# 第三章、查找

查找即通过给定的条件从数据集合找到所需要的数据

1. 符号表

即以键值对存储的数据结构，查找必须要通过键来查找

* 1. API
     1. 所有对于符号表中的某条具体的数据的操作都必须要通过键。
     2. 符号表是一种抽象数据类型，我们需要指定API，需要实现泛型和迭代
  2. 有序符号表
     1. 作用

对于有序符号表我们有更多可做的操作，例如操作最大最小键、查找键在某一范围的所有数据

* + 1. 实现

所有键都是Comparable的对象，这样可以在插入键值对的时候通过比较来对键排序

* 1. 用例举例
     1. 行为测试用例

符号表可以用于存储键值对，和迭代遍历键值对

* + 1. 性能测试用例

对于符号表处理大型问题，需要能够get()操作更高效，因为无序符号表的get()实现是搜索表中的所有键，对于一个几千条数据的表进行数据处理，其中可能有几百万此搜索。

* 1. 无序链表的顺序查找
     1. get（）的实现

通过比较要查找的键和表中的所有键，如果找到就返回值，否则返回null

* + 1. put（）的实现

也是通过遍历链表，通过equals()来比较要查找的键和表中的所有键，如果匹配成功就更新这个键的值，否则就插入新结点。

* + 1. 顺序查找（基于无序链表）

使用一个私有类来保存结点。get（）会遍历符号表。put（）也会遍历所有键，知道匹配成功就会修改值，匹配失败会添加一个新结点。

* + 1. 实现原理

每个结点保存键和值以及对下一个结点的引用，该对象还会保存一个首结点， 插入新结点的只需覆盖首结点，将原先的首结点的引用加入到新结点中。

* 1. 有序数组中的二分查找
     1. 二分查找（基于有序数组）
        1. 数据结构

将键和值分别用两个数组保存，

* + - 1. get()

使用rank（）找到键保存的位置

* + - 1. put()

如果能找到则更新，否则插入数据，并且将比它大的数据向后移

* + - 1. rank()

使用二分法递归查找键在数组keys的位置，将数组从中间键分为两部分，如果该键大于中间键则继续从右半部分查找，如果该键小于中间键则从左半部分查找，如果相同则直接返回该位置。

* 1. 对二分查找的分析

二分查找的查找速度很快，但是插入很慢。

* 1. 预览

1. 二叉查找树
   1. 基本实现
      1. 数据表示

每个结点含有一个键、值、一条左链接、一条右链接、一个结点计数器

* + 1. 查找

查找从根结点开始，对比查找键和结点键，如果查找键大的话，就递归地从根结点的右子结点开始查找。如果查找键小的话，就递归地从根结点的左子结点开始查找。相等则返回该结点的值

* + 1. 插入

与查找很相似，找到相同的结点就修改值，否则就根据给定键和当前结点键的大小关系来决定继续查找左子树或右子树，直到这个子树是空结点，就返回一个新结点的链接。这就实现了在相应的位置插入一个新结点

* + 1. 递归

对于递归的运行细节我们可以这样理解：对于递归调用前的代码，可以想象沿着树向下走。递归调用后的代码可以想象成沿树往上走

* 1. 分析
  2. 有序性相关的方法和删除操作
     1. 最大键和最小键

取最小键只需查看当前结点是否有左子结点。如果没有则当前结点就是最小键的结点，否则就递归的把左子结点作为当前结点

取最大键则与取最小键同理，只需把左子结点换成右子结点

如果根节点的左链接为空，那么根节点就是最小键。否则递归地从左子树中查找最小键

* + 1. 向下取整和向上取整

如果给定的键key小于根节点的键，那么小于key的最大键一定在根节点的左子树中；如果给定的键大于根结点的键，那么只有当根节点的右子树中存在小于等于key的结点时，目标结点才存在于根节点的右子树中，否则小于key的最大键就是根结点的键。

* + 1. 选择操作

选择操作就是找到排名为k的键（即键中正好右k个小于它的键），如果左子树的结点树t大于k，那么就递归地在左子树中查找排名为k的键。如果t等于k，我们就返回根结点中的键。如果t小于k，我们就递归地在右子树中查找排名为k – t – 1的键

* + 1. 排名

排名就是返回给定键key的排名（即右多少键小于给定键），rank()是select()的逆操作，。如果给定的键与根结点的键相等，则返回左子树的结点总数。如果给定键小于根节点的键，那么则递归地在左子树中查找给定键地排名。如果给定键大于根节点的键，则递归地在右子树中查找排名，并且将给定键在右子树中排名加上根结点左子树的总结点数加一。

* + 1. 删除最大键和最小键

删除最小键我们需要从根结点开始，一直递归地查看左子树，直到根节点没有左子树，这时候我们用根节点的右子树替代根结点（只需要将右子树的引用返回给上一个结点，作为上一个结点的左子树）。

同时递归调用后还需要重新计算结点计数器

* + 1. 删除操作

如果要删除的结点只有一个子结点，我们可以用类似于删除最小键的方法。但是如果有两个子结点就需要用这个结点的后继结点去替代该结点，比如待删除结点有右子结点，那么该结点的后继结点就是右子树的最小结点，并且使用executeDeleteMin()删除右子树的最小结点（后继结点）。然后在递归回调结束之后调整结点计数器就好。如果待删除结点有左子结点，方法也是类似

如果要删除的结点只有一个子结点，我们可以用子结点来代替该结点（只需返回子结点即可），但如果要删除的结点有两个子结点，找到一个后继结点来代替该结点，比如把该结点的右子树的最小结点作为后继结点，因为该结点和该后继结点之间没有任何键，这样不会影响树的结构。只需要一下几步即可完成此替换操作

* + - * 1. 将待删除的结点x保存为t
        2. 将x指向它的后继结点executeMin(t.right)
        3. 将x.right（原本指向空）指向executeDeleteMin(t.right)，也就是删除后所有结点都大于x.key的子树
        4. x.left设为t.left

在递归调用结束后需要重新计算每个结点的结点计数器

* + 1. 范围查找

要实现能够返回给定范围内键的keys()方法，我们首先实现一个遍历二叉查找树的遍历方法—中序遍历。中序遍历需要将所有键按顺序打印出来。要做到这一点我们需要先递归地打印出根结点的左子树所有键，再打印根结点的键，最后递归地打印根结点右子树所有键。

要实现能接受两个参数并且返回给定范围的键的keys()方法，我们需要能够将所有在范围内的键加入到queue中，并且跳过那些不存在给定范围键的子树。

* + 1. 性能分析

1. 平衡查找树

之前所学到的二叉树可能退化成链表

* 1. 2-3查找树

为了保证查找的平衡性，我们需要更加灵活，因此我们允许一个结点保存多个结点。我们将标准二叉树的结点称之为2结点（含有一个键和两个链接），现在我们引入3结点，它含有两个键和三个链接，2结点和3结点的每个连接都对应着由结点的键分割的一个区间。

确切地说，2结点，含有一个键和两个链接，左链接指向小于根结点的键的子树，右链接指向大于根结点键的子树。

3结点，含有两个键和三个链接，左链接指向小于根结点的左键的子树，中链接指向位于根结点的左键和右键之间的子树。右链接指向大于右键的子树。

我们将指向一个空树的链接称之为空链接

一棵完美平衡的2-3查找树中的所有空链接到根结点的距离都是相同的。

* + 1. 查找

将二叉查找树的查找算法一般化可以得到2-3树的查找算法。先将待查找键与根结点的键做对比，和其中任意一个相等则查找命中。否则通过比较的结结果找到相应的链接，再递归地在子树中继续查找

* + 1. 向2-结点插入新键

要在2-3树插入一个新结点，我们可以在树中进行一次未命中的查找，然后将新结点挂在树的底部，但这样无法保持完美平衡。如果未命中的查找结束与一个2结点，我们只需要将这个2结点改为含有2个键的3结点即可。

* + 1. 向一棵只含有一个3结点的树中插入新键

在考虑一般情况前，我们先假设我们需要向一棵只含有一个3-结点的树中插入一个新键。这棵树中有两个键，所以无法再向唯一的结点中插入键了。为了插入新结点，我们先临时的将新键插入到结点中，使之成为一个4-结点，然后我们可以很容易地将一个4-结点转化为含有3个2结点的树，只需要左链接指向左键所创建的2-结点，中键作为根结点键，右链接指向右键所创建的2-结点

* + 1. 向一个父节点为2-结点的3-结点中插入新键

假设一个未命中查找结束于一个3-结点并且父节点是2-结点，这时候我们依然将新键临时保存到3-结点中，使之成为一个4-结点并将其分解。但是我们不会将它转化为3个2-结点，而是将中键放入到父结点中，而父结点中的原中键左边的链接指向原左键所构建的2-结点，父结点中的原中键右边链接指向原右键构建的2-结点。你可以将这次转换看成是将指向原3-结点的链接替换成新父结点的原中键的左右两条链接，并分别指向两个新的2-结点。

* + 1. 向一个父结点为3-结点的3-结点中插入新键

假设一个未命中查找结束于一个3-结点并且父节点是3-结点，这时候我们依然将新键临时保存到3-结点中，使之成为一个4-结点并将其分解。将中键移动到父结点中，但是父结点也是一个3-结点，这样的话，父结点也变成了一个4结点，然后在父结点中进行相同的变换。推广到一般情况，我们就这样递归的一直向上不断分解临时的4-结点，直到遇到一个不需要继续分解的2-结点，或者一个3-结点的根结点。

* + 1. 分解根结点

如果从插入结点到根节点全都是3-结点，我们可以按照向一棵只含有一个3结点的树中插入新键的方式来分解，将一个4-结点分解为3个2-结点。

* + 1. 局部变换

将一个4-结点转换为一棵2-3树可能有6种情况。这个4结点可能是一个根结点，也可能是一个2结点的左子结点或右子结点，也可能是一个3-结点的左子结点或中子结点或右子结点。这些变换都是局部的，除了相关的结点和链接之外不需要再检查树的其他部分，需要指出的是，不光是在树的底部，任何符合这个模式的4-结点都可以使用这种变换。可以将4-结点的中键移动到父结点中，并且这个原中键的左右两个链接分别指向原左键构建成的结点和原右键构建的结点。

* + 1. 全局性质

这些局部变换不会影响全局的有序性和平衡性：任意空链接到根结点的距离都相等。假设变换之前任意空链接到根结点的距离是h，那么变换之后空结点到根结点的距离任然为h。即使是将一个4-结点分解为两个2-结点并且将中键移动到父结点中。因为在变换中树的高度没有发生改变，并且所有空链接任然在最底层。除了将根结点分解为3个2-结点，这个时候树的高度增加了1，所以空链接到根结点的距离变为h+1。

和标准的二叉查找树的向下生长不同，2-3树是向上生长的。它的插入方式只有将一个2-结点变成3-结点，或者一个3-结点临时变成4-结点，再将4-结点分解为2个2-结点以及一个移出的中键，如果这个结点有父结点就将这个移出的中键放到父结点中，如果这个结点是父结点那么移出的中键就单独作为一个根结点。

2-3树在最坏情况下任然有比较好的性能

但是距离真正的实现还有一段差距，因为要处理的情况实在太多，很多操作的实现不方便

* 1. 红黑二叉查找树

红黑树使用由左斜的红链接相连的两个2-结点来表示3-结点。这样的话，红黑树即可以通过转换表示为2-3树，又是一个标准的二叉树。好处在于即可以使用标准二叉的简洁方法来实现各种操作，又可以用2-3树的分解4-结点的方式来进行插入以保持完美的黑链接平衡

* + 1. 替换3-结点

红黑树的思想是用标准的二叉查找树（完全由2-结点构成）和一些额外的信息（替换3-结点）来表示2-3树。我们用由一个红色左链接相连的两个2-结点表示一个3-结点。这中表示法的优点是我们可以用标准二叉树的get（）方法进行查找。对于任意的2-3树，我们只要对结点进行转换，我们都可以立即派生出一棵对应的二叉查找树

* + 1. 一种等价的定义
       1. 任何一个结点不可能与两个红链接相连
       2. 任何一个空链接到根结点之间的黑链接数相同。
       3. 红链接均为左链接
    2. 一一对应

如果将所有的红链接放平的话，我们可以看到所有的空链接到根结点的距离相同。首先，我们可以将红黑树在二叉树和2-3树之间进行转换，如果我们将所有由左斜的红链接相连两个2-结点合并就变成了一个2-3树，反之如果将2-3树的所有3-结点表示为由左斜的红链接相连的两个2-结点，它就变成了一个标准二叉树。红黑树所有空链接到根结点之间的黑链接树相同，因为黑链接就是原本2-3树的普通链接。并且所有结点都不可能同时连接两个红链接。如果我们能够结合二叉树和2-3树的算法，我们就可以同时使用二叉树简洁的查找算法，和2-3树插入的完美平衡。算法

* + 1. 颜色表示

为了方便，因为每个结点都只会有一条指向自己的链接（由它的父结点指向它），我们将链接的颜色信息保存在这条链接指向的结点的color变量中，如果color的值为true则表示红色，如果为false则为黑色。为了代码简洁。我们将这两个值保存为常量RED和BLACK，我们用isRed()来判断一个结点和它的父结点之间的链接颜色。

* + 1. 旋转

在我们实现的某些操作可能会出现红色的右链接或者连续的红链接，但在完成操作前我们会通过旋转来修复它。旋转操作会改变红链接的指向。例如我们有一条红色的右链接需要被转化为红色左链接，这个操作叫做左旋转。这个方法接受一个指向红黑树某个结点的链接，它会返回一个左链接是红色的根结点的链接。这个方法只是将一个较小值作根结点改为由一个较大值为根结点。实现右旋转和左旋转的逻辑完全相同。

* + 1. 在旋转后重置父结点的链接

左旋转和右旋转都会返回一个链接，来重置父结点的相应链接。这个链接可能是红色也可能是黑色，这可能会产生两条连续的红链接，但我们的算法会使用继续用旋转来修正这种情况

在插入新的键我们可以用旋转来保证树的完美平衡性和有序性，下面我们需要保证红黑树的另外两条性质，不存在两条连续的红链接和不存在红色的右链接。

* + 1. 向2-结点中插入新键

一棵只有一个键的红黑树只有一个2-结点，向它插入一个结点后，我们需要立即进行旋转操作。如果插入的新键小于根结点的键，我们会新增一个红色左链接指向的结点。如果插入的新键大于根结点的键，我们会新增一个红色右链接指向的结点，这时候我们就需要root=rotateLeft(root)来进行左旋转，使之变成一个由一个根节点和一个红色左链接指向的结点，并修正根结点的链接。两种情况都会得到一棵和单个3-结点等价的红黑，其中含有两个键，一个红色左链接，黑链接的高度为1。

* + 1. 向树底部的2-结点插入新键

用和二叉查找树相同的方式向一棵红黑树插入一个新键会在树的底部新增一个结点（为了保证树的有序性）。然后将新结点用红链接向父结点相连，如果这个红链接是左链接，那么这就是一个3-结点。如果这个红链接是右链接，那么这个3-结点就是错误的，我们需要一次左旋转来修正它。

* + 1. 向一棵双键树(即3-结点)插入一个新键

这种情况又可以分为三种子情况：新键小于树中的两个键，新键在树的两个键之间，新键大于树中的两个键。每种情况都会产生一个连接两个红链接的新树。而我们的目标就是修复它

* + - 1. 新键大于树中的两个键

这是最简单的情况，此时树是平衡的，中键作为根结点，它有两个红色链接分别指向较大的结点和较小的结点。我们只需要将两个红链接变为黑链接即可，就类似于将一个4-结点分解为3个2-结点。，此时新树任然是完美平衡的，并且黑链接的高度+1。其他两种情况最终也会转化为这种情况

* + - 1. 新键小于原树中的两个键

此时原树中的较小键会有一个红色左链接指向新键，此时产生了两条连续的红色左链接，我们只需要将上层的红色左链接进行右旋转就变成了第一种情况（中键为根结点并且有两条红色链接指向其他两个结点）。

* + - 1. 新键介于原树中两个键之间

此时原树中较小键会有一条红色右链接指向新键。此时我们只需要将下层的红色右链接进行左旋转就变成了第二种情况（两条连续的红色左连接）

总之我们通过0次、1次、2次旋转得到了预期的结果

* + 1. 颜色转换

我们用一个专门的方法flipColors()来转换一个结点的两个红色子结点的颜色。除了将子结点的颜色由红转黑之外，还将父结点的颜色由黑转红。这个操作是局部变换，不会影响到全局的黑色平衡性。

* + 1. 根结点总是黑色

颜色转换会使根结点变为红色。红色的根结点说根结点是一个3-结点的一部分。但实际上，我们每次插入都会把根节点变为黑色。每次把根节点变为黑色时黑链接的高度就会+1。

* + 1. 向树底部的3-结点插入新键

现在假设我们向一个树底部的3-结点下插入一个新键。这时候我们之前提到的3中情况都会出现，可能向3-结点中插入一个右链接（只需要进行颜色转换即可），也可能向3-结点中插入一个左链接（只需将上层链接进行一次右旋，再进行颜色转换），也可能向3-结点中插入一个中链接（先将下层链接进行左旋，再将上层链接进行右旋，最后进行颜色转换）。颜色转换后根结点会变成红色，这相当于将一个送入父结点，相当于在父结点中插入一个新键，我们继续用相同的方法解决这个问题

* + 1. 将红链接在树中传递

2-3树的插入算法需要我们分解3-结点，将中键送入到父结点中，直到遇到一个2-结点或根结点。我们考虑的所有情况都是为了达成这个目标：每次必要的旋转之后我们都会进行颜色转换，这使得中结点变红。这对于父结点来说处理一个这样的红色结点和在底部结点插入新的红色结点是一样的，即继续把红色结点转移到中结点中去。在红黑树中实现2-3树的插入算法的关键操作所需的步骤：要在一个3-结点中插入新键，先创造一个临时的3-结点，再将其分解，由中结点将红色链接继续向上传递给它的父结点。重复这个过程，我们就能将红链接继续向上传递，直至遇到一个2-结点或 根结点。

我们只要谨慎地使用左旋转、右旋转、颜色转换，就可以实现和2-3树一一对应的插入操作。在沿着插入结点到根结点之间的所有结点顺序执行以下操作，就可以完成插入操作

* + - 1. 如果左子结点是黑色的，而右子结点是红色的，对右链接进行一此左旋转
      2. 如果左子结点是红色的，而且左子结点的左子结点也是红色的，对上层链接进行一次右旋转
      3. 如果左子结点是红色的，右子结点也是红色的，进行一次颜色转换

请注意：第一步操作包括两种情况，向一个2-结点中插入一个新结点，和插入新结点通过中链接向父结点相连

* + 1. 实现

保持树的平衡性是需要在插入的结点从下至上保持的。把这算法植入到代码中也很简单。只需要在递归调用结束之后进行结点颜色的检查，然后进行一两次的旋转或者一次颜色翻转。

* 1. 红黑树的插入算法

红黑树的插入算法除了三个if语句其他的与标准二叉树相同。第一条if语句将新插入结点或临时4-结点的的红色右链接左旋。第二条if语句两个连续的红色左链接中的上层链接进行右旋。第三条if语句将4-结点进行颜色翻转。

* + 1. 红黑树在插入时保持平衡

如果要插入一个新键，首先会在树中从根结点开始向下查找直到遇到一个相同的键，就更新这个键。或者遇到一个空链接就插入一个新结点并且通过一个红色链接与父结点相连，这个过程是黑链接平衡的，但是可能会有两个个问题。一有可能产生一个红色右链接，这时候我们将它进行左旋操作即可。二是有可能产生两个连续的红链接，这个时侯我们通过0次、1次或2次旋转操作，再加上一次颜色翻转操。递归地在父结点也做这些操作，即可解决这个问题

* 1. 删除操作

和插入操作一样，我们需要定义一系列局部变换来在删除一个结点的时候保持平衡。我们需要在（为了删除一个结点）构造临时4-结点的时候向下进行局部变换保持平衡，还要再删除后递归结束时向上处理遗留的4-结点

* + 1. 自顶向下的2-3-4树

2-3-4树的插入算法，我们需要实现沿着树既能向下变换也能向上变换的算法，沿树向下变换时为了保证底部的树不是4-结点，需要将结点进行分解，将结点中的一部分键送入到父结点中。再递归调用结束后，向上的过程中需要利用旋转将结点配平

* + 1. 删除最小键

我们注意到删除底部的3-结点是非常简单的，只需将3-结点变为2-结点即可。但是删除底部的2-结点则不然，我们可以把删除的2-结点用空链接替代，但是这样会破坏完美平衡性。所以我们要在沿根结点向下的过程中保持结点不是2-结点，可以是3-结点，也可以是临时的4-结点。对于根结点可能有两种情况：如果根结点是2-结点并且它的左右子树都是2结点那就直接将之合并为临时的4-结点。否则需要保证左子结点不是2-结点，必要时候可以从右子结点借一个键来。在沿着左链接向下的过程中，保证以下情况值以成立：

* + - 1. 如果当前结点的左子结点不是2-结点，完成。
      2. 如果当前结点的左子结点是2-结点，并且左子结点的亲兄弟结点（离它最近的兄弟结点）不是2-结点，则从它的亲兄弟结点借一个键过来。
      3. 如果当前结点的左子结点是2-结点，并且它的亲兄弟结点也是2-结点。把当前结点的最小键和左子结点和离它最近的兄弟结点合并为4-结点。把它的父结点由4-结点变为3-结点，或者由3-结点变为2-结点。

在遍历的过程中执行这个过程，我们最终可以得到一个在底部的3-结点或4-结点，然后从中删除一个键。最后向上分解临时的4-结点（将4-结点分解为2个2-结点并且将中键送到父结点中）。

对于红黑树的删除最小键我们只需要保证当前结点的颜色是红色就可以，因为到底部时最小结点为红色的时候，直接删除红色结点，不会影响黑色链接平衡性。

* + 1. 删除操作

和二叉树的删除操作一样，先向下找到相应的结点，然后将该结点右子树的最小键作为后继结点，再将该结点与它的后继结点交换位置，最后以删除右子树最小键的方式删除目标结点。

每次沿左子树或右子树向下查找时必须保证相应的子结点是红色，否则可以使用moveRedLeft()或moveRedRight()方法。

* 1. 红黑树的性质
     1. 性能分析
     2. 有序符号表API

1. 散列表

如果所有的键都是小整数，那么使用数组将键作为索引。将值存储在数组对应索引所指向的值中。

使用散列的查找算法分为两步。第一步是将键通过散列算法转换为相应的索引。在数组对应索引中存储键和值。但是有可能存在不同的键却有相同的散列值。因此第二步就是处理碰撞冲突。有两种方法可以解决：拉链法，线性探测法。

* 1. 散列函数

散列函数会将键转化为数组的索引，如果我们有一个能保存M个键值对的数组，我们的散列函数就需要能够将任意键转化为数组范围内的索引（【0，M-1】范围内的整数）。

散列函数与键的类型有关。比如键是整数直接用键进行计算即可。如果键是字符串，就需要先将字符串转化为数字。如果键包含多个部分，比如邮件地址，我们就需要将多个部分结合起来。

* + 1. 典型的例子

假设键是美国的社保号，它由三部分组成，比如123-456-789。第一部分是颁发的地区，其他部分是身份信息。假设我们的应用程序只需要处理几百个键值对。我们可以使用一个M=1000的数组。一种实现方法是直接使用第三部分的数组作为索引，因为它们的重复率更低。但更好的方法是将三个部分的数组转化为一个整数。

* + 1. 正整数

将正整数散列的最常用方法是除留余数法，对大小为M的数组，正整数k的散列值就是K除以M的余数。这种方式可以将任意整数k散列到【0，M-1】。但如果M不是素数，可能无法利用到k的所有位，如果k是十进制树并且M=10的n次方，那么除留余数法只能利用到k的后n位数字。所以M为素数（2的幂除外）会更好

* + 1. 浮点数

如果键是0到1之间的实数，我们将键乘以M再四舍五入可以得到0到M-1之间的整数。但这种方法的缺陷在于键的高位起的作用更大，最低位对结果没有影响。一种解决方法就是把k转化成二进制数再对M做除留余数法。

* + 1. 字符串

除留余数法也可以用来处理字符串，只要把它当作一个大整数即可，

int hash = 0;

for(int I = 0; i < s.lenth(); i++){

hash = (R \* hash + s.charAt(i)) % M

}

Java的charAr（）方法是返回字符串某一位字符的整数值，如果R大于所有字符的整数值，这段代码就代表把每个字符当作N位的R进制树。再对M做除留余数法

* + 1. 组合键

如果键的类型含有多个整型变量，我们可以和string类型一样将它们混合起来。例如我们查找的键的类型是Date，其中含有多个整型：day、month、year。可以用以下方法来计算散列值

int hash = (((day \* R + month) % M) \* R + year) % M;

只要R足够小不造成溢出，我们可以得到0到M-1的散列值，在这种情况下，我们选择一个较小的M比如31就可以省略括号内的%M计算

* + 1. java的约定

每个数据类型都需要散列函数，所有每种数据类型都有hashCode()方法，并且需要和equals()一致，就是说如果a.equals(b)返回true，那么a和b的散列值也要相等，反过来散列值相等a和b却不一定相等，还要通过equals来确定。

* + 1. 将hashCode()的返回值转化数组的索引

我们需要的是数组的索引而不是32位的散列值，我们可以通过hashCode方法和除留余数法来得到一个数组索引

function hash(key k){

int hash = (k.hashCode() & 0x7fffffff) % M;

return hash;

}

这段代码会把符号位屏蔽（把32位整数转化为31位的非负整数），然后对M取余，我们一般会将数组的大小M取为素数，以充分利用原散列值的所有位。

* + 1. 自定义hashCode方法

散列表的用例希望能够将键平均的散列为任意的32位整数也就是说，对于任意对象x，都可以调用x.hashCode()并有均等的机会得到一个随机的2的32次方整数。在java中每一种数据类型都实现了hashCode()方法，因此对于对象的一个简单的做法是将每一个变量调用hashCode()方法，将返回值转化为32位整数并计算得到散列值。如transaction所示

public class transaction{

String who;

Date when;

double amount;

public int hashCode(){

int hash = 17;

hash = hash \* 31 + who.hashCode();

hash = hash \* 31 + when.hashCode();

hash = hash \* 31 + ((double)amount).hashCode();

return hash;

}

}

* + 1. 软缓存

如果散列值的计算很耗时，我们可以把键的散列值缓存起来。第一次调用hashCode（）方法我们需要进行计算，然后将返回值保存到键的hash变量中，之后再调用就直接从hash变量中取值。

一个优秀的hash算法需要满足以下三个条件

* + - 1. 一致性

相同的键必然返回相等的散列值

* + - 1. 高效性

计算简便快速。

* + - 1. 均匀性

有均等的机会得到每一个可能的散列值

均匀的散列所有键

* + 1. 1
    2. 1
  1. 基于拉链法的散列表

一个散列函数可以把键转化为数组索引，第二步是处理冲突，也就是解决多个键的散列值相同的情况。一种简单的方法是把数组的每个元素指向一条链表。把所有散列值为该元素索引的键值对放入该链表。

算法3.5 SeparateChainingHashST

* + 1. 散列表的大小

在实现基于拉链法的散列表的目标是找到一个合适的数组大小M。要实现即不因M太大而浪费内存，也不因M太小而加大查找时间。一种方法是动态调整数组大小。

* + 1. 删除操作

要删除一个键值对，先通过键找到相应的SequentialSearchST对象，再调用该对象的delete()方法。

* + 1. 有序性相关的操作

散列表的目标就是将键值对散列到不同的位置，然后位置信息就丢失了。

* + 1. 1
  1. 基于线性探测法的散列表

实现散列表的另一种方式是用空元素代替链表，假设键值对个数为N，数组大小为M，使M>N，如果散列值在某个数组索引发生了碰撞，就将按照某个规则寻找下一个索引。基于这种思想的叫做开放寻址法。

查找下一个索引有可能出现下面三种情况

* + - 1. 查到一个空元素（未命中）
      2. 查到一个元素，键相同（命中）
      3. 查到一个元素，键不同（继续查找）

最简单的实现就是线性探测法，如果在散列值在某个索引发生了碰撞，就将索引加1，如果索引值已经达到最大，就从头开始查找。因为M>N，所以总能找到一个空元素。

* + 1. 删除操作

我们不能简单的把要删除的键设为null，因为这样会找不到下一个碰撞的键，我们应该把所有被删除键右侧的键重新插入一遍。

* + 1. 键簇

线性探测平均成本取决于元素在插入数组后聚集成的一组连续的条目。叫做键簇，也就是所有碰撞后形成的连续的不为空的元素。

* + 1. 线性探测法的性能分析
  1. 调整数组大小

我们可以动态调整数组大小来保证表的使用率不超过1/2.

* + 1. 拉链法
    2. 均摊分析
  1. 内存使用

1. 应用

符号表的应用非常广泛

* 1. 我应该使用符号表的哪种实现

散列表的优势在于代码结构简单，查找时间最优，但是不支持顺序操作。二叉树的优势在于抽象结构简单。红黑树的优势在于最坏情况下查找最优，且支持多种操作。多数情况下的第一选择是散列表，除非其他操作更重要。

* + 1. 原始数据类型

加入我们有一张符号表，整型的键对应浮点型的值。对于标准实现，我们会把键和值保存在Key和Value对象中。这会额外增加两个引用，在大型应用中会带来额外的开销。我们可以把它们保存到原始数据类型中。

* + 1. 重复键

符号表的实现需要考虑重复键的可能，许多应用都希望能够为一个键保存多个值。符号表不允许多个键，用例只能自己保管多个键。

* + 1. java标准库

在本书中用ST代表RedBlackBST,用HashST代表有序性操作无关紧要的LinearProbingdHashST。

* 1. 集合的API

某些应用程序不需要对值进行操作，只需要插入键，并检测键是否存在。

* + 1. 集合数据类型的一组API
       1. SET

创建一个空的集合

* + - 1. add

将键加入到集合

* + - 1. delete

删除键

* + - 1. contains

键key是否在集合中

* + - 1. isEmpty

集合是否为空

* + - 1. size

集合中键的数量

* + - 1. toString

对象的字符串表示

* 1. dedup

过滤器的原型是是dedup，用于检测输入的键是否存在，如果存在则忽略它，否则将它加入到dedup中并打印。

* + 1. 白名单和黑名单

过滤器的另一个经典应用是白名单和黑名单。白名单可以忽略不在白名单中的键，黑名单功能也类似。

* 1. 字典类用例

符号表最简单的应用就是连续向符号表中put(),并进行get（）操作。

* 1. 索引类用例

字典的主要特点是每个键都有一个关联的值，但有时候一个键可能关联多个值，这种情况下我们只需要用某种数据结构（例如queue）代替即可

* + 1. 反向索引

在某些系统中一个键对应一个SET类型的值，而反向索引就是通过这个SET类型中的某个关键字来查找到对应的键。

* 1. 稀疏向量

我们要考察的简单计算就是矩阵和向量的乘法。