项目说明文档

数据结构课程设计

——修理牧场

作 者 姓 名： 赵子昱

学 号： 1951459

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 1](#_Toc59806061)

[1.1 项目简介 1](#_Toc59806062)

[1.2 解题分析 1](#_Toc59806063)

[2 设计 2](#_Toc59806064)

[2.1 数据结构设计 2](#_Toc59806065)

[2.2 主程序设计 2](#_Toc59806066)

[2.3 成员与操作设计 2](#_Toc59806067)

[3 实现 3](#_Toc59806068)

[3.1 最小堆的向下调整功能**FliterDown()**的实现 3](#_Toc59806069)

[3.1.1 最小堆的向下调整功能流程图 3](#_Toc59806070)

[3.1.2 最小堆的向下调整功能代码 4](#_Toc59806071)

[3.2 最小堆的向上调整功能**FliterUp()**的实现 5](#_Toc59806072)

[3.2.1 最小堆的向上调整功能流程图 5](#_Toc59806073)

[3.2.2 最小堆的向上调整功能核心代码 6](#_Toc59806074)

[3.3 插入功能**Insert(…)**的实现 6](#_Toc59806075)

[3.3.1 插入功能代码 6](#_Toc59806076)

[3.4 弹出功能**Pop()**的实现 6](#_Toc59806077)

[3.4.1 弹出功能代码 7](#_Toc59806078)

[3.5 主函数的实现 7](#_Toc59806079)

[3.5.1 主函数流程图 7](#_Toc59806080)

[3.5.2 主函数代码 8](#_Toc59806081)

[4 测试 9](#_Toc59806082)

[4.1 测试1 9](#_Toc59806083)

[4.2 测试2 9](#_Toc59806084)

# 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）.

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。

## 1.2 解题分析

分析题目不难发现，得到每块木头需要付出的代价等于这块木头的长度乘这块木头被锯的次数。由于每块木头被锯一次可以得到两块木头，所以可以用二叉树来表示锯木头这一关系，根节点为初始最长的那根木头，每个叶节点为最终需要的木头，每个叶节点木头所需的代价为其长度乘以其深度，也就是其带权路径的长度，要让总花费最小，就是要让叶节点的带权路径长度最短，所以我们可以用霍夫曼树结构来解决这个问题。

# 2 设计

## 2.1 数据结构设计

由于最终我们想要的结果是总费用，所以不必得到完整的霍夫曼树模型，只需的到最小的带权路径长度，所以项目中只需用到最小堆。

## 2.2 主程序设计

首先将输入的N个正整数逐个插入到最小堆中。然后每次从弹出堆顶的两个元素，将他们的值相加，加入总费用中，并将得到的新元素插入堆中，循环直到堆中只有一个元素

直观上来说，就是在一堆木块碎片中，每次都把最小的两块碎片拼成一块，不断的拼，直到最后就只剩下最初的完整的木块了。

## 2.3 成员与操作设计

**最小堆MinHeap 类**

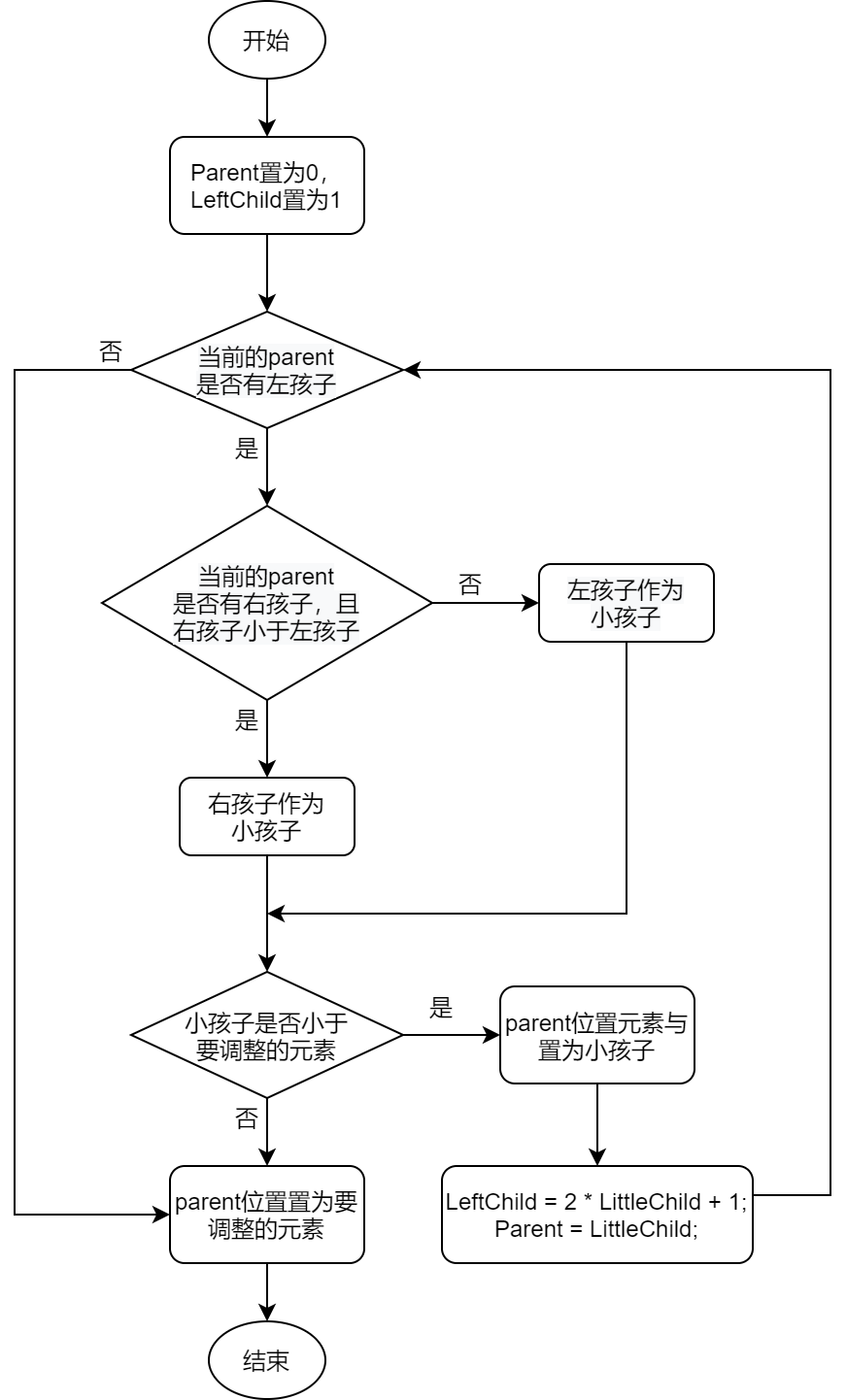
1. **template** <**class** T>
2. **class** MinHeap {
3. **public**:
4. MinHeap(**int** size) :MaxSize(size), CurrentSize(0) { Heap = **new** T[size]; }
5. ~MinHeap() { **delete**[]Heap; }
7. **void** FliterDown();
8. //向下调整
10. **void** FliterUp();
11. //向上调整
13. **void** Insert(T &x);
14. //插入新元素
16. T Pop();
17. //弹出堆顶元素
19. **private**:
20. T \*Heap;
21. //堆的元素
23. **int** MaxSize, CurrentSize;
24. //堆的最大容量，当前容量
26. };

# 3 实现

## 3.1 最小堆的向下调整功能**FliterDown()**的实现

* 从堆顶开始将堆调整为最小堆
* 被**Pop()**函数调用

### 3.1.1 最小堆的向下调整功能流程图



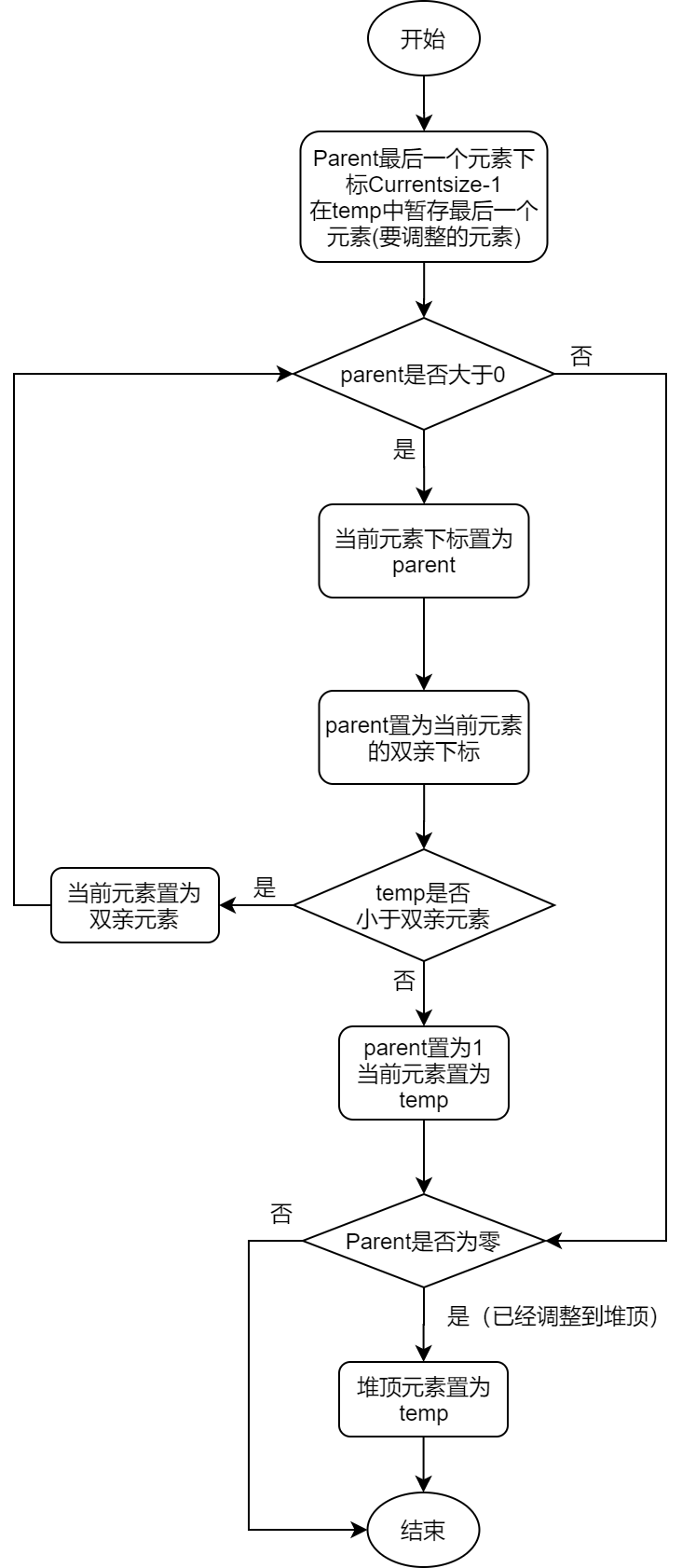
### 3.1.2 最小堆的向下调整功能代码

1. **template** <**class** T>
2. **void** MinHeap<T>::FliterDown() {
3. assert(CurrentSize > 0);
4. T temp = Heap[0];
5. **int** Parent = 0, LeftChild = 1, LittleChild;
6. **while** (LeftChild <= CurrentSize - 1) {
7. //选出较小的孩子
8. **if** (LeftChild + 1 <= CurrentSize - 1 && Heap[LeftChild] > Heap[LeftChild + 1]) {
9. LittleChild = LeftChild + 1;
10. }
11. **else** { LittleChild = LeftChild; }
12. //如果较小的孩子小于要调整的元素则调换位置
13. **if** (Heap[LittleChild] < temp) { Heap[Parent] = Heap[LittleChild]; }
14. **else** { **break**; }
15. LeftChild = 2 \* LittleChild + 1;
16. Parent = LittleChild;
17. }
18. Heap[Parent] = temp;
19. }

## 3.2 最小堆的向上调整功能**FliterUp()**的实现

* 从最后一个元素开始将堆调整为最小堆
* 被**Insert()**函数调用

### 3.2.1 最小堆的向上调整功能流程图



### 3.2.2 最小堆的向上调整功能核心代码

1. **template** <**class** T>
2. **void** MinHeap<T>::FliterUp() {
3. assert(CurrentSize > 0);
4. **int** Parent = CurrentSize - 1, CurrentLoc;
5. T temp = Heap[Parent];  //temp为要调整的元素（最后一个元素）
6. **while** (Parent > 0) {
7. CurrentLoc = Parent;
8. Parent = (CurrentLoc - 1) / 2;
9. **if** (temp < Heap[Parent]) {
10. Heap[CurrentLoc] = Heap[Parent];
11. }
12. **else** {
13. Parent = 1;
14. Heap[CurrentLoc] = temp;
15. **break**;
16. }
17. }
18. **if** (Parent == 0) { Heap[0] = temp; }
19. }

## 3.3 插入功能**Insert(…)**的实现

* x为要插入的元素
* 将新元素插在最后，再向上调整
* 被主函数调用

### 3.3.1 插入功能代码

1. **template** <**class** T>
2. **void** MinHeap<T>::Insert(T &x) {
3. assert(CurrentSize < MaxSize);
4. Heap[CurrentSize++] = x;
5. FliterUp();
6. }

## 3.4 弹出功能**Pop()**的实现

* 弹出堆顶元素，并将堆重新调整为最小堆
* 将最后一个元素移到堆顶，再向下调整
* 被主函数调用

### 3.4.1 弹出功能代码

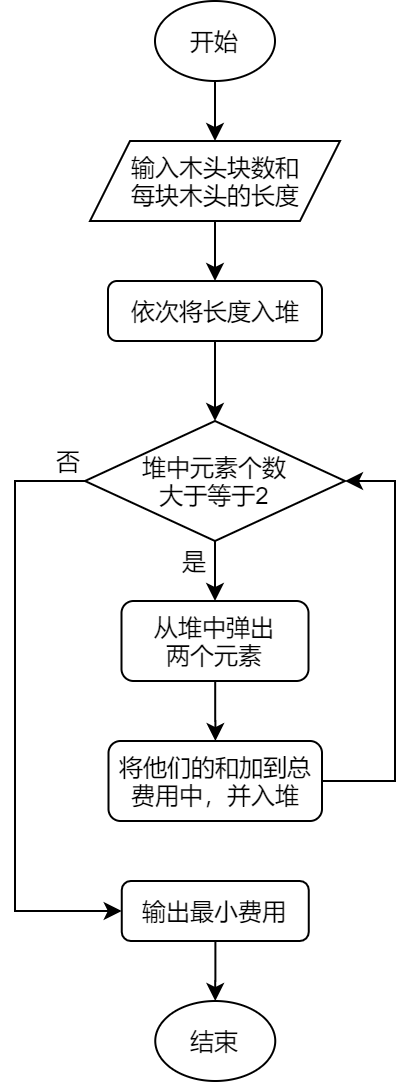
1. **template** <**class** T>
2. T MinHeap<T>::Pop() {
3. assert(CurrentSize > 0);
4. T temp = Heap[0];
5. Heap[0] = Heap[--CurrentSize];
6. **if** (CurrentSize != 0) { FliterDown(); }
7. **return** temp;
8. }

## 3.5 主函数的实现

首先将输入的N个正整数逐个插入到最小堆中。然后每次从弹出堆顶的两个元素，将他们的值相加，加入总费用中，并将得到的新元素插入堆中，循环直到堆中只有一个元素

直观上来说，就是在一堆木块碎片中，每次都把最小的两块碎片拼成一块，不断的拼，直到最后就只剩下最初的完整的木块了。

### 3.5.1 主函数流程图



### 3.5.2 主函数代码

1. **int** main() {
2. **int** n, Length;
3. cout << "请输入木头的块数：";
4. cin >> n;
5. assert(n > 0);
6. MinHeap<**int**> Lengthes(n);
7. cout << "请输入木头的长度：";
8. **for** (**int** i = 0; i < n; i++) {
9. cin >> Length;
10. Lengthes.Insert(Length);
11. }
12. **int** Cost = 0,currentCost;
13. **for** (**int** i = 1; i < n; i++) {
14. currentCost = Lengthes.Pop() + Lengthes.Pop();
15. Lengthes.Insert(currentCost);
16. Cost += currentCost;
17. }
18. cout << "最小费用为：" << Cost << "元" << endl;
19. system("pause");
20. **return** 0;
21. }

# 4 测试

## 4.1 测试1

**测试用例**：

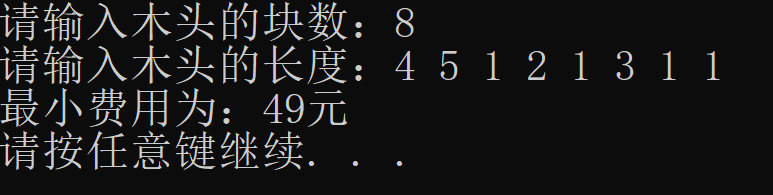
8

4 5 1 2 1 3 1 1

**预期结果**：

49

**实验结果**

****

## 4.2 测试2

**测试用例：**

9

1 3 2 6 5 9 3 4 2

**预期结果：**

104

**实验结果：**

