二叉查找树的函数式语义实现

The Binary Search Tree Implementation by Functional Semantics

谭杰锋　(湖北三峡职业技术学院　湖北宜昌 443000)

摘要:二叉查找树是数学建模、算法分析中常用的数据结构。函数式语义具有无副作用特性与类型高度抽象能力,用其表达数学模型简练明了。本文采用属于函数式语义的类 haskell 伪码实现了二叉查找树,并进行了相关算法分析。

关键词:二叉查找树　函数式语义　Haskell

1　函数式语义

函数式语义(Functional Semantics)是四种程序语义(命令式、函数式、逻辑式、面向对象式)之一。它的基本运算单位是函数,函数在此语义系统中和其他数据类型一样也是"普通公民",可以进行传递和高阶函数运算。因为每个变量在一个函数运算期间都是唯一绑定,不能改变其状态,所以此种语义具有无副作用(no side－effect)特性。同时因拥有强大的类型演算系统,具有类型高度抽象能力,所以函数式语义尤其适合表达数学运算、算法与数据结构等模型,而且十分简练。近年来众多函数式语言中,Haskell 语言日趋完善,发展良好。

2　二叉查找树

二叉查找树(Binary Search Tree,BST),或称二叉排序树(Binary Sort Tree,BST),是一种常用的数据结构,作为抽象数据类型动态查找表(DynamicSearchTable)的一种实现方式及二叉树排序算法的基本工具,是数学建模、计算机程序设计、算法分析等领域的常用数学模型。

二叉查找树是这样一类二叉树:或者为空,或者所有结点的左子结点(如果不为空)值都小于其父结点值并且所有结点的右子结点(如果不为空)值都大于其父结点值。

其中值的类型是任意的,其大小关系是可进行确定性再定义的。通常使用中值的类型是整数或字符串。整数按照其数学大小定义比较,字符串依次按照其字符序列的字典顺序(或 ASCII 码值或其他字符集编码值)进行比较。

下面是一棵二叉查找树:

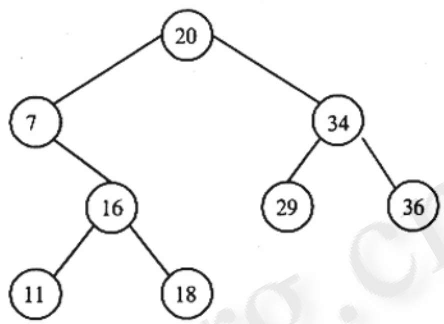

图 1　一棵二叉树

BST 十分适合内存中小量数据的快速查找和组织,并且数据往往是(key, value)二元组,数据的 key 作为结点值参与比较。基本的操作有查找、插入、删除。其中查找和插入算法比较简单,删除算法稍需讨论。三种对应算法用自然语言表述如下:

查找:给定欲查找 key 与 BST(的根结点)为当前结点,若当前结点值与 key 相同,则找到,返回此结点存储的 value。若 key 比当前结点值小(或大),则继续向左(或向右)查找,即将当前结点的左结点(右结点)作为当前结点递归调用本算法。若当前结点为空,说明树中没有此 key,返回未找到标志或根据需要返回当前结点(以便在此插入)。

插入:给定欲插入的(key, value)对与 BST(的根结点)为当前结点。首先按此 key 查找,如果找到可根据需要返回重复标志或者重复插入/替换或者简单忽略返回成功。若未找到,并由查找算法返回了合适的

结点，则在此处插入。

如要在图1所示的二叉查找树中插入数32，虚线所示为查找及插入路线：

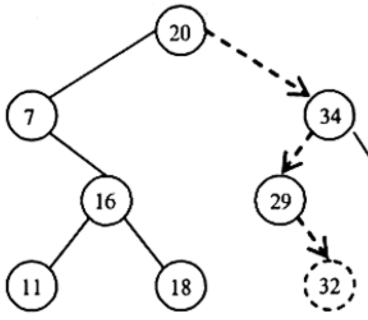

图 2 二叉查找树的查找与插入

删除：给定欲删除的 key 与 BST（的根结点）为当前结点。按此 key 查找，若未找到则返回相应标志；若找到应删除结点，则删除之，并进行调整，目的是删除结点后依然保持二叉查找树"左小右大"的性质。调整分为三种情况：

(1) 应删除结点是叶结点，即左右都为空，则直接删除。

(2) 应删除结点有且仅有一边子树为空，则删除后非空子树上提代替此位。

(3) 应删除结点两边子树均不为空，则删除后由左子树中最大结点（或右子树中最小结点，也即删除结点的线索/中序遍历的前驱或后继）换至此位。至于左子树中最大结点的寻找，即从左子树一直右行直至走不动为止。停下来的结点即是左子树中最大结点，且必定符合前两种情况之一，所以可以按前法去掉并替换至应删除结点位置。

3 实现

函数式语义十分适合表达算法与数据结构，相较于其他命令式语言（如 c），函数式程序显得十分简洁。下面用类 haskell 伪码来表述上述算法。

首先定义抽象数据类型：

BSTree :: Nil | Node a b (BSTree a b) (BSTree a b)

说明此树要么是空树，要么结点带有（key, value）对以及两边子结点。

查找函数 search 具有类型 a -> BSTree a b -> Maybe b 其中 a，b 任意类型，Maybe b 表示结果可能为 b，也可能找不到。

search :: a -> BSTree a b -> Maybe b

search _ Nil = Nothing -- 空树则返回找不到

search n (Node k v l r) -- k，v 为当前结点 key，value. l，r 分别为左右结点

-- 下面进行模式匹配

| n = = k = Just v -- 找到，返回 value

| n < k = search n l -- 进入左子树递归

| n > k = search n r -- 进入右子树递归

插入函数 ins 具有类型 a -> b -> BSTree a b -> BSTree a b，给定（key，value）和现有树，返回插入后新树。这里如果碰到已存在结点我们简单的忽略，返回成功。

ins :: a -> b -> BSTree a b -> BSTree a b

ins k v Nil = Node k v Nil Nil -- 空树插入新结点

ins k v (Node rk rv left right)

| k = = rk = Node k v left right -- 重复结点简单忽略

| k < rk = Node rk rv (ins k v left) right -- 进入左子树递归

| otherwise = Node rk rv left (ins k v right) -- 进入右子树递归

下面是结点删除算法，在给出此算法实现之前，需要一个找出左子树最大点的函数 max，以便后面替换到被删除节点。

max :: BSTree a b -> (a，b) -- 给定左子树返回最大 key 的（key，value）

max (Node rk rv _ Nil) = (rk，rv) -- 向右走到尽头了，说明当前结点即是结果

max (Node _ _ _ right) = max right -- 如果没有到尽头则继续向右递归

下面是删除算法：

del :: a -> BSTree a b -> BSTree a b -- 给定欲删除 key 和现有树，返回删除后新树

del _ Nil = Nil -- 空树，什么也不做

del n (Node root v left right)

-- 当前结点为欲删除结点，则判断三种情况：

-- 左右都为空，即本身为叶结点，直接删除自己

| n = = root && left = = Nil && right = = Nil = Nil

-- 有且仅有一边为空，则将非空一边子树提上代替自己

（下转第 94 页）

（上接第104页）

```
| n = = root && right = = Nil = left
| n = = root && left = = Nil = right
- - 左右都不空，则寻找左子树最大结点替换之
| n = = root = Node x y ( del x left) right
- - 当前结点未匹配欲删除结点，则根据大小进入两边子树递归
| n < root = Node root v ( del n left) right
| otherwise = Node root v left ( del n right)
where (x,y) = max left
```

在随机输入序列的情况下，二叉查找树趋向一棵完全二叉树，设 k 为树高，则节点数 $n = k(k-1)/2$，最高次项为 k2。查找的算法需要到达叶结点，故需要 k 步，即时间复杂度为 O(logn)。同理，其插入和删除算法时间复杂度也是 O(logn)。

4 结语

本文采用类 haskell 的伪码实现二叉查找树，充分利用了 haskell 语言的代数类型和模式匹配的功能，容易转换为实际的 haskell 代码，程序规模小巧，代码简明，容易维护。在实践中，为了提高搜索效率，减小平均查找长度，通常还需要结合平衡二叉树的技术来进一步完善此模型。

参考文献

1 Jacques Loeckx，Kurt Sieber. The Foundations of Program Verification.

2 严蔚敏、吴伟民，数据结构[M]，北京：清华大学出版社，1996.

3 Simon Thompson. Haskell：The Craft of Functional Programming，2ndEd[M] Addison – Wesley，1999.