树 Tree

1 二叉树 Binary Tree

1.1 实现

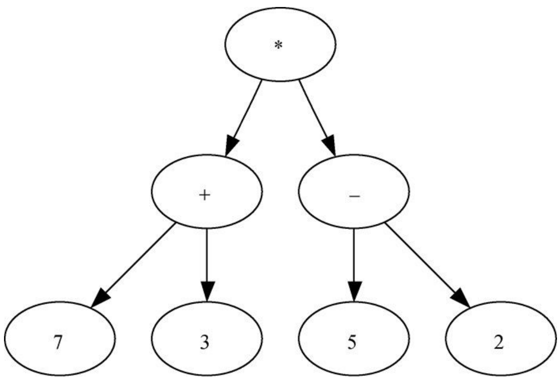
详见 这里（未记录父节点）

1.2 应用

1.2.1 解析树 Parse Tree

以完全括号表达式为例（要求全为正整数，且只含 `+-*/` 四种运算）

- 图示：对于表达式 $((7+3)*(5-2))$ ，



- 构建

```python
def buildParseTree(fpexp: str):
    fplist = fpexp.split()
    pStack = Stack() # 通过栈记录父节点
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree

    for i in fplist:
        if i == '(':
            currentTree.insertLeft('')
            pStack.push(currentTree)
            currentTree = currentTree.getLeftChild()
        elif i.isdigit():
            currentTree.setRootVal(int(i))
            currentTree = pStack.pop()
        elif i in '+-*/':
            currentTree.setRootVal(i)
            currentTree.insertRight('')
```

```python
            pStack.push(currentTree)
            currentTree = currentTree.getRightChild()
        elif i == ')':
            currentTree = pStack.pop()
        else:
            raise ValueError('Unknown Operator: {}'.format(i))
    return eTree
```

- 计算

```python
def evalParseTree(parseTree: BinaryTree):
    oprs = {'+': operator.add, '-': operator.sub,
           '*': operator.mul, '/': operator.truediv}
    leftChild = parseTree.getLeftChild()
    rightChild = parseTree.getRightChild()
    if leftChild and rightChild:
        opr = oprs[parseTree.getRootVal()]
        return opr(evalParseTree(leftChild), \
                   evalParseTree(rightChild))
    else:
        return parseTree.getRootVal()
```

1.2.2 遍历 Traversal

-
  - 前序遍历 (preorder traversal)：根节点、左子树、右子树
  - 中序遍历 (inorder traversal)：左子树、根节点、右子树
  - 后序遍历 (postorder traversal)：左子树、右子树、根节点
- 中序遍历应用：还原完全括号表达式（见上）

```python
def printParseTree(parseTree: BinaryTree):
    res = ''
    if parseTree:
        if not parseTree.getLeftChild():
            res = str(parseTree.getRootVal())
        else:
            res = '( ' + printParseTree(parseTree.getLeftChild()) \
                       + ' ' + str(parseTree.getRootVal()) + ' ' \
                       + printParseTree(parseTree.getRightChild()) \
                       + ' )'
    return res
```

2 二叉堆 Binary Heap

- 可用来实现优先级队列
- 二叉堆的入队和出队操作均可达 $O(\log n)$
- 最小堆（最小的元素一直在队首）、最大堆

2.1 完全二叉树 Complete Binary Tree

- **定义**：对于深度为 k ，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应时，称之为完全二叉树。

- 性质：

  - 可用**一个列表**表示（编号从 1 开始）

  - 若某左节点编号为 n，其父节点编号为 $n/2$ ；

    若某右节点编号为 n，其父节点编号为 $(n-1)/2$

    $\Longrightarrow$ 给定编号为 n 的节点，其父节点的编号为 $n//2$

2.2 二叉堆的实现（以最小堆为例）

- 用完全二叉树维持树的平衡（为使树高效工作）

- 堆的有序性：对于堆中任意元素 x 及其父元素 p ， p 都不大于 x

```python
class BinaryHeap:
    def __init__(self):
        self.heapList = [0] # 二叉堆编号从 1 开始
        self.size = 0

    # 增加元素：先加到列表最后一个，再对其使用 percUp 方法
    def percUp(self, i):
        while i > 1:
            if self.heapList[i] < self.heapList[i//2]:
                self.heapList[i], self.heapList[i//2] = \
                self.heapList[i//2], self.heapList[i]
            i //= 2
    def insert(self, key):
        self.heapList.append(key)
        self.size += 1
        self.percUp(self.size)

    # 删除最小数：将列表最后一个元素提到第一个，并对其使用 percDown 方法
    def percDown(self, i):
        def getMinChild(i):
```

```python
                if i * 2 + 1 > self.size:
                    return i * 2
                else:
                    if self.heapList[i*2] < self.heapList[i*2+1]:
                        return i * 2
                    else:
                        return i * 2 + 1
            while (i * 2) <= self.size:
                minChild = getMinChild(i)
                if self.heapList[i] > self.heapList[minChild]:
                    self.heapList[i], self.heapList[minChild] = \
                    self.heapList[minChild], self.heapList[i]
                i = minChild
        def delMin(self):
            minKey = self.heapList[1]
            self.heapList[1] = self.heapList[self.size]
            self.size -= 1
            self.heapList.pop()
            self.percDown(1)
            return minKey

        # 由列表构建堆的时间复杂度只有 O(n)
        def buildHeap(self, lst):
            i = len(lst) // 2 # 超过中点的节点都是叶节点，不必操作
            self.size = len(lst)
            self.heapList = [0] + lst[:]
            while i > 0:
                self.percDown(i)
                i -= 1
```

- 由于建堆的时间复杂度为 $O(n)$ ，可实现时间复杂度为 $O(n\log n)$ 的堆排序：

  1. 建立最大堆（ $O(n)$ ）
  2. 取出堆顶点（最大元），并将最后一个节点填充到堆顶点，堆大小减一
  3. 对新堆的堆顶点使用 `percDown` 方法（ $O(\log n)$ ）
  4. 不断重复 2 和 3 ，直至堆为空

  其中第 2~4 步的时间复杂度为 $O(\log n+\log(n-1)+\cdots+\log 2)=O(\log n!)$ ，由 Stirling公式知 $O(\log n!)$ 和 $O(n\log n)$ 是等价无穷大。故堆排序的复杂度为 $O(n\log n)$ 。

3 二叉搜索树 BST (Binary Search Tree)

- 依赖性质：**二叉搜索性**
    - 小于父节点的键都在左子树中，大于父节点的键都在右子树中

3.1 实现

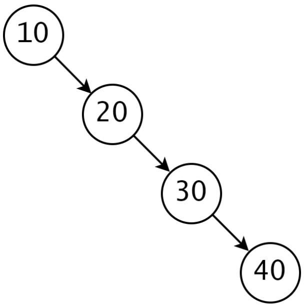
- 详见 这里
- 解释 `delete` 方法：

```
def delete(self, value):
    self.root = self._delete_recursive(self.root, value)

def _delete_recursive(self, node, value):
    if node is None:
        return node
    if value < node.value:
        node.left = self._delete_recursive(node.left, value)
    elif value > node.value:
        node.right = self._delete_recursive(node.right, value)
    else:
        # 当目标节点有儿子为空时，直接用另一个儿子替换目标节点
        if node.left is None:
            return node.right
        elif node.right is None:
            return node.left
        else:
            # 当目标节点的左右儿子均非空时，需要寻找目标节点的后继，
            # 后继为左子树的最大元或右子树的最小元。将目标节点的值
            # 赋为后继的值，再将后继节点删除。注意，以右子树的最小元
            # 为例，其左子树必然为空，故不会再次进入这一情况
            min_node = self._find_min_node(node.right)
            node.value = min_node.value
            node.right = self._delete_recursive(\
                node.right, min_node.value)
    return node

def _find_min_node(self, node):
    current = node
    while current.left is not None:
        current = current.left
    return current
```

3.2 应用

- 映射 (map) 的又一种实现方式
    - 详见 这里

3.3 分析

- 对于一棵平衡的 BST 树，`insert`，`delete`，`search` 等方法的时间复杂度均为 $O(\log n)$
- 当树严重偏斜时，时间复杂度迅速变为 $O(n)$
    - 

    - 解决：AVL 树（见下）

4 平衡二叉搜索树 AVL

- Named after G. M. **A**delson-**V**elsky & E. M. **L**andis

4.1 平衡因子 Balance Factor

- 定义：每个节点的平衡因子 = 左右子树的高度之差
- 平衡因子 >0： 左倾　平衡因子 <0： 右倾

  将每个节点的平衡因子为 -1, 0, 1 的树都定义为 AVL 树
- 一棵 AVL 树的 `insert`, `delete`, `search` 等方法的时间复杂度均为 $O(\log n)$
  - 最坏情况分析：

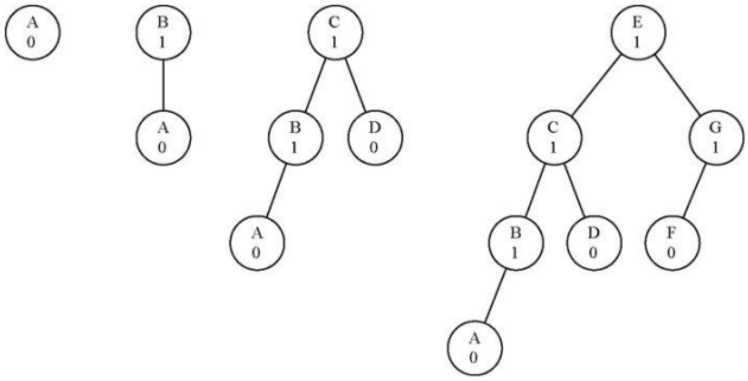


图6-27　左倾AVL树的最坏情况

查看树中的节点数之后可知，高度为0时有1个节点，高度为1时有2个节点（1 + 1 = 2），高度为2时有4个节点（1 + 1 + 2 = 4），高度为3时有7个节点（1 + 2 + 4 = 7）。也就是说，当高度为h时，节点数N_h是：

$$N_h = 1 + N_{h-1} + N_{h-2}$$

你或许觉得这个公式很眼熟，因为它与斐波那契数列很相似。可以根据它推导出由AVL树的节点数计算高度的公式。在斐波那契数列中，第i个数是：

$$\begin{aligned} &F_0 = 0 \\ &F_1 = 1 \\ &F_i = F_{i-1} + F_{i-2}, i \geqslant 2 \end{aligned}$$

一个重要的事实是，随着斐波那契数列的增长，F_i/F_{i-1}逐渐逼近黄金分割比例 Φ，$\Phi=\frac{1+\sqrt{5}}{2}$ 。如果你好奇这个等式的推导过程，可以找一本数学书看看。我们在此直接使用这个等式，将F_i近似为$F_i = \Phi^i / \sqrt{5}$。由此，可以将N_h的等式重写为：

$$N_h = F_{h+2} - 1, \mathrm{h} \geqslant 1$$

用黄金分割近似替换，得到：

$$N_h = \frac{\Phi^{h+2}}{\sqrt{5}} - 1$$

移项，两边以2为底取对数，求h，得到：

$$\begin{aligned} &\log N_h + 1 = (H+2)\log\Phi - \frac{1}{2}\log 5 \\ &h = \frac{\log N_h + 1 - 2\log\Phi + \frac{1}{2}\log 5}{\log\Phi} \\ &h = 1.44\log N_h \end{aligned}$$

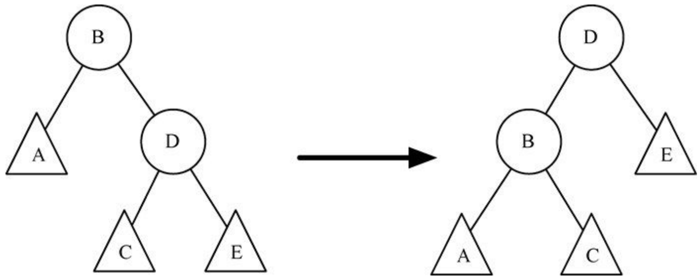
在任何时间，AVL树的高度都等于节点数取对数再乘以一个常数（1.44）。对于搜索AVL树来说，这是一件好事，因为时间复杂度被限制为$O(\log N)$。

4.2　实现

- 详见 这里 （不简单哈！）
- 和 BST 树相比，多了更新平衡因子和重新平衡的步骤。重新平衡：旋转

4.2.1　旋转

- 左旋：处理右倾的树
  - 图示



- 步骤

  1. 输入参数：给定节点 B
  2. 将 B 的右子节点 D 保存为临时变量，将 D 的左子节点 C 保存为临时变量
  3. 将 D 的左子节点更新为 B，将 B 的右子节点更新为 C
  4. 依次更新 B 和 D 的高度
  5. 返回新的根节点 D

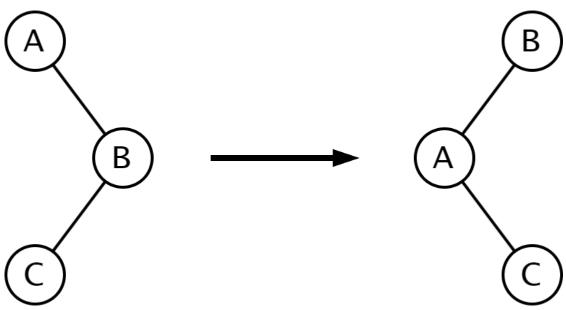
- 实现

```
def _rotate_left(self, z):
    y = z.right
    T2 = y.left

    y.left = z
    z.right = T2

    self._update_height(z)
    self._update_height(y)

    return y
```

- 右旋：处理左倾的树

- ……但是还没完

  - 例如，对这棵右倾的失衡树做一次左旋，

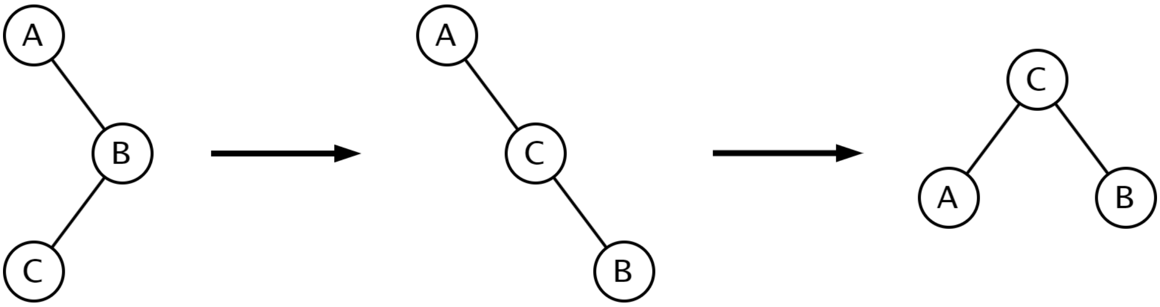


    得到了另一颗左倾的失衡树。若再做一次右旋，又回到了原来的状态！

  - 解决：先检查子树的平衡因子，确认是否需要先对子树进行旋转

- 原节点需要左旋：先检查右子树的平衡因子，如果右子树左倾，则先对右子树做一次右旋，再围绕原节点做一次左旋 (Right-Left Case)
- 原节点需要右旋：先检查左子树的平衡因子，如果左子树右倾，则先对左子树做一次左旋，再围绕原节点做一次右旋 (Left-Right Case)

上述方法解决了上图的问题：



- 实现

```
def _rebalance(self, node):
    self._update_height(node)

    balance = self._get_balance(node)

    if balance > 1:
        if self._get_balance(node.left) < 0:  # Left-Right case
            node.left = self._rotate_left(node.left)
        return self._rotate_right(node)

    if balance < -1:
        if self._get_balance(node.right) > 0:  # Right-Left case
            node.right = self._rotate_right(node.right)
        return self._rotate_left(node)

    return node
```

5 量化图片

在实践中，我们使用 1 字节（8位）表示像素的每个颜色构成。 8 位可以表示每种三原色的 256 种层次，共可表示 1670 万种颜色。所需的存储空间为每像素 3 字节。

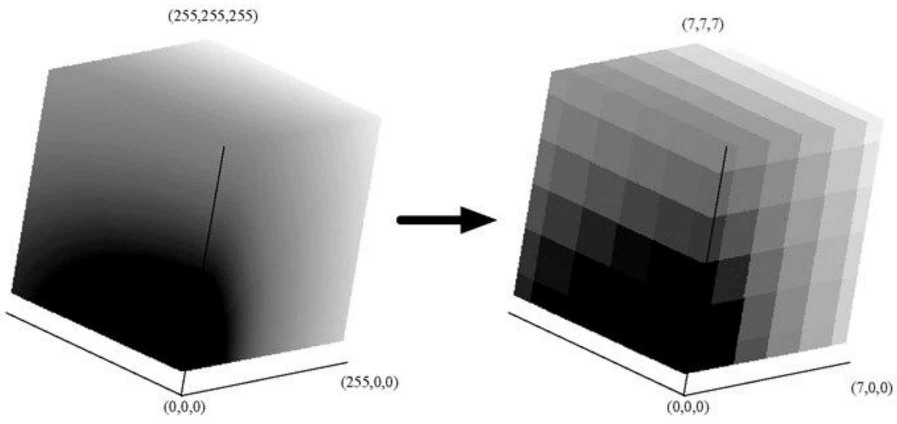
5.1 简单的图片量化算法

- ```python
  import sys
  import os
  from PIL import Image

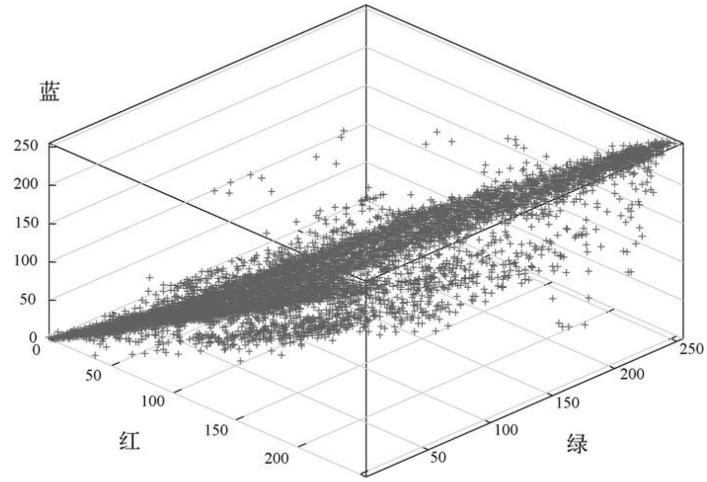
  def simpleQuant():
      im = Image.open('Path_of_the_file')
      w, h = im.size
      for row in range(h):
          for col in range(w):
              r, g, b = im.getpixel((col, row))
              r = r // 36 * 36
              g = g // 42 * 42
              b = b // 42 * 42
              im.putpixel((col, row), (r, g, b))
      im.save('Path_to_save') # or directly im.show()

  simpleQuant()
  ```

- 红色维度上有 7 个值，绿色和蓝色维度上有 6 个值，共 256 种颜色，每个像素只需用 1 字节存储。

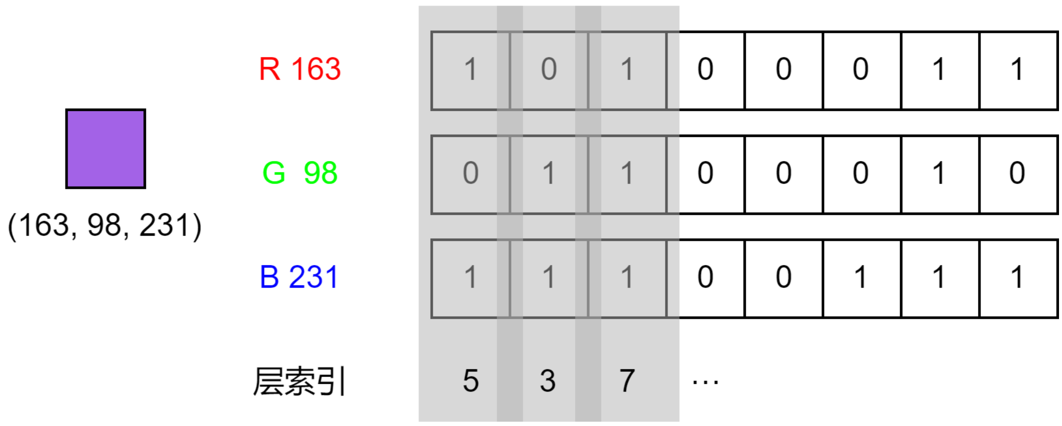


- 问题：大部分图片中的颜色不是均匀分布的，很多颜色可能没有出现在图片中，立方体中对应的部分并没有用到。在量化后的图片中分配没用到的颜色是浪费行为。
  ```



5.2 八叉树改进量化算法

- `OctTree` 的根代表整个立方体。每层代表每个维度上的一个切片，将父节点对应的子立方体等分成 8 块；第 8 层代表所有 1670 万种颜色。
- 注意：并不是完整创建一棵八叉树，而是在使用中逐步增加所需节点。并且，**可手动设置最深层数（例如 5 层）**，以大幅减小存储空间，而对最终图片质量没有大的影响。
- 颜色填入规则



5.2.1 实现操作

- 按照给定的颜色目标数目（例如上面的 256 种），选择图片的颜色子集。

  1. 遍历图片的每一个像素：
     (a) 在 OctTree 中查找该像素的颜色，这个颜色应该是位于第 8 层（或设置的最深层数）的一个叶子节点
     (b) 如果没找到，创建一个叶子节点（可能还需要在节点之上创建一些内部节点）
     (c) 如果找到了，将叶子节点的计数器加 1，以记录这个颜色用于多少个像素
  2. 重复以下步骤，直到叶子节点的数目小于等于颜色的目标数目：
     (a) 找到计数最少的叶子节点
     (b) 合并该叶子节点及其所有兄弟节点，形成一个新的叶子节点

3. 剩余的叶子节点形成图片的颜色集
4. 若要将初始的颜色映射为量化后的值，只需沿着树向下搜索到叶子节点，然后返回叶子节点存储的颜色值

- 实现

```python
def buildAndDisplay():
    im = Image.open('Path_of_the_file')
    w, h = im.size
    ot = OctTree()

    for row in range(0, h):
        for col in range(0, w):
            r, g, b = im.getpixel((col, row))
            ot.insert(r, g, b) # Step 1

    ot.reduce(256)  # Step 2

    for row in range(0, h):
        for col in range(0, w):
            r, g, b = im.getpixel((col, row))
            nr, ng, nb = ot.find(r, g, b) # Step 4
            im.putpixel((col, row), (nr, ng, nb))

    im.show()
```

5.3 具体实现

- 实现 `OctTree`, `otNode` 两个类
  - `otNode` 是 `OctTree` 的内部类
    - `OctTree` 的每个节点都需要访问一些存储于 `OctTree` 类实例中的信息
    - 没有任何在 `OctTree` 类之外使用 `otNode` 的必要
  - 上面代码中 `buildAndDisplay()` 用到的所有方法都在 `OctTree` 类中定义
  - `otNode` 构造方法中含有参数 `outer`，指向创建该节点的 `OctTree` 实例的引用
    - 其它参数有 red, green, blue, count, **level**, parent, children 等
    - red, green, blue 是属于该节点的所有像素的对应值的总和（合并时也一样）
- 详见 这里 （不简单哈！）

> 好像还有点问题... 处理有些图片正常，有些图片报错 `ValueError: list.remove(x): x not in list`

5.4 分析

- 合并节点时，通过遍历 `leafList` 寻找计数最少的节点，效率很低
    - 利用优先级队列代替列表
    - 注意重载 `__lt__` 方法（ < 运算符）
    - 改进后的实现见 这里
- 实验

原图
48.0KB

用简单算法 256 色压缩
38.2KB

用八叉图 256 色压缩
33.2KB