# 《离散数学》课程实验报告5 最优2元树的应用

# 题目简介；

# 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

# 二.解题思路；：

# 1. 初始化节点和权值：首先生成叶子节点，每个叶子节点包含一个通信符号的权值。这些节点被存储在数组中。

# 2. 构建哈夫曼树：构建哈夫曼树。该函数通过循环迭代的方式，每次选择权值最小的两个节点，合并为一个新的父节点，直到最终形成哈夫曼树的根节点。

# 3. 生成前缀码：通过对哈夫曼树进行前序遍历，生成通信符号对应的前缀码。在遍历过程中，使用递归的方式生成每个通信符号对应的前缀码，并输出到标准输出流中。

# 三.数据结构；

# 1. tree 结构体：

# - 用于表示哈夫曼树的节点，包括节点的权值 `num`，左子节点指针 `Lnode`，右子节点指针 `Rnode`。

# - 这个结构体定义了哈夫曼树节点的基本属性，包括权值和指向左右子节点的指针。

# 2. fp 数组：

# - `struct tree\* fp[N]` 数组用来保存节点。

# - 这个数组实际上代表了哈夫曼树的结构，在整个算法中扮演了保存节点的角色。

# 3. s 数组：

# - 用于存储生成的前缀码。

# - 其中 `s` 数组用来存储生成的前缀码，大小为 `2 \* N`，在 `preorder` 函数中被填充和输出。

# 四. 实验原理和实现过程（算法描述）；

## 1.实验原理

## 赫夫曼树（Huffman Tree）是一种经典的用于数据压缩的树形数据结构，它通过根据数据的出现频率来构建一套最优的编码方案。赫夫曼树的逻辑原理如下：

## 1. 构建赫夫曼树：

## - 赫夫曼树是一种特殊的二叉树，它的构建过程是基于贪心算法的。首先，将所有的数据看作是一个森林，每个数据对应一棵只包含一个节点的树。然后，通过不断合并森林中权值最小的两棵树（即节点权值最小的两个树），生成一棵新的树，直到森林中只剩下一棵树为止。这棵树就是赫夫曼树。

## 2. 构建赫夫曼编码：

## - 在赫夫曼树构建完成后，从根节点到每个叶子节点的路径上的0和1的序列就构成了该叶子节点对应的赫夫曼编码。赫夫曼编码的特点是没有一个编码是另一个编码的前缀，因此被称为前缀编码。这样的编码方案保证了解码的唯一性。

## 总的来说，赫夫曼树通过根据数据的出现频率构建一套最优的编码方案，实现了对数据的高效压缩和解压缩。其核心原理是利用数据的频率信息来设计编码方案，从而实现数据的无损压缩。

## 2.实验过程

# 通过哈夫曼树算法，根据输入的通信符号的使用频率，生成各通信符号对应的前缀码。

1. 初始化节点：在这一步中，程序根据输入的节点个数和权值，生成了对应数量的叶子节点，并将它们保存在`fp`数组中。每个叶子节点都有一个权值，表示该节点在哈夫曼树中的重要性。

2. 排序：在这一步中，程序对叶子节点按照权值进行排序。排序是为了后续的构建树过程做准备。程序使用了简单的插入排序算法，从第N-n个节点开始，依次将节点插入已排好序的序列中，以保证节点按照权值从小到大排列。

3. 构建树：在这一步中，程序通过不断合并权值最小的两个节点，构建哈夫曼树。合并的过程是这样的：程序创建一个新的非叶子节点，它的权值是两个待合并节点的权值之和，然后将这两个待合并节点分别作为新节点的左子节点和右子节点。合并后，程序将新节点放入数组中，并继续进行排序和合并的操作，直到只剩下一个根节点。

4. 前序遍历树：前序遍历是一种树的遍历方式，它以根节点为起点，先访问根节点，然后按照左子树、右子树的顺序依次遍历。在这个程序中，前序遍历的目的是为了生成每个叶子节点对应的前缀码。在遍历过程中，程序使用了一个字符数组`s`来记录前缀码。当遇到左子节点时，将字符'1'添加到`s`中，表示左子节点对应的前缀码为'1'；当遇到右子节点时，将字符'0'添加到`s`中，表示右子节点对应的前缀码为'0'。当遍历到叶子节点时，程序输出叶子节点的权值和对应的前缀码。

5. 输出结果：在这一步中，程序输出每个叶子节点的权值和对应的前缀码。对于每个叶子节点，程序先输出权值，然后输出前缀码。

# 五.部分核心代码；

#include <iostream>

#include <cstdlib>

#define N 13

struct tree

{

int num;

struct tree\* Lnode;

struct tree\* Rnode;

}\*fp[N]; //保存结点

char s[2 \* N]; //放前缀码

void init\_node(int f[], int n) //生成叶子结点

{

int i;

struct tree\* pt;

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

{

pt = new struct tree; //生成叶子结点

pt->num = f[i];

pt->Lnode = NULL;

pt->Rnode = NULL;

fp[i] = pt;

}

}

void sort(struct tree\* array[], int n) //将第N-n个点插入到已排好序的序列中

{

int i;

struct tree\* temp;

for (i = N - n; i < N - 1; i++)

{

if (array[i]->num > array[i + 1]->num) // 比较两个节点的权值大小

{

temp = array[i + 1]; // 交换节点位置

array[i + 1] = array[i];

array[i] = temp;

}

}

}

struct tree\* construct\_tree(int f[], int n) //建立树

{

int i;

struct tree\* pt;

for (i = 1; i < N; i++)

{

pt = new struct tree; //生成非叶子结点

pt->num = fp[i - 1]->num + fp[i]->num; // 非叶子节点的权值为左右子节点的权值之和

pt->Lnode = fp[i - 1];

pt->Rnode = fp[i];

fp[i] = pt; //wl+w2

sort(fp, N - i); // 对已排好序的序列进行插入排序

}

return fp[N - 1]; // 返回根节点

}

void preorder(struct tree\* p, int k, char c)

{

int j;

if (p != NULL)

{

if (c == '1')

s[k] = '0';

else

s[k] = '1';

if (p->Lnode == NULL)

{ //P 指向叶子

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

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

std::cout << s[j]; //输出前缀码

std::cout << std::endl;

}

preorder(p->Lnode, k + 1, '1'); // 遍历左子树

preorder(p->Rnode, k + 1, 'r'); // 遍历右子树

}

}

int main()

{

int n;

std::cout << "请输入节点个数(必须是正整数):";

std::cin >> n;

int f[80];

std::cout << "请输入节点(以空格分隔):";

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

{

std::cin >> f[i]; // 输入各个节点的权值

}

struct tree\* head;

init\_node(f, N); //初始化结点

head = construct\_tree(f, N); //生成最优树

s[0] = 0;

preorder(head, 0, '1'); //遍历树

return 0;

}