项目说明文档

数据结构课程设计

——修理牧场

作 者 姓 名： 肖杨

学 号： 1950430

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 分析 3

2 设计 3

3 实现

4 测试

1 分析

1.1 背景分析

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 功能分析

输入格式：输入第一行给出正整数N（N≤），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。

2 设计

2.1 算法设计

农夫让人锯木的酬金和这段木头的长度成正比，农夫要把一块长木头锯成给定的N块长度为整数 Li个单位长度木头。所以一块木头可能需要锯多次才能达到目的。因为木头长度越大，需要付的酬金也越多，一块木头被锯了多次，它所产生的酬金也会重复很多次。所以总体上来看需要避免对长的木头进行多次操作。而尽可能地通过把短的木头锯多次来达到目的。

根据这道题的思路分析来看，所以采用构造霍夫曼树可以方便地解决问题。直观地来看，把结果中长度较小的木头远离结点，把结果中长度较大地木头放在离根结点越近的位置。计算霍夫曼树的带权路径长度就可以得到将木头锯成N块的最小花费。

由于不需要知道其他子树的权值，所以本项目并不需要实现完整的霍夫曼树，只需要一个最小堆即可。

2.2 数据结构设计

如上功能分析所述， 本项目需要数据结构最小堆，本质只需实现优先级队列的功能而不需使用树，通过连续地址储存所有元素。同时使用模板类，使实现的最小堆具有泛用性。

2.3 成员与操作设计

最小堆类

类定义：

template<class T>

class MinHeap

{

public:

MinHeap(int maxSize);

MinHeap(T arr[], int num);

~MinHeap();

void push(const T& elem);

T pop();

bool isEmpty();

bool isFull();

int elemNumber();

private:

int size;

int number;

T\* buffer;

void down(int start, int end);

void up(int start);

};

私有成员：

int size;

//堆大小

int number;

//当前已储存的元素数量

T\* buffer;

//用于储存元素的连续地址

私有操作：

void down(int start, int end);

//向下调整，用于形成堆和移除最小元素

void up(int start);

//向上调整，用于添加元素

公有操作：

MinHeap(int maxSize);

//用堆大小来建立空堆

MinHeap(T arr[], int num);

//用已有的数组来建立一个堆，并对其大小进行了一定程度的扩充

void push(const T& elem);

//入堆尾，并向上调整

T pop();

//移除堆首，并取最后一个元素补上，再进行向下调整，返回移除的元素

bool isEmpty();

//判堆空

bool isFull();

//堆判满

int elemNumber();

//返回元素数量

3 实现

3.1 最小堆的实现

此处仅说明两个功能的实现，其余功能未使用。

3.1.1 最小堆的建立

从已有的数组创建最小堆，将从最后一个元素的前驱开始，逐步向前直到根结点调整为最小堆。主要分为向下调整和遍历至根结点两部分。

代码如下：

向下调整：

template<class T>

void MinHeap<T>::down(int start, int end)

{

int parent = start, child = 2 \* parent + 1;

T first = buffer[start];

while (child <= end)

{

if (child < end && buffer[child] > buffer[child + 1])

{

child++;

}

if (first < buffer[child])

{

break;

}

else

{

buffer[parent] = buffer[child];

parent = child;

child = parent \* 2 + 1;

}

}

buffer[parent] = first;

}

从最后一个元素前驱遍历至根结点

template<class T>

MinHeap<T>::MinHeap(T arr[], int num)

{

size = (num + 3)\*1.25;

number = num;

buffer = new T[size];

memcpy(buffer, arr, sizeof(T) \* num);

int now = (num - 2) / 2;

while (now >= 0)

{

down(now--, num - 1);

}

}

3.1.2 移除根结点的实现

首先，保存根结点元素，然后使最后一个元素填补根结点的空缺并使堆大小减1，随后自根结点向下调整一次使堆恢复最小堆的结构，最后输出根节点元素。

代码如下：

template<class T>

void MinHeap<T>::down(int start, int end)

{

int parent = start, child = 2 \* parent + 1;

T first = buffer[start];

while (child <= end)

{

if (child < end && buffer[child] > buffer[child + 1])

{

child++;

}

if (first < buffer[child])

{

break;

}

else

{

buffer[parent] = buffer[child];

parent = child;

child = parent \* 2 + 1;

}

}

buffer[parent] = first;

}

3.2 整体算法的实现

3.2.1 算法流程

首先，输入木块数量和木块长度，期间持续对输入进行检测。

随后，利用输入的木块数量和长度建立最小堆。

建立堆后，不断取堆顶两个元素，令结果加上两个元素的值后，再把两个元素相加存入堆。

不断重复这个过程，直到堆中仅剩一个元素，此时得到的结果即为最终结果。

3.2.2 核心代码

输入检测：

void getNumber(int& temp)

{

string checkIn;

cin >> checkIn;

if (checkIn.size() > 10)

{

temp = -1;

return;

}

for (auto elem : checkIn)

{

if (!(elem >= '0'&&elem <= '9'))

{

temp = -1;

return;

}

}

if (atoi(checkIn.c\_str()) < INT\_MAX)

{

temp = stoi(checkIn);

}

else

{

temp = -1;

}

}

主程序：

int size = -1;

cout << "请输入需要的木头数量" << endl;

getNumber(size);

while (size == -1 || size > 10000)

{

cout << "请输入正确的木头数量，为0-10000的正整数" << endl;

getNumber(size);

}

if (size == 0)

{

cout << "不需要木头，花费0元" << endl;

system("pause");

return 0;

}

int\* arr = new int[size];

cout << "请输入每块木头的长度" << endl;

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

{

int temp = 0;

getNumber(temp);

if (temp <= 0)

{

cout << "输入错误，请输入大于0小于INT\_MAX的整数" << endl;

i--;

continue;

}

arr[i] = temp;

}

MinHeap<int> heap(arr, size);

long long ans = 0;

int a = 0, b = 0;

while (heap.elemNumber() != 1)

{

a = heap.pop();

b = heap.pop();

ans += a + b;

heap.push(a + b);

}

cout << "需要" << size << "块木头，花费" << ans << "元" << endl;

system("pause");

4 测试

4.1 基本测试

**测试用例**：

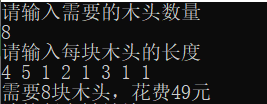
8

4 5 1 2 1 3 1 1

**预期结果**：

49

**实验结果：**



4.2 边界测试

4.2.1 木头数量为0

**测试用例：**

0

**预期结果：**

程序正常运行，不崩溃，给出错误提示。

**实验结果：**



4.2.2 只有一块木头

**测试用例：**

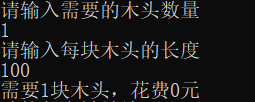
1

100

**预期结果：**

0

**实验结果：**



4.3 错误测试

4.3.1 木头数量错误

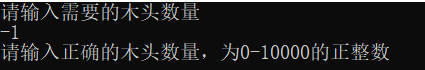
**测试用例：**

-1

**预期结果：**

程序正常运行，不崩溃，给出错误提示，并要求重新输入木头数量。

**实验结果：**



4.3.2 存在长度为0的木块

**测试用例：**

8

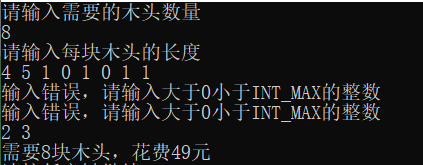
4 5 1 0 1 0 1 1

2 3

**预期结果：**

程序正常运行，不崩溃，给出错误提示，并要求重新输入错误木块长度。

**实验结果：**



4.3.3 存在长度为负数的木块

**测试用例：**

8

1 9 8 12 3 -77 1 -1

7 3

**预期结果：**

程序正常运行，不崩溃，给出错误提示，并要求重新输入错误木块长度。

**实验结果：**

