数据结构课程设计

项目说明文档

修理牧场

作者姓名: 汪明杰

学　　号: 1851055

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 项目分析

1.1 项目背景

哈夫曼（Huffman）树又称最优二叉树，是指对于一组带有确定权值的叶子结点所构造的具有带权路径长度最短的二叉树。哈夫曼编码是哈夫曼树的一个应用。在数字通信中，经常需要将传送的文字转换成由二进制字符 0 和 1 组成的二进制串，这一过程被称为编码。在传送电文时，总是希望电文代码尽可能短，采用哈夫曼编码构造的电文的总长最短。

哈夫曼树在解决许多现实问题中和最小相关的问题中都有着重要的应用，本项目实际上也是需要通过构造一棵哈夫曼树来解决问题。

1.2 项目需求分析

针对于修理牧场这一问题，本项目在实现的过程中，考虑并且满足了以下的需求：

- ✓ **功能完善**

  系统求解出的答案应该正确无误。

- ✓ **执行效率高**

  针对数据量比较大的情况，本系统也应该具有在较短时间内求解出正确答案的能力。

- ✓ **代码可读性强**

  本项目在实现过程中，将代码根据功能的不同划分为了不同的代码块，同时进行了合理封装。

1.3 项目要求

1.3.1 功能要求

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L_i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L_i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）。

1.3.2 输入格式

输入第一行给出正整数 N（N<<10^4），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。

1.3.3 输出格式

输出一个整数，即将木头锯成 N 块的最小花费。

1.3.4 项目示例

2 项目设计

2.1 数据结构设计

本项目实际上即需要构造一棵**哈夫曼树**（Huffman Tree）。哈夫曼树每次需要挑选两个最小的节点以组成新节点，因此本项目中定义了**优先级队列**（Priority Queue）。每次从优先级队列中出队两个元素，相加后入队，即完成了哈夫曼树构造的过程。

优先级队列中，项目使用**堆**（Heap）算法来对底层的**向量**（Vector）进行处理。各操作的时间复杂度如下所示：

- ✓ 入队操作：`O(log n)`
- ✓ 出队操作：`O(log n)`

2.2 类设计

2.2.1 向量类（Vector）

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence Container）。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，向量是一个能够存放任意类型的动态数组。需要注意到的是，向量具有以下两个特性：

（1） 顺序序列：

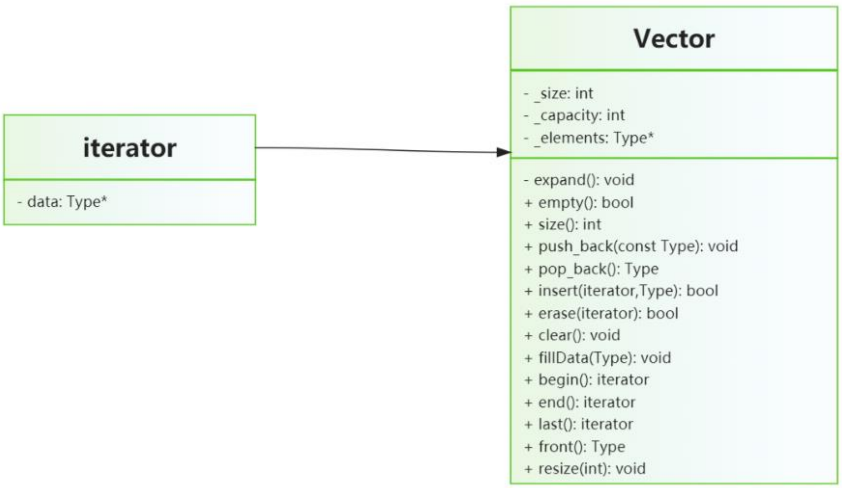
顺序容器中的元素按照严格的线性顺序排序，可以通过元素在序列中的位置访问对应的元素。本类中即通过重载取下标运算符[]实现了此功能。

（2） 动态数组：

支持对序列中的任意元素进行快速直接访问，甚至可以通过指针算术进行该操作，提供了在序列末尾相对快速地添加或者删除元素的操作。

本项目为了实现向量类，在内部定义了一个动态指针以及当前数组的大小。同时，封装了较多的函数以便使用。此外，为了便于对向量进行遍历、插入、删除、查找等操作，增加了一个 iterator 类。iterator 类内部存储一个数组元素指针，可以直接对向量的每个元素进行操作。同时，通过运算符重载相应的自增、自减、判等等操作。

该类和其内部的 iterator 类的 UML 图如下所示：



本类中的主要函数如下所示：

◆ `inline int size()const`

返回向量中元素的个数，也即_size 的大小。

◆ `inline bool empty()const`

判断向量是否为空，也即_size 是否为 0。

◆ `void push_back(const Type data)`

在向量尾端加入一个元素。

◆ `Type pop_back()`

删除向量最后一个元素，并且加以返回。

◆ `bool insert(const Vector<Type>::iterator place, Type item)`

在指定迭代器的位置插入元素，返回是否插入成功。

◆ `bool erase(const Vector<Type>::iterator place)`

删除指定迭代器位置的元素，返回是否删除成功。

◆ `void clear()`

清空向量中所有元素。

◆ `void fillData(const Type data)`

将向量中的元素统一赋值为同一个值。

◆ `iterator begin()`

返回向量首元素的迭代器。

◆ `iterator end()`

返回向量尾元素的下一个位置的迭代器。

◆ `iterator last()`

返回向量尾元素的迭代器。

◆ `Type& front()const`

返回向量的首个元素值。

◆ `void resize(int sz)`

将向量重新设定大小，如果变小则删除多余元素。

2.2.2 堆（Heap）

堆（Heap）是一类特殊数据结构的简称，通常是一个可以被看做一棵树的的数组对象。堆满足下述两个性质：

（1） 堆中某个节点的值总是不大于或者不小于其父节点的值；

（2） 堆总是一棵完全二叉树。

将根节点最大的堆叫做最大堆或者大根堆，根节点最小的堆叫做最小堆小根堆。常见的堆有二叉堆、斐波拉契堆等。

本项目中堆的底层数据结构使用向量来实现，通过定义建堆、调整堆来完成了一个堆。主要函数如下所示：

◆ 建堆函数

将[start,end)内的元素调整为堆

```
template <class Iter>
void makeHeap(Iter start, Iter end)
{
    typedef typename Iter::value_type value_type;
    ptrdiff_t len = end - start;
    if (len < 2) return;
    ptrdiff_t hole = (len - 2) / 2;
    while (hole >= 0) {
        //调整 hole 节点
        sinkFix(start, len, hole, *(start + hole), Less<value_type>());
        --hole;
    }
}
```

◆ 向上调整

向上调整 hole 的位置，找到合适的位置将 value 存入

```
template <class Iterator, typename T, class Compare>
void siftFix(Iterator first,
    ptrdiff_t top,
    ptrdiff_t hole,
    T value,
    Compare comp)
{
    ptrdiff_t parent = (hole - 1) / 2;
    while (parent >= top && comp(*(first + parent), value)) {
        *(first + hole) = *(first + parent);
        hole = parent;
        parent = (hole + 1) / 2 - 1;
```

```
    }
    *(first + hole) = value;
}
```

◆ 向下调整

向下调整 hole 的位置，找到合适的位置将 value 存入

```
template <class Iterator, typename T, class Compare>
void sinkFix(Iterator first,
    ptrdiff_t len,
    ptrdiff_t hole,
    T value,
    Compare comp)
{
    /* 1.将hole 下移到叶子处 */
    ptrdiff_t bigChild = 2 * hole + 2;  //从右孩子开始考虑
    while (bigChild < len) {
        if (comp(*(first + bigChild), *(first + bigChild - 1)))
            bigChild--;  //左孩子大于右孩子
        *(first + hole) = *(first + bigChild);
        hole = bigChild;
        bigChild = 2 * hole + 2;
    }
    if (len % 2 == 0 &&
        bigChild == len) {  //特殊情况：最后一个节点为左节点，没有兄弟节点
        *(first + hole) = *(first + bigChild - 1);
        hole = bigChild - 1;
    }
    /* 2.转化为向上调整问题 */
    siftFix(first, 0, hole, value, comp);
}
```

◆ 入堆

将*(end-1)调整到合适的位置，使得[start,end)变成堆

```
template <class Iter>
void pushHeap(Iter start, Iter end)
{
    typedef typename Iter::value_type value_type;
    siftFix(start, 0, end - start - 1, *(end - 1), Less<value_type>());
}
```

◆ 出堆

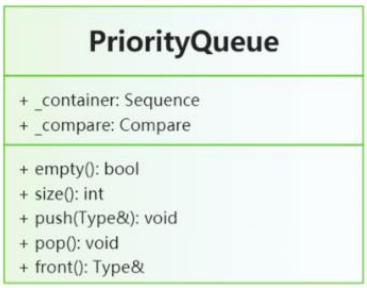
将*(start)交换到*(end-1)处(以便删去), [start,end-1)变成堆

```
template <class Iter>
void popHeap(Iter start, Iter end)
{
    typedef typename Iter::value_type value_type;
    value_type value = *(end - 1);  //将原来的末尾值保存起来
    *(end - 1) = *(start);  //将*(start)移动到*(end-1),*(start)处出现
hole
    sinkFix(start, end - start - 1, 0, value, Less<value_type>());
}
```

2.2.3 优先级队列（Priority Queue）

普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。在优先级队列（Priority Queue）中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先级队列具有最高级先出（First In, Largest Out）的行为特征。

本项目中的优先级队列类为外部提供了底层数据结构，同时默认使用向量作为数据结构，UML 图如下所示：



优先级队列中主要操作函数如下所示：

◆ `inline bool empty()const`

判断优先级队列是否为空，也即内部存储结构是否为空。

◆ `inline int size()const`

返回优先级队列中元素的个数。

◆ `void push(const Type& i)`

在优先级队列加入一个数据。

◆ `void pop()`

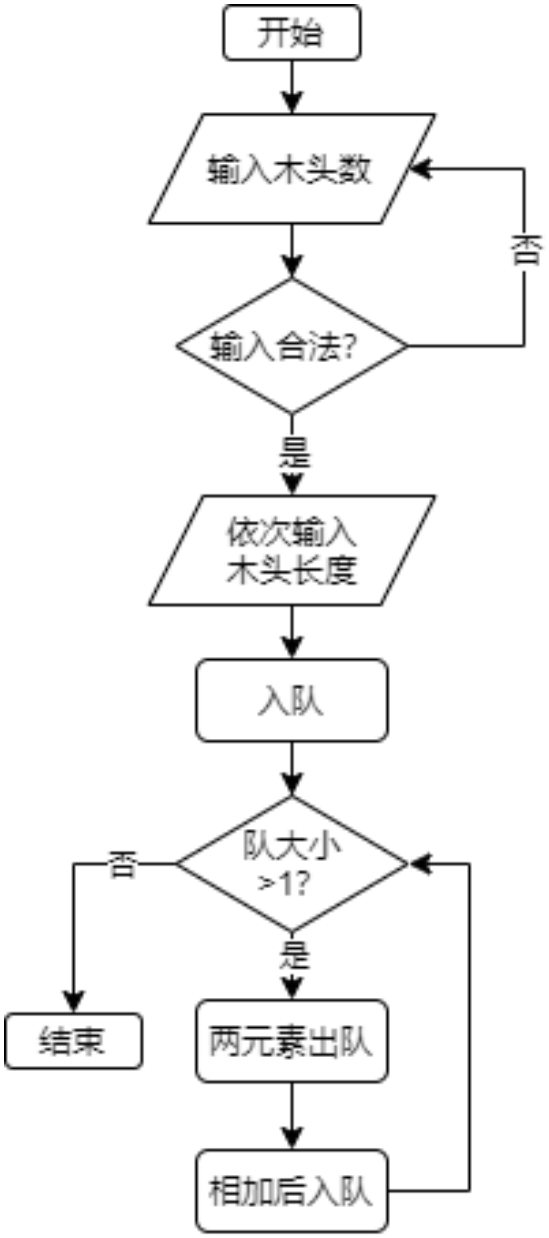
删除优先级队列中优先级最高的元素。

◆ `const Type& front()const`

获取优先级队列中优先级最高的元素。

3 项目实现

3.1 项目整体功能流程图



3.2 项目整体功能代码

```
int main()
{
    int N, temp;
    cin >> N;
    //优先级队列，也即最小堆
    PriorityQueue<int,Greater<int>> woodLength;
    for (int i = 0; i < N; ++i)
    {
        cin >> temp;
        woodLength.push(temp);
    }
    int sumCost = woodLength.size() == 1 ? woodLength.top() : 0;
    while (woodLength.size() > 1)
    {
        int length1 = woodLength.top();
        woodLength.pop();
        int length2 = woodLength.top();
        woodLength.pop();
        sumCost += length1 + length2;
        woodLength.push(length1 + length2);
    }
    cout << sumCost;
    return 0;
}
```

4 项目测试

4.1 样例测试

测试样例：8 4 5 1 2 1 3 1 1

预期结果：49

测试结果：

4.2 木头长度各不相同的情况测试

测试样例：10 1 2 3 4 5 6 7 8 9

预期结果：173

测试结果：

4.3 木头长度全部相同的情况测试

测试样例：5 1 1 1 1 1

预期结果：12

测试结果：

4.4 边界情况

4.4.1 只有两个输入情况测试

测试样例：2 5 6

预期结果：11

实验结果：

4.4.2 只有一个输入情况测试

测试样例：1 10

预期结果：10

实验结果：