Course 5: 二叉树基础（2）

三、二叉树的应用

（三） Huffman算法

哈夫曼算法（哈夫曼编码）是一种文本压缩方法。文本压缩是指将定义在一个字母表上的字符串有效地编码成一个小的二进制字符串，以便允许固定容量的存储设备容纳尽可能多的文档。

哈夫曼编码通过使用短的码字字符串来编码高频字符和使用长的码字字符串来编码低频字符。此外，哈夫曼编码基于对字符频率的使用，采用了一种针对任何给定字母表上的字符串X进行优化的可变长度编码。

为了编码字符串X，我们将X中的每个字符转换为可变长度的码字，连接起来产生X的编码Y。为了避免歧义，我们要求任何码字都不是另一个编码中的码字的前缀（满足这一条件的编码称为前缀码）

生成最佳可变长度前缀码的哈夫曼算法基于构造该编码的二叉树T。T中的每条边表示码字中的一位，指向左子结点的边表示0，指向右子结点的边表示1。每个叶子结点与特定字符相关联，该字符的码字即为从T的根到该叶子结点的路径上的位序列。每个叶子结点都有一个频率f，表示同该叶子结点相关联的字符在字符串X中的频率。此外，我们给T中的每个内部结点均赋予一个频率f，该频率是以该结点作为根结点的子树中所有叶子结点的频率之和。

1.哈夫曼编码算法

哈夫曼编码算法从待编码的字符串X中的每个独特字符开始，每个字符都是一个单结点的二叉树的根结点。该算法按照一系列轮次进行。在每一轮中，算法选择两个具有最小频率的二叉树，并将它们合并为一个单一的二叉树。算法重复这个过程直到只剩下一棵树。

命题：哈夫曼算法能够在O（n+dlogd）的时间内，为长度为n且包含d个独特字符的字符串构建一个最优的前缀编码。

3.贪心算法（Greedy method）

哈夫曼算法是一种称为**贪心**方法的算法设计模式的例子，这种设计模式应用于优化问题。

适用于贪心算法的问题一般称为具有贪心选择属性的问题。这是一种属性，即全局最优条件可以通过一系列局部最优选择达到。计算最优可变长度的前缀码问题，就是一个具有贪心选择属性的问题。

4.堆与堆的实现（heap）

堆是一种特殊的树状数据结构，通常用于实现优先队列。在最小堆中，父结点的值小于等于其子结点的值；在最大堆中，父结点的值大于等于其子结点的值。

heapq模块是python标准库中用于堆数据结构的工具，一种轻量级的方式来实现优先队列。它提供了一组函数，用于在列表上执行堆操作。heapq处理的是最小堆相关问题。

堆排序：heapq.heapify(iterable)函数【时间复杂度O(N)】，可以将一个可迭代对象转换为堆，该函数在原地进行操作。（对于给定的长度为n的初始序列，根据层次遍历后的完全二叉树，堆排序从n/2处开始索引，比较该结点的值与子结点的大小，不符条件就与子结点的最值交换，然后向上遍历。对于被交换的父结点，若原子结点位置还有子结点，则还需确定是否需要与子结点做二次交换，进行向下遍历）

插入元素：heapq.heappush(heap,item)函数【时间复杂度O(logN)】，将item插入heap堆中（操作方式上，将其插入最末叶子结点的位置，然后向上遍历，确定是否要交换）

弹出最小元素：heapq.heappop(heap)函数【时间复杂度O(logN)】，可以从堆中弹出并返回最小的元素（这时，将把堆底的元素放到堆顶填补空缺，然后由上到下依次比较，进行交换，保证优先队列的属性）

先推入后弹出：heapq.heappushpop(heap,item)函数【时间复杂度O(logN)】，先推入item入heap，整理后弹出并返回堆顶元素，并对堆排序

先弹出后替换：heapq.heapreplace(heap,item)函数【时间复杂度O(logN)】，可以弹出并返回堆中的最小元素，然后将新元素推入堆，并对堆排序

4.哈夫曼编码实现

要构建一个最优的哈夫曼编码树，首先需要对给定的字符及其权值进行排序，然后，通过重复合并权值最小的两个结点（子树），直到所有结点都合并为一棵树为止。

我们可以计算哈夫曼编码实现字符串所需要的总长度（带权外部路径长度:树中所有叶子结点的带权路径长度之和，结点的带权路径长度定义为该结点到树根之间的路径长度与该结点权值之间的乘积），采用heapq模块构建堆处理该问题：

```
import heapq

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq

def huffman_encoding(char_freq):
    heap = [Node(char, freq) for char, freq in char_freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq) # note: 合并之后 char 字典是空
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]
```

```
def external_path_length(node, depth=0):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return depth * node.freq
    return (external_path_length(node.left, depth + 1) +
            external_path_length(node.right, depth + 1))

def main():
    char_freq = {'a': 3, 'b': 4, 'c': 5, 'd': 6, 'e': 8, 'f': 9, 'g': 11, 'h': 12}
    huffman_tree = huffman_encoding(char_freq)
    external_length = external_path_length(huffman_tree)
    print("The weighted external path length of the Huffman tree is:", external_length)
```

5.Practice:哈夫曼编码树（openjudge 22161）

根据字符使用频率(权值)生成一棵唯一的哈夫曼编码树。生成树时需要遵循以下规则以确保唯一性：

选取最小的两个节点合并时，节点比大小的规则是:

1. 权值小的节点算小。权值相同的两个节点，字符集里最小字符小的，算小。 例如（{'c','k'},12) 和 ({'b','z'},12)，后者小。
2. 合并两个节点时，小的节点必须作为左子节点
3. 连接左子节点的边代表0,连接右子节点的边代表1 然后对输入的串进行编码或解码

输入

第一行是整数n，表示字符集有n个字符。接下来n行，每行是一个字符及其使用频率（权重）。字符都是英文字母。再接下来是若干行，有的是字母串，有的是01编码串。

输出

对输入中的字母串，输出该字符串的编码；对输入中的01串,将其解码，输出原始字符串

```
import heapq

class Node:
    def __init__(self, weight, char=None):
        self.weight = weight
        self.char = char
        self.left = None
        self.right = None

    def __lt__(self, other):
        if self.weight == other.weight:
            return self.char < other.char
        return self.weight < other.weight

def build_huffman_tree(characters):
    heap = []
```

```
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(left.weight + right.weight) #note: 合并后，char 字段默认值是空
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]

def encode_huffman_tree(root):
    codes = {}

    def traverse(node, code):
        if node.char:
            codes[node.char] = code
        else:
            traverse(node.left, code + '0')
            traverse(node.right, code + '1')

    traverse(root, '')
    return codes

def huffman_encoding(codes, string):
    encoded = ''
    for char in string:
        encoded += codes[char]
    return encoded

def huffman_decoding(root, encoded_string):
    decoded = ''
    node = root
    for bit in encoded_string:
        if bit == '0':
            node = node.left
        else:
            node = node.right

        if node.char:
            decoded += node.char
            node = root
    return decoded

# 读取输入
n = int(input())
characters = {}
for _ in range(n):
    char, weight = input().split()
    characters[char] = int(weight)

#string = input().strip()
#encoded_string = input().strip()
```

```
# 构建哈夫曼编码树
huffman_tree = build_huffman_tree(characters)

# 编码和解码
codes = encode_huffman_tree(huffman_tree)

strings = []
while True:
    try:
        line = input()
        strings.append(line)

    except EOFError:
        break

results = []
#print(strings)
for string in strings:
    if string[0] in ('0','1'):
        results.append(huffman_decoding(huffman_tree, string))
    else:
        results.append(huffman_encoding(codes, string))

for result in results:
    print(result)
```

四、利用二叉堆实现优先级队列

队列数据结构有一个重要的变体，叫做优先级队列。和队列一样，优先级队列从头部移除元素，不过元素的逻辑顺序是由优先级决定的。优先级最高的元素在最前，优先级最低的元素在最后。因此，入队的元素依据其优先级，也有可能直接被移到优先级队列的头部。

就时间复杂度而言，列表的插入操作和排序操作为O(n)&O(nlogn)。实际上，实现优先级队列的经典方法是使用二叉堆的数据结构，二叉堆的入队操作和出队操作均可达到O(logn)

（一） 二叉堆的实现

1.堆的结构性

为了使二叉堆高效工作，我们利用树的对数性质来表示它。为了保证对数性能，必须维持树的平衡。平衡的二叉树是指，其根结点的左右子树含有数量大致相等的结点。

因此，在实现二叉堆时，我们通过创建一棵**完全二叉树complete binary tree**来维持树的平衡。在完全二叉树中，除了最底层，其他每一层的结点都是满的；且在最底层，我们从左往右填充结点。

完全二叉树的另一个优势在于，可以用一个**列表**来表示，而不需要采用嵌套等复杂方法来表示。由于树的完全性，因此对于在列表中处于位置p（索引|p-1）的结点来说，它的左子结点正好处于位置2p，右子结点处于位置2p+1。若要找到树中任意结点的父结点，只需使用python的整除除法即可（位置n的结点（索引|n-1），父结点的位置是n//2(索引|n//2-1)）。树的列表表示和其“完全”的结构性质，有助于用简单的数学运算遍历完全二叉树，高效地实现二叉堆。

2.堆的有序性

堆的有序性是指，对于堆中任意元素x及其父元素p，p都不大于x（最小堆/小根堆）

3.堆操作

```
class BinHeap:
    #初始化这个列表，currentSize属性用于记录堆的当前大小
    #初始化列表元素为0，唯一用途是为了使后续的方法可以使用整除除法
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    #percUp算法，通过将一个新加入的元素与父结点进行循环比较，找到准确位置
    #这里可以看到初始[0]元素发挥的重要功能
    def percUp(self, i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i // 2]:
                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
            #else:
               #break
            i = i // 2

    #insert插入方法。将元素先追加到列表的末尾，但会破坏二叉堆的性质
    #通过比较新元素与其父元素来重新获得堆的性质
    #如果新元素小于其父元素，就将二者交换，并层层向上遍历直到正确位置
    def insert(self, k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    #percDown方法，帮助对于一个新的元素从上到下遍历，找到其正确位置
    #应用于取出堆顶元素、插入新的堆顶元素后对堆的调整
    #在可以遍历的情况下，对于每次遍历，先找到最小的child（minChild方法）
    #如果对于子结点最小的值mc，小于父结点，则做交换，再进行下一层循环
    def percDown(self, i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
           #else:
               #break
            i = mc

    #minChild用于索引出对于给定结点（在至少有子结点情况下）的min结点的索引
    def minChild(self, i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
                return i * 2
```

```
            else:
                return i * 2 + 1
    #注意这里确定min的方式，若左右子结点的值相同，则父结点应与交换右结点
    #经试验，heapq里的定义方式与此处相同

    #delMin方法，从二叉堆中删除最小元素
    #首先，查找最小元素（即根结点）十分简单，但重组堆的过程较为复杂
    #取出最小元素后，将列表的最后一个元素取出（可用pop）并移到根结点位置
    #索引最后一个元素可以用currentSize可（该size不算初始[0]）
    #调整currentSize的值（-1）
    #将新的根结点沿着树推到正确的位置，采用percDown的实现方式即可
    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    #buildHeap是根据元素列表构建整个堆的方法
    #从完整列表开始构建整个堆只需O(n)
    #找到有子结点的最大索引位置，用percDown调整子树的构造
    #从最大索引位置一直索引到1结束
    def buildHeap(self, alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            print(f'i = {i}, {self.heapList}')
            self.percDown(i)
            i = i - 1
        print(f'i = {i}, {self.heapList}')

bh = BinHeap()
bh.buildHeap([9, 5, 6, 2, 3])
#基于建堆的时间复杂度，可以构造一个使用堆为列表排序的方法，时间复杂度为O(nlogn)
"""
i = 2, [0, 9, 5, 6, 2, 3]
i = 1, [0, 9, 2, 6, 5, 3]
i = 0, [0, 2, 3, 6, 5, 9]
"""
for _ in range(bh.currentSize):
    print(bh.delMin())
"""
2
3
5
6
9
"""
```

注：以上算法给予了构建小顶堆的方法。当然，相同方法也可以应用于大顶堆的构造，只是比较的符号不相同。

另一种更为巧妙的方法是，可以通过在小顶堆中，对于给定x，输入-x，从而最小的-x意味着最大的x，输出后再取相反数，即可实现大顶堆的功能

（二） Practice:剪绳子（OJ 18164）

小张要将一根长度为L的绳子剪成N段。准备剪的绳子的长度为L1,L2,L3...,LN，未剪的绳子长度恰好为剪后所有绳子长度的和。

每次剪断绳子时，需要的开销是此段绳子的长度。

比如，长度为10的绳子要剪成长度为2,3,5的三段绳子。长度为10的绳子切成5和5的两段绳子时，开销为10。再将5切成长度为2和3的绳子，开销为5。因此总开销为15。

请按照目标要求将绳子剪完最小的开销时多少。

已知，1<=N <= 20000，0<=Li<= 50000

输入

第一行：N，将绳子剪成的段数。 第二行：准备剪成的各段绳子的长度。

输出

最小开销

可以用heapq包构造二叉堆：

```
import heapq
n = int(input())
mylist = [int(i) for i in input().split()]
heapq.heapify(mylist)
total = 0
a = heapq.heappop(mylist)
while mylist:
    b = heapq.heappop(mylist)
    total += a+b
    heapq.heappush(mylist,a+b)
    a = heapq.heappop(mylist)
print(total)
```

Note

1. python递归增加python允许的最大层数的方式：

```
import sys
sys.setrecursionlimit(1000000)
```

2. 利用二叉堆可以实现堆排序，时间复杂度为O（nlogn），其为不稳定排序