

《数据结构与算法》实验报告

实验名称	Huffman 树及 Huffman 编码的算法实现				
姓名	陈思睿	学号	21030021007	日期	2024. 5. 19
实验内容	<p>1、输入一段 100—200 字的英文短文，存入一文件 a 中。</p> <p>2、(2 分)写函数统计短文出现的字母个数 n 及每个字母的出现次数。</p> <p>3、(4 分)写函数以字母出现次数作权值，建 Huffman 树 (n 个叶子)，给出每个字母的 Huffman 编码。</p> <p>4、(2 分)用每个字母编码对原短文进行编码，码文存入文件 b 中。</p> <p>5、(2 分)用 Huffman 树对 b 中码文进行译码，结果存入文件 c 中，比较 a, c 是否一致，以检验编码、译码的正确性。</p>				
实验目的	<p>1、了解该树的应用实例，熟悉掌握 Huffman 树的构造方法及 Huffman 编码的应用</p> <p>2、了解 Huffman 树在通信、编码领域的应用过程。</p>				

实验步骤	<div data-bbox="331 212 903 257"><h3>(1) 输入一段短文，写入一个文件</h3></div> <div data-bbox="411 286 1329 324"><p>选取了狄更斯的小说《双城记》开篇短文，写进 test.txt 文件中</p></div> <div data-bbox="316 342 994 743"></div> <div data-bbox="331 772 715 817"><h3>(2) 字符频率统计函数</h3></div> <div data-bbox="331 824 1279 1529"><pre>FILE *file; int count = 0; count1 = 0; // int frequency[127] = {0}; // ASCII码来映射到每个字符的，统计其相应的概率 int ch; // 打开文件 file = fopen("D:\\test.txt", "r"); if (file == NULL) { fprintf(stderr, "无法打开文件\n"); return 0; } else { while ((ch = fgetc(file)) != EOF) { printf("%c",ch); if (ch >= 0 && ch < 127) { frequency[ch]++; // 更新相应字符的频率 } count++; } } fclose(file); printf("原文字符数: %d\n", count);</pre></div> <div data-bbox="316 1579 1340 1870"><p>打开文件 test.txt。这块统计字符频率时，由于是一片短文，因此除了字母，还有空格、逗号之类的符号，于是我做了两次统计，先采用哈希映射，根据 ASCII 码将能出现的所有可能的字符都考虑一遍，统计次数。</p></div>
------	---

[illegible]

但是存在问题，很多字符其实都是 0 的次数，大部分都集中在 26 个字母和空格，逗号之类的符号。于是定义了一个 array 优化数组，只统计出现的字符数。

```
for (int i = 0; i < 127; i++) {
    printf("%c:%d ", i, frequency[i]);
    if (frequency[i] != 0)
        count1++;
}

printf("\n");
printf("非零字符类型: %d\n", count1); // 统计非0字符个数，优化数组方便构建哈夫曼树
```

```
struct Char_frequency *array = (struct Char_frequency *)malloc(sizeof(struct Char_frequency) * count1);
if (array == NULL) {
    fprintf(stderr, "内存分配失败\n");
    return 0;
}
int j = 0;
for (int i = 0; i < 127; i++) {
    if (frequency[i] != 0) {
        array[j].f = frequency[i];
        array[j].s = i;
        j++;
    }
}
for (int i = 0; i < count1; i++) {
    printf("%c:%d ", array[i].s, array[i].f);
}
printf("\n");
```

```
非零字符类型: 37
:182 ,:20 -:2 .:4 ;:1 D:1 E:1 F:1 H:1 I:1 L:1 S:1 T:1 a:57 b:6 c:13 d:19 e:109 f:28 g:
20 h:44 i:58 j:2 k:4 l:21 m:5 n:43 o:58 p:12 q:2 r:48 s:53 t:72 u:8 v:8 w:31 y:5
```

```
struct Char_frequency {
    char s; // 字符
    int f; // 字符出现的频率
};
```

显然，优化后的 array 只有 37 个元素，array 数组采用结构体数组，方便后

续构建哈夫曼树。而且对比结果，根据常识也知道，例如 a, e,

i, o, u 之类的元音字母出现的概率应该是最高的，查看结果也是如此。

(3) 哈夫曼树的构建与编码

```
// 定义哈夫曼树节点结构体
typedef struct {
    char ch; // 字符
    int weight; // 权重
    int parent, lchild, rchild;
} HTNode, *HuffmanTree;
```

哈夫曼树构建时，由于最后需要翻译，如果再根据叶子节点去频率数组中找字符的话，我觉得比较麻烦，因此我又添加了一个 char 类型，方便在翻译时直接读取字符。

```
if (n <= 1) return;
int m = 2 * n - 1;
*HT = (HuffmanTree)malloc((m + 1) * sizeof(HTNode)); // 0号单元未用
// 初始化哈夫曼树的n个叶子节点的权值和字符
for (int i = 1; i <= n; i++) {
    (*HT)[i].ch = w[i - 1].s; // 字符
    (*HT)[i].weight = w[i - 1].f; // 字符频率数组下标是从0开始的
    (*HT)[i].lchild = 0;
    (*HT)[i].rchild = 0;
    (*HT)[i].parent = 0;
}
// 初始化哈夫曼树的n+1到m个分支节点
for (int i = n + 1; i <= m; i++) {
    (*HT)[i].weight = 0;
    (*HT)[i].lchild = 0;
    (*HT)[i].rchild = 0;
    (*HT)[i].parent = 0;
}
// 构建哈夫曼树
```

哈夫曼树的构建我采用了课堂讲过的算法，采用结构体数组来存储，其中 1-n 对应了哈夫曼树的叶子节点，也就是每个字符的结点。而其他非叶子结点根据哈夫曼树有 $n-1$ 个。初始化时，先初始化 1-n 的叶子结点，更新其权值和字符，其他都是 0，非

叶子结点类似，全为 0；

```
// 构建哈夫曼树
for (int i = n + 1; i <= m; i++) {
    int s1, s2;
    Select(*HT, i - 1, &s1, &s2); // 选择两个权值最小的节点
    (*HT)[s1].parent = i;
    (*HT)[s2].parent = i;
    (*HT)[i].lchild = s1;
    (*HT)[i].rchild = s2;
    (*HT)[i].weight = (*HT)[s1].weight + (*HT)[s2].weight;
}
```

构建哈夫曼树时，根据哈夫曼树的构成方法，每次选取权值最小的两个结点（通过判断父节点为空说可以选取构建），然后分别更新三个结点的相应的值。

哈夫曼编码：

```
// 生成哈夫曼编码,逆向求编码
*HC = (HuffmanCode)malloc((n + 1) * sizeof(char *));
char *cd = (char *)malloc(n * sizeof(char));
cd[n - 1] = '\0';
for (int i = 1; i <= n; i++) {
    int start = n - 1; // 编码结束位置
    for (int c = i, f = (*HT)[c].parent; f != 0; c = f, f = (*HT)[f].parent) {
        if ((*HT)[f].lchild == c) {
            cd[--start] = '0';
        } else {
            cd[--start] = '1';
        }
    }
    (*HC)[i] = (char *)malloc((n - start) * sizeof(char));
    strcpy((*HC)[i], &cd[start]);
    // 打印字符及其对应的哈夫曼编码
    printf("字符: %c 哈夫曼编码: %s\n", (*HT)[i].ch, (*HC)[i]);
}

free(cd);
```

生成哈夫曼编码时，采用从叶结点出发向上编码，定义左为 0，右为 1。

另外在该算法中还有一个 select 函数，用于选出未构建的结点，定义 min1 和 min2 两个值，遍历所有未构建的结点，找到

权值最小的两个。

```
void Select(HuffmanTree HT, int end, int *s1, int *s2) {
    int min1 = -1, min2 = -1;
    for (int i = 1; i <= end; i++) {
        if (HT[i].parent == 0) {
            if (min1 == -1 || HT[i].weight < HT[min1].weight) {
                min2 = min1;
                min1 = i;
            } else if (min2 == -1 || HT[i].weight < HT[min2].weight) {
                min2 = i;
            }
        }
    }
    *s1 = min1;
    *s2 = min2;
}
```

(4) 根据编码对原文进行转码写入 test2. txt

```
void write_encoded_text(struct Char_frequency* array, HuffmanCode HC, int count1) {
    FILE *inputFile = fopen("D:\\test.txt", "r");
    FILE *outputFile = fopen("D:\\test2.txt", "w");

    if (inputFile == NULL || outputFile == NULL) {
        fprintf(stderr, "无法打开文件\n");
        return;
    }

    char ch;
    while ((ch = fgetc(inputFile)) != EOF) {
        for (int i = 0; i < count1; i++) {
            if (ch == array[i].s) { // 在哈夫曼编码数组中找到对应的字符
                fprintf(outputFile, "%s", HC[i + 1]); // 写入对应的哈夫曼编码
                break;
            }
        }
    }

    fclose(inputFile);
    fclose(outputFile);
}
```

编码文件 test2. txt，根据 test. txt 文件逐字读入字符串，根据哈夫曼编码数组将其转化成哈夫曼编码，写入 test2. txt 文件中。

(5) 利用哈夫曼编码对 test2.txt 文件翻译, 写入 test3.txt 文件, 并检验是否一致。

```
void decode_text(HuffmanTree HT, int count1) {
    FILE *encodedFile = fopen("D:\\text2.txt", "r");
    FILE *outputFile = fopen("D:\\text3.txt", "w");

    if (encodedFile == NULL || outputFile == NULL) {
        fprintf(stderr, "无法打开文件\n");
        return;
    }

    int root = 2 * count1 - 1; // 赫夫曼树根节点的索引
    int current = root;
    char bit;

    while ((bit = fgetc(encodedFile)) != EOF) {
        if (bit == '0') {
            current = HT[current].lchild;
        } else if (bit == '1') {
            current = HT[current].rchild;
        }

        if (HT[current].lchild == 0 && HT[current].rchild == 0) { // 叶子节点
            fputc(HT[current].ch, outputFile); // 输出字符
            current = root; // 重置为根节点, 开始解码下一个字符
        }
    }
}
```

解码与编码相反, 编码从根节点开始, 根据读取的每个 1 或者 0, 从根节点开始向下寻找相对应的叶子结点, 然后输出对应的字符串。注意最后要重置根节点的值。写入 test3.txt 文件中。

对比函数: 分别读取原文件 test.txt 和解码后的文件 test3.txt 文件, 逐字对比是否相同, 存在不同就退出, 还要保障两个文件长度是一样的。

```

// 打开文件
file1 = fopen("D:\\test.txt", "r");
file2 = fopen("D:\\test3.txt", "r");
if (file1 == NULL || file2 == NULL) {
    fprintf(stderr, "无法打开文件\n");
    return;
} else {
    while ((ch1 = fgetc(file1)) != EOF && (ch2 = fgetc(file2)) != EOF) {
        if (ch1 != ch2) {
            printf("文件存在不同\n");
            fclose(file1);
            fclose(file2);
            return;
        }
    }

    if (feof(file1) && feof(file2)) {
        printf("文件完全一致\n");
    } else {
        printf("文件长度不同\n");
    }
}
}

```

源代码展示:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 定义哈夫曼树节点结构体
typedef struct {
    char ch; // 字符
    int weight; // 权重
    int parent, lchild, rchild;
} HTNode, *HuffmanTree;

typedef char **HuffmanCode; // 存放哈夫曼编码

struct Char_frequency {
    char s; // 字符
    int f; // 字符出现的频率
};

HuffmanTree HT; // 全局变量存储赫夫曼树
HuffmanCode HC; // 全局变量存储赫夫曼编码
int count1; // 非零字符类型个数

void Select(HuffmanTree HT, int end, int *s1, int *s2);
void creat_HuffmanTree(HuffmanTree *HT, HuffmanCode *HC, struct Char_frequency

```



```

*w, int n);
int calculate_frequency();
void write_encoded_text(struct Char_frequency* array, HuffmanCode HC, int count1);
void decode_text(HuffmanTree HT, int count1);
void compare_files();

int main() {
// 计算字符频率
if (calculate_frequency() != 0) {
// 解码文本
decode_text(HT, count1);
// 比较文件内容
compare_files();
}
return 0;
}

void write_encoded_text(struct Char_frequency* array, HuffmanCode HC, int count1)
{
FILE *inputFile = fopen("D:\\test.txt", "r");
FILE *outputFile = fopen("D:\\text2.txt", "w");

if (inputFile == NULL || outputFile == NULL) {
fprintf(stderr, "无法打开文件\n");
return;
}

char ch;
while ((ch = fgetc(inputFile)) != EOF) {
for (int i = 0; i < count1; i++) {
if (ch == array[i].s) { // 在哈夫曼编码数组中找到对应的字符
fprintf(outputFile, "%s", HC[i + 1]); // 写入对应的哈夫曼编码
break;
}
}
}

fclose(inputFile);
fclose(outputFile);
}

void Select(HuffmanTree HT, int end, int *s1, int *s2) {
int min1 = -1, min2 = -1;
for (int i = 1; i <= end; i++) {

```

```

if (HT[i].parent == 0) {
    if (min1 == -1 || HT[i].weight < HT[min1].weight) {
        min2 = min1;
        min1 = i;
    } else if (min2 == -1 || HT[i].weight < HT[min2].weight) {
        min2 = i;
    }
}
}
*s1 = min1;
*s2 = min2;
}
//创建哈夫曼树
void creat_HuffmanTree(HuffmanTree *HT, HuffmanCode *HC, struct Char_frequency
*w, int n) {
    if (n <= 1) return;
    int m = 2 * n - 1;
    *HT = (HuffmanTree)malloc((m + 1) * sizeof(HTNode)); // 0 号单元未用
    // 初始化哈夫曼树的 n 个叶子节点的权值和字符
    for (int i = 1; i <= n; i++) {
        (*HT)[i].ch = w[i - 1].s; // 字符
        (*HT)[i].weight = w[i - 1].f; // 字符频率数组下标是从 0 开始的
        (*HT)[i].lchild = 0;
        (*HT)[i].rchild = 0;
        (*HT)[i].parent = 0;
    }
    // 初始化哈夫曼树的 n+1 到 m 个分支节点
    for (int i = n + 1; i <= m; i++) {
        (*HT)[i].weight = 0;
        (*HT)[i].lchild = 0;
        (*HT)[i].rchild = 0;
        (*HT)[i].parent = 0;
    }
    // 构建哈夫曼树
    for (int i = n + 1; i <= m; i++) {
        int s1, s2;
        Select(*HT, i - 1, &s1, &s2); // 选择两个权值最小的节点
        (*HT)[s1].parent = i;
        (*HT)[s2].parent = i;
        (*HT)[i].lchild = s1;
        (*HT)[i].rchild = s2;
        (*HT)[i].weight = (*HT)[s1].weight + (*HT)[s2].weight;
    }
    // 生成哈夫曼编码,逆向求编码

```

```

*HC = (HuffmanCode)malloc((n + 1) * sizeof(char *));
char *cd = (char *)malloc(n * sizeof(char));
cd[n - 1] = '\0';
for (int i = 1; i <= n; i++) {
    int start = n - 1; // 编码结束位置
    for (int c = i, f = (*HT)[c].parent; f != 0; c = f, f = (*HT)[f].parent) {
        if ((*HT)[f].lchild == c) {
            cd[--start] = '0';
        } else {
            cd[--start] = '1';
        }
    }
    (*HC)[i] = (char *)malloc((n - start) * sizeof(char));
    strcpy((*HC)[i], &cd[start]);
    // 打印字符及其对应的哈夫曼编码
    printf("字符: %c  哈夫曼编码: %s\n", (*HT)[i].ch, (*HC)[i]);
}

free(cd);
}

int calculate_frequency() {
    FILE *file;
    int count = 0;
    count1 = 0; //
    int frequency[127] = {0}; // ASCII 码来映射到每个字符的，统计其相应的概率
    int ch;
    // 打开文件
    file = fopen("D:\\test.txt", "r");
    if (file == NULL) {
        fprintf(stderr, "无法打开文件\n");
        return 0;
    } else {
        while ((ch = fgetc(file)) != EOF) {
            printf("%c", ch);
            if (ch >= 0 && ch < 127) {
                frequency[ch]++; // 更新相应字符的频率
            }
            count++;
        }
    }
    fclose(file);

    printf("原文字符数: %d\n", count);
}

```

```

for (int i = 0; i < 127; i++) {
    printf("%c:%d ", i, frequency[i]);
    if (frequency[i] != 0)
        count1++;
}
printf("\n");
printf("非零字符类型: %d\n", count1); // 统计非 0 字符个数, 优化数组方便构建哈夫曼树

struct Char_frequency *array = (struct Char_frequency *)malloc(sizeof(struct Char_frequency) * count1);
if (array == NULL) {
    fprintf(stderr, "内存分配失败\n");
    return 0;
}
int j = 0;
for (int i = 0; i < 127; i++) {
    if (frequency[i] != 0) {
        array[j].f = frequency[i];
        array[j].s = i;
        j++;
    }
}
for (int i = 0; i < count1; i++) {
    printf("%c:%d ", array[i].s, array[i].f);
}
printf("\n");

// 接下来调用哈夫曼树的创建函数生成哈夫曼编码
creat_HuffmanTree(&HT, &HC, array, count1);

// 将原文根据哈夫曼编码写入文件
write_encoded_text(array, HC, count1);

// 释放内存
free(array);

return 0;
}

void decode_text(HuffmanTree HT, int count1) {
    FILE *encodedFile = fopen("D:\\text2.txt", "r");
    FILE *outputFile = fopen("D:\\text3.txt", "w");

```

```

if (encodedFile == NULL || outputFile == NULL) {
    fprintf(stderr, "无法打开文件\n");
    return;
}

int root = 2 * count1 - 1; // 赫夫曼树根节点的索引
int current = root;
char bit;

while ((bit = fgetc(encodedFile)) != EOF) {
    if (bit == '0') {
        current = HT[current].lchild;
    } else if (bit == '1') {
        current = HT[current].rchild;
    }
}

if (HT[current].lchild == 0 && HT[current].rchild == 0) { // 叶子节点
    fputc(HT[current].ch, outputFile); // 输出字符
    current = root; // 重置为根节点，开始解码下一个字符
}
}

fclose(encodedFile);
fclose(outputFile);
}

void compare_files() {
    FILE *file1;
    FILE *file2;
    int ch1, ch2;
    // 打开文件
    file1 = fopen("D:\\test.txt", "r");
    file2 = fopen("D:\\test3.txt", "r");
    if (file1 == NULL || file2 == NULL) {
        fprintf(stderr, "无法打开文件\n");
        return;
    } else {
        while ((ch1 = fgetc(file1)) != EOF && (ch2 = fgetc(file2)) != EOF) {
            if (ch1 != ch2) {
                printf("文件存在不同\n");
                fclose(file1);
                fclose(file2);
                return;
            }
        }
    }
}

```

```

}

if (feof(file1) && feof(file2)) {
printf("文件完全一致\n");
} else {
printf("文件长度不同\n");
}
}

// 关闭文件
fclose(file1);
fclose(file2);
}

```

实验结果

每个字符的出现次数和相对应的哈夫曼编码：

```

40 s:35 t:72 u:8 v:8 w:31 x:0 y:5 z:0 { :0 } :0 ~:0
非零字符类型: 37
:182 ,:20 -:2 .:4 ;:1 D:1 E:1 F:1 H:1 I:1 L:1 S:1 T:1 a:57 b:6 c:13 d:19 e:109 f:28
g:20 h:44 i:58 j:2 k:4 l:21 m:5 n:43 o:58 p:12 q:2 r:48 s:53 t:72 u:8 v:8 w:31 y:5

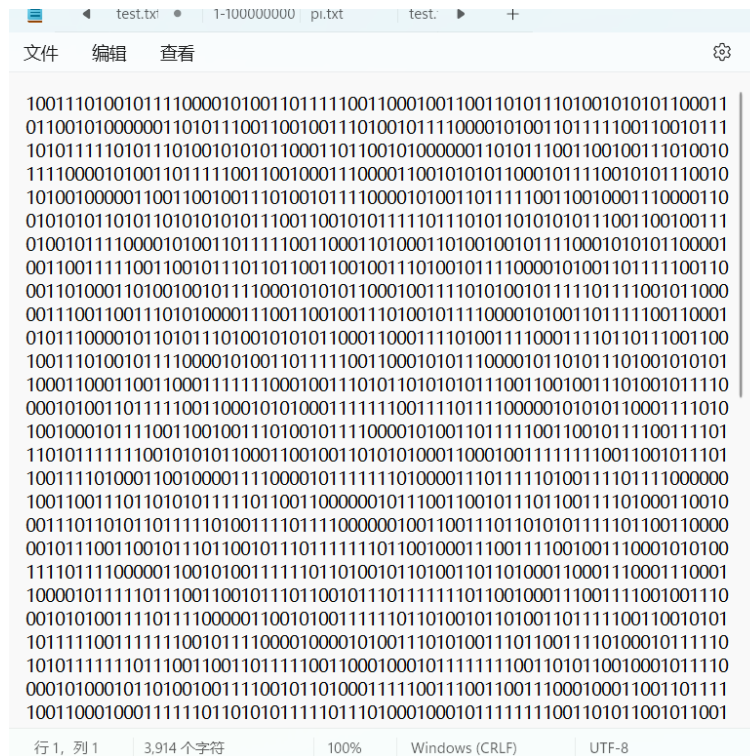
字符:   哈夫曼编码: 00
字符: , 哈夫曼编码: 110011
字符: - 哈夫曼编码: 010011101
字符: . 哈夫曼编码: 01001111
字符: ; 哈夫曼编码: 1100011000
字符: D 哈夫曼编码: 1100011001
字符: E 哈夫曼编码: 1100011010
字符: F 哈夫曼编码: 1100011011
字符: H 哈夫曼编码: 1100011100
字符: I 哈夫曼编码: 1100011101
字符: L 哈夫曼编码: 1100011110
字符: S 哈夫曼编码: 1100011111
字符: T 哈夫曼编码: 010011100
字符: a 哈夫曼编码: 1000
字符: b 哈夫曼编码: 0100110
字符: c 哈夫曼编码: 010010
字符: d 哈夫曼编码: 110010
字符: e 哈夫曼编码: 011
字符: f 哈夫曼编码: 10110
字符: g 哈夫曼编码: 111000
字符: h 哈夫曼编码: 11110
字符: i 哈夫曼编码: 1001
字符: j 哈夫曼编码: 110001010
字符: k 哈夫曼编码: 11000100
字符: l 哈夫曼编码: 111001
字符: m 哈夫曼编码: 0100000
字符: n 哈夫曼编码: 11101
字符: o 哈夫曼编码: 1010
字符: p 哈夫曼编码: 010001
字符: q 哈夫曼编码: 110001011
字符: r 哈夫曼编码: 11111
字符: s 哈夫曼编码: 0101
字符: t 哈夫曼编码: 1101
字符: u 哈夫曼编码: 1100000
字符: v 哈夫曼编码: 1100001
字符: w 哈夫曼编码: 10111
字符: y 哈夫曼编码: 0100001
PS D:\code>

```

根据出现次数也能看出，空格出现的次数最多，因此哈夫曼编码最短，而且元音字母如 a，e，i 等本身出现的次数就多，因

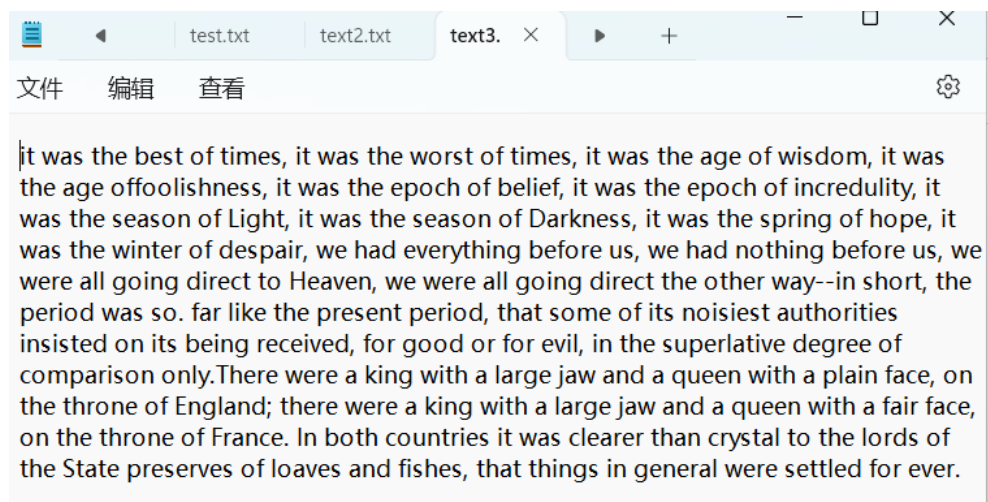
此哈夫曼编码也相对较短。

转码后的 test2.txt 文件



The screenshot shows a text editor window with a single tab labeled 'test.txt'. The editor displays a long string of binary code (0s and 1s) representing a Huffman-coded message. The status bar at the bottom indicates '行 1, 列 1' (Line 1, Column 1), '3,914 个字符' (3,914 characters), '100%' zoom, 'Windows (CRLF)' line endings, and 'UTF-8' encoding.

解码的 test.3 文件



The screenshot shows a text editor window with three tabs: 'test.txt', 'text2.txt', and 'text3. x'. The 'text3. x' tab is active and displays the decoded text, which is a paragraph from Charles Dickens' 'A Christmas Carol'. The status bar at the bottom indicates '文件' (File), '编辑' (Edit), '查看' (View), and a gear icon for settings.

	<p>可以手动对比几个字母，例如</p> <div><div><div>i</div><div>t</div></div><div>100111010010111100001010011011101100101000000110101110011001001010111101011101001010101100011111000010100110111110011001000</div><div>字符: i 哈夫曼编码: 1001</div><div>字符: t 哈夫曼编码: 1101</div><div>字符: 哈夫曼编码: 00</div><div>字符: w 哈夫曼编码: 10111</div><div>字符: a 哈夫曼编码: 1000</div><div>字符: s 哈夫曼编码: 0101</div><div>it was the</div><p>判别函数也输出完全一致, 转码无误</p><div>文件完全一致</div></div>
实验总结	<p>本次实验在完成哈夫曼编码的实验过程中，我了解到了关于数据压缩和信息编码的重要概念和技术，统计了原文和转码后的文件原文有 943 个字符，而哈夫曼编码仅仅 3914 个字符，平均一个字符只有不到 5 个 01 序列的长度来表示。通过本次实验，我也熟悉了二叉树的构建操作以及表示方法，加深了我对于数据结构的操作和使用。</p> <p>在实验过程中，我面临了一些困难，例如正确处理文件的读取与写入、设计高效的数据结构以及调试算法逻辑等。但最终都解决了。</p>