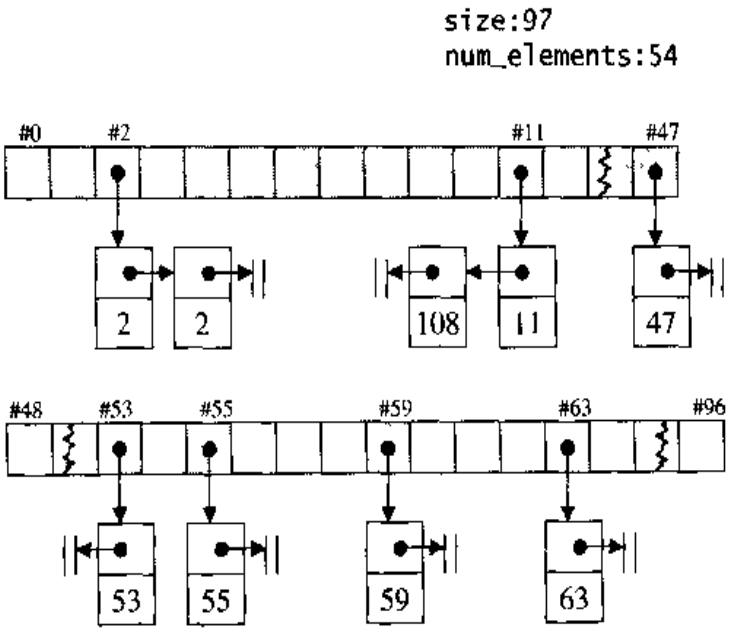


图 **5-27** hash table 重整结果。注意，bucket #2 和 bucket #11 的节点个数都是 2，其余的灰色 buckets，节点个数都是 1。白色 buckets 表示节点个数为 0。灰色和白色只是为了方便表示，因为如果把节点全画出来，画面过挤摆不下。图中的 bucket #47 和 bucket #48 应该连续，也是因为画面的关系，分为两段显示。

程序最后分别使用了 hash table 提供的 find 和 count 函数，搜寻键值为 2 的元素，以及计算键值为 2 的元素个数。请注意，键值相同的元素，一定落在同一个 bucket list 之中。下面是 find 和 count 的源代码：

```
iterator find(const key_type& key)
{
  size_type n = bkt_num_key(key);  // 首先寻找落在哪一个 bucket 内
  node* first;
  // 以下，从 bucket list 的头开始，一一比对每个元素的键值。比对成功就跳出
  for ( first = buckets[n];
        first && !equals(get_key(first->val), key);
        first = first->next)
    {}
  return iterator(first, this);
}

size_type count(const key_type& key) const
{
  const size_type n = bkt_num_key(key); // 首先寻找落在哪一个 bucket 内
  size_type result = 0;
```

```
    // 以下，从 bucket list 的头开始，一一比对每个元素的键值。比对成功就累加 1。
    for (const node* cur = buckets[n]; cur; cur = cur->next)
      if (equals(get_key(cur->val), key))
        ++result;
    return result;
  }
```

5.7.7 hash functions

<stl_hash_fun.h> 定义有数个现成的 hash functions，全都是仿函数（第 7 章）。先前谈及概念时，我们说 hash function 是计算元素位置的函数，SGI 将这项任务赋予了先前提过的 bkt_num()，再由它来调用这里提供的 hash function，取得一个可以对 hashtable 进行模运算的值。针对 char, int, long 等整数型别，这里大部分的 hash functions 什么也没做，只是忠实返回原值。但对于字符字符串（const char*），就设计了一个转换函数如下：

```
// 以下定义于 <stl_hash_fun.h>
template <class Key> struct hash { };

inline size_t __stl_hash_string(const char* s)
{
  unsigned long h = 0;
  for ( ; *s; ++s)
     h = 5*h + *s;

  return size_t(h);
}

// 以下所有的 __STL_TEMPLATE_NULL，在 <stl_config.h> 中皆被定义为
template<>，详见 1.9.1 节

__STL_TEMPLATE_NULL struct hash<char*>
{
  size_t operator()(const char* s) const { return __stl_hash_string(s); }
};
__STL_TEMPLATE_NULL struct hash<const char*>
{
  size_t operator()(const char* s) const { return __stl_hash_string(s); }
};
__STL_TEMPLATE_NULL struct hash<char> {
  size_t operator()(char x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned char> {
  size_t operator()(unsigned char x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<signed char> {
```

```
  size_t operator()(unsigned char x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<short> {
  size_t operator()(short x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned short> {
  size_t operator()(unsigned short x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<int> {
  size_t operator()(int x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned int> {
  size_t operator()(unsigned int x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<long> {
  size_t operator()(long x) const { return x; }
};
__STL_TEMPLATE_NULL struct hash<unsigned long> {
  size_t operator()(unsigned long x) const { return x; }
};
```

由此观之，SGI hashtable 无法处理上述所列各项型别以外的元素，例如 string, double, float。欲处理这些型别，用户必须自行为它们定义 hash function。下面是直接以 SGI hashtable 处理 string 所获得的错误现象：

```
#include <hash_set>  // for hashtable and hash_set
#include <iostream>
#include <string>
using namespace std;

int main()
{
  // hash-table
  // <value, key, hash-func, extract-key, equal-key, allocator>
  // note: hashtable has no default ctor
  hashtable<string, string, hash<string>, identity<string>,
            equal_to<string>, alloc>
    iht(50,hash<string>(),equal_to<string>());

  cout<< iht.size() << endl;              // 0
  cout<< iht.bucket_count() << endl;      // 53
  iht.insert_unique(string("jjhou"));     // error

  // hashtable 无法处理的型别，hash_set 当然也无法处理
  hash_set<string> shs;
  hash_set<double> dhs;

  shs.insert(string("jjhou"));  // error
```

```
  dhs.insert(15.0);              // error
}
```

5.8 hash_set

虽然 STL 只规范复杂度与接口，并不规范实现方法，但 STL set 多半以 RB-tree 为底层机制，SGI 则是在 STL 标准规格之外另又提供了一个所谓的 hash_set，以 hashtable 为底层机制。由于 hash_set 所供应的操作接口，hashtable 都提供了，所以几乎所有的 hash_set 操作行为，都只是转调用 hashtable 的操作行为而已。

运用 set，为的是能够快速搜寻元素。这一点，不论其底层是 RB-tree 或是 hashtable，都可以达成任务，但是请注意，RB-tree 有自动排序功能而 hashtable 没有，反应出来的结果就是，set 的元素有自动排序功能而 hash_set 没有。

set 的元素不像 map 那样可以同时拥有实值（*value*）和键值（*key*），set 元素的键值就是实值，实值就是键值。这一点在 hash_set 中也是一样的。

hash_set 的使用方式，与 set 完全相同。

下面是 hash_set 的源代码摘录，其中的注释几乎说明了一切，本节不再另做文字解释。

请注意，5.7.5 节最后谈到，hashtable 有一些无法处理的型别（除非用户为那些型别撰写 hash function）。凡是 hashtable 无法处理者，hash_set 也无法处理。

```
template <class Value,
          class HashFcn = hash<Value>,
          class EqualKey = equal_to<Value>,
          class Alloc = alloc>
class hash_set
{
private:
  // 以下使用的 identity<> 定义于 <stl_function.h> 中
  typedef hashtable<Value, Value, HashFcn, identity<Value>,
                    EqualKey, Alloc> ht;
  ht rep;     // 底层机制以 hash table 完成
```

```
public:
  typedef typename ht::key_type key_type;
  typedef typename ht::value_type value_type;
  typedef typename ht::hasher hasher;
  typedef typename ht::key_equal key_equal;

  typedef typename ht::size_type size_type;
  typedef typename ht::difference_type difference_type;
  typedef typename ht::const_pointer pointer;
  typedef typename ht::const_pointer const_pointer;
  typedef typename ht::const_reference reference;
  typedef typename ht::const_reference const_reference;

  typedef typename ht::const_iterator iterator;
  typedef typename ht::const_iterator const_iterator;

  hasher hash_funct() const { return rep.hash_funct(); }
  key_equal key_eq() const { return rep.key_eq(); }

public:
  // 缺省使用大小为 100 的表格，将被 hash table 调整为最接近且较大之质数
  hash_set() : rep(100, hasher(), key_equal()) {}
  explicit hash_set(size_type n) : rep(n, hasher(), key_equal()) {}
  hash_set(size_type n, const hasher& hf) : rep(n, hf, key_equal()) {}
  hash_set(size_type n, const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) {}

  // 以下，插入操作全部使用 insert_unique()，不允许键值重复
  template <class InputIterator>
  hash_set(InputIterator f, InputIterator l)
    : rep(100, hasher(), key_equal()) { rep.insert_unique(f, l); }
  template <class InputIterator>
  hash_set(InputIterator f, InputIterator l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_unique(f, l); }
  template <class InputIterator>
  hash_set(InputIterator f, InputIterator l, size_type n,
           const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_unique(f, l); }
  template <class InputIterator>
  hash_set(InputIterator f, InputIterator l, size_type n,
           const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) { rep.insert_unique(f, l); }

public:
  // 所有操作几乎都有 hash table 对应版本，传递调用就行
  size_type size() const { return rep.size(); }
  size_type max_size() const { return rep.max_size(); }
  bool empty() const { return rep.empty(); }
```

```
  void swap(hash_set& hs) { rep.swap(hs.rep); }
  friend bool operator== __STL_NULL_TMPL_ARGS (const hash_set&,
                                               const hash_set&);

  iterator begin() const { return rep.begin(); }
  iterator end() const { return rep.end(); }

public:
  pair<iterator, bool> insert(const value_type& obj)
    {
      pair<typename ht::iterator, bool> p = rep.insert_unique(obj);
      return pair<iterator, bool>(p.first, p.second);
    }
  template <class InputIterator>
  void insert(InputIterator f, InputIterator l) { rep.insert_unique(f,l); }
  pair<iterator, bool> insert_noresize(const value_type& obj)
  {
    pair<typename ht::iterator, bool> p = rep.insert_unique_noresize(obj);
    return pair<iterator, bool>(p.first, p.second);
  }

  iterator find(const key_type& key) const { return rep.find(key); }

  size_type count(const key_type& key) const { return rep.count(key); }

  pair<iterator, iterator> equal_range(const key_type& key) const
    { return rep.equal_range(key); }

  size_type erase(const key_type& key) {return rep.erase(key); }
  void erase(iterator it) { rep.erase(it); }
  void erase(iterator f, iterator l) { rep.erase(f, l); }
  void clear() { rep.clear(); }

public:
  void resize(size_type hint) { rep.resize(hint); }
  size_type bucket_count() const { return rep.bucket_count(); }
  size_type max_bucket_count() const { return rep.max_bucket_count(); }
  size_type elems_in_bucket(size_type n) const
    { return rep.elems_in_bucket(n); }
};

template <class Value, class HashFcn, class EqualKey, class Alloc>
inline bool operator==(const hash_set<Value, HashFcn, EqualKey, Alloc>& hs1,
                       const hash_set<Value, HashFcn, EqualKey, Alloc>& hs2)
{
  return hs1.rep == hs2.rep;
}
```

下面这个例子，取材自 [Austern98] 16.2.5 节。我特别在程序最后新加一段遍历操作，为的是验证 hash_set 内的元素并无任何特定排序，但是这样的安排不尽合理，稍后我有进一步的说明。程序中对于 hash_set 的 EqualKey 必须有特别的设计，不能沿用缺省的 equal_to<T>，因为此例之中的元素是 C 字符串（C style charactors string），而 C 字符串的相等与否，必须一个字符一个字符地比较（可使用 C 标准函数 strcmp()），不能直接以 const char* 做比较。

```
// file: 5hastset-test.cpp
#include <iostream>
#include <hash_set>
#include <cstring>
using namespace std;

struct eqstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) == 0;
  }
};

void lookup(const hash_set<const char*, hash<const char*>, eqstr>& Set,
            const char* word)
{
  hash_set<const char*, hash<const char*>, eqstr>::const_iterator it
    = Set.find(word);
  cout << "  " << word << ": "
       << (it != Set.end() ? "present" : "not present")
       << endl;
}

int main()
{
  hash_set<const char*, hash<const char*>, eqstr> Set;
  Set.insert("kiwi");
  Set.insert("plum");
  Set.insert("apple");
  Set.insert("mango");
  Set.insert("apricot");
  Set.insert("banana");

  lookup(Set, "mango");    //  mango: present
  lookup(Set, "apple");    //  apple: present
  lookup(Set, "durian");   //  durian: not present
```

```
hash_set<const char*, hash<const char*>, eqstr>::iterator ite1
  = Set.begin();
hash_set<const char*, hash<const char*>, eqstr>::iterator ite2
  = Set.end();
for(; ite1 != ite2; ++ite1)
  cout << *ite1 << ' ';   // banana plum mango apple kiwi apricot
}
```

最后执行结果虽然显示 hash_set 内的字符串并没有排序，但这其实不是一个良好的测试，因为即使有排序，也是以元素型别 const char* 为排序对象，而非对 const char* 所代表的字符串进行排序。要测试 hash_set 是否排序，最好是以 int 作为元素型别，如下：

```
hash_set<int> Set;
Set.insert(59);
Set.insert(63);
Set.insert(108);
Set.insert(2);
Set.insert(53);
Set.insert(55);

hash_set<int>::iterator ite1 = Set.begin();
hash_set<int>::iterator ite2 = Set.end();
for(; ite1 != ite2; ++ite1)
  cout << *ite1 << ' ';  // 2 53 55 59 63 108
cout << endl;
```

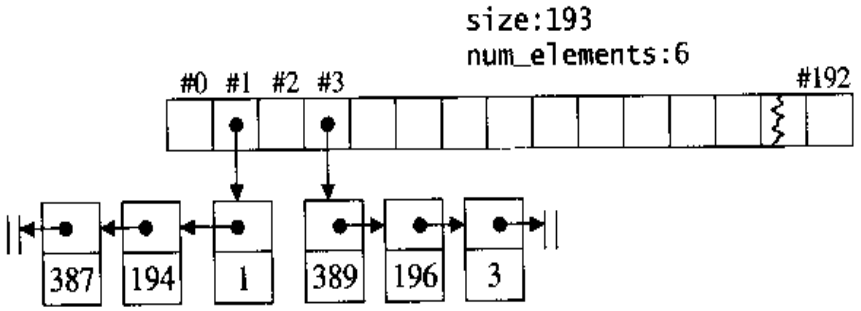
奇怪了，为什么也有排序呢？hash_set 的底层不就是一个 hashtable 吗？先前 5.7.6 节的例子也是以相同的次序将相同的 6 个整数插入到 hashtable 内，获得的结果为什么和这里不同？

这真是一个令人迷惑的问题。答案是，5.7.6 节的 hashtable 大小被指定为 50（根据 SGI 的设计，采用质数 53），而这里所使用的 hash_set 缺省情况下指定 hashtable 的大小为 100（根据 SGI 的设计，采用质数 193），由于 buckets 够多，才造成排序假象。如果以下面这样的次序输入这些数值：

```
hash_set<int> Set;   // 底层 hashtable 缺省大小为 100
Set.insert(3);       // 实际大小为 193
Set.insert(196);
Set.insert(1);
Set.insert(389);
Set.insert(194);
Set.insert(387);
```

```
hash_set<int>::iterator ite1 = Set.begin();
hash_set<int>::iterator ite2 = Set.end();
for(; ite1 != ite2; ++ite1)
   cout << *ite1 << ' ';      // 387 194 1 389 196 3
```

就呈现出未排序的状态了。此时底层的 hashtable 构造如下：



5.9 hash_map

SGI 在 STL 标准规格之外，另提供了一个所谓的 hash_map，以 hashtable 为底层机制。由于 hash_map 所供应的操作接口，hashtable 都提供了，所以几乎所有的 hash_map 操作行为，都只是转调用 hashtable 的操作行为而已。

运用 map，为的是能够根据键值快速搜寻元素。这一点，不论其底层是 RB-tree 或是 hashtable，都可以达成任务。但是请注意，RB-tree 有自动排序功能而 hashtable 没有，反应出来的结果就是，map 的元素有自动排序功能而 hash_map 没有。

map 的特性是，每一个元素都同时拥有一个实值（*value*）和一个键值（*key*）。这一点在 hash_map 中也是一样的。hash_map 的使用方式，和 map 完全相同。

下面是 hash_map 的源代码摘录，其中的注释几乎说明了一切，本节不再另做文字解释。

请注意，5.7.5 节最后谈到，hashtable 有一些无法处理的型别（除非用户为那些型别撰写 hash function）。凡是 hashtable 无法处理者，hash_map 也无法处理。

```
// 以下的 hash<> 是个 function object，定义于 <stl_hash_fun.h> 中
```

```
// 例: hash<int>::operator()(int x) const { return x; }
template <class Key,
          class T,
          class HashFcn = hash<Key>,
          class EqualKey = equal_to<Key>,
          class Alloc = alloc>
class hash_map
{
private:
  // 以下使用的 select1st<> 定义于 <stl_function.h> 中
  typedef hashtable<pair<const Key, T>, Key, HashFcn,
                    select1st<pair<const Key, T> >, EqualKey, Alloc> ht;
  ht rep;     // 底层机制以 hash table 完成

public:
  typedef typename ht::key_type key_type;
  typedef T data_type;
  typedef T mapped_type;
  typedef typename ht::value_type value_type;
  typedef typename ht::hasher hasher;
  typedef typename ht::key_equal key_equal;

  typedef typename ht::size_type size_type;
  typedef typename ht::difference_type difference_type;
  typedef typename ht::pointer pointer;
  typedef typename ht::const_pointer const_pointer;
  typedef typename ht::reference reference;
  typedef typename ht::const_reference const_reference;

  typedef typename ht::iterator iterator;
  typedef typename ht::const_iterator const_iterator;

  hasher hash_funct() const { return rep.hash_funct(); }
  key_equal key_eq() const { return rep.key_eq(); }

public:
  // 缺省使用大小为 100 的表格。将由 hash table 调整为最接近且较大之质数
  hash_map() : rep(100, hasher(), key_equal()) {}
  explicit hash_map(size_type n) : rep(n, hasher(), key_equal()) {}
  hash_map(size_type n, const hasher& hf) : rep(n, hf, key_equal()) {}
  hash_map(size_type n, const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) {}

  // 以下，插入操作全部使用 insert_unique()，不允许键值重复
  template <class InputIterator>
  hash_map(InputIterator f, InputIterator l)
    : rep(100, hasher(), key_equal()) { rep.insert_unique(f, l); }
  template <class InputIterator>
  hash_map(InputIterator f, InputIterator l, size_type n)
```

```
    : rep(n, hasher(), key_equal()) { rep.insert_unique(f, l); }
  template <class InputIterator>
  hash_map(InputIterator f, InputIterator l, size_type n,
           const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_unique(f, l); }
  template <class InputIterator>
  hash_map(InputIterator f, InputIterator l, size_type n,
           const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) { rep.insert_unique(f, l); }

public:
  // 所有操作几乎都有 hash table 对应版本，传递调用就行
  size_type size() const { return rep.size(); }
  size_type max_size() const { return rep.max_size(); }
  bool empty() const { return rep.empty(); }
  void swap(hash_map& hs) { rep.swap(hs.rep); }
  friend bool
  operator== __STL_NULL_TMPL_ARGS (const hash_map&, const hash_map&);

  iterator begin() { return rep.begin(); }
  iterator end() { return rep.end(); }
  const_iterator begin() const { return rep.begin(); }
  const_iterator end() const { return rep.end(); }

public:
  pair<iterator, bool> insert(const value_type& obj)
    { return rep.insert_unique(obj); }
  template <class InputIterator>
  void insert(InputIterator f, InputIterator l) { rep.insert_unique(f,l); }
  pair<iterator, bool> insert_noresize(const value_type& obj)
    { return rep.insert_unique_noresize(obj); }

  iterator find(const key_type& key) { return rep.find(key); }
  const_iterator find(const key_type& key) const { return rep.find(key); }

  T& operator[](const key_type& key) {
    return rep.find_or_insert(value_type(key, T())).second;
  }

  size_type count(const key_type& key) const { return rep.count(key); }

  pair<iterator, iterator> equal_range(const key_type& key)
    { return rep.equal_range(key); }
  pair<const_iterator, const_iterator> equal_range(const key_type& key) const
    { return rep.equal_range(key); }

  size_type erase(const key_type& key) {return rep.erase(key); }
  void erase(iterator it) { rep.erase(it); }
  void erase(iterator f, iterator l) { rep.erase(f, l); }
```

```
  void clear() { rep.clear(); }

public:
  void resize(size_type hint) { rep.resize(hint); }
  size_type bucket_count() const { return rep.bucket_count(); }
  size_type max_bucket_count() const { return rep.max_bucket_count(); }
  size_type elems_in_bucket(size_type n) const
    { return rep.elems_in_bucket(n); }
};

template <class Key, class T, class HashFcn, class EqualKey, class Alloc>
inline bool operator==(const hash_map<Key, T, HashFcn, EqualKey, Alloc>& hm1,
                       const hash_map<Key, T, HashFcn, EqualKey, Alloc>& hm2)
{
  return hm1.rep == hm2.rep;
}
```

下面这个例子，取材自 [Austern98] 16.2.6 节。我特别在程序最后新加了一段遍历操作，为的是验证 **hash_map** 内的元素并无任何特定排序。程序中对于 **hash_map** 的 EqualKey 必须有特别的设计，不能沿用缺省的 equal_to<T>，因为此例之中的元素是字符串，面字符串的相等与否，必须一个字符一个字符地比较（可使用C标准函数 strcmp()），不能直接以 const char* 做比较。

```
// file : 5hashmap-test.cpp
#include <iostream>
#include <hash_map>
#include <cstring>
using namespace std;

struct eqstr
{
  bool operator()(const char* s1, const char* s2) const {
    return strcmp(s1, s2) == 0;
  }
};

int main()
{
  hash_map<const char*, int, hash<const char*>, eqstr> days;

  days["january"] = 31;
  days["february"] = 28;
  days["march"] = 31;
  days["april"] = 30;
  days["may"] = 31;
  days["june"] = 30;
```

```
  days["july"] = 31;
  days["august"] = 31;
  days["september"] = 30;
  days["october"] = 31;
  days["november"] = 30;
  days["december"] = 31;

  cout << "september -> " << days["september"] << endl;  // 30
  cout << "june      -> " << days["june"] << endl;       // 30
  cout << "february  -> " << days["february"] << endl;   // 28
  cout << "december  -> " << days["december"] << endl;   // 31

  hash_map<const char*, int, hash<const char*>, eqstr>::iterator
    ite1 = days.begin();
  hash_map<const char*, int, hash<const char*>, eqstr>::iterator
    ite2 = days.end();
  for(; ite1 != ite2; ++ite1)
     cout << ite1->first << ' ';
  // september june july may january february december march
  // april november october august
}
```

5.10 hash_multiset

hash_multiset 的特性与 multiset 完全相同，唯一的差别在于它的底层机制是 hashtable。也因此，hash_multiset 的元素并不会被自动排序。

hash_multiset 和 hash_set 实现上的唯一差别在于，前者的元素插入操作采用底层机制 hashtable 的 insert_equal()，后者则是采用 insert_unique()。

下面是 hash_multiset 的源代码摘要，其中的注释几乎说明了一切，本节不再另做文字解释。

请注意，5.7.5 节最后谈到，hashtable 有一些无法处理的型别（除非用户为那些型别撰写 hash function）。凡是 hashtable 无法处理者，hash_multiset 也无法处理。

```
template <class Value,
          class HashFcn = hash<Value>,
          class EqualKey = equal_to<Value>,
          class Alloc = alloc>
class hash_multiset
{
```

```
private:
  typedef hashtable<Value, Value, HashFcn, identity<Value>,
                    EqualKey, Alloc> ht;
  ht rep;

public:
  typedef typename ht::key_type key_type;
  typedef typename ht::value_type value_type;
  typedef typename ht::hasher hasher;
  typedef typename ht::key_equal key_equal;

  typedef typename ht::size_type size_type;
  typedef typename ht::difference_type difference_type;
  typedef typename ht::const_pointer pointer;
  typedef typename ht::const_pointer const_pointer;
  typedef typename ht::const_reference reference;
  typedef typename ht::const_reference const_reference;

  typedef typename ht::const_iterator iterator;
  typedef typename ht::const_iterator const_iterator;

  hasher hash_funct() const { return rep.hash_funct(); }
  key_equal key_eq() const { return rep.key_eq(); }

public:
  // 缺省使用大小为100的表格。将被hash table调整为最接近且较大之质数
  hash_multiset() : rep(100, hasher(), key_equal()) {}
  explicit hash_multiset(size_type n) : rep(n, hasher(), key_equal()) {}
  hash_multiset(size_type n, const hasher& hf) : rep(n, hf, key_equal()) {}
  hash_multiset(size_type n, const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) {}

// 以下，插入操作全部使用 insert_equal()，允许键值重复
  template <class InputIterator>
  hash_multiset(InputIterator f, InputIterator l)
    : rep(100, hasher(), key_equal()) { rep.insert_equal(f, l); }
  template <class InputIterator>
  hash_multiset(InputIterator f, InputIterator l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_equal(f, l); }
  template <class InputIterator>
  hash_multiset(InputIterator f, InputIterator l, size_type n,
                const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_equal(f, l); }
  template <class InputIterator>
  hash_multiset(InputIterator f, InputIterator l, size_type n,
                const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) { rep.insert_equal(f, l); }

public:
```

```
  // 所有操作几乎都有 hash table 的对应版本，传递调用即可
  size_type size() const { return rep.size(); }
  size_type max_size() const { return rep.max_size(); }
  bool empty() const { return rep.empty(); }
  void swap(hash_multiset& hs) { rep.swap(hs.rep); }
  friend bool operator== __STL_NULL_TMPL_ARGS (const hash_multiset&,
                                               const hash_multiset&);

  iterator begin() const { return rep.begin(); }
  iterator end() const { return rep.end(); }

public:
  iterator insert(const value_type& obj) { return rep.insert_equal(obj); }
  template <class InputIterator>
  void insert(InputIterator f, InputIterator l) { rep.insert_equal(f,l); }
  iterator insert_noresize(const value_type& obj)
    { return rep.insert_equal_noresize(obj); }

  iterator find(const key_type& key) const { return rep.find(key); }

  size_type count(const key_type& key) const { return rep.count(key); }

  pair<iterator, iterator> equal_range(const key_type& key) const
    { return rep.equal_range(key); }

  size_type erase(const key_type& key) {return rep.erase(key); }
  void erase(iterator it) { rep.erase(it); }
  void erase(iterator f, iterator l) { rep.erase(f, l); }
  void clear() { rep.clear(); }

public:
  void resize(size_type hint) { rep.resize(hint); }
  size_type bucket_count() const { return rep.bucket_count(); }
  size_type max_bucket_count() const { return rep.max_bucket_count(); }
  size_type elems_in_bucket(size_type n) const
    { return rep.elems_in_bucket(n); }
};

template <class Val, class HashFcn, class EqualKey, class Alloc>
inline bool operator==(const hash_multiset<Val, HashFcn, EqualKey, Alloc>& hs1,
                       const hash_multiset<Val, HashFcn, EqualKey, Alloc>& hs2)
{
  return hs1.rep == hs2.rep;
}
```

hash_multiset 的使用方式，与 hash_set 完全相同。

5.11 hash_multimap

hash_multimap 的特性与 multimap 完全相同，唯一的差别在于它的底层机制是 hashtable。也因此，hash_multimap 的元素并不会被自动排序。

hash_multimap 和 hash_map 实现上的唯一差别在于，前者的元素插入操作采用底层机制 hashtable 的 insert_equal()，后者则是采用 insert_unique()。

下面是 hash_multimap 的源代码摘要，其中的注释几乎说明了一切，本节不再另做文字解释。

请注意，5.7.5 节最后谈到，hashtable 有一些无法处理的型别（除非用户为那些型别撰写 hash function）。凡是 hashtable 无法处理者，hash_multimap 也无法处理。

```
template <class Key,
          class T,
          class HashFcn = hash<Key>,
          class EqualKey = equal_to<Key>,
          class Alloc = alloc>
class hash_multimap
{
private:
  typedef hashtable<pair<const Key, T>, Key, HashFcn,
                    selectlst<pair<const Key, T> >, EqualKey, Alloc> ht;
  ht rep;

public:
  typedef typename ht::key_type key_type;
  typedef T data_type;
  typedef T mapped_type;
  typedef typename ht::value_type value_type;
  typedef typename ht::hasher hasher;
  typedef typename ht::key_equal key_equal;

  typedef typename ht::size_type size_type;
  typedef typename ht::difference_type difference_type;
  typedef typename ht::pointer pointer;
  typedef typename ht::const_pointer const_pointer;
  typedef typename ht::reference reference;
  typedef typename ht::const_reference const_reference;

  typedef typename ht::iterator iterator;
```

```
  typedef typename ht::const_iterator const_iterator;

  hasher hash_funct() const { return rep.hash_funct(); }
  key_equal key_eq() const { return rep.key_eq(); }

public:
  // 缺省使用大小为 100 的表格，将被 hash table 调整为最接近且较大之质数
  hash_multimap() : rep(100, hasher(), key_equal()) {}
  explicit hash_multimap(size_type n) : rep(n, hasher(), key_equal()) {}
  hash_multimap(size_type n, const hasher& hf) : rep(n, hf, key_equal()) {}
  hash_multimap(size_type n, const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) {}

// 以下，插入操作全部使用 insert_equal()，允许键值重复
  template <class InputIterator>
  hash_multimap(InputIterator f, InputIterator l)
    : rep(100, hasher(), key_equal()) { rep.insert_equal(f, l); }
  template <class InputIterator>
  hash_multimap(InputIterator f, InputIterator l, size_type n)
    : rep(n, hasher(), key_equal()) { rep.insert_equal(f, l); }
  template <class InputIterator>
  hash_multimap(InputIterator f, InputIterator l, size_type n,
                const hasher& hf)
    : rep(n, hf, key_equal()) { rep.insert_equal(f, l); }
  template <class InputIterator>
  hash_multimap(InputIterator f, InputIterator l, size_type n,
                const hasher& hf, const key_equal& eql)
    : rep(n, hf, eql) { rep.insert_equal(f, l); }

public:
  // 所有操作几乎都有 hash table 的对应版本，传递调用即可
  size_type size() const { return rep.size(); }
  size_type max_size() const { return rep.max_size(); }
  bool empty() const { return rep.empty(); }
  void swap(hash_multimap& hs) { rep.swap(hs.rep); }
  friend bool
  operator== __STL_NULL_TMPL_ARGS (const hash_multimap&, const hash_multimap&);

  iterator begin() { return rep.begin(); }
  iterator end() { return rep.end(); }
  const_iterator begin() const { return rep.begin(); }
  const_iterator end() const { return rep.end(); }

public:
  iterator insert(const value_type& obj) { return rep.insert_equal(obj); }
  template <class InputIterator>
  void insert(InputIterator f, InputIterator l) { rep.insert_equal(f,l); }
  iterator insert_noresize(const value_type& obj)
    { return rep.insert_equal_noresize(obj); }
```

```
  iterator find(const key_type& key) { return rep.find(key); }
  const_iterator find(const key_type& key) const { return rep.find(key); }

  size_type count(const key_type& key) const { return rep.count(key); }

  pair<iterator, iterator> equal_range(const key_type& key)
    { return rep.equal_range(key); }
  pair<const_iterator, const_iterator> equal_range(const key_type& key) const
    { return rep.equal_range(key); }

  size_type erase(const key_type& key) {return rep.erase(key); }
  void erase(iterator it) { rep.erase(it); }
  void erase(iterator f, iterator l) { rep.erase(f, l); }
  void clear() { rep.clear(); }

public:
  void resize(size_type hint) { rep.resize(hint); }
  size_type bucket_count() const { return rep.bucket_count(); }
  size_type max_bucket_count() const { return rep.max_bucket_count(); }
  size_type elems_in_bucket(size_type n) const
    { return rep.elems_in_bucket(n); }
};

template <class Key, class T, class HF, class EqKey, class Alloc>
inline bool operator==(const hash_multimap<Key, T, HF, EqKey, Alloc>& hm1,
                       const hash_multimap<Key, T, HF, EqKey, Alloc>& hm2)
{
  return hm1.rep == hm2.rep;
}
```

hash_multimap 的使用方式，与 hash_map 完全相同。