项目说明文档

数据结构课程设计

——修理牧场

作 者 姓 名： 杨滕超

学 号： 2151298

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 项目概述](#_Toc495668153) 3

[1.1 项目背景](#_Toc495668154) 3

[1.2 项目功能](#_Toc495668155) 3

[1.3 项目分析](#_Toc495668154) 3

[2 项目设计](#_Toc495668156) 4

[2.1 类设计](#_Toc495668158) 4

[2.1.1 Vector类](#_Toc495668163) 4

[2.1.2 Vector游标类](#_Toc495668163) 5

[2.1.3 Heap类](#_Toc495668163) 6

[2.1.4 Priority\_queue类](#_Toc495668163) 6

[3 项目实现](#_Toc495668161) 8

[3.1 总体实现](#_Toc495668162) 8

[3.1.1 总体实现流程图](#_Toc495668167) 8

[3.1.2 总体实现思路](#_Toc495668167) 9

[3.1.3 总体实现主要代码](#_Toc495668167) 9

[3.2 构造哈夫曼树实现 1](#_Toc495668166)0

[3.2.1 构造哈夫曼树实现流程图 1](#_Toc495668167)0

[3.2.2 构造哈夫曼树实现思路](#_Toc495668167) 10

[3.2.3 构造哈夫曼树实现主要代码](#_Toc495668168) 11

[3.3 堆设计实现](#_Toc495668162) 11

[3.3.1 堆设计实现思路](#_Toc495668167) 11

[3.3.2 堆设计实现仿函数](#_Toc495668167) 12

[3.3.3 堆设计实现向上调整流程图](#_Toc495668167) 12

[3.3.4 堆设计实现向上调整思路](#_Toc495668167) 12

[3.3.5 堆设计实现向上调整主要代码](#_Toc495668167) 12

[3.3.6 堆设计实现向下调整流程图](#_Toc495668167) 13

[3.3.7 堆设计实现向下调整思路](#_Toc495668167) 13

[3.3.8 堆设计实现向下调整主要代码](#_Toc495668167) 14

[4 项目测试](#_Toc495668161) 15

[4.1 输入错误测试——非法字符](#_Toc495668174) 15

[4.2 输入错误测试——超过范围](#_Toc495668174) 15

[4.3 输入错误测试——非正整数](#_Toc495668174) 15

[4.4 正确测试——样例](#_Toc495668174) 15

[4.5 正确测试——只有两个木头](#_Toc495668174) 15

[4.6 正确测试——只有一个木头](#_Toc495668174) 16

[4.7 正确测试——多段相同长度的木头](#_Toc495668174) 16

1 项目概述

* 1. 项目背景

该项目在切割木材的过程中，抽象化过程其实是哈夫曼树的构造。哈夫曼树又称最优二叉树，是一种带权路径长度最短的[二叉树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E5%93%88%E5%A4%AB%E6%9B%BC%E6%A0%91/_blank)。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层，叶结点到根结点的路径长度为叶结点的层数）。

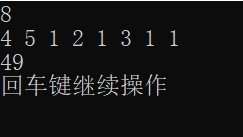
利用哈夫曼树的构造，可以得到一些问题的最优解，在相关问题中具有重要作用。例如哈夫曼编码，作为哈夫曼树的典型应用，它实在数字通信中，将传送的文字转换0和1组成的二进制编码，人们希望总体编码尽可能地简短，哈夫曼树的构造正好满足这一要求。

* 1. 项目功能

输入第一行给出正整数N，其中要求N<=1e4，表示需要将木头锯成N块。第二行给出N个正整数，表示每块木头的长度。当然，在输入的过程中需要我们需要对错误输入的检查，若检测到输入错误，程序将输出相关提示，并让用户重新输入。

程序将输出一个整数，即为将木头锯成N块的最小花费。

程序示例如下：



* 1. 项目分析

1. 对于程序的效率，本项目在构建哈夫曼树的时候，采用优先队列这一数据结构，保证时间复杂度尽可能小。
2. 对于程序健壮性与用户友好性，程序在用户输入错误时候将输出输入错误提示，并提示用户重新输入，而不会引起程序的崩溃。
3. 对于功能完整性，本程序按照题目要求，将求解出正确无误的答案。

2 项目设计

2.1 类设计

2.1.1 Vector类

向量作为存储同一种类型数据的一维数组，相当于是数组的扩展，在其基础上增加方便程序员工作的操作。其中的优点在于：对于查找某个位置的元素的时间复杂度为O(1)，但是它的缺点也十分明显：向量的删除与插入的时间代价是巨大的，时间复杂度基本是O(n)级别；但特殊的是，在最后插入元素时，时间复杂度为O(1)。

我们设计Vector类的同时，同样设计了它的迭代器。

主要成员函数如下：

·//构造函数

Vector();

Vector(int size);

Vector(const Vector<T>& V);

Vector(int size, const T& val);

·//清空Vector

void clear();

·//最后添加元素

void emplace\_back(const T& x);

·//最后删除元素

void pop\_back();

·//返回最后元素

const T back() const ;

T& back() ;

·//返回开头元素

const T front() const ;

T& front() ;

·//插入

void insert(const Vector<T>::vector\_iterator& it, const T& x);

·//删除

void remove(const Vector<T>::vector\_iterator& it);

·//迭代器begin

inline Vector<T>::vector\_iterator begin() ;

·//迭代器end

inline Vector<T>::vector\_iterator end() ;

·//返回迭代器的查找

typename Vector<T>::vector\_iterator find(const T& x);

·//重载[]

T& operator[](const int index) ;

·//重载=

Vector<T>& operator=(const Vector<T>& V);

·//重新设置大小

void resize(int size);

void resize(int size, const T& val);

2.1.2 Vector游标类

其中数据成员包含了指向Vector中数组元素的指针，并实现的重载\*、->、++、--、<、>等操作符。

主要函数如下：

·//重载\*

T& operator\*();

·//重载->

T\* operator->();

·//重载==

inline bool operator==(const vector\_iterator& it) const ;

·//重载!=

inline bool operator!=(const vector\_iterator& it) const;

·//重载不等号

inline bool operator<(const vector\_iterator& it) const;

inline bool operator>(const vector\_iterator& it) const ;

inline bool operator<=(const vector\_iterator& it) const;

inline bool operator>=(const vector\_iterator& it) const ;

·//重载自加

inline Vector<T>::vector\_iterator& operator++();

·//重载自减

inline Vector<T>::vector\_iterator& operator--();

·//后置自加

Vector<T>::vector\_iterator operator++(int);

·//后置自减

Vector<T>::vector\_iterator operator--(int);

·//与数字相加

friend Vector<T>::vector\_iterator operator+(const vector\_iterator& it,int num);

·//与数字相减

friend Vector<T>::vector\_iterator operator-(const vector\_iterator& it,int num);

·//自加

Vector<T>::vector\_iterator operator+=(int num);

·//自减

Vector<T>::vector\_iterator operator-=(int num);

·//两个相减

friend int operator-(const vector\_iterator& it1, const vector\_iterator& it2);

2.1.3 Heap类

堆是一种树形结构，是完全树，因此常常利用数组储存。分为大根堆和小根堆，以小根堆为例，任意一个根的孩子结点的值总是不小于其父节点，同时两个孩子也是一个小根堆。

我们利用向量作为堆的基础，通过迭代方式，实现堆的操作，主要成员函数如下：

·上浮调整void siftFix(Iter beg, Iter end, Compare cmp);

·下沉调整void sinkFix(Iter beg, Iter end, Compare cmp);

2.1.4 Priority\_queue类

基于Vector类作为存储数据的数据结构，Heap类中的方法，形成优先队列，其中参数模板包括数据类型、数据底层结构（默认为Vector）以及元素之间的比较方法。比较方法作为仿函数传入类中作为类的数据成员。其中入队出队的时间复杂度均为O(logn)

主要代码如下：

template<class T, class Seq = Vector<T>, class Compare = Less<T>>

class Priority\_queue {

private:

Seq data;

Compare cmp;

public:

//构造函数

Priority\_queue() :data(), cmp() {}

Priority\_queue(const Compare& cmp) :data(), cmp(cmp) {}

//析构函数

~Priority\_queue() {}

//判断是否为空

bool empty() const;

//返回大小

int size() const;

//顶

const T top() const;

//加入元素

void emplace(const T& x){

data.emplace\_back(x);

pushHeap(data.begin(), data.end(), cmp);

}

//弹出元素

void pop(){

popHeap(data.begin(), data.end(), cmp);

data.pop\_back();

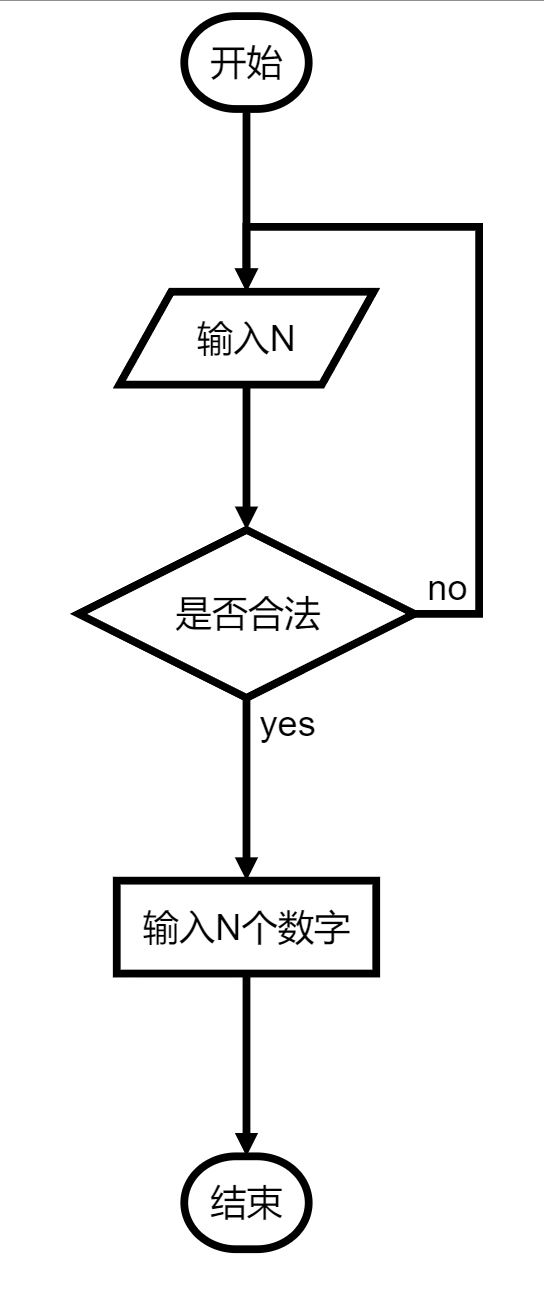
}

};

3 项目实现

3.1 总体实现

3.1.1 总体实现流程图



3.1.2 总体实现思路

首先请用户输入正整数N（N <= 1e4），在此过程中检查输入是否正确，若错误则输出输入错误提示，并请用户重新输入。

3.1.3 总体实现主要代码

while (1){

cin >> N;

if (cin.fail() || N < 1 || N > 1e4){

cout << "输入有误，请重新输入" << endl;

cin.clear();

cin.ignore(65536, '\n');

}

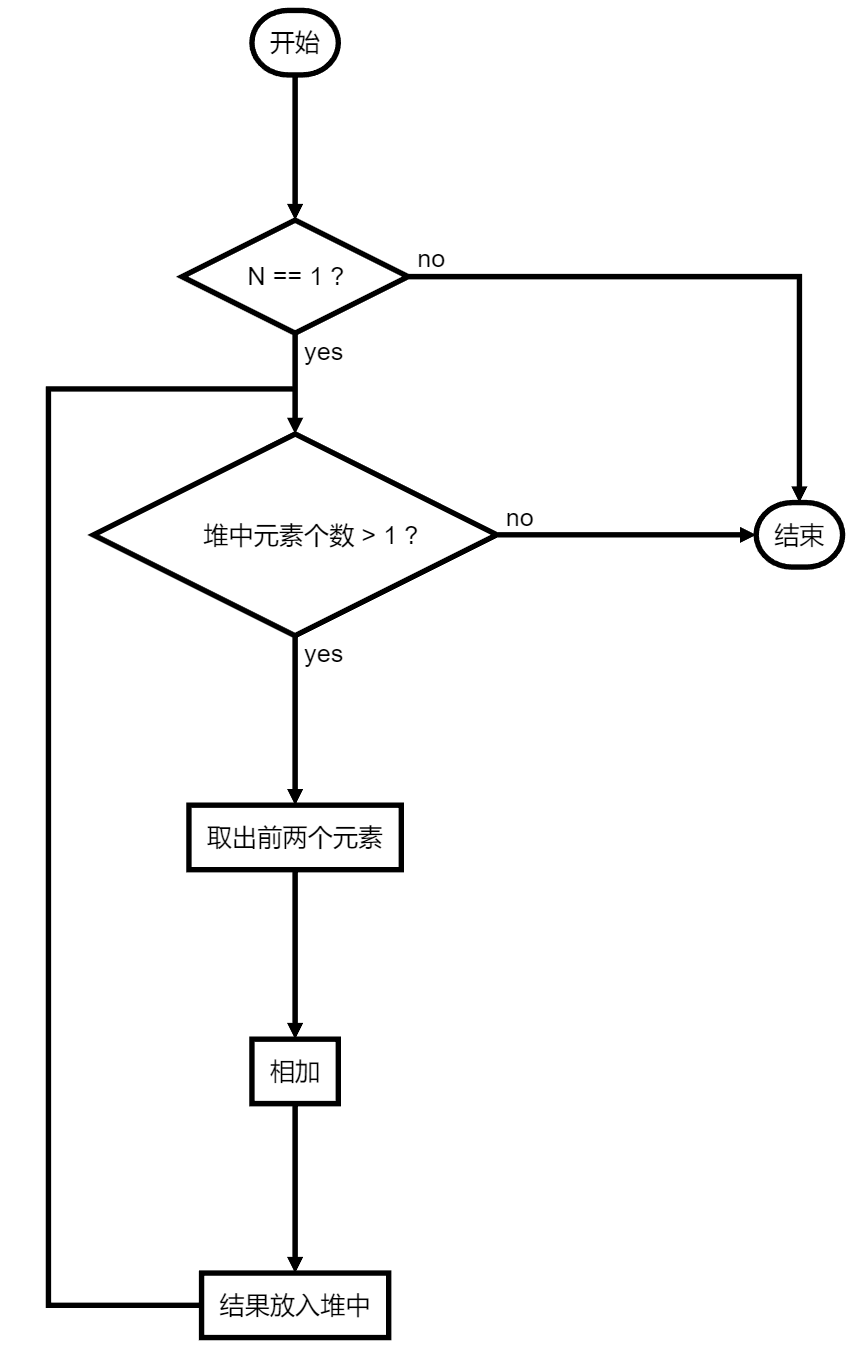
else

break;

}

3.2 构造哈夫曼树实现

3.2.1 构造哈夫曼树实现流程图



3.2,2 构造哈夫曼树实现思路

请用户输入N个正整数，每得到一个正确输入就将该数放入优先队列中，否则输出输入错误提示，并要求用户重新输入。直到输入完成，检查N值是否等于1，若等于则直接输出花费代价为0，否则进入哈夫曼树的构造。

构造具体过程为：只要优先队列中的元素个数大于1，进入循环，取出前两个优先级最高的数字，即两块最短的木头，合成一个木头，使得花费最少，再将合成的木头压入队列中。

上述过程实际上是锯木头的逆过程，将切割好的木头重新合在一起，由于每次取得木头都是最短的，因此整个过程总的花费是最少的。

3.2.3 构造哈夫曼树实现主要代码

Priority\_queue<int,Vector<int>, Greater<int>> woods;

if (N == 1)

cost = 0;

else{

//逆向模拟该过程，不断从切好的集合中选取两条代价最小的拼接在一起

while (woods.size() > 1){

int len1 = woods.top();

woods.pop();

int len2 = woods.top();

woods.pop();

//记录花费代价

cost += len1 + len2;

woods.emplace(len1 + len2);

}

}

3.3 堆设计实现

3.3.1 堆设计实现思路

堆的操作主要分为入堆和出堆。入堆的主要过程为：将要压入的元素放到数组的最后一个位置，之后进行向上调整。出堆的主要过程为：将堆顶的元素与最后一个元素交换，再对堆顶元素进行向下调整。

上文提到的向上调整和向下调整，可以实现的最重要的条件在于堆是一颗基于数组存储的树。因此，我们可以很容易地找到一个结点对应的双亲结点和孩子结点，若没有越界，一个结点的下标为i，则其双亲结点的下标为(i - 1) / 2，左孩子结点的下标为2 \* i + 1，右孩子结点的下标为2 \* i + 2。

对于用户需要的是大根堆还是小根堆，这里可以实现依据用户的要求而定。这里提供两种方法：其一是重载操作符大于小于号，即< 和 >，由于这里的堆默认调用Less作为比较仿函数，若不想使用Great仿函数，可以将<重载为>，>重载为<，实现大小的相反。其二是使用Greater仿函数或自己构造仿函数，自定义比较方法，通过参数模板传入Heap类中。

这里还有一个小细节：向上调整或向下调整的过程中，并不用每次都进行交换，而是将要调整的值保存在临时变量中，找到最终的目标位置之后再进行赋值，由此可以减少交换值的次数，提高程序的效率。

3.3.2 堆设计实现仿函数

内置比较函数主要代码如下：

template<class T>

struct Less {

bool operator()(const T& x, const T& y) {

return x < y;

}

};

template<class T>

struct Greater {

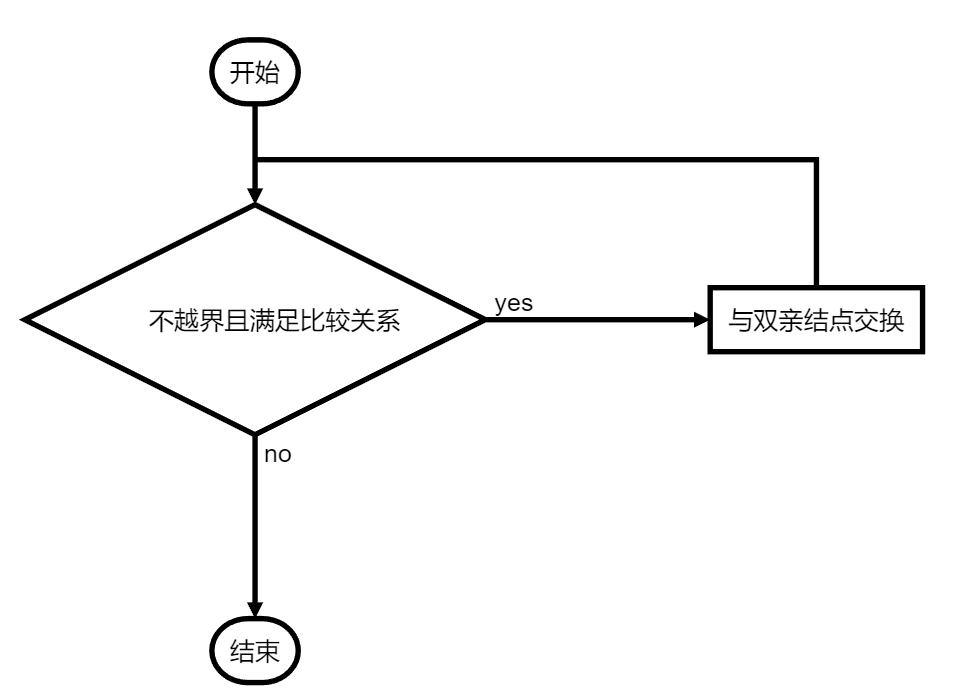
bool operator()(const T& x, const T& y) {

return x > y;

}

};

3.3.3 堆设计实现向上调整流程图



3.3.4 堆设计实现向上调整思路

对于向上调整具体过程，这里将其存入临时变量val中，设置变量hole表示val最终目标位置。在循环过程中，val与其双亲结点的值比较，满足比较条件则将双亲结点的值下移，hole位置上移，将双亲结点的位置赋给hole，计算hole新一轮的双亲，直到不满足比较条件，退出循环，这时候将最终目标位置hole处的值用val填充。

在上述过程中，需要注意的是要时刻保证下标不会越界。

3.3.5 堆设计实现向上调整主要代码

template<class Iter, class Compare>

void siftFix(Iter beg, Iter end, Compare cmp){

typedef typename Iter::value\_type value\_type;

//新加入的元素的值

value\_type val = \*(end - 1);

//要处理的hole的位置

int hole = (end - 1) - beg;

//hole的父节点

int parent = (hole - 1) / 2;

//上溯 结束 条件 hole越界 or 不满足cmp关系

while (hole >= 0 && hole != parent && cmp( \*(beg + parent), val)){

//父节点的值下来

\*(beg + hole) = \*(beg + parent);

//hole上去

hole = parent;

parent = (hole - 1) / 2;

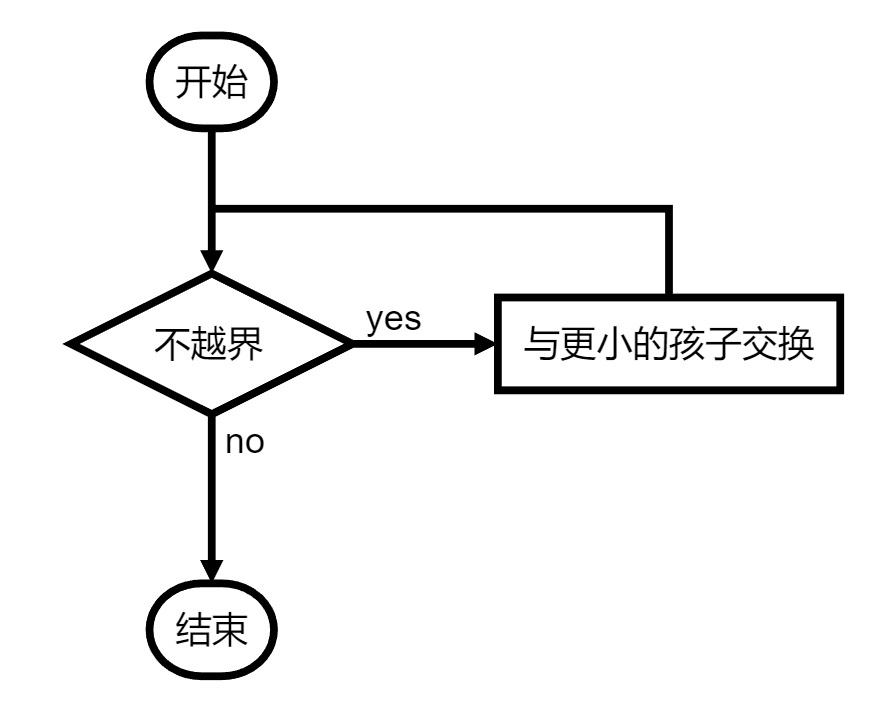
}

//hole上去到能上去的最高位置

\*(beg + hole) = val;

}

3.3.6 堆设计实现向下调整流程图



3.3.6 堆设计实现向下调整思路

对于向下调整，首先需要选择两个孩子中更满足比较条件的孩子，沿着该孩子的路径下沉到最后。这里同样利用hole标记最终的目标位置，以减少值交换的次数。最后同样要注意的的是，对于堆是一棵完全二叉树，因此可能会出现只有一个左孩子的情况，这时候需要单独判断。

3.3.6 堆设计实现向下调整主要代码

template<class Iter, class Compare>

void sinkFix(Iter beg, Iter end, Compare cmp){

typedef typename Iter::value\_type value\_type;

//要处理的元素的值

value\_type val = \*beg;

int hole = 0;

int leftNode = hole \* 2 + 1;

int rightNode = leftNode + 1;

int len = end - beg;

//寻找最后一个元素要去的hole，因为第一个元素要占用

while (rightNode < len){

//选中符合条件的那个孩子交换，hole下沉

if (cmp(\*(beg + rightNode), \*(beg + leftNode))){

\*(beg + hole) = \*(beg + leftNode);

hole = leftNode;

}

else{

\*(beg + hole) = \*(beg + rightNode);

hole = rightNode;

}

leftNode = hole \* 2 + 1;

rightNode = leftNode + 1;

}

//还有一个单独的左孩子，继续hole下沉

if (leftNode < len){

\*(beg + hole) = \*(beg + leftNode);

hole = leftNode;

}

//最后一个位置要被val占用，

\*(beg + hole) = \*(end - 1);

//下来会打乱顺序，再进行上溯

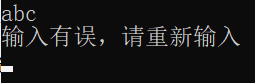
siftFix(beg, beg + hole + 1, cmp);

\*(end - 1) = val;

}

4 项目测试

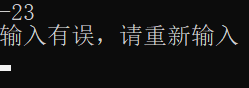
4.1 输入错误测试——非法字符



4.2 输入错误测试——超过范围



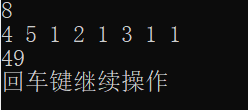
4.3 输入错误测试——非正整数



4.4 正确测试——样例

目标输出：49

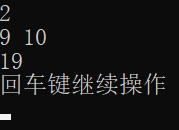
测试结果：



4.5 正确测试——只有两个木头

目标输出：19

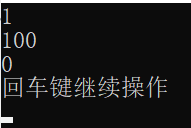
测试结果：



4.6 正确测试——只有一个木头

目标输出：0

测试结果：



4.7 正确测试——多段相同长度木头

目标输出：24

测试结果：

