数据结构与算法第四次作业

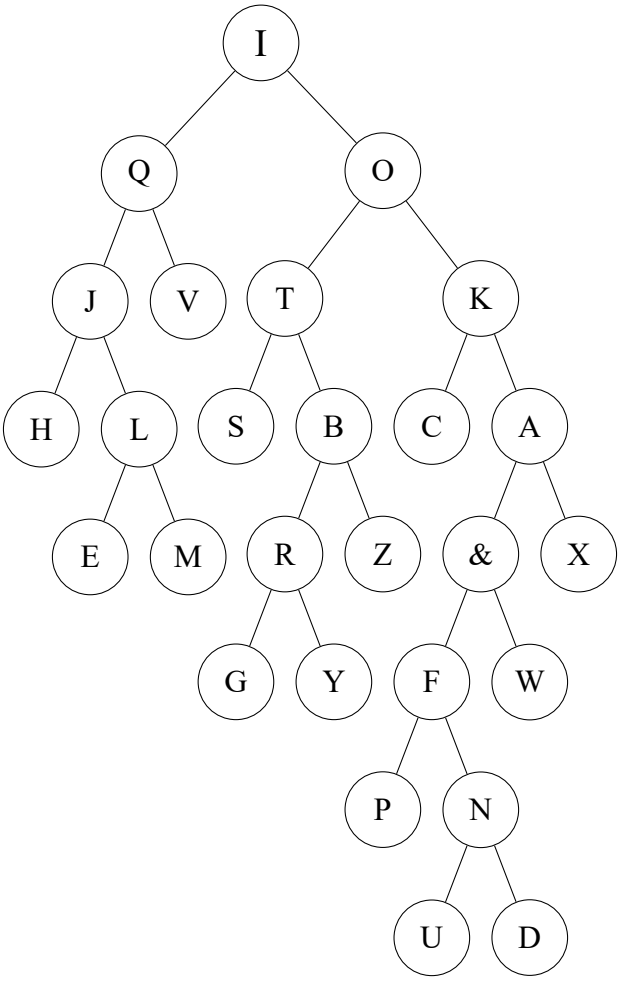
麻超　201300066

南京大学人工智能学院

2021 年 10 月 28 日

Problem 1

a



b

解: 前序遍历的结构是 [根节点, 左子树, 右子树], 后序遍历的结构是 [左子树, 右子树, 根节点]. 每一个子树又可以划分为一个完整的 [根节点, 左子树, 右子树] 的结构. 因此可以考虑用递归的方法解决问题.

首先,preorder 的第一个元素和 postorder 的最后一个元素必然是整个树的根节点. 接下来 preorder 的第二个元素必然是其左子树的根节点, 在 postorder 里面找到它 (如在 (a) 中的元素'Q'),postorder 在 Q 之前的元素即是该二叉树的左子树中的元素,postorder 中其他的元素就是该二叉树右子树中的元素. 接下来用同样的方法对 preorder 中左子树和右子树分别作为一个完整的树进行递归即可.

伪代码如下: 假设一共有 n 个节点, 则其时间复杂度为 $O(n^2)$.

Algorithm 1 Construct the tree

Input: $pre[], post[]$
  function CONSTRUCT($pre[], post[]$)
    if pre.length=0 **then return** *null*
    end if
    $root \leftarrow pre[0]$
    if pre.length=1 **then return** *root*
    end if
    L← i+1 and post[i]=pre[1]
    $root.left \leftarrow Construct(pre[1, L+1], post[0, L])$
    $root.right \leftarrow Construct(pre[L+1, N], post[L, N-1])$ **return** *root*
  end function

c

如果该树为非满二叉树, 则无法确定其值. 假设有如下 tree:

这两个树的前序遍历都是 [1,2,3,4], 后序遍历都是 [3,4,2,1], 但这两个树不同, 故如果一个树为非满二叉树, 无法通过前序遍历和后序遍历构建这个树.

Problem 2

解: 按照中序遍历的顺序, 从这个树的根结点开始右旋, 直至左子树为空, 然后对右子树进行上述操作, 直至二叉树变成升序排列的链. 而每进行一次上述的右旋操作, 都会将一个不在这样的链上的结点旋转到这样的链上, 所以该操作的步数为 $O(n)$.

每一棵含 n 个结点的二叉搜索树都能通过这样的 O(n) 的步数旋转成右侧伸展的链, 接下来将这个链伸展成为一个平衡二叉树即可. 当然, 由于树的左旋和右旋互为逆操作, 在用旋转操作将一个树变为链的时候, 必然可以同样地从一个链变为一个树. 由于平衡二叉树也是二叉搜索树的一种, 所以它也满足同样的性质.

故总共需要 $O(n)+O(n)=O(n)$ 次旋转操作即可以完成这个算法.

Problem 3

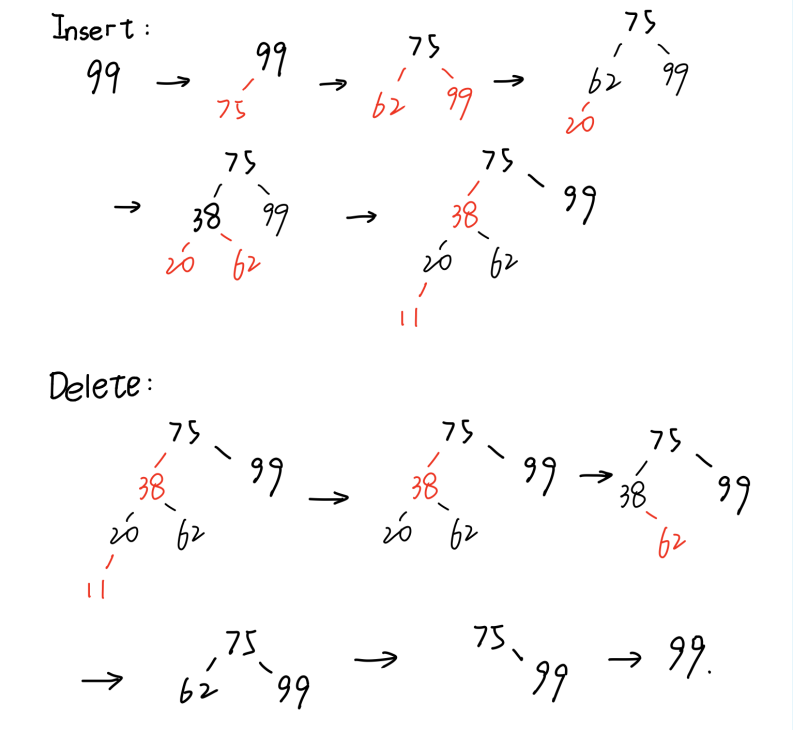
对于任意一个二叉搜索树, 由第二题中的结论, 可以知道将其转化为一条链. 在这之后, 从这条链的底部节点 A 开始, 将 A 与其父节点进行比较, 若 A 大于父节点, 则对 A 进行左旋, 否则对 A 的子树进行 swap 操作后左旋, 再对 B 的子树进行 swap 操作. 不断将 A 与其父节点进行比较可以获得一个根节点为 A,A 的左子树中所有节点都小于 A, 右子树的所有节点都大于 A 的树. 且 A 的左右子节点分别都是向右延展的链, 再次对两个子树进行递归即可得到二叉搜索树.

Algorithm 2 Build binary search tree

```
Input: T
  function BUILD(T.root)
      Turn to a chain(T)
      if T.length=1 or T.root=null then return
      else
          A = T.bottom()
          while A.parent ≠ null do
              if A.parent<A then
                  LeftRotation(A)
              else
                  swap(A)
                  LeftRotation(A)
                  swap(A)
              end if
          end while
          Build(A->left)
          Build(A->right)
      end if
  end function
```

时间复杂度分析: 将一个树转化为链的过程需要进行 O(n) 次旋转, 在计算过程中需要将底部元素与链中其他所有元素进行比较, 以及 swap 操作, 总共需要 $O(n^2)$ 次操作. 故总的时间复杂度为 $O(n^2)$.

Problem 4



Problem 5

a

假设有一个高度为 h 的树, 则这个树所包含的最小的节点数, 设为 T(h). 则对于这棵 AVL 树而言, 其左右子树中必然至少有一个子树的最大高度等于 h-1. 由于 AVL 树的性质 (高度平衡), 则另一棵 AVL 树的高度至少为 h-2. 由于此时 T(h) 表示的是最小节点数, 所以当左右子树都最小时, 可以表示 T(h), 再加上根节点的 1, 可以知道 T(h)=T(h-1)+T(h-2)+1. 易得当树的高度为 0 时, 最小结点数为 0(空树), 当树的高度为 1 时, 最小结点数为 1(只有根结点), 即 T(0)=0,T(1)=1.

由等式 (3.25) 可得 $T(h) = \left\lfloor \frac{\phi^h}{\sqrt{5}} + \frac{1}{2} \right\rfloor \leq n$

解得 $h \leq \frac{\lg(\sqrt{5})+\lg(n+\frac{1}{2})}{\lg\phi} \in O(\lg n)$

故有 n 个节点的 AVL tree 有 O(lg n) 的高度.

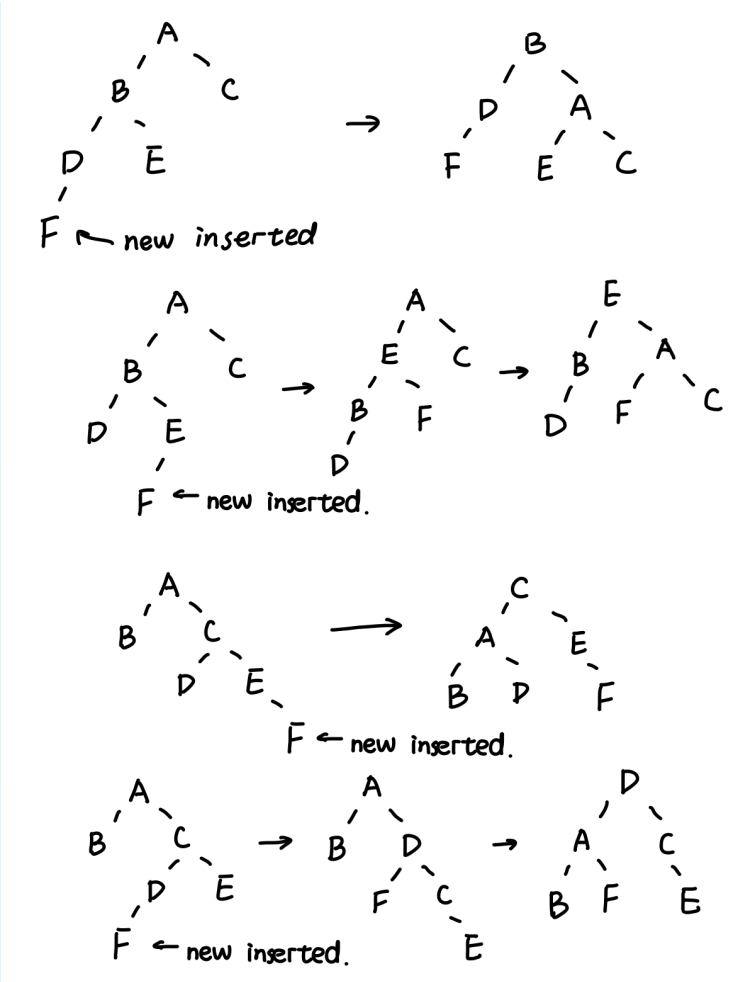
b

将一个元素首先按二叉搜索树的顺序添加到 AVL 树中时, 其可能会导致该树高度的变化, 导致其不再平衡. 通过树的左旋和右旋操作, 可以修正由于插入元素导

致的平衡性被破坏.

当平衡性被破坏时分为以下四种情况:

- 插入到了左子树的左边, 可以通过一次右旋操作修正
- 插入到了左子树的右边, 可以通过先对左子树左旋转化为第一种情况再右旋修正
- 插入到了右子树的右边, 可以通过一次左旋操作修正
- 插入到了右子树的左边, 可以通过先对左子树右旋转化为第三种情况再左旋修正



伪代码如下:

Algorithm 3 Balance(x)

Input: $treeA, x$

```
function BALANCE(A.root, x)
    if root.key < x then
        if lefttree.height − righttree.height = 2 then        ▷ AVL tree is not balance
            if root− > left.key > x then                      ▷ Condition 1
                Right − Rotation(root)
            else                                              ▷ Condition 2
                Left − Rotation(root− > left)
                Right − Rotation(root)
            end if
        end if
    else                                                      ▷ 插入到右子树
        if righttree.height − lefttree.height = 2 then        ▷ AVL tree is not balance
            if root− > right.key < x then                     ▷ Condition 3
                Left − Rotation(root)
            else                                              ▷ Condition 4
                Right − Rotation(root− > right)
                Letf − Rotation(root)
            end if
        end if
    end if
end function
```

c

首先以和二叉搜索树相同的方式将元素插入到 AVL tree 中, 从根节点开始逐次递归地比较每个节点和 z 的大小关系 (如果 z 大于该节点的值, 则应当插入到该节点的右子树中, 否则为左子树), 直到到达叶节点. 最后, 从被插入的该节点开始依次递归地向上检查看是否插入操作使得某个子树违背了 height-balance 的性质, 如果违背的话, 就通过左旋及右旋操作修正这个错误. 直到根节点为止, 这样可以首先保证其满足二叉搜索树的性质, 其次在旋转过程中, 一方面可以保证其 height-balance 的性质, 另一方面也不会改变二叉搜索树的性质.

d

在算法中, 出现了两次循环, 易知这两次循环都只发生了 $O(h) = O(\lg n)$ 次迭代, 故运行时间为 $O(\lg n)$. 在第二个循环, 即递归向上检查是否满足 AVL 的性质时, 只需要发生一次旋转, 在发生这次旋转之后, 该子树的高度将会降低, 则对于总的所有元素来说, 该树必然已经满足了高度平衡的性质 (因为插入过程中只插入了一个元素, 最多只会将树的高度增加 1), 即需要做 $O(1)$ 次旋转.

Algorithm 4 Insert(x,z)

Input: $treeA.root\ x, z$

```
if x=null then
    z.h = 0 return
end if
if z.key > x.key then
    Insert(x->right, z)
    if x->right = null then
        x->right = z
    end if
else
    Insert(x->left, z)
    if x->left = null then
        x->left = z
    end if
end if
left-height = -1, right-height = -1
if x->left ≠ null then
    left-height = x->left.h
end if
if x->right ≠ null then
    right-height = x->right.h
end if
x.h ← max(left-height, right-height) + 1
Balance(x)
```

Problem 6

a

meld 操作的基本目标是取两个堆 Q1 和 Q2，并合并它们，结果返回单个堆节点。这个堆节点是包含来自以 Q1 和 Q2 为根的两个子树的所有元素的堆的根节点。

如果任一堆为空，则合并将使用空集进行，并且该方法仅返回非空堆的根节点。如果 Q1 和 Q2 都不为零，检查是否有 Q1>Q2。如果是，交换两者。因此确保 Q1<Q2 并且合并堆的根节点将包含 Q1。然后递归地将 Q2 与 Q1.left 或 Q1.right 合并。在每一次合并时, 都会降低一个树的高度, 直至某个树为空.

调用 meld(Q1,Q2) 的期望次数为 $\lg x + \lg y \leq 2\lg n$

故期望时间为 $O(\lg n)$

b

FindMin() 操作只需要返回该 queue 的根节点. 需要 O(1) 时间.

DeleteMin() 操作只需合并根节点的左右子树, 因为最小节点为根节点.

Insert(Q,x) 操作首先创建一个只含有 x 的节点 a, 然后合并 Q 和 a.

DecreaseKey(Q,x,y): 对 x 的节点的值修改为 y, 将原节点的父节点指向该节点的指针变为 null, 显然需要 O(1) 步操作. 然后将 Q(new) 与该节点再次合并即可.

Delete(Q,x) 操作将 x 的父节点指向 x 的指针指向 x.left, 并设置 x.left 的父节点为 x 的父节点. 然后合并 Q(new) 与 x.right