项目说明文档

离散数学课程项目

——最优二元树的应用

作 者 姓 名： 陈奕名

学 号： 2351883

指 导 教 师： 李冰

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

# 1 项目分析

## 1.1 项目要求

输入一组通信符号的使用频率，求各通信符号对应的前缀码。

# 2 项目设计

## 2.1 数据结构设计

1. TreeNode 结构体

TreeNode 结构体用于表示赫夫曼树的节点。每个节点包含一个频率值、指向左子树和右子树的指针。

int num：该节点的频率（或权重），用来进行赫夫曼树的合并与排序。

shared\_ptr<TreeNode> Lnode：指向左子节点的智能指针，表示当前节点的左子树。

shared\_ptr<TreeNode> Rnode：指向右子节点的智能指针，表示当前节点的右子树。

2.**vector<shared\_ptr<TreeNode>> fp**：

存储赫夫曼树的所有节点，包含叶子节点和最终的根节点。

使用 shared\_ptr 管理内存，避免内存泄漏。

3.**char s[2 \* 13]**：

存储前缀编码（赫夫曼编码），可以容纳最多 13 个节点的编码。

## 2.2 算法设计

1. 初始化节点：init\_node

目标：根据输入的频率列表，创建对应的叶子节点，并将其存储在 fp 向量中。

输入：频率列表 f。

输出：初始化 fp 向量，其中每个元素是 shared\_ptr<TreeNode> 类型的叶子节点。

设计思路：

遍历频率数组，创建每个叶子节点。

将每个节点的频率（f[i]）赋值给 TreeNode，并用 shared\_ptr<TreeNode> 管理该节点。

2. 排序函数：sort

目标：对当前节点列表按照频率排序，以便赫夫曼算法在每次合并节点时可以选择频率最小的两个节点。

输入：fp 向量，存储当前的所有节点（包括叶子节点和内部节点）。

输出：排序后的 fp 向量，按频率升序排列。

设计思路：

使用 std::sort 或手动排序（按需求）将频率最小的节点放在前面。

3. 构建赫夫曼树：construct\_tree

目标：通过赫夫曼算法构建最优树，将两个频率最小的节点合并成一个新的父节点，直到所有节点合并为一棵树。

输入：频率数组 f。

输出：赫夫曼树的根节点，类型为 shared\_ptr<TreeNode>。

设计思路：

每次合并最小的两个节点，形成一个新的节点。

新节点的频率为合并节点的频率之和，合并后的节点成为新的父节点。

使用 sort 函数重新排序，确保下次合并时是频率最小的两个节点。

# 3 代码实现

#include <iostream>

#include <vector>

#include <memory> // 用于智能指针

using namespace std;

// 定义树节点结构

struct TreeNode {

int num; // 节点的频率

shared\_ptr<TreeNode> Lnode; // 左子树（智能指针）

shared\_ptr<TreeNode> Rnode; // 右子树（智能指针）

// 构造函数初始化节点

TreeNode(int val) : num(val), Lnode(nullptr), Rnode(nullptr) {}

};

// 全局变量

vector<shared\_ptr<TreeNode>> fp; // 保存树节点（使用智能指针）

char s[2 \* 13]; // 存储前缀码（数组长度可以根据实际需要调整）

// 初始化树节点

void init\_node(const vector<int>& f) {

for (int i = 0; i < f.size(); ++i) {

fp.push\_back(make\_shared<TreeNode>(f[i])); // 创建叶子节点

}

}

// 排序函数：将第 N-n 个点插入到已排序的序列中

void sort(vector<shared\_ptr<TreeNode>>& array) {

for (size\_t i = 0; i < array.size() - 1; ++i) {

for (size\_t j = i + 1; j < array.size(); ++j) {

if (array[i]->num > array[j]->num) {

swap(array[i], array[j]);

}

}

}

}

// 构建赫夫曼树

shared\_ptr<TreeNode> construct\_tree(const vector<int>& f) {

int n = f.size();

vector<shared\_ptr<TreeNode>> nodes = fp;

for (int i = 1; i < n; ++i) {

// 创建一个新的父节点

shared\_ptr<TreeNode> pt = make\_shared<TreeNode>(nodes[i - 1]->num + nodes[i]->num);

pt->Lnode = nodes[i - 1]; // 左子树

pt->Rnode = nodes[i]; // 右子树

// 替换掉原来的节点

nodes[i] = pt;

sort(nodes); // 重新排序

}

return nodes[n - 1]; // 返回赫夫曼树的根节点

}

// 前序遍历赫夫曼树并生成编码

void preorder(shared\_ptr<TreeNode> p, int k, char c) {

if (p != nullptr) {

// 生成前缀码

s[k] = (c == '1') ? '0' : '1';

// 如果是叶子节点，输出其频率和编码

if (p->Lnode == nullptr && p->Rnode == nullptr) {

cout << p->num << ": ";

for (int j = 0; j <= k; ++j) {

cout << s[j];

}

cout << endl;

}

// 递归遍历左右子树

preorder(p->Lnode, k + 1, '1'); // Lnode 是 shared\_ptr<TreeNode>

preorder(p->Rnode, k + 1, 'r'); // Rnode 是 shared\_ptr<TreeNode>

}

}

int main() {

int n;

cout << "请输入节点个数(必须是正整数): ";

cin >> n;

vector<int> f(n);

cout << "请输入节点的频率(以空格分隔): ";

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

cin >> f[i];

}

// 初始化树节点并构建赫夫曼树

init\_node(f);

shared\_ptr<TreeNode> head = construct\_tree(f);

// 遍历赫夫曼树并输出前缀码

preorder(head, 0, '1');

return 0;

}

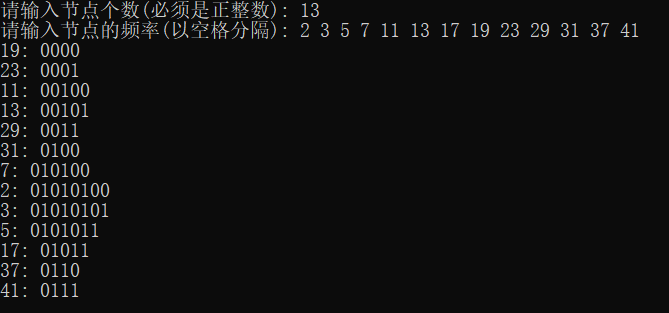
# 4 测试

**测试用例**：

13

2 3 5 7 11 13 17 19 23 39 31 37 41

**实验结果:**



# 5 心得体会

通过实现赫夫曼编码算法，我对数据结构和编码原理有了更深入的理解。在实现过程中，我不仅学习了如何构建和操作树形数据结构，还体会到了赫夫曼树在数据压缩和优化方面的应用。

首先，使用二叉树来表示赫夫曼树的结构让我更好地理解了树的递归性质和左右子树的关系。每次合并两个最小频率节点时，形成的新的父节点总是频率最小的，这个过程让我更加明白了贪心算法的思想。通过每次选择最小的节点进行合并，我们最终能够得到一个最优的编码方案。

其次，代码中使用了 shared\_ptr 智能指针来管理内存，避免了手动管理内存可能带来的错误，这让我意识到现代 C++ 的内存管理优势和必要性。在使用智能指针的过程中，我学习到了如何通过智能指针自动管理资源，避免了内存泄漏和野指针问题。

在实现过程中，我也锻炼了对复杂问题的抽象能力。在赫夫曼编码的实现中，我需要将频率、树的结构以及前缀码结合起来，这需要不断思考和调试代码。在调试过程中，能够准确输出每个节点的赫夫曼编码，解决了编码正确性的问题，使我对算法的实现充满信心。

总的来说，这个实验不仅让我加深了对赫夫曼算法的理解，还提高了我的编程技巧，特别是在数据结构和内存管理方面。通过这个实践，我体会到了算法设计和优化的乐趣，也更加意识到在解决实际问题时，合理选择数据结构和算法的重要性

通过这次实践，我不仅增强了对图论和算法的理论知识，还提高了自己处理复杂数据结构和算法的能力。编写代码的过程中，我更加注重了算法的时间复杂度和空间复杂度，力求在保证正确性的前提下，优化代码的执行效率。

总的来说，这次实现不仅锻炼了我的编程技巧，尤其是数据结构和算法的实现能力，也让我在解决实际问题时更加注重算法的选择和优化，进一步提升了我的问题分析和解决能力。