项目说明文档

数据结构课程设计

——修理牧场

作 者 姓 名： 陈翔飞

学 号： 1851756

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](#_Toc27608523)

[1.1 背景分析 3](#_Toc27608524)

[1.2 功能分析 3](#_Toc27608525)

[2 设计 3](#_Toc27608526)

[2.1 算法设计 3](#_Toc27608527)

[2.1 数据结构设计 3](#_Toc27608528)

[2.2 类结构设计 3](#_Toc27608529)

[2.3 成员与操作设计 4](#_Toc27608530)

[3 实现 7](#_Toc27608531)

[3.1算法流程图 7](#_Toc27608532)

[3.1.2 算法核心代码 7](#_Toc27608533)

[4 测试 9](#_Toc27608534)

[4.1 基本测试 9](#_Toc27608535)

[4.2 边界测试 10](#_Toc27608536)

[4.2.1 木头数量为0 10](#_Toc27608537)

[4.2.1 只有一块木头 10](#_Toc27608538)

[4.3 错误测试 11](#_Toc27608539)

[4.3.1 木头数量错误 11](#_Toc27608540)

[4.3.1 存在长度为0的木块 11](#_Toc27608541)

[4.3.1 存在长度为负数的木块 12](#_Toc27608542)

# 1 分析

## 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。

## 2.1 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。如果要求堆的大小灵活可变，那么还需要动态数组，所以本项目还设计了数据结构Vector。

## 2.2 类结构设计

为了保证设计的数据结构的泛用性，本项目选择将Heap类和Vector类都设计为模板类，还设计了Greater和Less两个比较器类，使得Heap类可支持最小堆和最大堆。

## 2.3 成员与操作设计

**向量类（Vector）**

**类定义：**

1. **template**<**typename** ElementType> **class** Vector
2. {
3. **public**:
4. ~Vector<ElementType>();
5. Vector<ElementType>() = **default**;
6. Vector<ElementType>(**const** Vector<ElementType> & v);
7. Vector<ElementType>& operator = (**const** Vector<ElementType>&v);
8. **void** PushBack(**const** ElementType& t);
9. **void** PopBack();
10. **void** Clear();
11. **int** GetSize() **const**;
12. **void** ReSize(**int** NewSize);
13. ElementType& operator[](**int** Index) **const**;
14. **bool** Empty()**const**;
15. **const** ElementType& Back() **const**;
16. **private**:
17. **void** Extend();
18. **int** Size = 0;
19. **int** Capacity = 0;
20. ElementType\* Array = nullptr;
21. };

**私有成员：**

int Size;//Vector中实际储存的元素数量

int Capacity;//Vector已经申请的空间

ElementType\* Array;//储存的数据的起始地址

**私有操作：**

void Extend();

//扩容函数，当容量不足时调用

**公有操作：**

~Vector<ElementType>();

//析构函数，通过调用Clear()来释放内存

Vector<ElementType>() = default;

//默认构造函数

Vector<ElementType>(const Vector<ElementType> & v);

//拷贝构造函数

Vector<ElementType>& operator = (const Vector<ElementType>&v);

//重载=运算符，使该类支持赋值运算

void PushBack(const ElementType& t);

//向Vector末尾添加一个元素

void PopBack();

//删除末尾的元素

void Clear();

//清空Vector，释放内存

int GetSize() const;

//返回储存元素的数量

void ReSize(int NewSize);

//重设Vector的大小

ElementType& operator[](int Index) const;

//重载[]运算符，使Vector可以像数组一样使用

bool Empty()const;

//判断Vector是否为空

const ElementType& Back() const;

//返回末尾的元素

**比较器类（Greater/Less）**

**类定义：**

1. **template**<**typename** ElementType> **class** Greater
2. {
3. **public**:
4. **bool** operator ()(**const** ElementType& E1, **const** ElementType& E2)
5. {
6. **return**  E2 < E1;
7. }
8. };
9. **template**<**typename** ElementType> **class** Less
10. {
11. **public**:
12. **bool** operator ()(**const** ElementType& E1, **const** ElementType& E2)
13. {
14. **return** E1 < E2;
15. }
16. };

**堆类（Heap）**

**类定义：**

1. **template**<**typename** ElementType, **typename** Comparator=Greater<ElementType>> **class** Heap
2. {
3. **public**:
4. ~Heap<ElementType, Comparator>() = **default**;
5. Heap<ElementType, Comparator>() ;
6. Heap<ElementType, Comparator>(**int** Size);
7. **int** GetSize() { **return** Elements.GetSize() - 1; };
8. **bool** Empty() { **return** Elements.GetSize() <= 1; };
9. **void** Push(**const** ElementType& E);
10. **void** Build(**int** Size);
11. **void** Pop();
12. **void** Clear();
13. ElementType Top();
14. **private**:
15. Vector<ElementType> Elements;
16. };

**私有成员：**

Vector<ElementType> Elements;//向量类存储堆中实际数据

**公有操作：**

~Heap<ElementType, Comparator>() = default;

//析构函数

Heap<ElementType, Comparator>() ;

//默认构造函数

Heap<ElementType, Comparator>(int Size);

//带大小参数的初始化函数

int GetSize() { return Elements.GetSize() - 1; };

//返回堆大小

bool Empty() { return Elements.GetSize() <= 1; };

//判断堆是否为空

void Push(const ElementType& E);

//向堆中推入数据

void Build(int Size);

//重设堆的大小

void Pop();

//删除堆顶元素

void Clear();

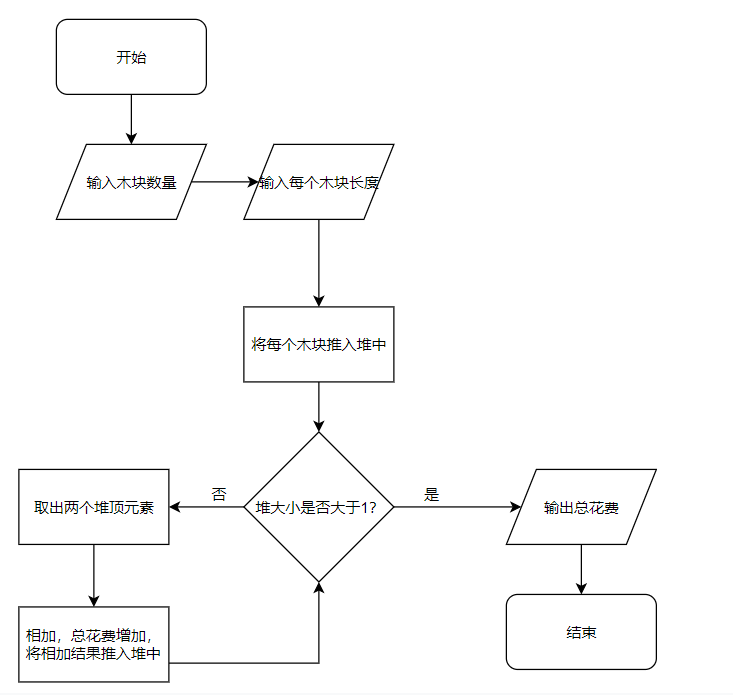
//清空堆

ElementType Top();

//返回堆顶元素

# 3 实现

### 3.1算法流程图



### 3.1.2 算法核心代码

**主程序：**

1. **int** main(**void**)
2. {
3. Heap<**int**> H;
4. Vector<**int**> Input;
5. cout << "请输入木块数量:" << endl;
6. **int** i, Num;
7. cin >> Num;
8. **while** (Num <= 0)
9. {
10. cout << "请输入一个正整数！" << endl;
11. cout << "请重新输入：" << endl;
12. cin >> Num;
13. }
14. **int** Temp, Ans=0;
15. cout << "请输入每块木块的长度：" << endl;
16. **for** (i = 0; i < Num; ++i)
17. {
18. cin >> Temp;
19. Input.PushBack(Temp);
20. }
21. **while** (Check(Input)==**false**)
22. {
23. cout << "木块长度中不应该0或负数！" << endl;
24. cout << "请重新输入：" << endl;
25. Input.Clear();
26. **for** (i = 0; i < Num; ++i)
27. {
28. cin >> Temp;
29. Input.PushBack(Temp);
30. }
31. }
32. **for** (i = 0; i < Num; ++i)
33. {
34. H.Push(Input[i]);
35. }
36. **while** (H.GetSize() > 1)
37. {
38. **int** First = H.Top(); H.Pop();//取出第一个最小值作为左子树根节点
39. **int** Second = H.Top(); H.Pop();//取出第二个最小值作为右子树根节点
40. **int** NewCost = First + Second;//左右子树根节点的和作为根节点
41. Ans += NewCost;//费用加上这个根节点的值
42. H.Push(NewCost);//将新节点推入堆中
43. }
44. cout << "最小花费为:" << Ans << endl;
45. **while** (getchar() != '\n') **continue**;//消除行末换行符
46. cout << "请按任意键退出！" << endl;
47. getchar();
48. **return** 0;
49. }

**堆Push功能内部实现：**

1. **template**<**typename** ElementType, **typename** Comparator>
2. **void** Heap<ElementType, Comparator>::Push(**const** ElementType & E)
3. {
4. **int** i;
5. Elements.PushBack(E);
6. Comparator Cmp;
7. **for** (i=Elements.GetSize()-1; i > 1; i /= 2)
8. {
9. **if**(Cmp(Elements[i/2],E)) Elements[i]=Elements[i/2];
10. **else** **break**;
11. }
12. Elements[i]=E;
13. }

**堆Pop功能内部实现：**

1. **template**<**typename** ElementType, **typename** Comparator>
2. **void** Heap<ElementType, Comparator>::Pop()
3. {
4. **if**(Empty()) **return**;
5. **int** i, Child, CurSize = Elements.GetSize();
6. Comparator Cmp;
7. ElementType  LastElement;
8. LastElement = Elements.Back();
9. **for** (i = 1; i \* 2 < CurSize; i = Child)
10. {
11. Child = i \* 2;
12. **if** (Child != CurSize - 1 && Cmp(Elements[Child], Elements[Child + 1]))
13. ++Child;
14. **if** (Cmp(LastElement, Elements[Child])) Elements[i] = Elements[Child];
15. **else** **break**;
16. }
17. Elements[i] = LastElement;
18. Elements.PopBack();
19. }

# 4 测试

## 4.1 基本测试

**测试用例**：

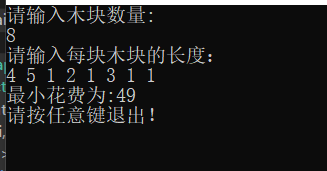
8

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

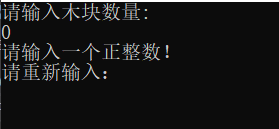
**测试用例：**

0

**预期结果：**

程序正常运行，不崩溃，给出错误提示。

**实验结果：**



### 4.2.1 只有一块木头

**测试用例：**

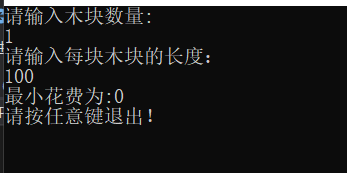
1

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

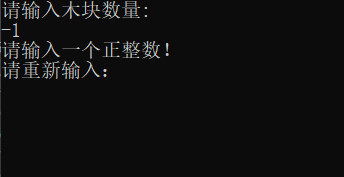
**测试用例：**

-1

**预期结果：**

程序正常运行，不崩溃，给出错误提示。

**实验结果：**



### 4.3.1 存在长度为0的木块

**测试用例：**

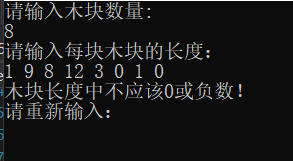
8

1 9 8 12 3 0 1 0

**预期结果：**

程序正常运行，不崩溃，给出错误提示。

**实验结果：**



### 4.3.1 存在长度为负数的木块

**测试用例：**

8

1 9 8 12 3 -77 1 -1

**预期结果：**

程序正常运行，不崩溃，给出错误提示。

**实验结果：**

