

《离散数学课程项目文档》

——最优二元树的应用

作者姓名: _____ 胡峻玮

学 号: _____ 2153393

指导教师: _____ 唐剑锋

学院、专业: _____ 软件学院 软件工程



目 录

1 项目分析.....	1
1.1 项目背景.....	1
1.2 项目要求.....	1
2 项目设计.....	2
2.1 数据结构设计.....	2
2.2 算法设计.....	2
2.3 性能评估.....	3
2.4 项目主体部分代码实现.....	3
3 项目测试.....	7
3.1 项目示例.....	7
3.2 仅有一个结点的情况.....	7
4 心得体会.....	8

1. 项目分析

1.1 项目背景

在通信领域，编码选择至关重要。发送方必须思考如何按照特定规则有效转换所需传递的信息。一个优秀的策略是根据字符频率将字符编码为二进制串，并确保没有任何一个二进制串是另一个的前缀(以消除歧义)。此外，调节二进制串的长度可以在保持信息完整性的同时减小期望的编码长度。这个方法就是著名的霍夫曼编码。

1.2 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。

项目示例：

```
输入节点个数:13
输入节点:2 3 5 7 11 13 17 19 23 29 31 37 41
19: 0000
23: 0001
11: 00100
13: 00101
29: 0011
31: 0100
7: 010100
2: 01010100
3: 01010101
5: 0101011
17: 01011
37: 0110
41: 0111
请按任意键继续...
```

2. 项目设计

2.1 数据结构设计

`TreeNode` 结构体：作为基本的数据结构单元，表示树的节点。每个节点包含一个整数 `num` 和指向其左右子节点的智能指针。

`std::shared_ptr<TreeNode>`：用于管理 `TreeNode` 结构体的指针，自动处理内存释放，防止内存泄漏。

`std::vector<std::shared_ptr<TreeNode>>`：用于存储指向树节点的智能指针，方便动态地管理树的各个节点。

`std::vector<char>`：用于存储前序遍历生成的前缀编码。

2.2 算法设计

2.2.1 初始化节点(`init_node`)：

- 创建叶子节点并将其存储在一个数组中。
- 使用输入的整数数组 `f`，为每个数值创建一个单独的 `TreeNode` 对象。
- 利用智能指针自动管理节点内存，防止内存泄漏。

2.2.2 排序(`sort`)：

- 对树节点数组进行排序，确保每次构建新的树节点时，能够选取最小的两个节点。
- 采用简单的插入排序算法，适用于部分已经排序的小数组。

2.2.3 构建树(`construct_tree`)：

- 使用贪心算法，从初始化的叶子节点开始，逐步构建霍夫曼树。

- 在每一步中，创建一个新的非叶子节点，其数值为两个最小叶子节点数值之和，然后更新树节点数组，并重新排序。

- 重复这个过程直到只剩下一个节点，它将成为霍夫曼树的根节点。

2.2.4 前序遍历 (preorder) :

- 使用递归方法遍历霍夫曼树，并在途中构建前缀编码。
- 当访问到叶子节点时，输出节点数值及其对应的前缀编码。

2.3 性能评估

2.3.1 时间复杂度:

- 初始化节点的时间复杂度为 $O(n)$ ，其中 n 为节点的数量。
- 排序过程的时间复杂度最坏为 $O(N^2)$ ，但由于每次只插入一个元素，实际性能可能接近 $O(N)$ 。
- 树的构建过程的时间复杂度为 $O(N^2)$ ，因为每次插入新节点后都需要排序。
- 前序遍历的时间复杂度为 $O(n)$ ，因为每个节点只被访问一次。
- 整体时间复杂度为 $O(N^2)$ ，这主要受到排序算法的影响。

2.3.2 空间复杂度:

- 需要为每个节点和前缀编码分配空间，总空间复杂度为 $O(n)$ 。

2.4 完整代码如下:

```
#include <iostream>
#include <vector>
#include <memory>
#include <algorithm>
#include <iomanip>
```

```
const int N = 13;
```

```
int n;
// 定义树节点结构体
struct TreeNode
{
    int num;
    std::shared_ptr<TreeNode> Lnode; // 左孩子
    std::shared_ptr<TreeNode> Rnode; // 右孩子
};

std::vector<std::shared_ptr<TreeNode>> fp(N); // 保存节点
std::vector<char> s(2 * N); // 存储前缀码

// 初始化节点函数，用于生成叶子节点
void init_node(const std::vector<int>& f)
{
    for (size_t i = 0; i < f.size(); ++i)
    {
        auto pt = std::make_shared<TreeNode>(); // 创建一个叶子节点
        pt->num = f[i];
        fp[i] = pt;
    }
}

// 排序函数，将第 N-n 个点插入到已排序的序列中
void sort(std::vector<std::shared_ptr<TreeNode>>& array)
{
    for (int i = 0; i < n - 1; ++i)
    {
        if (array[i]->num > array[i + 1]->num)
        {
            std::swap(array[i], array[i + 1]);
        }
    }
}

// 构建树的函数
std::shared_ptr<TreeNode> construct_tree(std::vector<int>& f)
{
    if (f.size() == 1)
    {
        // 如果只有一个节点，直接返回这个节点
        return std::make_shared<TreeNode>(TreeNode{ f[0], nullptr,
        nullptr });
    }
}
```

```
}
init_node(f); // 初始化叶子节点
for (int i = 1; i < n; ++i)
{
    auto pt = std::make_shared<TreeNode>(); // 创建一个非叶子节点
    pt->num = fp[i - 1]->num + fp[i]->num;
    pt->Lnode = fp[i - 1];
    pt->Rnode = fp[i];
    fp[i] = pt; // 更新节点
    sort(fp); // 排序
}
return fp[n - 1]; // 返回树的根节点
}

// 前序遍历函数
void preorder(const std::shared_ptr<TreeNode>& p, int k, char c)
{
    if (!p)
        return;
    if (p)
    {
        s[k] = (c == 'l' ? '0' : '1');
        if (!p->Lnode)
        { // 叶子节点
            std::cout << std::left << std::setw(2) << p->num << ": ";
            for (int j = 0; j <= k; ++j)
                std::cout << s[j];
            std::cout << '\n';
        }
        if (p->Lnode)
            preorder(p->Lnode, k + 1, 'l');
        if (p->Rnode)
            preorder(p->Rnode, k + 1, 'r');
    }
}

int main()
{
    std::cout << "请输入节点个数(必须是正整数): ";
    while (1)
    {
        std::cin >> n;
        if (n <= 0 || n > N || std::cin.fail())
```

```
{
    std::cin.clear();
    std::cin.ignore(INT_MAX, '\n'); // 忽略错误输入直到下一个
换行符
    std::cout << "节点个数输入无效, 请重新输入: ";
    continue;
}
break;
}

std::vector<int> f(n);
std::cout << "请输入节点(以空格分隔): ";

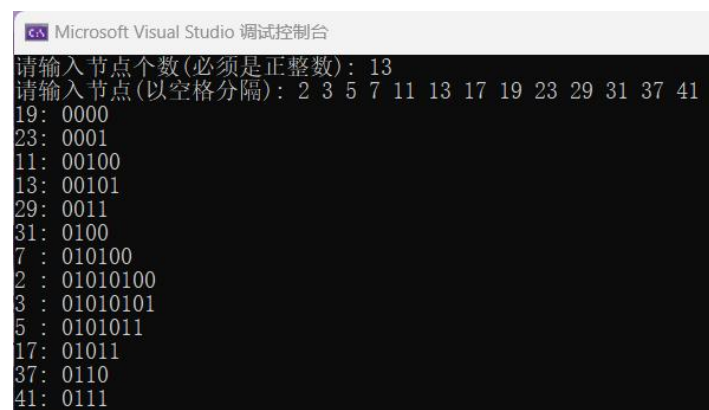
for (int i = 0; i < n; ++i)
{
    while (!(std::cin >> f[i]))
    {
        if (std::cin.fail())
        {
            std::cin.clear(); // 清除 cin 的错误标志
            std::cin.ignore(INT_MAX, '\n'); // 忽略错误输入直到下
一个换行符
            std::cout << "输入错误, 请从第" << i + 1 << "个节点开
始重新输入: ";
        }
    }
}

auto head = construct_tree(f); // 构建树
s[0] = '0'; // 初始化前缀码
preorder(head, 0, '1'); // 前序遍历树
return 0;
}
```


3. 项目测试

3.1 示例情况

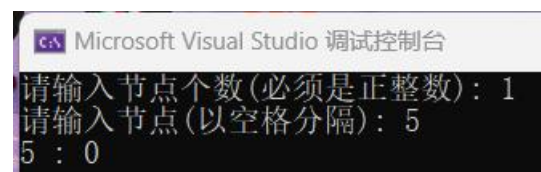
测试结果：



```
Microsoft Visual Studio 调试控制台
请输入节点个数(必须是正整数): 13
请输入节点(以空格分隔): 2 3 5 7 11 13 17 19 23 29 31 37 41
19: 0000
23: 0001
11: 00100
13: 00101
29: 0011
31: 0100
7 : 010100
2 : 01010100
3 : 01010101
5 : 0101011
17: 01011
37: 0110
41: 0111
```

3.2 仅有一个结点的情况

测试结果：



```
Microsoft Visual Studio 调试控制台
请输入节点个数(必须是正整数): 1
请输入节点(以空格分隔): 5
5 : 0
```

4. 心得体会

在我转换这段代码到现代 C++ 风格时，我采用了多个关键实践，以确保代码的健壮性和可读性。首先，我决定使用 `std::shared_ptr` 来管理 `TreeNode` 的内存，这样可以避免手动释放内存，减少了内存泄露的风险。我注意到，通过使用 `std::make_shared`，我不仅能够简化智能指针的创建，同时还提高了内存分配的效率。

接着，我将数组替换为 `std::vector`。这个决定基于 `vector` 能够提供自动的内存管理和灵活的容器大小。我特别欣赏 `vector` 在处理动态数据集时的弹性，它使我能够避免在数组大小和内存分配上的常见错误。

在输入验证方面，我仔细地设计了循环和条件语句，以确保用户输入的正确性。每次用户的输入不符合预期时，我都会清除错误状态，然后提示他们重新输入，这样增加了程序的健壮性，并提升了用户体验。