

## 1. 题目分析

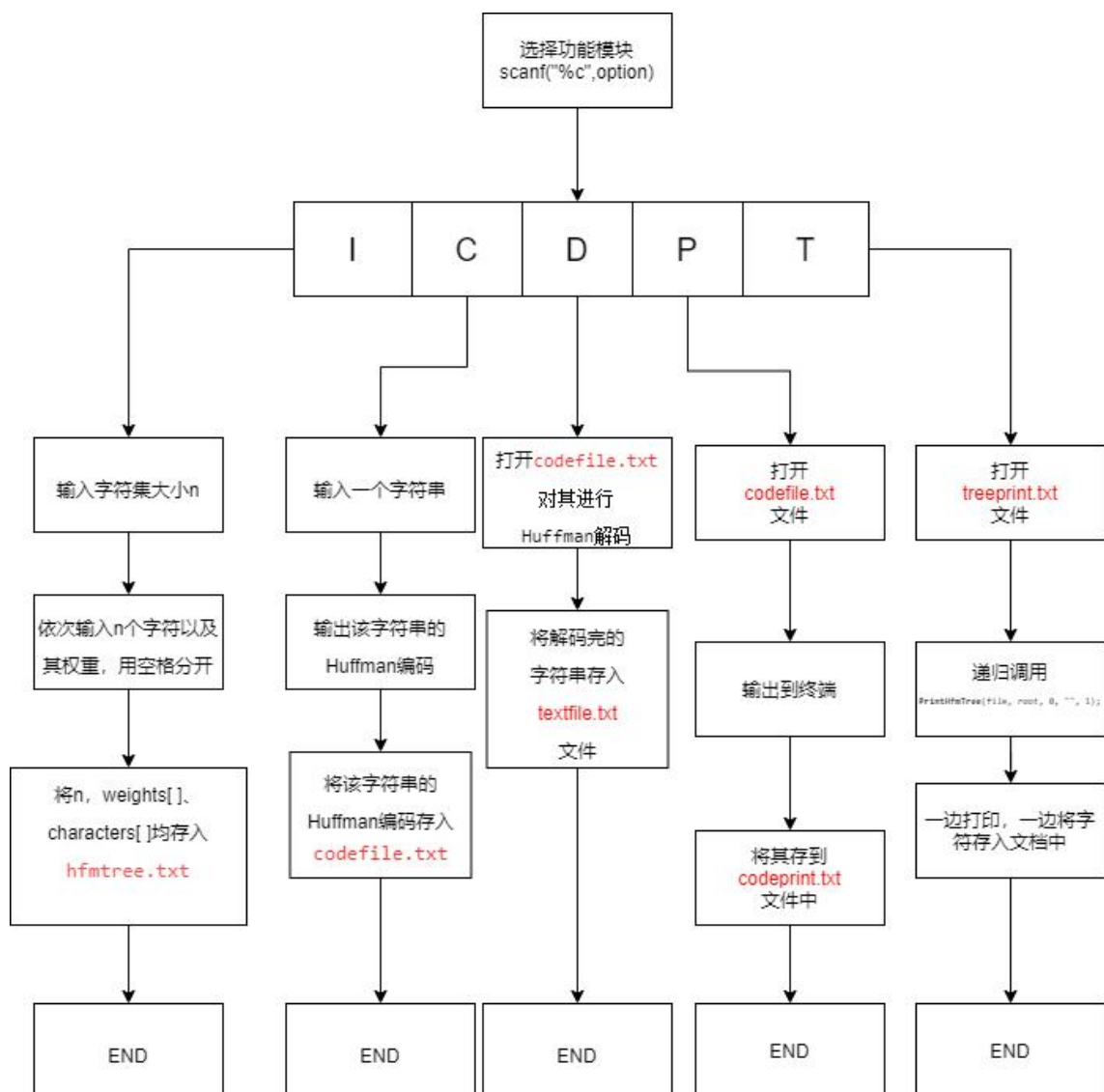
说明程序设计的任务，强调的是程序要做什么，此外列出各成员分工

- 任务:设计一个 Huffman 编码译码系统，有 I: 初始化；C: 编码；D: 译码；P: 印代码文件；T: 印 Huffman 树五个功能模块。

- 成员分工:

## 2. 数据结构设计

该程序主要用到的数据结构是 Huffman 树（最优二叉树），主程序的流程如下：



### 3. 程序设计

实现概要设计中的数据类型，对主程序、模块及主要操作写出伪代码，画出函数的调用关系

各模块伪代码如下：

初始化：

```
1 HuffmanTree initialization()
2 {
3     int n; // 字符集大小
4     printf("请输入字符集大小: \n");
5     scanf("%d", &n);
6     int *weights = (int *)malloc(sizeof(int) * n); // 动态分配n个权重
7     值 char *characters = (char *)malloc(sizeof(char) * n); // 动态分配n个字符
8     printf("请输入字符和权值: \n");
9     for (int i = 0; i < n; i++)
10    {
11        scanf(" %c %d", &characters[i], &weights[i]);
12    }
13    HuffmanTree root = buildHuffmanTree(weights, characters, n);
14    FILE *fp = fopen("hfmtree.txt", "w");
15    fprintf(fp, "%d\n", n);
16    for (int i = 0; i < n; i++)
17    {
18        fprintf(fp, "%c%s", characters[i], i == n - 1 ? "\n" : " ");
19    }
20    for (int i = 0; i < n; i++)
21    {
22        fprintf(fp, "%d%s", weights[i], i == n - 1 ? "\n" : " ");
23    }
24    fclose(fp);
25    return root;
26 }
```

编码:

```
1 void EnCode(HuffmanTree root)
2 {
3     char tobetrans[100];
4     char *result;
5     printf("请输入一个字符串: \n");
6     scanf("%s", tobetrans); // 读取输入的字符串, 存储到字符数组
7     result = encode(root, tobetrans);
8     printf("%s\n", result);
9     FILE *fp = fopen("codefile.txt", "w");
10    for (int i = 0; i < strlen(result); i++)
11    {
12        fprintf(fp, "%c", result[i]);
13    }
14    fclose(fp);
15 }
```

解码:

```
1 void DeCode(HuffmanTree root)
2 {
3     // 读取译码文件
4     FILE *fp_code = fopen("codefile.txt", "r");
5     char *code = (char *)malloc(1000 * sizeof(char)); // 申请存储代码的空间
6     fscanf(fp_code, "%s", code); // 读取代码
7     fclose(fp_code);
8
9     // 译码
10    char *text = (char *)malloc(1000 * sizeof(char)); // 申请存储译文的空间
11    int i = 0, j = 0;
12    while (code[i] != '\0')
13    {
14        char *tmp = (char *)malloc(100 * sizeof(char)); // 申请临时空间存储单个字符的编
15        int k = 0;
16        while (DeCodeChar(root, tmp) == '\0')
17        {
18            tmp[k++] = code[i++];
19        }
20        text[j++] = DeCodeChar(root, tmp); // 译码并存储译文
21        free(tmp); // 释放临时空间
22    }
23    text[j] = '\0';
24    // 存储译文到文件中
25    FILE *fp_text = fopen("textfile.txt", "w");
26    fprintf(fp_text, "%s", text);
27    fclose(fp_text);
28    // 释放申请的空间
29    free(code);
30    free(text);
31 }
```

印代码:

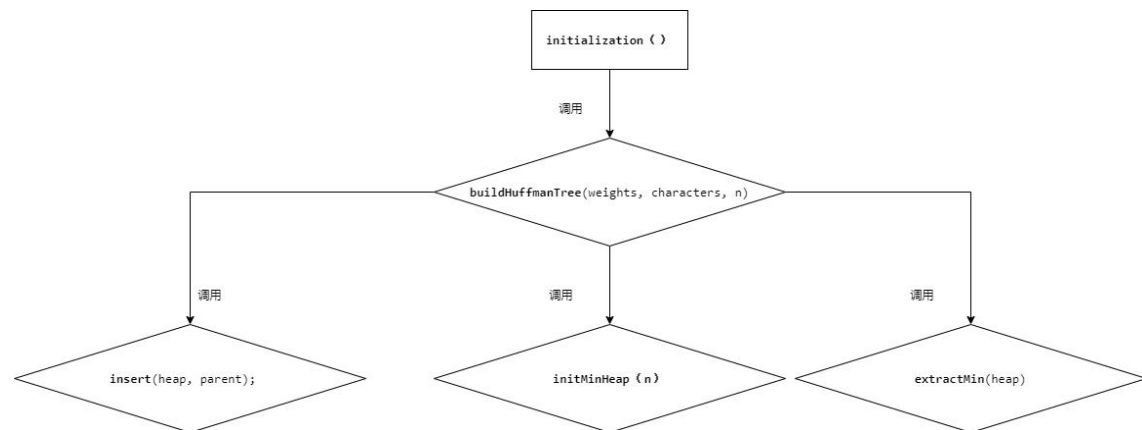
```
1 void Print()
2 {
3     FILE *fp;
4     char buffer[100];
5     fp = fopen("codefile.txt", "r");
6     if (fp == NULL)
7     {
8         printf("文件打开失败\n");
9         exit(1);
10    }
11    // 读取数据
12    fgets(buffer, 100, fp);
13    printf("%s", buffer);
14    fclose(fp);
15    fp = fopen("codeprint.txt", "w");
16    for (int i = 0; i < strlen(buffer); i++)
17    {
18        fprintf(fp, "%c", buffer[i]);
19    }
20    fclose(fp);
21 }
```

打印并存入 Huffman 树:

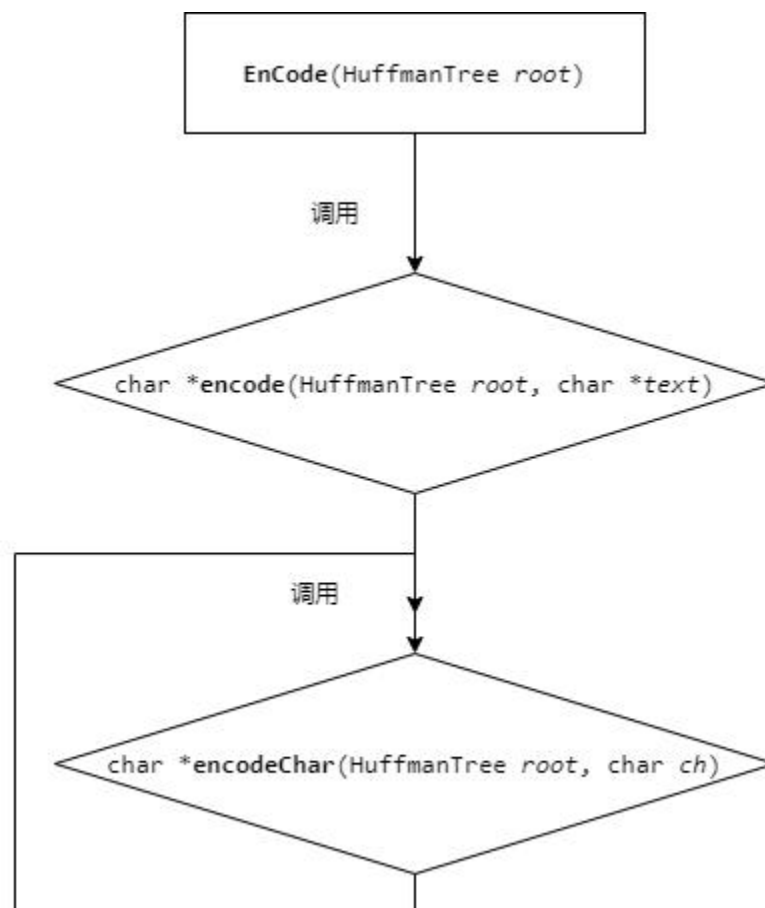
```
1 void Tree(HuffmanTree root)
2 {
3     // 打开文件treeprint
4     FILE *file = fopen("treeprint.txt", "w");
5     // 调用打印函数打印哈夫曼树并写入文件
6     PrintHfmTree(file, root, 0, "", 1);
7     // 关闭文件
8     fclose(file);
9 }
```

各函数层次关系图:

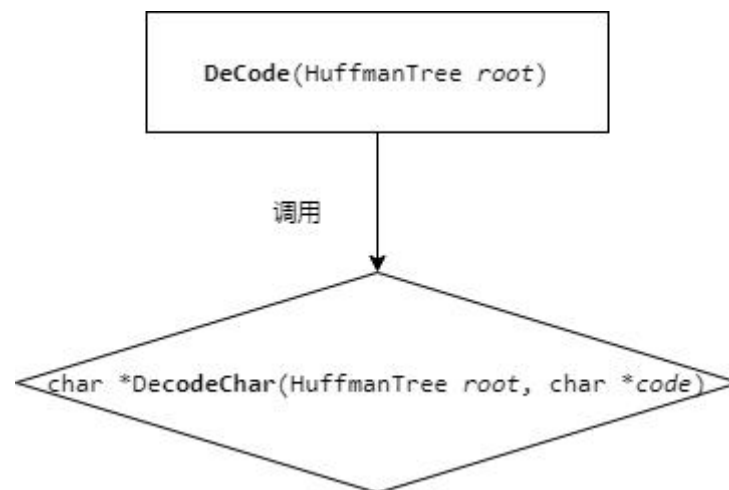
Initialization:



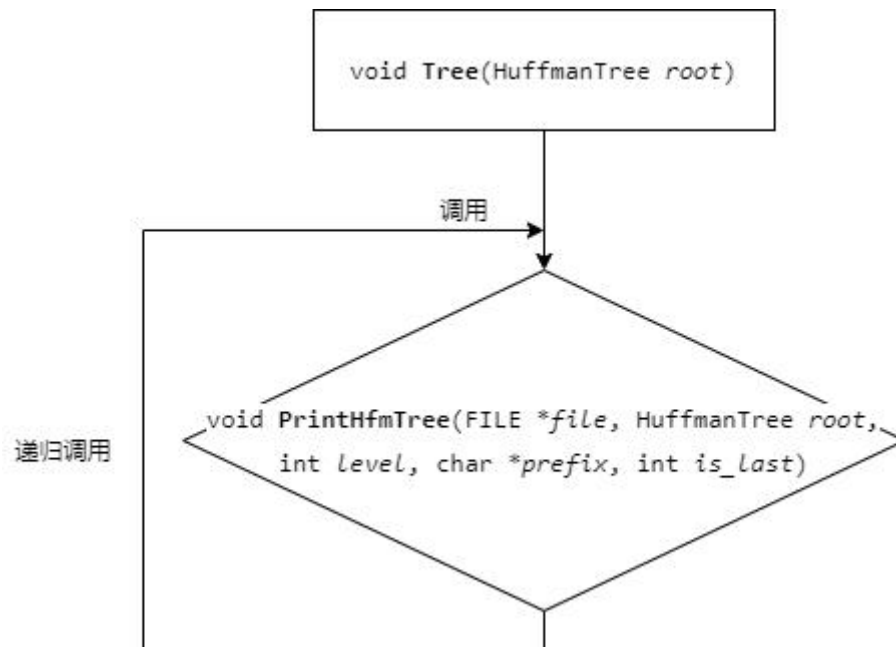
EnCode:



DeCode:



Tree:



#### 4. 调试分析

- 问题 1:内存泄露问题, 动态分配的空间未释放。
- 解决:在函数解放使用 free 来释放动态分配的空间
- 问题 2:按格式读取文件信息
- 解决:使用 buffer 和 characters 两个数组, buffer 用于接收所有的 txt 文件中的内容, 将 buffer 中有用的信息赋值给 characters。

时间复杂度:  $O(n \log n)$

explain: 由于建立哈夫曼树的过程中需要使用最小堆来实现节点的排序和合并。最小堆的插入和删除操作都需要  $O(\log n)$  的时间复杂度, 而需要进行  $n-1$  次操作, 因此总的时间复杂度为  $O(n \log n)$

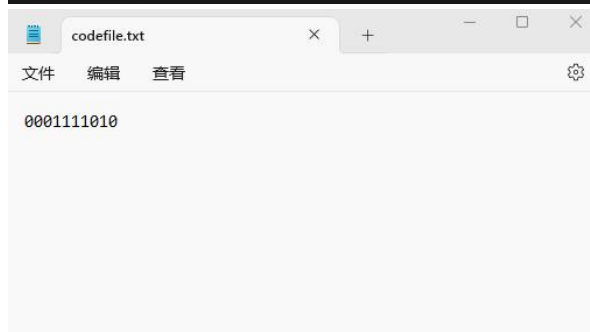
空间复杂度:  $O(n)$

explain: 空间复杂度主要是由节点的内存空间和最小堆的内存空间所占据的空间所决定的，因此总的空间复杂度为  $O(n)$ 。

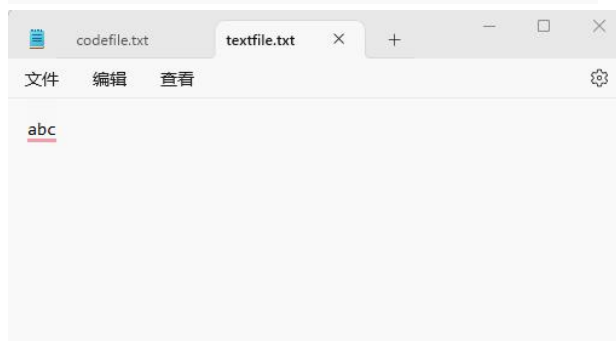
## 5. 测试结果

```
请选择您的操作：
I. 初始化
C. 编码
D. 译码
P. 印代码文件
T. 印哈夫曼树
E. 退出
I
您选择了初始化操作。
请输入字符集大小：
8
请输入字符和权值：
a 5
b 29
c 7
d 8
e 14
f 23
g 3
h 11
```

```
请选择您的操作：
I. 初始化
C. 编码
D. 译码
P. 印代码文件
T. 印哈夫曼树
E. 退出
C
您选择了编码操作。
请输入一个字符串：
abc
0001111010
```



A screenshot of a code editor window titled 'codefile.txt'. The window has a menu bar with '文件' (File), '编辑' (Edit), and '查看' (View). The main text area contains the binary string '0001111010'.



A screenshot of a code editor window with two tabs: 'codefile.txt' and 'textfile.txt'. The 'textfile.txt' tab is active and shows the string 'abc' with a red underline.





## huffman.cpp 程序

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <iostream>

using namespace std;

// 定义哈夫曼树的节点结构体

typedef struct node

{

    int weight;          // 权值

    char character;      // 字符

    struct node *left;   // 左子树指针

    struct node *right;  // 右子树指针

} Node;

// 定义哈夫曼树的节点类型

typedef Node *HuffmanTree;

// 定义哈夫曼树的节点最小堆

typedef struct heap
```

```

{

    int size;          // 堆的大小

    int capacity;      // 堆的容量

    HuffmanTree *data; // 堆数据存储指针
} MinHeap;

// 初始化最小堆

MinHeap *initMinHeap(int capacity)

{

    // 动态分配最小堆

    MinHeap *heap = (MinHeap *)malloc(sizeof(MinHeap));

    heap->capacity = capacity;

    heap->size = 0;

    heap->data = (HuffmanTree *)malloc(sizeof(HuffmanTree) * capacity);

    return heap;

}

// 最小堆中插入元素

void insert(MinHeap *heap, HuffmanTree node)

{

    if (heap->size >= heap->capacity)

    {

```

```

        return; // 如果堆已满，直接退出
    }

    heap->data[heap->size] = node; // 将元素插入堆底

    int current = heap->size++; // 更新堆大小

    int parent = (current - 1) / 2; // 父节点的下标

    // 自下往上调整堆，直到找到合适的位置插入新元素

    while (parent >= 0 && heap->data[current]->weight <
heap->data[parent]->weight)
    {
        // 如果当前元素比父节点的权值小，则交换两个元素的位置

        HuffmanTree temp = heap->data[parent];

        heap->data[parent] = heap->data[current];

        heap->data[current] = temp;

        current = parent;

        parent = (current - 1) / 2;
    }
}

// 从最小堆中取出最小元素

HuffmanTree extractMin(MinHeap *heap)
{
    if (heap->size == 0) // 如果堆为空，直接返回空指针

```

```

{
    return NULL;
}

HuffmanTree min = heap->data[0];          // 最小元素即为堆顶元素

heap->data[0] = heap->data[--heap->size]; // 将堆底元素移到堆顶，并更新堆大小

int current = 0;                          // 当前节点的下标

int child = current * 2 + 1;              // 当前节点的左孩子的下标

// 自上往下调整堆，直到找到合适的位置插入堆底元素

while (child < heap->size) // 当前节点还有孩子节点
{
    if (child + 1 < heap->size && heap->data[child + 1]->weight <
heap->data[child]->weight)
    {
        child++; // 找到当前节点的左右孩子中较小的一个
    }

    if (heap->data[child]->weight < heap->data[current]->weight)
    {
        // 将当前节点和较小孩子节点交换位置

        HuffmanTree temp = heap->data[child];

        heap->data[child] = heap->data[current];

        heap->data[current] = temp;

        current = child;          // 更新当前节点的下标
    }
}

```

```

        child = current * 2 + 1; // 更新当前节点的左孩子下标

    }

    else

    {

        break; // 如果已经满足最小堆的性质，则退出循环

    }

}

return min; // 返回被取出的最小元素
}

// 用最小堆构建哈夫曼树

HuffmanTree buildHuffmanTree(int *weights, char *characters, int n)

{

    // 初始化最小堆，将每个字符及其权重转换成节点，并插入堆中

    MinHeap *heap = initMinHeap(n);

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

    {

        Node *node = (Node *)malloc(sizeof(Node));

        node->weight = weights[i];

        node->character = characters[i];

        node->left = NULL;

        node->right = NULL;

```

```

        insert(heap, node); // 将节点插入堆中

    }

    // 不断从最小堆中取出权重最小的两个节点，合并成一个新节点，再插入堆中

    while (heap->size > 1)

    {

        HuffmanTree left = extractMin(heap); // 取出堆顶节点，即最小权重节点

        HuffmanTree right = extractMin(heap); // 再次取出最小权重节点

        Node *parent = (Node *)malloc(sizeof(Node));

        parent->weight = left->weight + right->weight; // 新节点的权重为左右节点的
权重之和

        parent->left = left; // 将左节点作为新节点的左孩
子

        parent->right = right; // 将右节点作为新节点的右孩
子

        insert(heap, parent); // 将新节点插入堆中

    }

    HuffmanTree root = extractMin(heap); // 最后堆中只剩下根节点，即为哈夫曼树的根
节点

    free(heap->data); // 释放堆数组占用的空间

    free(heap); // 释放最小堆结构体占用的空间

    return root; // 返回哈夫曼树的根节点指针

}

// 对单个字符进行编码模块

char *encodeChar(HuffmanTree root, char ch)

```

```

{

    static char code[100]; // 申请存储编码的空间

    static int index = 0; // 记录编码位数


    if (root == NULL)

    {

        return NULL;

    }


    if (root->character == ch)

    {

        code[index] = '\0'; // 编码结尾

        index = 0;          // 编码位数归零

        return code;

    }


    code[index++] = '0';

    char *leftCode = encodeChar(root->left, ch);

    if (leftCode != NULL)

    {

        return leftCode;

    }

}

```

```

    index--; // 回溯

    code[index++] = '1';

    char *rightCode = encodeChar(root->right, ch);

    if (rightCode != NULL)
    {
        return rightCode;
    }

    index--; // 回溯

    return NULL;
}

// 对文本进行编码模块

char *encode(HuffmanTree root, char *text)
{
    char *result = (char *)malloc(strlen(text) * 100 * sizeof(char)); // 申请存储编码结果
    的空间

    result[0] = '\0'; // 初始化

    for (int i = 0; i < strlen(text); i++)
    {
        char *code = encodeChar(root, text[i]); // 对单个字符编码

        if (code)

```



```

        {

            strcat(result, code); // 将编码拼接到结果中

        }

    }

    return result;
}

// 初始化函数
HuffmanTree initialization()
{

    int n; // 字符集大小

    printf("请输入字符集大小: \n");

    scanf("%d", &n);

    int *weights = (int *)malloc(sizeof(int) * n); // 动态分配 n 个权重值

    char *characters = (char *)malloc(sizeof(char) * n); // 动态分配 n 个字符

    printf("请输入字符和权值: \n");

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

    {

        scanf(" %c %d", &characters[i], &weights[i]);

    }

    HuffmanTree root = buildHuffmanTree(weights, characters, n);

    FILE *fp = fopen("hfmtree.txt", "w");

```

```

    fprintf(fp, "%d\n", n);

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

    {

        fprintf(fp, "%c%s", characters[i], i == n - 1 ? "\n" : " ");

    }

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

    {

        fprintf(fp, "%d%s", weights[i], i == n - 1 ? "\n" : " ");

    }

    fclose(fp);

    return root;
}

void EnCodeChar(HuffmanTree root)
{
    FILE *fp;

    char *characters;

    char buffer[100];

    int n;

    // 打开文件

    fp = fopen("hfmtree.txt", "r");

    if (fp == NULL)

```

```

{

    printf("文件打开失败\n");

    exit(1);

}

// 读取第一行，获取字符集大小

fscanf(fp, "%d", &n);

// 分配空间

characters = (char *)malloc(n * sizeof(char));

// 读取第二行数据

fgets(buffer, sizeof(characters) * 2, fp);

fgets(buffer, sizeof(characters) * 2, fp);

int i = 0;

int j = 0;

while (buffer[i] != NULL)

{

    if (buffer[i] != ' ')

    {

        characters[j] = buffer[i];

        j++;

    }

    i++;

}

```

```

    fclose(fp);

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

        char *res = encodeChar(root, characters[i]);

        printf("%c: %s\n", characters[i], res);
    }
}

void EnCode(HuffmanTree root)
{
    char tobetrans[100];

    char *result;

    printf("请输入一个字符串: \n");

    scanf("%s", tobetrans); // 读取输入的字符串, 存储到字符数组中

    result = encode(root, tobetrans);

    printf("%s\n", result);

    FILE *fp = fopen("codefile.txt", "w");

    for (int i = 0; i < strlen(result); i++)
    {
        fprintf(fp, "%c", result[i]);
    }
}

```

```

    fclose(fp);
}

char DeCodeChar(HuffmanTree root, char *code)
{
    HuffmanTree p = root;

    while (*code != '\0')
    {
        if (*code == '0')
        {
            p = p->left;
        }

        else if (*code == '1')
        {
            p = p->right;
        }

        if (p->left == NULL && p->right == NULL)
        {
            return p->character;
        }

        code++;
    }
}

```

```

        return '\0';
    }

void DeCode(HuffmanTree root)
{
    // 读取译码文件

    FILE *fp_code = fopen("codefile.txt", "r");

    char *code = (char *)malloc(1000 * sizeof(char)); // 申请存储代码的空间

    fscanf(fp_code, "%s", code); // 读取代码

    fclose(fp_code);

    // 译码

    char *text = (char *)malloc(1000 * sizeof(char)); // 申请存储译文的空间

    int i = 0, j = 0;

    while (code[i] != '\0')
    {
        char *tmp = (char *)malloc(100 * sizeof(char)); // 申请临时空间存储单个字符的
        编码

        int k = 0;

        while (DeCodeChar(root, tmp) == '\0')
        {
            tmp[k++] = code[i++];
        }
    }
}

```

```

        text[j++] = DeCodeChar(root, tmp); // 译码并存储译文

        free(tmp);                          // 释放临时空间
    }

    text[j] = '\0';

    // 存储译文到文件中

    FILE *fp_text = fopen("textfile.txt", "w");

    fprintf(fp_text, "%s", text);

    fclose(fp_text);

    // 释放申请的空间

    free(code);

    free(text);
}

void Print()
{
    FILE *fp;

    char buffer[100];

    fp = fopen("codefile.txt", "r");

    if (fp == NULL)
    {
        printf("文件打开失败\n");

        exit(1);
    }
}

```

```

    }

    // 读取数据

    fgets(buffer, 100, fp);

    printf("%s", buffer);

    fclose(fp);

    fp = fopen("codeprint.txt", "w");

    for (int i = 0; i < strlen(buffer); i++)

    {

        fprintf(fp, "%c", buffer[i]);

    }

    fclose(fp);
}

// 打印哈夫曼树的函数

void PrintHfmTree(FILE *file, HuffmanTree root, int level, char *prefix, int is_last)
{

    // 如果根节点为空，则返回

    if (root == NULL)

    {

        return;

    }

    // 打印当前节点的值

```



```

printf("%s", prefix);

printf(is_last ? "└── " : "│── ");

printf("(%c:%d)\n", root->character, root->weight);

fprintf(file, "%s", prefix);

fprintf(file, is_last ? "└── " : "│── ");

fprintf(file,("(%c:%d)\n", root->character, root->weight);


// 更新前缀

char new_prefix[128];

sprintf(new_prefix, "%s%s", prefix, is_last ? "    " : " |    ");


// 递归打印左右子树

PrintHfmTree(file, root->left, level + 1, new_prefix, (root->right == NULL));

PrintHfmTree(file, root->right, level + 1, new_prefix, 1);
}

void Tree(HuffmanTree root)
{

    // 打开文件 treeprint

    FILE *file = fopen("treeprint.txt", "w");

    // 调用打印函数打印哈夫曼树并写入文件

    PrintHfmTree(file, root, 0, "", 1);

```

```

    // 关闭文件

    fclose(file);
}

HuffmanTree root = nullptr; // 将 huffman 树根设置为全局变量

void Window()
{
    char choice;

    char exit_choice;

    while (1)
    {
        printf("请选择您的操作: \n");

        printf("I. 初始化\n");

        printf("C. 编码\n");

        printf("D. 译码\n");

        printf("P. 印代码文件\n");

        printf("T. 印哈夫曼树\n");

        printf("E. 退出\n");

        scanf(" %c", &choice); // 加上空格忽略换行符

        switch (choice)

```

```
{

case 'I':

    printf("您选择了初始化操作。 \n");

    root = initialization();

    break;

case 'C':

    if (root == NULL)

    {

        printf("请先进行初始化操作! \n");

        break;

    }

    printf("您选择了编码操作。 \n");

    EnCode(root);

    break;

case 'D':

    if (root == NULL)

    {

        printf("请先进行初始化操作! \n");

        break;

    }

    printf("您选择了译码操作。 \n");

    DeCode(root);
```

```

        break;

case 'P':

    printf("您选择了打印操作。\\n");

    Print();

    break;

case 'T':

    printf("您选择了打印操作。\\n");

    Tree(root);

    break;

case 'E':

    printf("您确定要退出吗？按 E 键确认退出，按其他键返回上级菜单。\\n");

    scanf(" %c", &exit_choice); // 加上空格忽略换行符

    if (exit_choice == 'E' || exit_choice == 'e')

    {

        printf("谢谢使用，再见！\\n");

        return;

    }

    break;

default:

    printf("无效的选择，请重新选择。\\n");

    break;

}

```

```
    }  
}  
  
int main()  
{  
    Window();  
    return 0;  
} return words;  
}
```