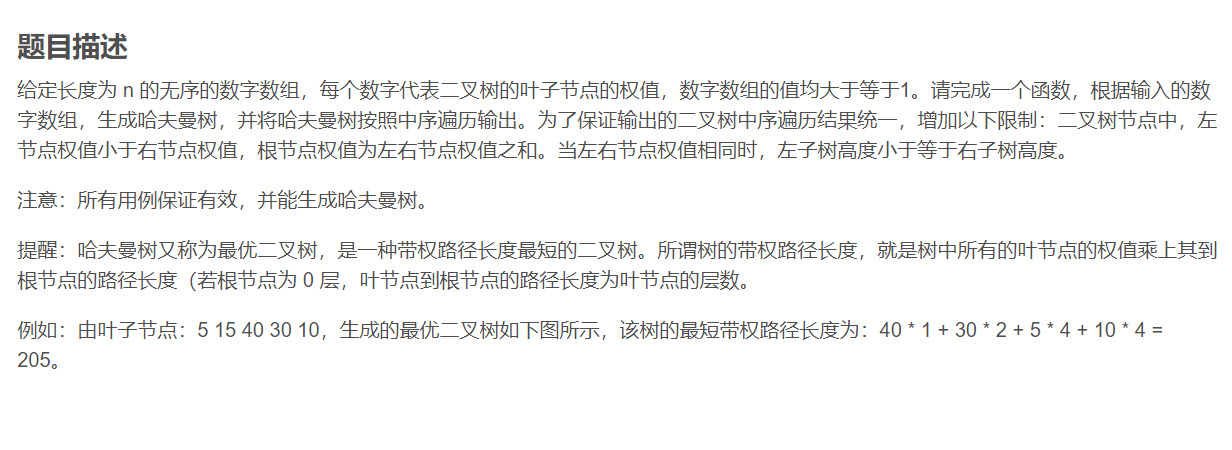
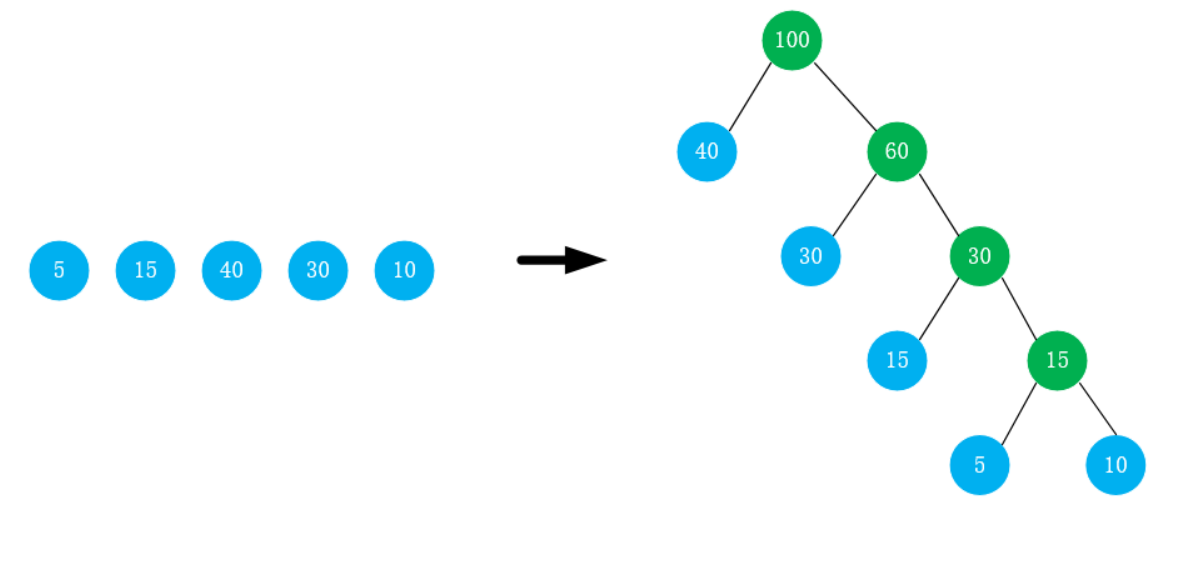
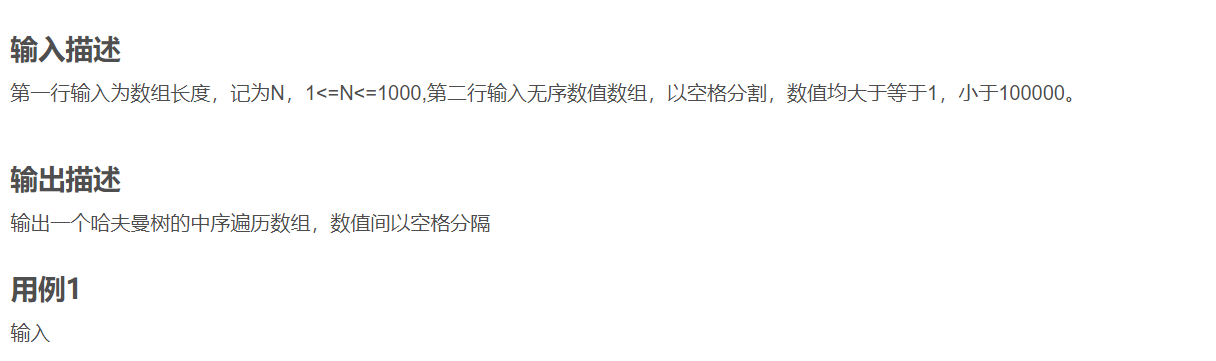
# **E卷-生成哈夫曼树[100分]（ Java | Python3 | C++ | C语言 | JsNode | Go ）**





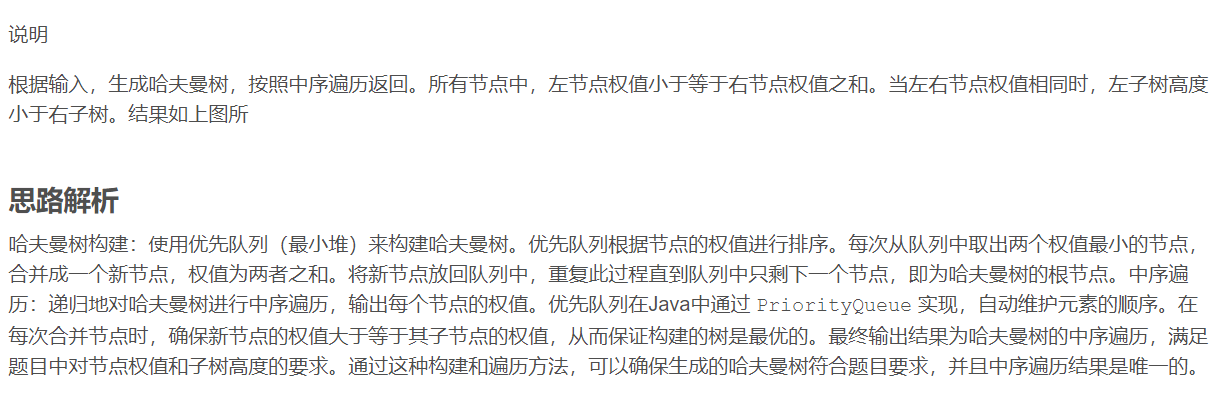


5

5 15 40 30 10



40 100 30 60 15 30 5 15 10





import java.util.PriorityQueue;

import java.util.Scanner;

import java.util.StringJoiner;

public class Main {

static StringJoiner res = new

StringJoiner(" "); // 用于拼接输出结果的中序遍历

static PriorityQueue<Tree> pq; // 优先队列，用于构建哈夫曼树

// 定义树节点类

static class Tree {

Tree lch; // 左子节点

Tree rch; // 右子节点

int cost; // 节点权值

// 构造函数

public Tree(Tree lch, Tree rch, int cost) {

this.lch = lch; // 初始化左子节点

this.rch = rch; // 初始化右子节点

this.cost = cost; // 初始化节点权值

}

}

// 中序遍历函数

public static void mid(Tree x) {

if (x.lch != null) {

mid(x.lch); // 递归遍历左子树

}

res.add("" + x.cost); // 添加当前节点权值到结果中

if (x.rch != null) {

mid(x.rch); // 递归遍历右子树

}

}

// 添加节点到优先队列

static void add(Tree a, Tree b, int c) {

Tree node = new Tree(a, b, c); // 创建新节点

pq.add(node); // 将节点加入优先队列

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in); // 创建扫描器读取输入

int n = sc.nextInt(); // 读取数组长度

pq = new PriorityQueue<>((a,

b) -> a.cost - b.cost); // 初始化优先队列，按权值升序排序

// 读取每个叶子节点权值并添加到优先队列

for (int i = 0; i < n; i++) {

int w = sc.nextInt();

add(null, null, w); // 添加叶子节点

}

// 构建哈夫曼树

while (pq.size() > 1) {

Tree lc = pq.poll(); // 取出权值最小的节点作为左子节点

Tree rc = pq.poll(); // 取出权值第二小的节点作为右子节点

int dcost = lc.cost + rc.cost; // 计算新节点的权值

add(lc, rc, dcost); // 创建新节点并加入优先队列

}

mid(pq.poll()); // 对最终的哈夫曼树进行中序遍历

System.out.println(res); // 输出中序遍历结果

}

}



import heapq

# 定义树节点类

class Tree:

def \_\_init\_\_(self, lch=None, rch=None, cost=None):

# 左子节点

self.lch = lch

# 右子节点

self.rch = rch

# 节点权值

self.cost = cost

# 定义小于比较方法，用于堆的比较

def \_\_lt\_\_(self, other):

return self.cost < other.cost

# 中序遍历函数

def mid(x):

"""

对给定的树进行中序遍历，将节点权值添加到结果列表中。

参数：

x (Tree)：树的根节点。

"""

if x.lch:

mid(x.lch)

res.append(x.cost)

if x.rch:

mid(x.rch)

n = int(input())

pq = []

heapq.heapify(pq)

arr = [int(x) for x in input().split()]

for i in range(n):

w = arr[i]

heapq.heappush(pq, Tree(cost=w))

# 构建哈夫曼树

while len(pq) > 1:

lc = heapq.heappop(pq)

rc = heapq.heappop(pq)

dcost = lc.cost + rc.cost

# 创建新节点并加入优先队列

heapq.heappush(pq, Tree(lch=lc, rch=rc, cost=dcost))

res = []

mid(pq[0])

# 输出中序遍历结果

print(" ".join(map(str, res)))



#include <iostream>

#include <queue>

#include <vector>

using namespace std;

struct Tree {

Tree\* lch; // 左子节点

Tree\* rch; // 右子节点

int cost; // 节点权值

Tree(Tree\* lch, Tree\* rch, int cost) : lch(lch), rch(rch), cost(cost) {}

};

void mid(Tree\* x, vector<int>& res) {

if (x->lch != nullptr) {

mid(x->lch, res); // 递归遍历左子树

}

res.push\_back(x->cost); // 添加当前节点权值到结果中

if (x->rch != nullptr) {

mid(x->rch, res); // 递归遍历右子树

}

}

int main() {

int n;

cin >> n; // 读取数组长度

priority\_queue<pair<int, Tree\*>, vector<pair<int, Tree\*>>, greater<>>

pq; // 优先队列

for (int i = 0; i < n; ++i) {

int w;

cin >> w;

pq.push({w, new Tree(nullptr, nullptr, w)}); // 添加叶子节点

}

// 构建哈夫曼树

while (pq.size() > 1) {

auto lc = pq.top();

pq.pop(); // 取出权值最小的节点作为左子节点

auto rc = pq.top();

pq.pop(); // 取出权值第二小的节点作为右子节点

int dcost = lc.first + rc.first; // 计算新节点的权值

pq.push({dcost, new Tree(lc.second, rc.second, dcost)}); // 创建新节点并加入优先队列

}

vector<int> res;

mid(pq.top().second, res); // 对最终的哈夫曼树进行中序遍历

for (int cost : res) {

cout << cost << " "; // 输出中序遍历结果

}

cout << endl;

return 0;

}



#include <stdio.h>

#include <stdlib.h>

// 定义树节点结构体

typedef struct Tree {

struct Tree\* lch;

struct Tree\* rch;

int cost;

} Tree;

// 定义最小堆结构体

typedef struct MinHeap {

Tree\*\* array;

int size;

int capacity;

} MinHeap;

// 创建新树节点

Tree\* createTree(Tree\* lch, Tree\* rch, int cost) {

Tree\* newTree = (Tree\*)malloc(sizeof(Tree));

newTree->lch = lch;

newTree->rch = rch;

newTree->cost = cost;

return newTree;

}

// 创建最小堆

MinHeap\* createMinHeap(int capacity) {

MinHeap\* minHeap = (MinHeap\*)malloc(sizeof(MinHeap));

minHeap->array = (Tree\*\*)malloc(capacity \* sizeof(Tree\*));

minHeap->size = 0;

minHeap->capacity = capacity;

return minHeap;

}

// 交换两个树节点指针

void swapTree(Tree\*\* a, Tree\*\* b) {

Tree\* temp = \*a;

\*a = \*b;

\*b = temp;

}

// 最小堆的下滤操作

void minHeapify(MinHeap\* minHeap, int index) {

int smallest = index;

int left = 2 \* index + 1;

int right = 2 \* index + 2;

if (left < minHeap->size && minHeap->array[left]->cost < minHeap->array[smallest]->cost)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->cost < minHeap->array[smallest]->cost)

smallest = right;

if (smallest!= index) {

swapTree(&minHeap->array[index], &minHeap->array[smallest]);

minHeapify(minHeap, smallest);

}

}

// 插入树节点到最小堆

void insertMinHeap(MinHeap\* minHeap, Tree\* tree) {

if (minHeap->size == minHeap->capacity) {

return;

}

int i = minHeap->size;

minHeap->array[i] = tree;

minHeap->size++;

while (i!= 0 && minHeap->array[(i - 1) / 2]->cost > minHeap->array[i]->cost) {

swapTree(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

// 获取并删除最小堆的根节点

Tree\* extractMin(MinHeap\* minHeap) {

if (minHeap->size <= 0)

return NULL;

if (minHeap->size == 1) {

minHeap->size--;

return minHeap->array[0];

}

Tree\* root = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

minHeap->size--;

minHeapify(minHeap, 0);

return root;

}

// 中序遍历树

void inorderTraversal(Tree\* root) {

if (root == NULL)

return;

inorderTraversal(root->lch);

printf("%d ", root->cost);

inorderTraversal(root->rch);

}

int main() {

int n;

scanf("%d", &n);

MinHeap\* minHeap = createMinHeap(n);

for (int i = 0; i < n; i++) {

int w;

scanf("%d", &w);

Tree\* tree = createTree(NULL, NULL, w);

insertMinHeap(minHeap, tree);

}

// 构建哈夫曼树

while (minHeap->size > 1) {

Tree\* lc = extractMin(minHeap);

Tree\* rc = extractMin(minHeap);

int dcost = lc->cost + rc->cost;

Tree\* newTree = createTree(lc, rc, dcost);

insertMinHeap(minHeap, newTree);

}

Tree\* root = extractMin(minHeap);

// 中序遍历输出结果

inorderTraversal(root);

printf("\n");

return 0;

}



const readline = require('readline');

// 创建读取接口

const rl = readline.createInterface({

input: process.stdin,

output: process.stdout

});

// 定义树节点类

class Tree {

constructor(lch = null, rch = null, cost = null) {

// 左子节点

this.lch = lch;

// 右子节点

this.rch = rch;

// 节点权值

this.cost = cost;

}

}

// 中序遍历函数

function mid(x, res) {

/\*\*

\* 对给定的树进行中序遍历，将节点权值添加到结果列表中。

\*

\* 参数：

\* x (Tree)：树的根节点。

\* res (Array)：存储遍历结果的数组。

\*/

if (x.lch) {

mid(x.lch, res);

}

res.push(x.cost);

if (x.rch) {

mid(x.rch, res);

}

}

// 优先队列（最小堆）实现

class MinHeap {

constructor() {

this.heap = [];

}

// 向堆中插入元素

push(node) {

this.heap.push(node);

this.bubbleUp(this.heap.length - 1);

}

// 从堆中移除并返回最小元素

pop() {

if (this.heap.length === 1) return this.heap.pop();

const min = this.heap[0];

this.heap[0] = this.heap.pop();

this.bubbleDown(0);

return min;

}

// 获取堆的大小

size() {

return this.heap.length;

}

// 上浮操作

bubbleUp(index) {

while (index > 0) {

const parentIndex = Math.floor((index - 1) / 2);

if (this.heap[index].cost >= this.heap[parentIndex].cost) break;

[this.heap[index], this.heap[parentIndex]] = [this.heap[parentIndex], this.heap[index]];

index = parentIndex;

}

}

// 下沉操作

bubbleDown(index) {

const length = this.heap.length;

while (true) {

let leftChildIndex = 2 \* index + 1;

let rightChildIndex = 2 \* index + 2;

let smallest = index;

if (leftChildIndex < length && this.heap[leftChildIndex].cost < this.heap[smallest].cost) {

smallest = leftChildIndex;

}

if (rightChildIndex < length && this.heap[rightChildIndex].cost < this.heap[smallest].cost) {

smallest = rightChildIndex;

}

if (smallest === index) break;

[this.heap[index], this.heap[smallest]] = [this.heap[smallest], this.heap[index]];

index = smallest;

}

}

}

// 读取输入

rl.question('', (n) => {

rl.question('', (line) => {

const arr = line.split(' ').map(Number);

const pq = new MinHeap();

// 初始化优先队列

for (let i = 0; i < n; i++) {

pq.push(new Tree(null, null, arr[i]));

}

// 构建哈夫曼树

while (pq.size() > 1) {

const lc = pq.pop();

const rc = pq.pop();

const dcost = lc.cost + rc.cost;

pq.push(new Tree(lc, rc, dcost));

}

const res = [];

mid(pq.heap[0], res);

// 输出中序遍历结果

console.log(res.join(' '));

rl.close();

});

});



package main

import (

"container/heap"

"fmt"

)

type Tree struct {

lch \*Tree // 左子节点

rch \*Tree // 右子节点

cost int // 节点权值

}

type Element struct {

cost int

node \*Tree

}

type PriorityQueue []Element

func (pq PriorityQueue) Len() int { return len(pq) }

func (pq PriorityQueue) Less(i, j int) bool { return pq[i].cost < pq[j].cost }

func (pq PriorityQueue) Swap(i, j int) { pq[i], pq[j] = pq[j], pq[i] }

func (pq \*PriorityQueue) Push(x interface{}) {

\*pq = append(\*pq, x.(Element))

}

func (pq \*PriorityQueue) Pop() interface{} {

old := \*pq

n := len(old)

item := old[n-1]

\*pq = old[0 : n-1]

return item

}

func mid(x \*Tree, res \*[]int) {

if x.lch != nil {

mid(x.lch, res) // 递归遍历左子树

}

\*res = append(\*res, x.cost) // 添加当前节点权值到结果中

if x.rch != nil {

mid(x.rch, res) // 递归遍历右子树

}

}

func main() {

var n int

fmt.Scan(&n) // 读取数组长度

pq := &PriorityQueue{}

heap.Init(pq)

for i := 0; i < n; i++ {

var w int

fmt.Scan(&w)

heap.Push(pq, Element{cost: w, node: &Tree{cost: w}}) // 添加叶子节点

}

// 构建哈夫曼树

for pq.Len() > 1 {

lc := heap.Pop(pq).(Element) // 取出权值最小的节点作为左子节点

rc := heap.Pop(pq).(Element) // 取出权值第二小的节点作为右子节点

dcost := lc.cost + rc.cost // 计算新节点的权值

heap.Push(pq, Element{cost: dcost, node: &Tree{lch: lc.node, rch: rc.node, cost: dcost}}) // 创建新节点并加入优先队列

}

res := []int{}

mid((\*pq)[0].node, &res) // 对最终的哈夫曼树进行中序遍历

for \_, v := range res {

fmt.Printf("%d ", v) // 输出中序遍历结果

}

fmt.Println()

}