【华为OD机考 统一考试机试C卷】生成哈夫曼树（C++ Java JavaScript Python C语言）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

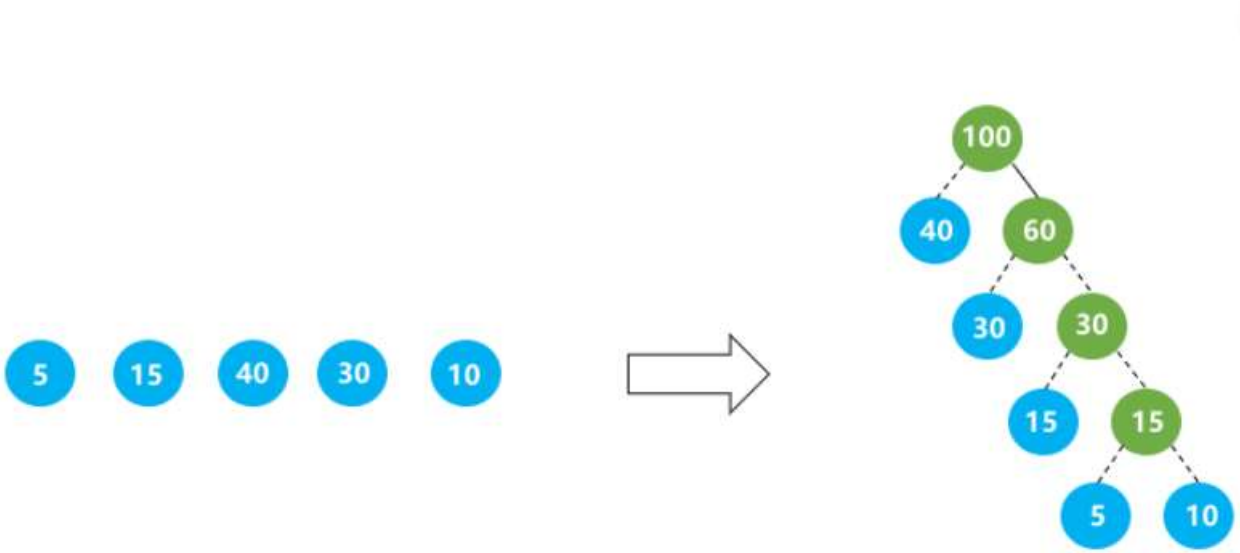
目前在考C卷，经过两个月的收集整理，C卷真题已基本整理完毕
抽到原题的概率为2/3到3/3，也就是最少抽到两道原题。请注意：大家刷完C卷真题，最好要把B卷的真题刷一下，因为C卷的部分真题来自B卷。
另外订阅专栏还可以联系笔者开通在线 OJ 进行刷题，提高刷题效率。
真题目录：华为OD机考机试 真题目录（C卷 + D卷 + B卷 + A卷） + 考点说明
专栏：2023华为OD机试(B卷+C卷+D卷)（C++JavaJSPy）
华为OD面试真题精选：华为OD面试真题精选
在线OJ：点击立即刷题，模拟真实机考环境

题目描述

给定长度为 n nn 的无序的数字数组，每个数字代表二叉树的叶子节点的权值，数字数组的值均大于等于 1 11 。请完成一个函数，根据输入的数字数组，生成哈夫曼树，并将哈夫曼树按照中序遍历输出。
为了保证输出的二叉树中序遍历结果统一，增加以下限制:又树节点中，左节点权值小于等于右节点权值，根节点权值为左右节点权值之和。当左右节点权值相同时，左子树高度高度小于等于右子树。
注意:所有用例保证有效，并能生成哈夫曼树提醒:哈夫曼树又称最优二叉树，是一种带权路径长度最短的一叉树。所谓树的带权路径长度，就是树中所有的叶结点的权值乘上其到根结点的路径长度(若根结点为 0 00 层，叶结点到根结点的路径长度为叶结点的层数)

输入描述

例如：由叶子节点 5 15 40 30 10 生成的最优二叉树如下图所示，该树的最短带权路径长度为 40 * 1 + 30 * 2 +5 * 4 + 10 * 4 = 205 。



输出描述

输出一个哈夫曼的中序遍历数组，数值间以空格分隔

用例

输入

```
5
5 15 40 30 10
```

输出

```
40 100 30 60 15 30 5 15 10
```

模拟计算

请结合上图阅读！ 计算过程如下：

1. 输入的5个数是：5, 15, 40, 30, 10。

2. 将这些数作为节点值创建节点，并将节点添加到优先队列中。

   构建哈夫曼树：

   - 弹出两个最小的节点，值为5和10，合并为一个新节点值为15，将新节点添加回优先队列。
   - 弹出两个最小的节点，值为15（新合成的）和15（原始的），合并为一个新节点值为30，将新节点添加回优先队列。
   - 弹出两个最小的节点，值为30（新合成的）和30（原始的），合并为一个新节点值为60，将新节点添加回优先队列。
   - 弹出两个最小的节点，值为40和60，合并为一个新节点值为100，将新节点添加回优先队列。
   - 此时队列中只剩下一个节点，这就是树的根节点，值为100。

   对哈夫曼树进行中序遍历：

   - 访问左子树，值为40，它是一个叶子节点，输出40。
   - 访问根节点，值为100，输出100。

     访问右子树，值为60，它不是叶子节点，继续中序遍历：

     访问左子树，值为30，它不是叶子节点，继续中序遍历：

     - 访问左子树，值为15，它是一个叶子节点，输出15。
     - 访问根节点，值为30，输出30。
     - 访问右子树，值为15，它是一个叶子节点，输出15。

   - 访问根节点，值为60，输出60。

     访问右子树，值为30，它不是叶子节点，继续中序遍历：

     - 访问左子树，值为10，它是一个叶子节点，输出10。
     - 访问根节点，值为30，输出30。
     - 右子树为空，无输出。

5. 最终输出的结果是：40 100 15 30 60 10 30。

解题思路

小根堆（最小堆）：实现一个小根堆，用于在构建哈夫曼树的过程中维护节点的顺序，确保每次都能从中取出权值最小的节点。

贪心算法：构建哈夫曼树的过程本身是一个贪心算法的应用，每次选择两个权值最小的节点合并，以确保最终树的带权路径长度最短。

DFS（深度优先搜索）：在进行中序遍历时，使用了递归方法。

C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <functional>

// 定义Node结构体，用于构建哈夫曼树的节点
struct Node {
    int value; // 节点存储的数值
    Node* left; // 节点的左子节点
    Node* right; // 节点的右子节点
    // 构造函数，初始化节点的数值和左右子节点指针
    Node(int v) : value(v), left(nullptr), right(nullptr) {}
};

// 比较函数对象，用于优先队列中比较Node指针
struct Compare {
    // 重载操作符()，定义Node指针的比较规则
    bool operator()(Node* a, Node* b) {
        // 返回a的数值是否大于b的数值
        return a->value > b->value;
    }
};

// 构建哈夫曼树的函数
Node* buildHuffmanTree(const std::vector<int>& values) {
    // 创建优先队列，用于存储节点并按数值大小排序
    std::priority_queue<Node*, std::vector<Node*>, Compare> pq;
    // 遍历数值数组，为每个数值创建一个节点并加入优先队列
    for (int value : values) {
        pq.push(new Node(value));
    }
    // 当优先队列中的节点数大于1时，执行循环
    while (pq.size() > 1) {
        // 取出两个数值最小的节点
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();
        // 创建新节点，其数值为两个子节点数值之和
        Node* parent = new Node(left->value + right->value);

```

```
        // 设置新节点的左右子节点
        parent->left = left;
        parent->right = right;
        // 将新节点加入到优先队列中
        pq.push(parent);
    }
    // 返回优先队列中剩下的最后一个节点，即哈夫曼树的根节点
    return pq.top();
}

// 中序遍历哈夫曼树的函数
void inorderTraversal(Node* root, std::string& result) {
    if (root) {
        // 递归遍历左子树
        inorderTraversal(root->left, result);
        // 访问当前节点，将数值转为字符串并追加到结果字符串中
        result += std::to_string(root->value) + " ";
        // 递归遍历右子树
        inorderTraversal(root->right, result);
    }
}

int main() {
    int n; // 定义一个整数变量n，用于存储将要输入的数值数量
    std::cin >> n; // 从标准输入读取n的值
    std::vector<int> values(n); // 创建一个整数向量，用于存储输入的数值
    // 循环读取n个数值，并存储到向量values中
    for (int i = 0; i < n; ++i) {
        std::cin >> values[i];
    }

    // 调用buildHuffmanTree函数构建哈夫曼树，并获取根节点
    Node* root = buildHuffmanTree(values);
    std::string result; // 定义一个字符串，用于存储中序遍历的结果
    // 调用inorderTraversal函数进行中序遍历，并将结果存储到result中
    inorderTraversal(root, result);
    // 如果结果字符串不为空，移除最后一个空格
    if (!result.empty()) {
        result.pop_back();
    }

```

```
    // 输出中序遍历的结果
    std::cout << result << std::endl;
    return 0; // 程序正常结束
}
```

Java

```java
import java.util.PriorityQueue;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        // 创建Scanner对象读取输入
        Scanner scanner = new Scanner(System.in);
        // 读取第一个数字，表示后续将输入多少个数字
        int n = scanner.nextInt();
        // 创建数组存储输入的数字
        int[] values = new int[n];
        // 循环读取输入的数字并存储到数组中
        for (int i = 0; i < n; i++) {
            values[i] = scanner.nextInt();
        }
        // 关闭Scanner对象
        scanner.close();

        // 构建哈夫曼树，并返回根节点
        Node root = buildHuffmanTree(values);
        // 使用StringBuilder来构建输出结果
        StringBuilder result = new StringBuilder();
        // 对哈夫曼树进行中序遍历，并将结果存储到result中
        inorderTraversal(root, result);
        // 输出结果，并去除末尾的空格
        System.out.println(result.toString().trim());
    }

    // 定义Node类，用于构建哈夫曼树的节点
    private static class Node implements Comparable<Node> {
        int value; // 节点存储的数值
        Node left; // 节点的左子节点
        Node right; // 节点的右子节点
```

```
    // 构造函数，初始化节点的数值
    Node(int value) {
        this.value = value;
    }

    // 实现Comparable接口的compareTo方法，用于比较节点的数值大小
    @Override
    public int compareTo(Node other) {
        return this.value - other.value;
    }
}

// 构建哈夫曼树的方法
private static Node buildHuffmanTree(int[] values) {
    // 创建优先队列，用于存储节点并按数值大小排序
    PriorityQueue<Node> pq = new PriorityQueue<>();
    // 将输入的每个数值作为节点加入到优先队列中
    for (int value : values) {
        pq.add(new Node(value));
    }
    // 循环处理，直到优先队列中只剩下一个节点
    while (pq.size() > 1) {
        // 弹出两个数值最小的节点
        Node left = pq.poll();
        Node right = pq.poll();
        // 创建新节点，其数值为两个子节点数值之和
        Node parent = new Node(left.value + right.value);
        // 设置新节点的左右子节点
        parent.left = left;
        parent.right = right;
        // 将新节点加入到优先队列中
        pq.add(parent);
    }
    // 返回优先队列中剩下的最后一个节点，即哈夫曼树的根节点
    return pq.poll();
}

// 中序遍历哈夫曼树的方法
private static void inorderTraversal(Node root, StringBuilder result) {

```

```
        // 如果当前节点不为空，则进行遍历
        if (root != null) {
            // 递归遍历左子树
            inorderTraversal(root.left, result);
            // 访问当前节点，将数值加入到结果中
            result.append(root.value).append(" ");
            // 递归遍历右子树
            inorderTraversal(root.right, result);
        }
    }
}
```

javaScript

```
const readline = require('readline');

// 定义节点类，用于构建哈夫曼树
class Node {
    constructor(value) {
        this.value = value; // 节点的值
        this.left = null;   // 节点的左子节点
        this.right = null;  // 节点的右子节点
    }
}

// 定义最小优先队列类
class MinPriorityQueue {
    constructor() {
        this.elements = []; // 存储队列元素的数组
    }

    // 入队操作，将元素按照优先级（这里是值的大小）插入队列
    enqueue(element) {
        this.elements.push(element); // 将新元素添加到数组的末尾
        // 对数组进行排序，确保最小的元素在数组的开头
        this.elements.sort((a, b) => a.value - b.value);
    }

    // 出队操作，移除并返回队列中优先级最高（值最小）的元素
    dequeue() {
```

```
        return this.elements.shift(); // 移除并返回数组的第一个元素
    }

    // 检查队列是否为空
    isEmpty() {
        return this.elements.length === 0; // 当数组长度为0时，队列为空
    }
}

// 构建哈夫曼树的函数
function buildHuffmanTree(values) {
    const pq = new MinPriorityQueue(); // 创建最小优先队列实例
    // 将所有值封装成节点并加入优先队列
    values.forEach(value => pq.enqueue(new Node(value)));

    // 当队列中有多于一个节点时，执行循环
    while (pq.elements.length > 1) {
        const left = pq.dequeue(); // 弹出两个优先级最高的节点
        const right = pq.dequeue();
        const parent = new Node(left.value + right.value); // 创建新的父节点，其值为两个子节点的和
        parent.left = left;  // 设置新节点的左子节点
        parent.right = right; // 设置新节点的右子节点
        pq.enqueue(parent); // 将新节点加入优先队列
    }

    return pq.dequeue(); // 返回队列中剩下的最后一个节点，即哈夫曼树的根节点
}

// 中序遍历哈夫曼树的生成器函数
function* inorderTraversal(root) {
    if (root) {
        yield* inorderTraversal(root.left); // 递归遍历左子树
        yield root.value; // 返回当前节点的值
        yield* inorderTraversal(root.right); // 递归遍历右子树
    }
}

// 创建命令行读取接口实例
const rl = readline.createInterface({
    input: process.stdin, // 标准输入流
```

```
    output: process.stdout // 标准输出流
});

// 监听输入事件
rl.on('line', (n) => {
    rl.on('line', (line) => {
        const values = line.split(' ').map(Number); // 将输入的行按空格分割，并将每个元素转换为数字
        const root = buildHuffmanTree(values); // 使用输入的值构建哈夫曼树，并获取根节点
        const result = [...inorderTraversal(root)].join(' '); // 对哈夫曼树进行中序遍历，并将结果转换为字符串
        console.log(result); // 打印中序遍历的结果
        rl.close(); // 关闭读取接口
    });
});
```

Python

```
import heapq

# 定义Node类，用于构建哈夫曼树的节点
class Node:
    def __init__(self, value):
        self.value = value  # 节点存储的数值
        self.left = None  # 节点的左子节点
        self.right = None  # 节点的右子节点

    # 重载小于操作符，用于优先队列中比较Node对象
    def __lt__(self, other):
        return self.value < other.value  # 返回当前节点的值是否小于另一个节点的值

# 构建哈夫曼树的函数
def build_huffman_tree(values):
    pq = [Node(value) for value in values]  # 创建Node对象列表
    heapq.heapify(pq)  # 将列表转换为最小堆
    while len(pq) > 1:  # 当堆中有多于一个节点时
        left = heapq.heappop(pq)  # 弹出两个数值最小的节点
        right = heapq.heappop(pq)
        parent = Node(left.value + right.value)  # 创建新节点，其数值为两个子节点数值之和
        parent.left = left  # 设置新节点的左子节点
        parent.right = right  # 设置新节点的右子节点
        heapq.heappush(pq, parent)  # 将新节点加入堆中
```

```
    return pq[0]  # 返回堆中剩下的最后一个节点，即哈夫曼树的根节点

# 中序遍历哈夫曼树的函数
def inorder_traversal(root):
    if root is not None:  # 如果当前节点不为空
        yield from inorder_traversal(root.left)  # 递归遍历左子树
        yield root.value  # 返回当前节点的值
        yield from inorder_traversal(root.right)  # 递归遍历右子树

# 主函数
def main():
    n = int(input())  # 从标准输入读取数字的个数
    values = list(map(int, input().split()))  # 从标准输入读取数字，并转换为整数列表
    root = build_huffman_tree(values)  # 构建哈夫曼树，并获取根节点
    result = ' '.join(map(str, inorder_traversal(root)))  # 对哈夫曼树进行中序遍历，并将结果转换为字符串
    print(result)  # 打印中序遍历的结果

# 当脚本作为主程序运行时，调用main函数
if __name__ == '__main__':
    main()
```

C语言

复制

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 1000 // 定义最大节点数量，用于限制优先队列的最大容量

// 定义节点结构体，用于构建哈夫曼树的节点
typedef struct Node {
    int value;          // 节点存储的数值
    struct Node *left;   // 节点的左子节点
    struct Node *right;  // 节点的右子节点
} Node;

// 优先队列的结构定义
typedef struct {
    Node *data[MAX_SIZE]; // 存储队列元素的数组
    int size;             // 队列当前的元素个数
} PriorityQueue;
```

```
18
19  // 初始化优先队列
20  void initPriorityQueue(PriorityQueue *pq) {
21      pq->size = 0; // 初始时队列为空
22  }
23
24  // 向优先队列中添加元素
25  void push(PriorityQueue *pq, Node *node) {
26      int i = pq->size++; // 获取新元素的插入位置
27      // 调整队列，保持最小堆的性质
28      while (i > 0) {
29          int parent = (i - 1) / 2; // 找到父节点
30          // 如果父节点的值小于或等于当前节点，停止调整
31          if (pq->data[parent]->value <= node->value) break;
32          pq->data[i] = pq->data[parent]; // 否则，交换父节点和当前节点
33          i = parent; // 更新当前节点位置，继续向上调整
34      }
35      pq->data[i] = node; // 将新节点插入到正确位置
36  }
37
38  // 从优先队列中弹出最小元素
39  Node *pop(PriorityQueue *pq) {
40      Node *min = pq->data[0]; // 取出队列中最小的元素
41      Node *last = pq->data[--pq->size]; // 取出队列中最后一个元素
42
43      int i = 0;
44      // 调整队列，保持最小堆的性质
45      while (i * 2 + 1 < pq->size) {
46          int left = i * 2 + 1, right = i * 2 + 2; // 左右子节点位置
47          int smallest = left; // 假设左子节点是最小的
48          // 如果右子节点存在且更小，更新最小节点位置
49          if (right < pq->size && pq->data[right]->value < pq->data[left]->value) {
50              smallest = right;
51          }
52          // 如果最后一个元素小于或等于最小子节点的值，停止调整
53          if (pq->data[smallest]->value >= last->value) break;
54          pq->data[i] = pq->data[smallest]; // 将最小子节点上移
55          i = smallest; // 更新当前节点位置
56      }
57      pq->data[i] = last; // 将最后一个元素放入正确位置
58
```

```
    return min; // 返回最小元素
}

// 创建新节点
Node *createNode(int value) {
    Node *newNode = (Node *)malloc(sizeof(Node)); // 分配内存
    newNode->value = value; // 设置节点值
    newNode->left = newNode->right = NULL; // 初始化左右子节点为NULL
    return newNode; // 返回新创建的节点
}

// 构建哈夫曼树
Node *buildHuffmanTree(int values[], int n) {
    PriorityQueue pq;
    initPriorityQueue(&pq); // 初始化优先队列

    // 将所有值加入优先队列
    for (int i = 0; i < n; i++) {
        push(&pq, createNode(values[i]));
    }

    // 构建哈夫曼树
    while (pq.size > 1) {
        Node *left = pop(&pq);  // 弹出最小元素作为左子节点
        Node *right = pop(&pq); // 弹出下一个最小元素作为右子节点
        Node *parent = createNode(left->value + right->value); // 创建新节点作为父节点
        parent->left = left;   // 设置左子节点
        parent->right = right; // 设置右子节点
        push(&pq, parent);     // 将新创建的父节点加入优先队列
    }
    return pop(&pq); // 返回哈夫曼树的根节点
}

// 中序遍历哈夫曼树
void inorderTraversal(Node *root) {
    if (root != NULL) {
        inorderTraversal(root->left);   // 遍历左子树
        printf("%d ", root->value);     // 打印节点值
        inorderTraversal(root->right);  // 遍历右子树
    }

```

```
}

int main() {
    int n; // 输入的节点数
    scanf("%d", &n); // 读取节点数
    int values[MAX_SIZE]; // 存储节点值的数组

    // 循环读取所有节点的值
    for (int i = 0; i < n; ++i) {
        scanf("%d", &values[i]);
    }

    Node *root = buildHuffmanTree(values, n); // 构建哈夫曼树
    inorderTraversal(root); // 中序遍历哈夫曼树
    printf("\n"); // 打印换行

    return 0; // 程序结束
}
```

文章目录


机考真题
华为OD
No No No
CSDN @算法大师