# 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. 12
2. 12
3. 1