搜索结构(search)

2019 年 3 月 15 日

1 搜索

搜索，就是在数据集合中寻找满足某种条件的数据元素。可以用于搜索的数据集合称为搜索结构，在搜索结构中所有元素（对象）均为同一数据类型。

所有对象中可能有若干属性，其中必须一个可以用于标识这个对象，即key。对于key的搜索，结果必须是唯一的；而对于其他属性的搜索，结果可以不唯一。

静态环境就是在执行插入和删除操作时数据结构不发生变化，在这种环境中进行搜索称为*静态搜索*；*动态环境*就是在执行插入和删除操作时数据结构会进行一定的调整，在这种环境中进行搜索称为*动态搜索*。

为提高搜索效率，对于不同数据采用不同的搜索结构。如搜索电话号码时，可将电话号码分段，即分块搜索；对于英文字典，可以按字母进行折半搜索。

在搜索过程中，可以用关键码的平均比较次数或磁盘的平均读写次数衡量搜索效率，这个标准可以统称为*平均搜索长度*（*ASL, average search length*）。

经典的搜索结构 静态搜索结构、二叉搜索树、AVL树、伸展树、红黑树。

静态搜索结构 基于数组的数据表类，即静态搜索表。

> 根据给定值k，在数组中进行搜索，直至找到k的位置或确定找不到k值。如果是典型的数据类型，可以直接比较；如果是自定义的数据类型，可以对key值进行比较。

基于有序顺序表的：顺序搜索和折半搜索 有序顺序表需要保证元素在插入时保证有序，比如从小到大排列。如此可以增加搜索效率。

顺序搜索时间复杂度为$O(n)$，而折半搜索时间复杂度为$O(\log_2 n)$；但插入操作时间代价为$O(n)$。

二叉搜索树 折半搜索对于顺序搜索有着性能上的优势，因此可以构造类似于使用折半搜索的树形结构实现快速搜索，即二叉搜索树。

AVL树 当二叉搜索树某子树过长，将会增加平均搜索长度。为了尽可能缩短平均搜索长度，可以使用高度平衡搜索二叉树。

伸展树 将经常访问的结点逐渐上移，以提高搜索速度。

红黑树 将每个结点附件了颜色信息，并且：1）根结点和所有外结点均为黑色；2）根到外部结点没有连续的红色；3）根到外部结点有相同数量的黑色结点。

2 二叉搜索树（binary search tree）

二叉搜索树：1）每个结点上附有key；2）**左子树**上所有结点key均比根结点**小**；3）**右子树**上所有结点key均比根结点**大**；4）左右子树也是二叉搜索树。

根据上述特征，一棵二叉搜索树按中序遍历后，可以将key按从小到大排列，故二叉搜索树也称二叉排序树（binary sorting tree）。

2.1 抽象数据类型接口（ADTI）

二叉搜索树和二叉树在诸多方面十分类似，这里重点关注与普通二叉树不同的地方，如搜索、插入、删除操作。

搜索 搜索key值为x的结点：先检查根节点如果根节点为NULL，则搜索不成功；否则根据x和根节点的比较结果：1）如果相等，返回结果；2）如果小，搜索左结点；3）如果大，搜索右结点；4）递归进行上述过程直至叶节点。

插入 插入操作需要在搜索的基础上进行。插入元素之前，需要检查该元素是否存在1）存在的话，返回错误，不插入；2）不存在的话，则在搜索位置处，插入新元素。

并且插入函数的输入参数应至少含有开始搜索结点，以**引用**的方式作为输入。如果开始搜索结点为空，直接修改其值指向新结点；如果开始结点不为空，且待插入值<当前结点，则递归向左子树插入；如果开始结点不为空，且待插入值>当前结点，则递归向右子树插入。

插入操作，每次必须从根节点开始，以保证整棵树有序。

删除 删除操作除了删除结点外，最重要的是需要考虑*重链*问题。删除结点后，断链如何在保证有序的条件下重新链接，都是在删除时要考虑的问题。

当删除某结点时，1）如果其左右子树均为NULL（叶节点），将其父结点指向当前结点的指针置NULL，并删除当前结点即可；2）如果当前结点仅左子树为NULL，将其右子树链接到当前结点的父结点；3）如果当前结点仅右子树为NULL，将其左子树链接到当前结点的父结点（即如果只有一个子树，将该子树代替当前结点）；4）*如果左右结点均不为NULL，需要在*

其右子树中，选出关键码最小[1]的一个结点，替换当前结点。随后调整其右子树使其有序。

根据二叉搜索树特性，最左下角结点的值最小，最右下角的值最大。因此在右子树中找最小结点，只需找到其最左下角结点即可。

因为涉及到断链重链，因此输入必须是引用。

当两子树均不为空时，具体操作为：1）寻找右子树的最左下角结点（右子树中最小值）；2）将该结点的值赋给当前结点

[1]大于当前结点且最小的；或者在左子树中找关键码最大的

3 高度平衡的二叉搜索树（AVL）

高度平衡的二叉搜索树由G.M.Adel'son-Vel'skii和E.M.Landis于1962年提出，因此取姓名首字母称之为AVL。AVL相较于二叉搜索树有着较短的平均搜索长度，因而具有较高的搜索效率。

对于AVL，其左右子树均是AVL树；且左右子树高度之差的绝对值小于等于1。对于每个结点，其*右子树高度*减去*左子树高度*所得高度差，称为平衡因子bf（balance factor）。因此，每个结点的平衡因子只能是-1,0,1。

AVL树如果有n个结点，其高度可以保持在$O(\log_2 n)$，平均搜索长度为$O(\log_2 n)$。

3.1 平衡化旋转

当插入新结点时，容易使原本平衡的二叉树变得*不平衡*，此时必须调整树的结构，使之平衡。平衡化旋转有两种：*单旋转*（左单旋转、右单旋转）和*双旋转*（先左后右双旋转、先右后左双旋转）。

当插入新结点时，需要依层检查父结点的左右子树是否平衡。当发现某结点的左右子树不平衡（高度差≥ 2）时，回溯刚才检查路径上的两层结点，如果是“不平衡结点”$\rightarrow$“右子树”$\rightarrow$“右子树”，即一条直线，则可以使用（左）单旋转；如果是一条折线，如“不平衡结点”$\rightarrow$“左子树”$\rightarrow$“右子树”，则需要使用（先左后右）双旋转。

直线使用单旋转，折线使用双旋转。

单旋转 单旋转的情况如下图所示。原本树是平衡的（结点A的左右子树高度差为1），当在右下角插入新结点C时，破坏了原本树的平衡性。此时可以从新结点C开始，向上层检查各层结点的左右子树是否平衡，在本例中，可以检查出A结点的左右子树高度差≥ 2不平衡。此时回看检查路径是一条直线，因此可以使用单旋转，使原树平衡。

左旋转过程是：1）将不平衡结点A的右子树B的左子树链接到结点A上；2）将结点A改为B的左子树，形成右图中平衡的树。即对A的右结点，和B的左结点做了修改。

左右旋转互为镜像。

直线和折线的判断 平衡因子（bf）是有符号的，一般是右子树高

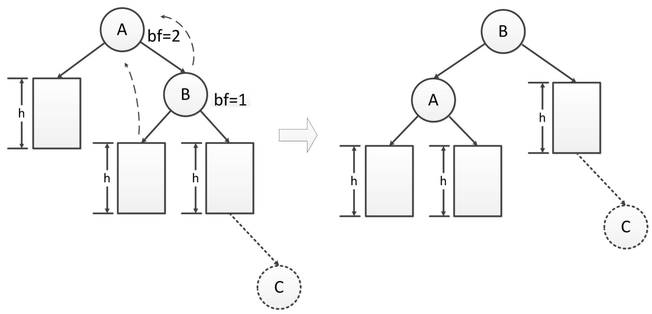

图 1: 左旋转示意图

度−左子树高度，因此如果bf>0，右子树较高。所以当上图中A和B的平衡因子同号（均>0），为直线；如果异号，为折线。

双旋转

4 伸展树（splaying tree）

5 红黑树（red-black tree）

参考文献

[1] 严蔚敏. 数据结构（C语言版）. 北京: 清华大学出版社, 2007.

[2] 邓俊辉. 数据结构（C++语言版）(第三版). 北京: 清华大学出版社, 2013.

[3] 李春葆. 数据结构考研指导. 北京: 清华大学出版社, 2002.

[4] 殷人昆. 数据结构：用面向对象方法与C++描述. 北京: 清华大学出版社,1999.