同济大学
TONGJI UNIVERSITY

离散数学

课程实验报告

最优二元树

姓　　名：　　马小龙

学　　号：　　2353814

学　　院：计算机科学与技术学院（软件学院）

专　　业：　　软件工程

指导教师：　　李冰

二〇二四年十一月二十九日

目录

一 实验目的

本实验的目的是通过构建最优二元树（哈夫曼树）来理解如何根据通信符号的使用频率生成对应的前缀码。通过此实验，学生将掌握哈夫曼编码的基本原理，学习如何利用该算法对数据进行压缩优化，减少冗余，从而提高数据传输效率。学生还将加深对二元树结构、优先队列和编码策略的理解，并能够在实际通信场景中应用哈夫曼编码技术来提高性能。

二 实验内容

实验中，输入一组通信符号及其名称和使用频率，要求根据符号的频率为每个符号生成一个唯一的前缀码。要求为高频符号分配较短的编码，而为低频符号分配较长的编码，从而有效压缩数据。这些前缀码的设计应满足无歧义性，确保在传输过程中解码的唯一性和准确性。

三 实验环境

编程开发语言：C++
文 件 编 码：UTF-8
集成开发环境：Clion 2024
工 具 集：MinGW 11.0 w64

四 实验原理

给定 N 个权值作为 N 个叶子结点，构造一棵二叉树，若该树的带权路径长度达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman Tree)。哈夫曼树是带权路径长度最短的树，权值较大的结点离根较近。

最优二元树（又称哈夫曼树）是为了解决在数据传输和存储中如何有效压缩信息的问题而提出的。传统的二进制编码方法将每个符号映射到固定长度的二进制码，这样会导致频繁出现的符号用较长的编码表示，而不常出现的符号用较短

的编码表示，造成信息传输效率低下。为了解决这一问题，哈夫曼树提出了一种基于符号出现频率的动态编码方式。

最优二元树的主要思想是根据每个符号的出现频率来构建一颗树，频率高的符号对应树的较浅部分，频率低的符号对应树的较深部分。这样，符号的编码长度与其出现频率成反比，从而优化了编码的平均长度，减少了信息的冗余。哈夫曼树广泛应用于数据压缩领域，特别是在无损压缩算法（如 ZIP 文件格式、PNG 图像格式等）中，帮助提高数据存储和传输效率。

使用最优二元树进行编码，主要是基于哈夫曼树的结构来为每个符号分配一个二进制编码。每个符号的编码对应的是从根节点到达该符号所在叶子节点的路径。路径上的每一步往左走时分配“0”，往右走时分配“1”，最终生成每个符号的唯一编码。通过这种方法，高频符号对应较短的编码，低频符号对应较长的编码，从而实现了数据的压缩和冗余减少。

哈夫曼编码的优点在于它的编码是前缀编码，即任何一个编码都不是另一个编码的前缀，保证了编码的唯一解码性。

通过最优二元树，哈夫曼编码不仅在理论上具有最优的压缩效果，而且在实际应用中也被广泛采用，尤其是在通信和存储领域。

五 实验过程

5.1 实验思路

该程序的主要目标是实现哈夫曼树的构建，并根据输入的符号频率生成对应的前缀编码。实验的核心在于哈夫曼树的构建，通过链表的方式存储哈夫曼树，当建立完成后，从根节点开始遍历哈夫曼树，左则编码加“0”，向右则编码加“1”，当到达叶节点之后，此时的编码即为叶节点对应通信符号的编码。

5.2 程序框架设计

程序的框架大致分为几个部分：

1.程序启动与欢迎界面显示：程序启动时，首先清除屏幕并显示一个简单的欢迎界面，提示用户进入哈夫曼树的构建和前缀编码生成模块。

2.用户输入符号：程序要求用户输入符号的个数（节点的数量），并确保输入有效。如果用户输入不合法，程序会提示用户重新输入。接着，用户输入每个符号的名称和频率，程序验证每个频率是否合法并存储。

3.构建哈夫曼树：根据用户输入的频率数据，程序通过 HuffmanTree 类的 BuildHuffmanTree 方法构建哈夫曼树。

4.生成前缀编码：构建完哈夫曼树后，程序调用 GeneratePrefixCode 方法，通过递归遍历哈夫曼树生成前缀编码。

5.输出结果：格式化输出符号的频率和它们对应的前缀编码。

6.是否继续执行：在每次输出完结果后，程序询问用户是否继续执行。如果用户输入“Y”或“y”，则重新进入输入阶段，开始新一轮的哈夫曼树构建和前缀编码生成。如果用户输入“N”或“n”，则程序结束，显示结束语。

5.3 数据结构设计

5.3.1 哈夫曼树结点(TreeNode)设计

5.3.1.1 概述

TreeNode 结构体用于表示哈夫曼树的节点，包含符号、权重及左右子节点指针。

5.3.1.2 结构体定义

```
struct TreeNode
{
    std::string name;
    int value;
    TreeNode *left;
    TreeNode *right;
     TreeNode(std::string n, int x, TreeNode* l = nullptr, TreeNode* r = nullptr) ;
};
```

5.3.1.3 构造函数

TreeNode(std::string n, int x, TreeNode* l = nullptr, TreeNode* r = nullptr)

构造函数，初始化节点的名称、权重、左子节点和右子节点指针。

5.3.1.4 数据成员

name: 存储节点的符号名称。

value: 存储节点的频率或权重。

left、right: 分别指向节点的左子树和右子树。

5.3.2 Compare 结构体

5.3.2.1 概述

Compare 结构体定义了节点的比较方式，主要用于在优先队列中进行节点的排序，确保最小堆结构。

5.3.2.2 成员函数

bool operator()；

重载了 () 运算符，比较两个 TreeNode 节点的权重，确保优先队列中较小权重的节点具有更高的优先级。

5.3.3 哈夫曼树(HuffmanTree)类设计

5.3.3.1 概述

HuffmanTree 类负责构建哈夫曼树、生成前缀编码、获取编码映射等。

5.3.3.2 HuffmanTree 类定义

```
class HuffmanTree {
public:
    HuffmanTree();
    ~HuffmanTree();
    void BuildHuffmanTree(const std::vector<std::string>&names,const std::vector<int>& v);
    void GeneratePrefixCode();
    std::unordered_map<std::string, std::string> GetPrefixCode();
private:
    TreeNode *root;
    std::unordered_map<std::string,std::string> prefix_code;
    void Destroy(TreeNode * subTree);
    void GeneratePrefixCodePrivate(TreeNode *subTree, const std::string& prefix);
};
```

5.3.3.3 构造函数与析构函数

HuffmanTree();

构造函数，初始化头节点为 nullptr；

~HuffmanTree();

析构函数，释放哈夫曼树所占用内存资源，防止内存泄漏。

5.3.3.4 数据成员

root: 哈夫曼树的根节点。

prefix_code: 存储符号及其对应前缀编码的映射。

5.3.3.5 成员函数

BuildHuffmanTree(): 根据符号名称和权重构建哈夫曼树。

GeneratePrefixCode(): 生成前缀编码。

GetPrefixCode(): 获取前缀编码的映射。

Destroy(): 递归销毁树并释放内存。

GeneratePrefixCodePrivate(): 递归生成前缀编码。

5.4 程序功能实现

5.4.1 通信符号数目及符号频率输入功能

在该程序中，输入功能的主要任务是确保用户能够正确地输入符号的数量和每个符号的频率，并且对输入的有效性进行检查，保证输入数据的合法性。以下是对输入功能的详细分析：

1. 输入符号数量：

进入一个 while 循环，开始输入符号数量，std::cin >> dN 尝试读取一个 double 类型的输入（防止用户输入小数）。

输入的符号数量必须是一个正整数。当输入的数据无效时，或者输入的数据无法转换为整数或小于等于零，则程序清除缓冲区和输入错误状态，提示用户重新输入，开始新的输入循环。

输入成功后，将输入的数据转化为 int 型，作为通信符号数。

2.输入符号名称和频率：

根据输入的符号数目进入一个 for 循环，每次输入符号名称和频率时，程序检查用户输入是否符合合法条件：先检查频率输入 std::cin.fail() 判断是否为有效输入，element != static_cast<int>(element) 检查是否为整数，element <= 0 检查是否为正整数；在检查名称输入，检查其是否已经存在。

如果用户输入无效，或名称不存在时程序清空缓冲区并提示用户重新输入，从当前输入的符号开始重新输入。

如果输入值合法，那么通过 v.push_back(static_cast<int>(element)) 将有效的频率添加到 std::vector<int> v 中，以供后续处理。

5.4.3 程序退出功能的实现

在每次逻辑运算完成后，程序询问用户是否继续进行计算，要求用户输入一个字符（Y、y、N 或 n），以决定是否继续执行下一轮的运算。

用户输入通过 do-while 循环进行处理。程序使用_getch() 函数获取用户的输入，当用户输入的不是 Y、y、N 或 n 四个字符之一时，do-while 会继续执行，直到用户输入有效字符。

如果用户输入的是 Y 或 y，程序会继续执行主循环，开始新的命题逻辑运算。

如果用户输入的是 N 或 n，程序退出主循环，输出提示信息并退出程序。

5.5 核心算法实现

5.5.1 最优二叉树构建

首先，将输入的每个节点名称和权重（即符号频率）创建为 TreeNode 对象，并将这些节点加入优先队列 pq（最小堆）。优先队列会根据节点的权重自动进行排序，保证最小权重的节点优先出队。

然后，在优先队列中，重复以下步骤直到只剩下一个节点：

1.从优先队列中取出两个权重最小的节点（即最小堆的堆顶元素）。

2.创建一个新的节点，该节点的权重是这两个节点权重的和，且将这两个节点作为新节点的左右子节点。

3.将新创建的节点加入到优先队列中。

最后，当优先队列中只剩下一个节点时，这个节点就是哈夫曼树的根节点。

5.5.2 前缀码生成

1.从根节点开始，递归遍历哈夫曼树的每个节点。

2.在遍历过程中，为每个节点维护一个字符串 prefix，用于表示当前节点的路径。根节点的 prefix 为 "0"。每次访问左子节点时，将 prefix 加上字符 '0'，每次访问右子节点时，将 prefix 加上字符 '1'。

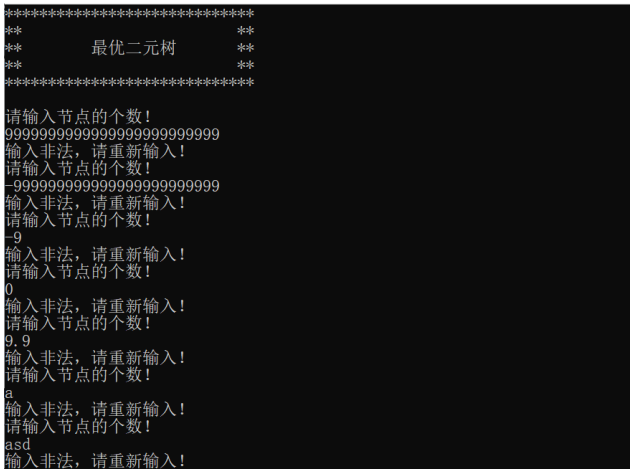
3. 如果当前节点是叶子节点（即没有左子节点和右子节点），则将该节点的名称和对应的 prefix 路径存储在前缀编码映射表 prefix_code 中。

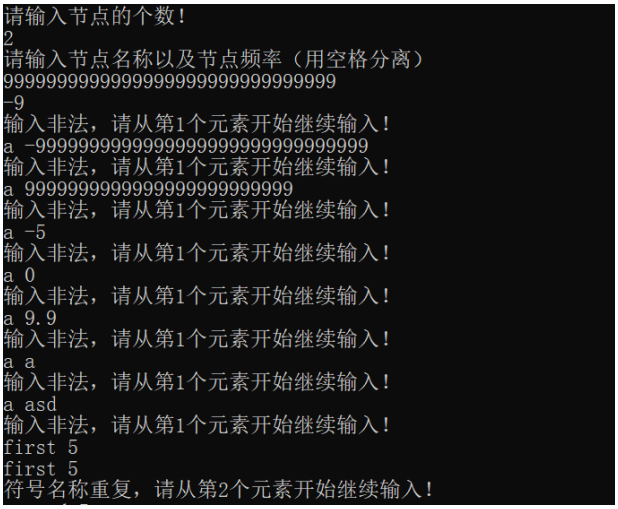
4. 对左子树和右子树分别递归调用，直到遍历完整棵树；当树遍历完毕，所有叶子节点的前缀编码都被记录在 prefix_code 中。

当前缀码生成完成后，可以获取前缀编码映射表，通过名称获得对应的前缀码从而进行输出工作。

六 实验结果测试

输入符号数目、符号名称和频率时，分别输入超过 int 上下限的整数、超范围的数，浮点数、字符、字符串，或者输入重复符号名称可以验证程序对输入非法的情况进行了处理。





当输入无误时，可以构建最优二叉树，同时为每个符号生成前缀编码。

```
请输入节点的个数!
13
请输入节点名称以及节点频率（用空格分离）
A 2
B 3
C 5
D 7
E 11
F 13
G 17
H 19
I 23
J 29
K 31
L 37
M 41
```

Communication Symbol Name	Usage Frequency	Prefix Code
A	2	1011110
B	3	1011111
C	5	101110
D	7	10110
E	11	0100
F	13	0101
G	17	1010
H	19	000
I	23	001
J	29	011
K	31	100
L	37	110
M	41	111

```
是否继续执行程序？[Y/n] _
```

七 实验心得

通过完成哈夫曼编码实验，我在多个方面得到了提升，尤其是在算法设计、数据结构应用、递归思维和编码原理等方面。

首先，哈夫曼编码作为一种贪心算法，要求每次合并最小的两个节点，从而达到最优编码。通过实现这一过程，我更好地理解了贪心算法的思路和实现方法。在程序中，我使用了优先队列（最小堆）来确保每次选取权重最小的节点进行合并，这不仅提高了算法的效率，还加深了我对数据结构的理解。尤其是在优先队列的使用上，我学会了如何通过合理的数据结构高效地解决实际问题。

此外，哈夫曼树的递归实现让我加深了对递归思想的理解。通过递归地遍历树形结构，我能够在每个叶子节点存储前缀编码。递归方法的使用让我更加熟悉树形结构的操作，也提升了我在其他复杂问题中使用递归的能力。这种思维方式帮助我处理了树的遍历和合并等复杂操作。

实验中，我还深入了解了哈夫曼编码的压缩原理。哈夫曼编码通过为高频符号分配较短的编码，为低频符号分配较长的编码，有效实现了数据压缩。这一设计不仅保证了编码的无歧义性，也提高了数据传输的效率。通过这次实验，我意

识到在实际应用中，数据压缩和编码的重要性，特别是在处理大量数据时，如何有效节省存储空间和提高传输效率。

此外，实验中的调试过程也提高了我的细节关注能力。在编写代码时，我不断调试并检查每个节点合并的正确性，确保程序输出符合预期。通过细致的调试，我学会了如何应对代码中的潜在问题，并提高了编程的稳定性和健壮性。

通过这次实验，我不仅掌握了哈夫曼树的构建与前缀编码的生成方法，还提升了在实际编程中解决复杂问题的能力。这些经验不仅有助于我在今后的学习中应用相关算法，还为我在实际工程中解决编码和压缩问题提供了思路。总的来说，这次实验增强了我对算法、数据结构和编码原理的理解，也让我在调试和编程过程中变得更加细心和高效。

附录 · 源码

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
#include <string>
#include <iomanip>
#include <algorithm>
#include <cassert>
#include <conio.h>

/*************************************************************
* Struct: TreeNode
* Function: 表示哈夫曼树的节点，每个节点包含两个值和两个子节点指针。
*************************************************************/
struct TreeNode {
    std::string name;
    int value;// 节点的值
    TreeNode *left;// 左子节点指针
    TreeNode *right;// 右子节点指针
    TreeNode(std::string n,int x,TreeNode*l=nullptr,TreeNode *r=nullptr ) :
name(n),value(x), left(l), right(r) {}
};

/*************************************************************
* Struct: Compare
* Function: 用于优先队列中比较两个节点的值，构建最小堆。
*************************************************************/
struct Compare {
    // 比较节点的值，值较小的优先
    bool operator()(const TreeNode * a, const TreeNode * b) const { return
a->value > b->value; }
};

/*************************************************************
* Class: HuffmanTree
* Function: 管理哈夫曼树的构建及生成前缀编码。
*************************************************************/
class HuffmanTree {
public:
    HuffmanTree(){root=nullptr;}
```

```
    ~HuffmanTree(){Destroy(root);}// 析构函数，释放内存
    void BuildHuffmanTree(const std::vector<std::string>&names,const
std::vector<int>& v);// 构建哈夫曼树
    void GeneratePrefixCode(){GeneratePrefixCodePrivate(root,"");} // 生成前缀编码
    std::unordered_map<std::string, std::string> GetPrefixCode(){return
prefix_code;}
private:
    TreeNode *root; // 哈夫曼树的根节点
    std::unordered_map<std::string,std::string> prefix_code;// 存储前缀编码的映射
    void Destroy(TreeNode * subTree);
    void GeneratePrefixCodePrivate(TreeNode *subTree, const std::string& prefix);
};

/*************************************************************
* Function Name: Destroy
* Function: 销毁哈夫曼树的所有节点并释放内存。
* Input Parameter: subTree - 当前递归处理的子树根节点。
*************************************************************/
void HuffmanTree::Destroy(TreeNode *subTree)
{
    if(subTree==nullptr)
        return;
    Destroy(subTree->left);
    Destroy(subTree->right);
    delete subTree;
    subTree=nullptr;
}

/*************************************************************
* Function Name: BuildHuffmanTree
* Function: 根据节点权重构建哈夫曼树，使用最小堆来选择最小的两个节点。
* Input Parameter: names - 存储节点名称的数组。
*                  v - 存储节点权重的数组。
*************************************************************/
void HuffmanTree::BuildHuffmanTree(const std::vector<std::string> &names,const
std::vector<int> &v)
{
    std::priority_queue<TreeNode*,std::vector<TreeNode *>,Compare> pq;

    // 将每个节点加入优先队列
    for(size_t i=0;i<names.size();i++) {
        auto *node = new(std::nothrow) TreeNode(names[i],v[i]); // 创建节点
        assert(node!=nullptr);
        assert(node!=nullptr);
```

```
        pq.push(node); // 将节点推入优先队列
    }
    // 合并两个最小节点直到队列中只剩下一个节点
    while(pq.size()>1) {
        auto left = pq.top();// 获取最小的节点
        pq.pop();
        auto right = pq.top();// 获取第二小的节点
        pq.pop();
        auto *node = new(std::nothrow)
TreeNode("",left->value+right->value,left,right);
        assert(node!=nullptr);
        pq.push(node);// 将新节点加入队列
    }
    root=pq.top();// 根节点是队列中剩余的唯一节点
    pq.pop();
}

/*************************************************************
* Function Name:   GeneratePrefixCodePrivate
* Function:        递归生成哈夫曼树的前缀编码。
* Input Parameter: subTree - 当前递归处理的子树根节点。
*                  prefix - 当前节点的前缀路径。
*************************************************************/
void HuffmanTree::GeneratePrefixCodePrivate(TreeNode* subTree,const std::string&
prefix)
{
    if(subTree==nullptr)// 如果当前节点为空，返回
        return;
    // 叶子节点
    if(subTree->left==nullptr&&subTree->right==nullptr) {
        prefix_code[subTree->name]=prefix;// 将当前节点的前缀编码存入映射
        return;
    }
    // 递归处理左右子树，分别添加 '0' 和 '1' 到前缀中
    GeneratePrefixCodePrivate(subTree->left, prefix+"0");
    GeneratePrefixCodePrivate(subTree->right, prefix+"1");
}

/*************************************************************
* Function Name:    Input
* Function:        从用户输入获取节点的个数和权重。
* Input Parameter: names - 存储节点名称的数组。
*                  v - 存储节点权重的数组。
*************************************************************/
```

```
void Input (std::vector<std::string> &names,std::vector<int>&v)
{
    int N;
    while(true) {
        std::cout << "请输入节点的个数！\n";
        double dN;
        std::cin >> dN;
        if (std::cin.fail()|| dN != static_cast<int>(dN)||dN<=0) {
            std::cout<<"输入非法，请重新输入！\n";
            std::cin.clear();
            std::cin.ignore(INT_MAX,'\n');
            continue;
        }
        std::cin.ignore(INT_MAX,'\n');
        N=static_cast<int>(dN);
        break;
    }

    std::cout<<"请输入节点名称以及节点频率（用空格分离）\n";
    for(int i=0;i<N;i++) {
        std::string name;
        double element;
        std::cin >>name >> element;
        if (std::cin.fail() || element != static_cast<int>(element) || element <=
0) {
            std::cout << "输入非法，请从第"<<i+1<<"个元素开始继续输入！\n";
            std::cin.clear();
            std::cin.ignore(INT_MAX, '\n');
            i--;
            continue;
        }
        bool is_continue=false;
        for (const auto& it:names)
            if (it==name) {
                std::cout << "符号名称重复，请从第" << i + 1 << "个元素开始继续输
入！\n";
                std::cin.clear();
                std::cin.ignore(INT_MAX, '\n');
                i--;
                is_continue=true;
            }
        if (is_continue)
            continue;
        names.push_back(name);
```

```
        v.push_back(static_cast<int>(element));
    }
}

/*************************************************************
* Function Name:   MaxElement
* Function:        获取数组中最大元素。
* Input Parameter: nums - 存储元素的数组。
* Returned Value:  返回数组中的最大元素。
*************************************************************/
int MaxElement(const std::vector<int> &nums)
{
    // 使用 std::max_element 获取最大元素的迭代器
    auto maxElementIt = std::max_element(nums.begin(), nums.end());
    return *maxElementIt; // 解引用迭代器获取最大元素的值
}

/*************************************************************
* Function Name:  countDigits
* Function:        计算数字的位数。
* Input Parameter:  num - 要计算位数的数字。
* Returned Value:  返回数字的位数。
*************************************************************/
int countDigits(const int num)
{
    // 将数字转换为字符串，然后返回字符串的长度（即位数）
    return static_cast<int>(std::to_string(num).length());
}

/*************************************************************
* Function Name:    MaxNameLength
* Function:        获取数组中的字符串的最大长度。
* Input Parameter: names - 存储元素的数组。
* Returned Value:  返回数组中的字符串的最大长度。
*************************************************************/
int MaxNameLength(const std::vector<std::string> &names)
{
    int max_length=0;
    for(const auto & name : names)

max_length=static_cast<int>(name.size())>max_length?static_cast<int>(name.size()):m
ax_length;
    return max_length;
}
```

```
/*************************************************************
* Function Name:   Print
* Function:        打印节点的频率和前缀编码。
* Input Parameter: names - 存储节点名称的数组
*                  nums - 存储节点频率的数组。
*                  prefix_code - 存储前缀编码的映射。
*************************************************************/
void Print(const std::vector<std::string>& names, const std::vector<int>
&nums,std::unordered_map<std::string, std::string>& prefix_code)
{
    int max_length=MaxNameLength(names);
    int digit_count = countDigits(MaxElement(nums));
    int max_size = 0;
    for (const auto &it: prefix_code) {
        int size = static_cast<int>(it.second.size());
        max_size=size > max_size ? size : max_size;
    }
    const std::string symbol_name="Communication Symbol Name";
    const std::string fluency = "Usage Frequency";
    const std::string prefix = "Prefix Code";
    const int len0=
max_length+2>static_cast<int>(symbol_name.size())?max_length+2:static_cast<int>(sym
bol_name.size());
    const int len1 = digit_count + 2 > static_cast<int>(fluency.size()) ?
digit_count + 2 : static_cast<int>(fluency.size());
    const int len2 = max_size + 2 > static_cast<int>(prefix.size()) ? max_size +
2 : static_cast<int>(prefix.size());
    std::cout << std::setfill('-') << std::left;
    std::cout << "+" << std::setw(len0) << '-' << '+' << std::setw(len1) << '-' <<
'+' << std::setw(len2) << '-' << '+'<< '\n';
    std::cout << std::setfill(' ');
    std::cout << "|" << std::setw(len0) << symbol_name << '|'<< std::setw(len1) <<
fluency << '|' << std::setw(len2) << prefix<< "|\n";
    std::cout << std::setfill('-');
    std::cout << "+" << std::setw(len0) << '-'<< '+' << std::setw(len1) << '-' <<
'+' << std::setw(len2) << '-' << '+' <<'\n';

    for (size_t i =0;i<names.size();i++) {
        std::cout << std::setfill(' ');
        std::cout << '|' << std::setw(len0) << names[i] << '|'<<std::setw(len1) <<
nums[i] << '|' << std::setw(len2) << prefix_code[names[i]] << '|' << '\n';
        std::cout << std::setfill('-');
        std::cout << "+" << std::setw(len0) << '-' << '+' << std::setw(len1) << '-'
```

```
<< '+' << std::setw(len2) << '-' <<'+' << '\n';
    }
}

/*************************************************************
* Function Name: main
* Function: 主程序入口，控制哈夫曼树的构建和前缀编码生成过程。
*           提供用户交互界面，允许用户多次输入数据并查看结果。
*************************************************************/
int main()
{
    while (true) {
        system("cls");// 清屏，刷新界面
        std::cout << "*****************************\n" //标语
                  << "**                         **\n"
                  << "**        最优二元树       **\n"
                  << "**                         **\n"
                  << "*****************************\n\n";


        std::vector<int> v;
        std::vector<std::string> names;

        Input(names,v);
        // 创建一个 HuffmanTree 对象
        HuffmanTree huffmanTree;
        // 使用输入的节点数据构建哈夫曼树
        huffmanTree.BuildHuffmanTree(names,v);
        // 生成前缀编码
        huffmanTree.GeneratePrefixCode();
        // 获取生成的前缀编码
        std::unordered_map<std::string, std::string> prefix_code =
huffmanTree.GetPrefixCode();
        // 打印节点的频率和对应的前缀编码
        Print(names,v, prefix_code);

        // 提示用户是否继续执行程序
        std::cout<<"是否继续执行程序？[Y/n] ";
        char ch;
        do {
            ch = static_cast<char>(_getch());
        }while (ch != 'Y' && ch != 'y' && ch != 'N' && ch != 'n');
        std::cout<<ch<<'\n';
```

```
        // 如果用户输入 N 或 n，则退出循环，结束程序
        if (ch == 'N' || ch == 'n')
            break;
    }

    std::cout << "\n 欢迎下次使用！ \n";
    return 0;
}
```