项目说明文档

数据结构课程设计

——修理牧场

同济大学

Tongji University

目 录

[1 分析 1](#_Toc60095890)

[1.1 项目简介 1](#_Toc60095891)

[1.2项目要求 1](#_Toc60095892)

[1.3 功能分析 1](#_Toc60095893)

[2 设计 2](#_Toc60095894)

[2.1 数据结构设计 2](#_Toc60095895)

[2.2 类结构设计 2](#_Toc60095896)

[2.3 成员与操作设计 2](#_Toc60095897)

[2.4 系统设计 4](#_Toc60095898)

[3 实现 4](#_Toc60095899)

[3.1 堆插入功能的实现 4](#_Toc60095900)

[3.1.1 堆插入的思想： 4](#_Toc60095901)

[3.1.2 堆插入的流程图： 4](#_Toc60095902)

[3.1.3 堆插入功能核心代码 6](#_Toc60095903)

[3.2 去顶功能的实现 6](#_Toc60095904)

[3.2.1 去顶功能思想 6](#_Toc60095905)

[3.2.2 去顶功能流程图 6](#_Toc60095906)

[3.2.3 去顶功能核心代码 8](#_Toc60095907)

[3.3 总体系统的实现 9](#_Toc60095908)

[3.3.1 总体系统流程图 9](#_Toc60095909)

[3.3.2 总体系统核心代码 11](#_Toc60095910)

[3.3.3 总体系统截屏示例 11](#_Toc60095911)

[4 测试 12](#_Toc60095912)

[4.1 堆功能测试 12](#_Toc60095913)

[4.1.1测试原理说明 12](#_Toc60095914)

[4.1.2 测试核心代码 12](#_Toc60095915)

[4.1.3 测试结果截图 13](#_Toc60095916)

[4.2 主函数测试 13](#_Toc60095917)

[4.2.1 样例测试 13](#_Toc60095918)

[4.2.2 二分测试 14](#_Toc60095919)

[4.2.3 边界测试 14](#_Toc60095920)

[4.3 出错测试 15](#_Toc60095921)

[4.3.1 输入范围错误 15](#_Toc60095922)

[4.3.2 输入非法字符 15](#_Toc60095923)

# 1 分析

## 1.1 项目简介

农夫要修理牧场的一段栅栏，他测量了栅栏，发现需要N块木头，每块木头长度为整数*Li*个长度单位，于是他购买了一个很长的，能锯成N块的木头，即该木头的长度是*Li*的总和。

但是农夫自己没有锯子，请人锯木的酬金跟这段木头的长度成正比。为简单起见，不妨就设酬金等于所锯木头的长度。例如，要将长度为20的木头锯成长度为8，7和5的三段，第一次锯木头将木头锯成12和8，花费20；第二次锯木头将长度为12的木头锯成7和5花费12，总花费32元。如果第一次将木头锯成15和5，则第二次将木头锯成7和8，那么总的花费是35（大于32）。

## 1.2项目要求

输入格式：输入第一行给出正整数N（N<104），表示要将木头锯成N块。第二行给出N个正整数，表示每块木头的长度。

输出格式：输出一个整数，即将木头锯成N块的最小花费。

## 1.3 功能分析

统计将木头锯成N块确定的长度所需要话费的最小时间，这个过程需要用到哈夫曼编码的思想，即每一次从堆中取出值最小的两个，加以合并，再将合并后的结果重新置入堆中。

同时，考虑到N较大（最大可达104），如果在逻辑上使用线性结构，如果是无序的情况，从无序的线性表中找出最小值的过程需要O(n)的最坏时间复杂度，删除的过程需要O(1)(如果存储结构是链表)或者O(n)(如果存储结构是顺序表)的时间复杂度，将合并后的结果插入线性表又需要O(n)的时间复杂度，由简单的计算可知，需要的时间复杂度；如果是有序的线性表，排序需要的时间复杂度，取出最小值仅需要O(1)的时间复杂度，但是去除最小值后需要进行重新排序，由于除了被去除的元素，其他元素均已有序，可以使用简单插入排序，但仍然需要O(n)的时间复杂度，所以删除元素的时间复杂度仍未O(n)。而插入元素虽然可以使用二分查找(要求存储结构为顺序表)，但是找到之后需要进行调整，调整过程仍然是O(n)的时间复杂度。综上，不论是有序线性表，还是无序线性表，都需要的时间复杂度，鉴于数据量较大，需要使用更高效的数据结构。

在本项目中，使用了树形结构中的堆（小根堆）。这种逻辑结构可以实现效率为的插入，删除最小值操作，总复杂度为。

# 2 设计

## 2.1 数据结构设计

如上功能分析所述，该项目要求大量的插入，删除最小值等操作，同时，为了实现简易，采用顺序存储。

## 2.2 类结构设计

由于堆结构应用广泛，为了使程序能够泛用，使用类模板。

## 2.3 成员与操作设计

**（小根）堆类（Heap）**

**私有成员：**

T\* Data; //存放数据

int \_size; //数据量的大小

int \_capacity; //申请的内存大小，当内存不足时会动态更新

const int INIT\_SIZE = 100; //初始内存申请大小

**ADT接口：**

Heap()

//无参数的构造函数

Heap(int )

//将读取n个元素，并依次插入堆中的构造函数

Heap& operator=(const Heap&)

//复制构造函数

Heap(const Heap& )

//拷贝构造函数

void insert(T)

//将新元素插入堆中

T top()

//取堆首元素

T pop()

//弹出堆首元素，并调整堆结构使其仍为小根堆

int size()

//堆中元素的数量

bool empty()

//返回堆是否为空

void print()

//层次遍历打印堆中元素

## 2.4 系统设计

系统输出提示信息，读入数据（并进行合法性检查），进行运算，最后输出结果。

# 3 实现

## 3.1 堆插入功能的实现

### 3.1.1 堆插入的思想：

先满足结构性，将要插入元素插到最后一个，再不断地与其父节点比较调整位置，直到满足有序性或者到达根节点为止。

### 3.1.2 堆插入的流程图：

开始

插入堆尾

堆大小加一

内存不足？

申请新空间

i不为堆首且小于其父节点？

i指向堆尾元素

i节点与其父节点交换

i指向其父节点

结束

y

n

y

n

### 3.1.3 堆插入功能核心代码

void insert(T e)

{

//内存不足

if (\_size + 2 >= \_capacity)

{

T\* oldData = Data;

Data = new T[(\_capacity << 1)];

for (int i = 1; i <= \_size; ++i)Data[i] = oldData[i];

delete[]oldData;

\_capacity <<= 1;

}

Data[++\_size] = e;

int i = \_size;

while (i > 1 && Data[i] < Data[i / 2])

{

swap(Data[i], Data[i / 2]);

i >>= 1;

}

}

## 3.2 去顶功能的实现

### 3.2.1 去顶功能思想

取出根结点元素，同时用最后一个结点代替根结点，删除最后一个节点。对于每一个子树结构，取根结点两孩子的较小的一个，若较小值小于此时的根节点，将该较小值移动至根结点，不断重复，直到满足根节点小于两个孩子节点为止。

### 3.2.2 去顶功能流程图

开始

记录堆首

堆首与堆尾交换

堆大小减一

堆中无元素？

报错

i\*2<=\_size?

i指向堆首元素

i节点与k节点交换

i=k

结束

y

n

y

n

取i中两孩子中小者，以k指向它

Data[k]<Data[i]

n

y

### 3.2.3 去顶功能核心代码

T pop()

{

if (!\_size) { printf("堆中没有元素！\n"); exit(-1); }

//栈空情况，报错

T e = Data[1];

//记录栈首

swap(Data[1], Data[\_size--]);

//交换，大小减一

int i = 1;

while (i \* 2 <= \_size)

{

int k;

if (i \* 2 == \_size || Data[i \* 2] < Data[i \* 2 + 1])

k = (i << 1);

else k = (i << 1) + 1;

//取出两孩子中的最小值

if (Data[k] < Data[i])

{

swap(Data[i], Data[k]);

i = k;

}

else break;

}

return e;

}

## 3.3 总体系统的实现

### 3.3.1 总体系统流程图

开始

堆的大小大于1？

读入数据并检查合法性

新建堆H，将数据插入堆中

结束

SUM=0

连续两次取出堆首，并记录其和为len

SUM+=len

将len重新插入到堆中

输出SUM

y

n

### 3.3.2 总体系统核心代码

int n;

printf("请输入要锯成的块数：\t");

n = readInt(range);

printf("请输入各个块数的长度：\n");

Heap<ll> H(n);

ll SUM = 0;

while (H.size() > 1)

{

ll len = H.pop() + H.pop();

SUM += len;

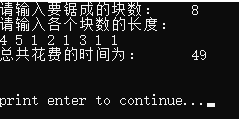
H.insert(len);

}

cout << "总共花费的时间为：\t" <<SUM;

return 0;

### 3.3.3 总体系统截屏示例



# 4 测试

## 4.1 堆功能测试

### 4.1.1测试原理说明

该测试针对堆的无参数构造函数，有参数构造函数，复制构造函数，赋值构造函数，插入和去顶函数进行测试，同时对于堆的插入和去顶函数的效率进行测试。

先使用rand()构造无序数组，然后无序数组的元素插入无参数构造的H1，再调用有参数构造函数构造H2，调用复制构造函数将H1赋值到H2，再用赋值构造函数用H2构造H3。最后将H3中的去顶，弹出的元素从0到n-1依次赋值给一开始构造的无序数组。最后调用检查排序函数检查数组此时是否已经有序，并且统计整个过程的时间。（整个过程总共包括N次插入，N次去顶，kN次赋值）

### 4.1.2 测试核心代码

const int maxn = 1e6 + 5;

//数组的最大值

int arr[maxn];

bool isSorted(int\* arr, int n)

{

for (int i = 0; i < n - 1; ++i)if (arr[i + 1] < arr[i])return 0;

return 1;

}

//检查数组是否已经有序

int main()

{

int n;

printf("请输入测试数n（不大于一百万）:");

cin >> n;

clock\_t t1 = clock();

//生成无序数组

srand(time(0));

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

arr[i] = rand();

Heap<int> H1;

//无参数构造函数

Heap<int>H2(0);

//有参数构造函数

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

H1.insert(arr[i]);

//把元素插入H1

H2 = H1;

//调用赋值构造函数

Heap<int>H3(H2);

//调用赋值构造函数

for (int i = 0; i < n; ++i)arr[i] = H3.pop();

//对数组重新赋值

clock\_t t2 = clock();

cout << "是否排序："<<isSorted(arr, n) << endl;

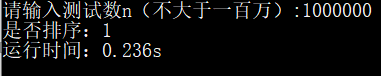
//检查数组是否有序

cout << "运行时间：" << (double)(t2 - t1) / CLOCKS\_PER\_SEC <<"s\n" ;

return 0;

}

### 4.1.3 测试结果截图



## 4.2 主函数测试

### 4.2.1 样例测试

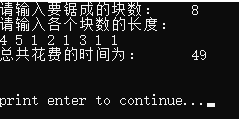
**测试用例：**

8

4 5 1 2 1 3 1 1

**预期结果：**49

**实验结果：**



### 4.2.2 二分测试

**测试用例：**

将个1插入堆中，作为输入数据

**预期结果：**

可以推算出：

**实验结果：**



### 4.2.3 边界测试

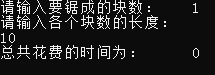
**测试用例：**

1

10

**预期结果：**0

**实验结果：**



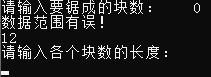
## 4.3 出错测试

### 4.3.1 输入范围错误

**测试用例：**0

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**

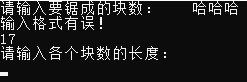
****

### 4.3.2 输入非法字符

**测试用例：**哈哈哈

**预期结果：**程序给出提示信息，程序正常运行不崩溃。

**实验结果：**

****