二叉树 哈夫曼树 堆

吴益强

Python 版

日期：2022 年 4 月 22 日

1 二叉树

1.1 二叉树的定义

1) 节点：由三部分组成：数据、左子节点指针、右子节点指针 g_2

2) 一个左右子节点指针均为空的节点，叫叶子节点

3) 叶子节点是一棵二叉树，树根即是该节点

4) 若有一个节点 X 的左子节点指针或右子节点指针指向一棵不包含 X 的二叉树的根，或两者分别指向两棵没有公共节点且不包含 X 的二叉树的根指向 X 不算))，则 X 和其指向的一棵或两棵二叉树构成一棵二叉树，根为 X 。X 的左子树是左子节点指向的树，右子树是右子节点指向的树：

1.2 二叉树相关概念

```
父节点、祖先节点c
    如果a 是 b 的子节点，则 b 是 a 的父节点
    父节点是祖先节点。祖先节点的祖先节点也是祖先节点。
度
    节点的子树的个数
树的边
    连接父节点和子节点的指针
节点 的 层次
    从根出发到达节点所经过的边数。根节点为第 0 层
树的 高度
    即节点层数，为节点最大层次 +1 。树的高度是 >=1 的
满二叉树
    每一层节点数目都达到最大。即第 i 层有 2^i 个节点。高为 h 的满二叉树，有 2^h
       -1 个节点
完全二叉树
    除最后一层外，其余层的节点数目均达到最大。而且，最后一层节点若不满，则缺的
       节点定是在最右边的连续若干个
```

1.3 二叉树的性质

1) 第 i 层最个多 2^i 个节点。高为 h 的满二叉树节点总数 $2^h - 1$

2) 节点数为 n 的树，边的数目为 n 1

3) 包含 n 个结点的二叉树的高度至少为 log 2 n+1) 向上取整

4) 在任意一棵二叉树中，若叶子节点的个数为 n 0 ，度为 2 的节点个数为 n 2 ，则 n 0 =n 2 +1 。

5) 任何两个节点之间只有一条路径

2),4) 按树的高度用数学归纳法证明

1.4 二叉树的实现和遍历

二叉树的遍历，指的是如何按某种搜索路径巡防树中的每个结点，使得每个结点均被访问一次，而且仅被访问一次。对于二叉树，常见的遍历方法有：先序遍历，中序遍历，后序遍历，层序遍历。这些遍历方法一般使用递归算法实现。

```
先序遍历的操作定义为:
若二叉树为空，为空操作；否则（1）访问根节点；（2）先序遍历左子树；（3）先序遍历
    右子树。
中序遍历的操作定义为:
若二叉树为空，为空操作；否则（1）中序遍历左子树；（2）访问根结点；（3）中序遍历
    右子树。
后序遍历的操作定义为:
若二叉树为空，为空操作；否则（1）后序遍历左子树；（2）后序遍历右子树；（3）访问
    根结点。
层序遍历的操作定义为:
若二叉树为空，为空操作；否则从上到下、从左到右按层次进行访问。
```

```
from graphviz import Digraph
import uuid
from random import sample

# 二叉树类
class BTree(object):

    # 初始化
    def __init__(self, data=None, left=None, right=None):
        self.data = data    # 数据域
        self.left = left    # 左子树
        self.right = right  # 右子树
        self.dot = Digraph(comment='Binary Tree')

    # 前序遍历
    def preorder(self):

        if self.data is not None:
```

```
            print(self.data, end=' ')
        if self.left is not None:
            self.left.preorder()
        if self.right is not None:
            self.right.preorder()

    # 中序遍历
    def inorder(self):

        if self.left is not None:
            self.left.inorder()
        if self.data is not None:
            print(self.data, end=' ')
        if self.right is not None:
            self.right.inorder()

    # 后序遍历
    def postorder(self):

        if self.left is not None:
            self.left.postorder()
        if self.right is not None:
            self.right.postorder()
        if self.data is not None:
            print(self.data, end=' ')

    # 层序遍历
    def levelorder(self):

        # 返回某个节点的左孩子
        def LChild_Of_Node(node):
            return node.left if node.left is not None else None
        # 返回某个节点的右孩子
        def RChild_Of_Node(node):
            return node.right if node.right is not None else None

        # 层序遍历列表
        level_order = []
        # 是否添加根节点中的数据
        if self.data is not None:
            level_order.append([self])

        # 二叉树的高度
        height = self.height()
        if height >= 1:
            # 对第二层及其以后的层数进行操作, 在level_order中添加节点而不是数据
            for _ in range(2, height + 1):
```

```
            level = []  # 该层的节点
            for node in level_order[-1]:
                # 如果左孩子非空，则添加左孩子
                if LChild_Of_Node(node):
                    level.append(LChild_Of_Node(node))
                # 如果右孩子非空，则添加右孩子
                if RChild_Of_Node(node):
                    level.append(RChild_Of_Node(node))
            # 如果该层非空，则添加该层
            if level:
                level_order.append(level)

        # 取出每层中的数据
        for i in range(0, height):  # 层数
            for index in range(len(level_order[i])):
                level_order[i][index] = level_order[i][index].data

    return level_order

# 二叉树的高度
def height(self):
    # 空的树高度为0, 只有root节点的树高度为1
    if self.data is None:
        return 0
    elif self.left is None and self.right is None:
        return 1
    elif self.left is None and self.right is not None:
        return 1 + self.right.height()
    elif self.left is not None and self.right is None:
        return 1 + self.left.height()
    else:
        return 1 + max(self.left.height(), self.right.height())

# 二叉树的叶子节点
def leaves(self):

    if self.data is None:
        return None
    elif self.left is None and self.right is None:
        print(self.data, end=' ')
    elif self.left is None and self.right is not None:
        self.right.leaves()
    elif self.right is None and self.left is not None:
        self.left.leaves()
    else:
        self.left.leaves()
        self.right.leaves()
```

```
# 利用Graphviz实现二叉树的可视化
def print_tree(self, save_path='./Binary_Tree.gv', label=False):

    # colors for labels of nodes
    colors = ['skyblue', 'tomato', 'orange', 'purple', 'green', 'yellow', '
        pink', 'red']

    # 绘制以某个节点为根节点的二叉树
    def print_node(node, node_tag):
        # 节点颜色
        color = sample(colors,1)[0]
        if node.left is not None:
            left_tag = str(uuid.uuid1())            # 左节点的数据
            self.dot.node(left_tag, str(node.left.data), style='filled',
                color=color)    # 左节点
            label_string = 'L' if label else ''    # 是否在连接线上写上标
                签，表明为左子树
            self.dot.edge(node_tag, left_tag, label=label_string)   # 左节
                点与其父节点的连线
            print_node(node.left, left_tag)

        if node.right is not None:
            right_tag = str(uuid.uuid1())
            self.dot.node(right_tag, str(node.right.data), style='filled',
                color=color)
            label_string = 'R' if label else ''  # 是否在连接线上写上标签，
                表明为右子树
            self.dot.edge(node_tag, right_tag, label=label_string)
            print_node(node.right, right_tag)

    # 如果树非空
    if self.data is not None:
        root_tag = str(uuid.uuid1())                # 根节点标签
        self.dot.node(root_tag, str(self.data), style='filled', color=
            sample(colors,1)[0])     # 创建根节点
        print_node(self, root_tag)

    self.dot.render(save_path)                              # 保存文件为指
        定文件
```

2 哈夫曼树

2.1 定常编码方案

定长编码方案每个字符编码的比特数都相同。比如 ASCII 编码方案。

2.2 熵编码方案

熵编码使用频率高的字符，给予较短编码，使用频率低的字符，给予较长编码，如哈夫曼编码。

2.3 哈夫曼编码树

使用可变长编码，需要解决的问题是：如何区分一个编码是一个字符的完整编码，还是另一个字符的编码的前缀。解决办法之一就是采用前缀编码任何一个字符的编码，都不会是其他字符编码的前缀。

```
二叉树

叶子代表字符，且每个叶子节点有个权值，权值即该字符的出现频率

非叶子节点里存放着以它为根的子树中的所有字符，以及这些字符的权值之和

权值仅用来建树，对于字符串的解码和编码没有用处
```

字符的编码过程：

从树根开始，每次往包含该字符的子树走。往左子树走，则编码加上比特 1 往右子树走，则编码加上比特 0

字符的解码过程：

从树根开始，在字符串编码中碰到一个 0，就往左子树走，碰到 1，就往右子树走。走到叶子，即解码出一个字符。然后回到树根重复前面的过程。

基本思想：使用频率越高的字符，离树根越近。

过程：

```
1.开始时，若有 n 个字符，则就有 n 个节点。每个节点的权值就是字符的频
率，每个节点的字符集就是一个字符。
2.取出权值最小的两个节点，合并为一棵子树。子树的树根的权值为两个
节点的权值之和，字符集为两个节点字符集之并。在节点集合中删除取出的
两个节点，加入新生成的树根。
3.如果节点集合中只有一个节点，则建树结束。否则， goto 2
```

注意：哈夫曼编码树不唯一

3 堆

3.1 堆的概念

1、堆（二叉堆）是一个完全二叉树

2、堆中任何节点优先级都高于或等于其两个子节点（什么叫优先级高可以自己定义）

3.2 堆的储存

用列表存放堆。堆顶元素下标是 0 。下标为 i 的节点，其左右子节点下标分别为 i *2+1 , i *2 + 2 。

3.3 堆的性质

```
1)堆顶元素是优先级最高的
2)堆中的任何一棵子树都是堆
3)往堆中添加一个元素，并维持堆性质，复杂度 O(log(n))
4)删除堆顶元素，剩余元素依然维持堆性质，复杂度 O(log(n))
5)在无序列表中原地建堆，复杂度 O(n)
```

3.4 堆的作用

堆用于需要经常从一个集合中取走即删除优先级最高元素，而且还要经常往集合中添加元素的场合堆可以用来实现优先队列）

可以用堆进行排序，复杂度 O(nlog (n))，且只需要 O(1) 的额外空间称为“堆排序”。递归写法需要 o(log(n)) 额外空间，非递归写法需要 O(1) 额外空间。

3.5 堆的操作——添加一个元素

```
1)假设堆存放在列表 a 中，长度为 n
2)添加元素 x 到列表 a 尾部，使其成为 a[n]
3)若 x 优先级高于其父节点，则令其和父节点交换，直到 x 优先级不高于
其父节点，或 x 被交换到 a[0] 0]，变成堆顶为止。此过程称为将 x" 上移
4)x 停止交换后，新的堆形成，长度为 n+1
```

注意：显然，交换过程中，以 x 为根的子树，一直都是个堆由于 n 个元素的完全二叉树高度为 $log_2(n+1)$ 向上取整，每交换一次 x 就上升一层，因此上移操作复杂度 $O(log(n))$，即添加元素复杂度 $O(log(n))$

3.6 堆的操作——删除堆顶元素

```
1)假设堆存放在列表 a 中，长度为 n
2)将 a[0] 和 a[n-1] 交换
3)将 a[n-1] 删除 (pop)
4)记此时的 a[0] 为 x ，则将 x 和它两个儿子中优先级较高的，且优先级高
于 x 的那个交换，直到 x 变成叶子节点，或者 x 的儿子优先级都不高于 x
为止。将此整个过程称为将 x 下移
5)x 停止交换后，新的堆形成，长度为 n-1
```

注意：下移过程复杂度为 $O(log(n))$，因此删除堆顶元素复杂度 $O(log(n))$

3.7 堆的操作—-建堆

一个长度为 n 的列表 a, 要原地将 a 变成一个堆

```
将a 看作一个完全二叉树。假设有 H 层。根在第 0 层, 第 H-1 层都是叶子
对第H-2 层的每个元素执行下移操作
对第H-3 层的每个元素执行下移操作
.....
对第0 层的元素执行下移操作
堆即建好。复杂度O(n)
```

3.8 堆的应用

1、哈夫曼编码树的构造

2、堆排序（见上次课 pdf）

3.9 堆的实现

```
from heapq import *

Python 的 heapq 包实现的仅仅是最小堆!

heappush(heap, item): 将 item 元素加入堆。
heappop(heap): 将堆中最小元素弹出。
heapify(heap): 将堆属性应用到列表上。
heapreplace(heap, x): 将堆中最小元素弹出, 并将元素x 入堆。
merge(*iterables, key=None, reverse=False): 将多个有序的堆合并成一个大的有序
    堆, 然后再输出。
heappushpop(heap, item): 将item 入堆, 然后弹出并返回堆中最小的元素。
nlargest(n, iterable, key=None): 返回堆中最大的 n 个元素。
nsmallest(n, iterable, key=None): 返回堆中最小的 n 个元素。
```