

# 河南工业大学 数据结构 实验报告

课程名称	数据结构	实验项目	实验三 二叉树的典型算法实现
院 系	信息学院计科系	专业班级	计科 XXXX 班
姓 名	COS	学 号	XXXXXXXXXX
指导老师	XXXXXX	日 期	2020.11.2
批改日期		成 绩	

## 一 实验目的

1. 理解二叉树的类型定义与性质。
2. 掌握二叉树的二叉链表存储结构的表示和实现方法。
3. 掌握二叉树遍历操作的算法实现。
4. 熟悉二叉树遍历操作的应用。

## 二 实验内容及要求

### 实验内容：

采用二叉链表存储，实现二叉树的创建、遍历（递归）、赫夫曼编码和译码等典型操作。

#### 1. 编程实现如下功能：

(1) 假设二叉树的结点值是字符型，根据输入的一棵二叉树的完整先序遍历序列（子树空用‘#’表示），建立一棵以二叉链表存储表示的二叉树。

(2) 对二叉树进行先序、中序和后序遍历操作，并输出遍历序列，观察输出的序列是否与逻辑上的序列一致。

(3) 主程序中要求设计一个菜单，允许用户通过菜单来多次选择执行哪一种遍历操作。

#### 2. 编程实现如下功能：

(1) 建立由英文字符组成的文件 f1（字符种类 $\geq 10$ ，长度 $\geq 100$ ），并统计不同字符出现的次数；

(2) 按字符出现的次数对其建立哈夫曼树，并求出各个字符的哈夫曼编码；

(3) 读入要编码的文件 f1，编码后存入另一个文件 f2；

(4) 接着再调出编码后的文件 f2，对其进行译码输出，最后存入文件 f3。

### 实验要求：

1. 键盘输入数据；
2. 屏幕输出运行结果。
3. 要求记录实验源代码及运行结果。
4. 运行环境：CodeBlocks/Dev c++/VC6.0 等 C 编译环境

## 三 实验过程及运行结果

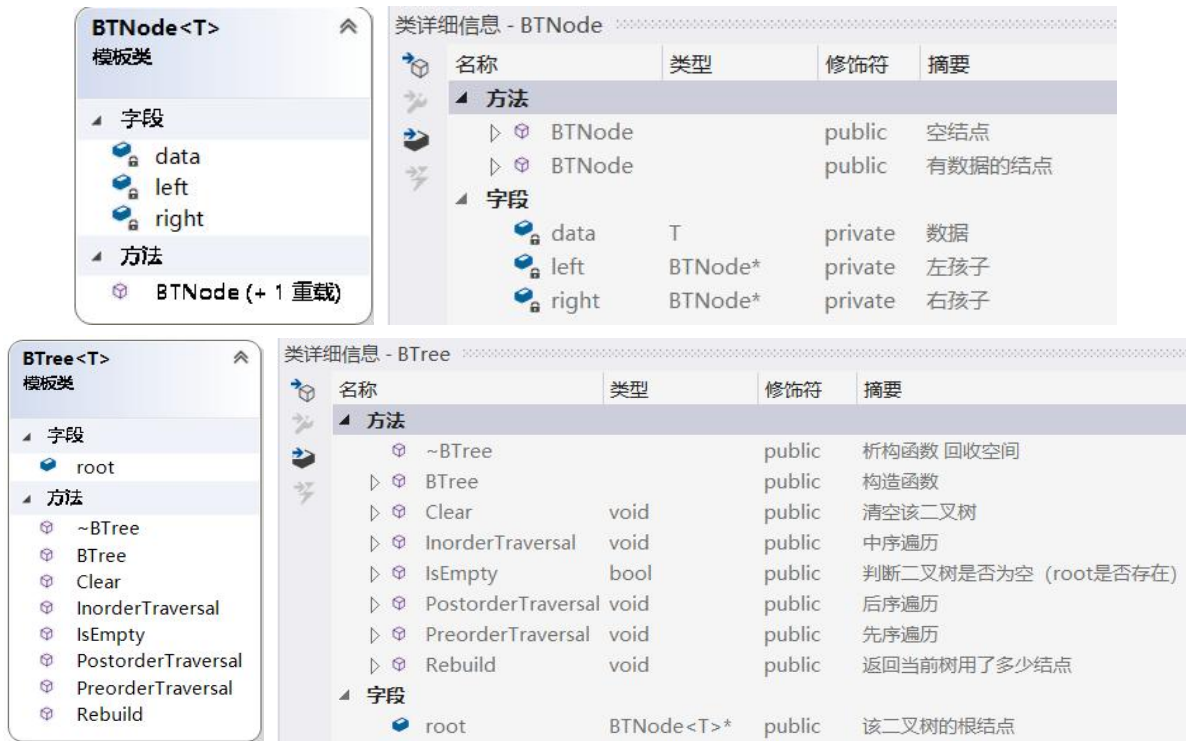
题 1：根据输入的一棵二叉树的完整先序遍历序列（子树空用‘#’表示），建立一棵以二叉链表存储表示的二叉树，并进行先序、中序、后序遍历。

### (1) 整体设计

采用 c++ 语言，一是实现二叉树基本操作：清空二叉树（Clear）、判断树空（IsEmpty）、先

序遍历（PreorderTraversal）、中序遍历（InorderTraversal）、后序遍历（PostorderTraversal）。二是扩展操作：根据完整先序序列重建二叉树（Rebuild）。封装一个模板类 BTreeNode 表示结点的定义、封装另一个模板类 BTree 作为二叉树操作类，存放头结点，初始化和销毁操作均可在 BTree 类的构造函数和析取函数中实现。

类图：



## (2) 具体实现

### 1. 构造函数和析构函数

```
BTree():root(nullptr) {};           //构造函数 初始化，根节点为空，还没有任何元素
~BTree() { Clear(); };              //析构函数 清空二叉树，回收空间，相当于手动调用了一次 Clear
```

### 2. 判断树空 (IsEmpty)

核心思想：无根结点既为空 shutdown，返回 true,反之返回 false

核心代码：

```
if (!root) return true;

else return false;

if (R) {
    if (R->left) Clear(R->left);
    if (R->right) Clear(R->right);
    delete R;
}
```

### 4. 先序遍历 (PreorderTraversal)

核心思想：递归思想，根左右，先访问根节点输出其值，再访问左子树、右子树

核心代码：

```
if (R) { //根左右
```

```

    cout << R->data; //先访问根节点 进行访问或其他操作
    PreorderTraversal(R->left);
    PreorderTraversal(R->right);
}

```

## 5. 中序遍历 (InorderTraversal)

核心思想：递归思想，左根右，先访问左子树，再访问根节点、右子树

核心代码：

```

if (R) { //左根右
    InorderTraversal(R->left);
    cout << R->data; //访问根节点 进行访问或其他操作
    InorderTraversal(R->right);
}

```

## 6. 后序遍历 (PostorderTraversal)

核心思想：递归思想，左右根，先访问左子树，再访问右子树、最后访问根结点。

核心代码：

```

if (R) { //左右根
    PostorderTraversal(R->left);
    PostorderTraversal(R->right);
    cout << R->data; //最后访问根节点 进行访问或其他操作
}

```

## 7. 根据完整先序序列重建二叉树 (Rebuild)

注意：函数原型为 `void Rebuild(BTNode<T>** R)`、

参数为一个二级指针，这是为了避免传空指针！

核心思想：跟着输入序列构建结点便可，若输入#则为空结点，不为#则申请空间，建立新结点。

核心代码：

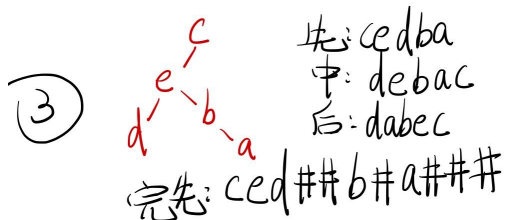
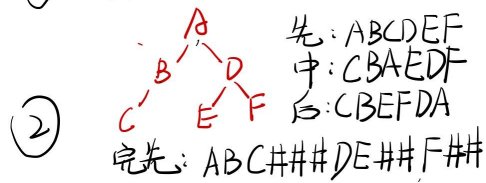
```

char ch;
//scanf("%c", &ch);
ch = getchar();
if (ch == '#') *R = nullptr;
else {
    (*R) = new BTNode<T>;
    (*R)->data = ch;
    Rebuild(&((*R)->left));
    Rebuild(&((*R)->right));
}

```

### (3) 测试数据

① # 空树



(4) 运行结果

```
C:\Users\whx\Desktop\201916010728余思娴实验3\BinaryTree\Debug\BinaryTree.exe
Please input complete preorder traversal sequence:
#
Successfully Rebuild!
You can do the following operation
1 Rebuild By PreorderTraversal
2 PreorderTraversal
3 InorderTraversal
4 PostOrderTraversal
5 Exit
Please chose:2
The preorder traversal sequence is:The BinaryTree is Empty!

Press Enter to continue...

Please chose:3
The inorder traversal sequence is:The BinaryTree is Empty!

Press Enter to continue...

Please chose:4
The postorder traversal sequence is:The BinaryTree is Empty!

Press Enter to continue...
```

图 1 测试数据 1 结果

```
C:\Users\whx\Desktop\201916010728余思娴实验3\BinaryTree\Debug\BinaryTree.exe
Please chose:1
Please input complete preorder traversal sequence:
ABC###DE##F##
Successfully Rebuild!
Press Enter to continue...

Please chose:2
The preorder traversal sequence is:ABCDEF
Press Enter to continue...

Please chose:3
The inorder traversal sequence is:CBAEDF
Press Enter to continue...

Please chose:4
The postorder traversal sequence is:CBEFDA
Press Enter to continue...
```

图 2 测试数据 2 结果

```
C:\Users\whx\Desktop\201916010728余思娴实验3\BinaryTree\Debug\BinaryTree.exe
Please chose:1
Please input complete preorder traversal sequence:
ced##b#q###
Successfully Rebuild!
Press Enter to continue...

Please chose:2
The preorder traversal sequence is:cedbq
Press Enter to continue...

Please chose:3
The inorder traversal sequence is:debqc
Press Enter to continue...

Please chose:4
The postorder traversal sequence is:dqbec
Press Enter to continue...
```

图 3 测试数据 3 结果

## 2. 编程实现如下功能:

- (1) 建立由英文字符组成的文件 f1 (字符种类 $\geq 10$ , 长度 $\geq 100$ ), 并统计不同字符出现的次数;
- (2) 按字符出现的次数对其建立哈夫曼树, 并求出各个字符的哈夫曼编码;
- (3) 读入要编码的文件 f1, 编码后存入另一个文件 f2;
- (4) 接着再调出编码后的文件 f2, 对其进行译码输出, 最后存入文件 f3。

### (1) 整体设计

这里我用结构数组来存储哈夫曼树, 所以数组下标即为索引, 定义宏 Null 为 -1 表示不存在。然后由于哈夫曼树每次要选取两个最小的结点合并后将新的结点作为父结点, 故空结点要保证不被一开始选到权值要尽可能大。整体上, 首先统计 f1 中不同字符出现次数, 读取文件 f1 返回一个存储了字符和其出现次数的 map 映射 ch (Readfile1), 封装一个函数展示各字符出现次数 (PrintFrequency), 然后将 ch 作为参数传入, 对其建立哈夫曼树并返回根节点下标 (Huffman) 给出该哈夫曼树的先序和中序遍历 (PreOrderPrintHuffman、InOrderPrintHuffman), 根据哈夫曼树求出各个字符的哈夫曼编码存在 (ComputingHuffmanCode), 展示所有字符对应的哈夫曼编码 (PrintHuffmanCode)。将 f1 读入, 编码后存入 f2 (StoreAfterCoding)。将 f2 读入, 解码后存入 f3 (GeneratedAfterDecoding)

#### ① 一些宏定义

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <map>
#include <queue>
#define f1_Name "f1.txt"
#define f2_Name "f2.txt"
#define f3_Name "f3.txt"
#define Error -1
#define Null -1
#define MaxSize 1000
#define MaxFrequency 200
using namespace std;
typedef int Position;
```



## ② 哈夫曼结点定义

```
struct HNode {
    int weight; //该结点的权值 出现频率越高 权值越大
    char data; //该结点实际代表的字符, '#' 表示该结点只有权值, 无需要编码的字符
    Position left; //指向左孩子 -1表示无
    Position right; //指向右孩子 -1表示无
    Position parent; //父结点的下标 -1表示无
    HNode() :weight(MaxFrequency + 1), data('#'), left(Null), right(Null), parent(Null) {}
    //构造函数 权值为Null (-1) 说明无这个节点
} *H;
```

## ③ 各函数原型及主程序展示

```
25 map<char, string> HuffmanCode; //各字符对应的哈夫曼编码
26 map<string, char> DeHuffmanCode; //各哈夫曼编码 对应的字符
27 //读取f1文件 返回f1中字符出现频率
28 map<char, int> Readfile1() { ... }
49
50 //计算有多少种字符
51 int CountSize(map<char, int> ch) { ... }
58
59 //展示各字符出现次数
60 void PrintFrequency(const map<char, int> m) { ... }
71
72 //将字符及其频率作为参数传入 构造出哈夫曼树 返回根节点下标 无则返回Null(-1)
73 Position Huffman(map<char, int> ch) { ... }
118
119 //先序遍历展示哈夫曼树
120 void PreOrderPrintHuffman(Position root) { ... }
128
129 //中序遍历展示哈夫曼树
130 void InOrderPrintHuffman(Position root) { ... }
138
139 //计算哈夫曼编码 递归到叶子结点 参数path为根结点到这个结点的路径 (左0右1)
140 void ComputingHuffmanCode(Position root, string path) { ... }
153
154 //展示各字符哈夫曼编码
155 void PrintHuffmanCode() { ... }
162
163 //读入f1文件, 全部编码后存入另一个文件f2;
164 void StoreAfterCoding() { ... }
186
187 //读入f2文件, 全部解码后存入另一个文件f3;
188 void GeneratedAfterDecoding() { ... }
207
208 int main() {
209     map<char, int> ch;
210     //(1)建立由英文字符组成的文件f1 (字符种类≥10, 长度≥100), 并统计不同字
211     ch = Readfile1();
212     PrintFrequency(ch);
213
214     //(2)按字符出现的次数对其建立哈夫曼树, 并求出各个字符的哈夫曼编码
215     Position root = Huffman(ch);
216     cout << "先序遍历展示哈夫曼树" << endl;
217     PreOrderPrintHuffman(root);
218     cout << "\n中序遍历展示哈夫曼树" << endl;
219     InOrderPrintHuffman(root);
220     string str;
221     ComputingHuffmanCode(root, str);
222     PrintHuffmanCode();
223
224     //(3)读入要编码的文件f1, 编码后存入另一个文件f2
225     StoreAfterCoding();
226
227     //(4)读入要解码的文件f2, 解码后存入另一个文件f3
228     GeneratedAfterDecoding();
229     return 0;
230 }
```

## (2) 具体实现

### 1. 读取 f1 文件 返回 f1 中字符出现频率 (Readfile1)

核心思想：读文件时利用 `getline` 函数读一行到 `string` 类型的 `str` 里，再对 `str` 里每一个字符进行处理（若不存在则新插入，存在则出现次数++）

核心代码：

```
map<char, int> ch;
string str;
fstream f1;
f1.open(f1_Name, ios::in);
if (!f1) {
    cout << "f1.txt文件打开失败!" << endl;
    return ch;
}
while (getline(f1, str)) {
    int len = str.length();
    for (int i = 0; i < len; ++i) {
        if (ch.end() == ch.find(str[i]))
            ch.insert(make_pair(str[i], 1));
        else
            ++ch[str[i]];
    }
}
f1.close();
return ch;
```

## 2. 计算 ch 共统计了多少种字符（CountSize）

核心代码：

```
int cnt = 0;
for (auto& i : ch) { //前size个节点 存数据和权值
    ++cnt;
}
return cnt;
```

## 3. 展示各字符出现次数（PrintFrequence）

核心代码：

```
cout << "字符\t出现次数" << endl;
int cnt = 0, sum = 0;
for (const auto& i : m) {
    cout << i.first << '\t' << i.second << endl;
    ++cnt;
    sum += i.second;
}
cout << "共有" << cnt << "种字符。" << endl;
cout << "共有" << sum << "个字符。" << endl;
```

#### 4. 将字符及其频率作为参数传入 构造出哈夫曼树 返回根节点下标 无则返回 Null(-1) (Huffman)

核心思想: 利用 CountSize 函数计算出有 size 个字符需要进行编码, 则需要  $2 * \text{size} - 1$  个结点的空间, 因为要进行 size-1 次合并, 每次选择两个权值最小且无父结点的结点的合并, 所以要特判 0 个元素和 1 个元素是的情况, 合并完后, 找到根节点下标并返回。

核心代码:

```
int size = CountSize(ch); //ch中共统计了多少种字符
if (size == 0) return Null;
if (H) delete[] H;
H = new HTNode[2 * size - 1];
int cnt = 0;
for (auto& i : ch) { //前size个节点 存数据和权值
    H[cnt].data = i.first;
    H[cnt].weight = i.second;
    ++cnt;
}
if (size == 1) return 0; //只有一个元素
Position EmptyNode = cnt; //最前边的空节点下标
for (int k = 1; k <= size - 1; ++k) { //size-1次合并 每次合并选两个在所有无父节点中权值最小的节点
    int min1 = Null, min2 = Null; //最小权值节点下标、次小权值节点下标
    int minw1 = MaxFrequence + 1, minw2 = MaxFrequence + 1; //最小权值1、次小权值2
    for (int i = 0; i < 2 * size - 1; ++i) { //找两个最小无父节点的节点
        if (H[i].parent != Null) continue; //有父节点了不用看
        if (H[i].weight < minw1) { //当前节点权值比最小值小
            minw2 = minw1;
            min2 = min1;
            minw1 = H[i].weight;
            min1 = i;
        } else if (H[i].weight < minw2) { //比次小值小
            minw2 = H[i].weight;
            min2 = i;
        }
    }
    //合并操作
    H[min1].parent = EmptyNode;
    H[min2].parent = EmptyNode;
    H[EmptyNode].left = min1;
    H[EmptyNode].right = min2;
    H[EmptyNode].weight = H[min1].weight + H[min2].weight;
    H[EmptyNode].parent = Null;
    EmptyNode++;
}
Position root = 0;
while (H[root].parent != -1) {
    root = H[root].parent; //有父节点的时候循环继续, 找到根节点即当前节点没有父节点的时候循环退出
}
return root;
```



## 5. 先序遍历、中序遍历展示哈夫曼树(PreOrderPrintHuffman、InOrderPrintHuffman)

核心思想：递归

核心代码：

先序遍历

```
if (root != Null) {
    cout << H[root].data << ": " << H[root].weight << endl;
    if (H[root].left != Null) PreOrderPrintHuffman(H[root].left);
    if (H[root].right != Null) PreOrderPrintHuffman(H[root].right);
}
else cout << "The HuffmanTree is Empty" << endl;
```

中序遍历

```
if (root != -1) {
    if (H[root].left != -1) InOrderPrintHuffman(H[root].left);
    cout << H[root].data << ": " << H[root].weight << endl;
    if (H[root].right != -1) InOrderPrintHuffman(H[root].right);
}
else cout << "The HuffmanTree is Empty" << endl;
```

## 6. 计算哈夫曼编码（左0右1）（ComputingHuffmanCode）

核心思想：递归到叶子结点 参数 path 为根结点到这个结点的路径，注意这里将反编码也求下方便后边译码

核心代码：

```
if (root == Null)
    cout << "Fail to computing! The HuffmanTree is Empty" << endl;
if (H[root].left == Null && H[root].right == Null) { //为叶子结点
    HuffmanCode[H[root].data] = path;
    DeHuffmanCode[path] = H[root].data; //这里将哈夫曼编码对应的字符也存下方便译码
    return;
}
string lcode = path + '0';
string rcode = path + '1';
if (H[root].left != Null) ComputingHuffmanCode(H[root].left, lcode);
if (H[root].right != Null) ComputingHuffmanCode(H[root].right, rcode);
```

## 7. 展示各字符哈夫曼编码

```
cout << "字符\t哈夫曼编码" << endl;
int cnt = 0, sum = 0;
for (const auto& i : HuffmanCode) {
    cout << i.first << '\t' << i.second << endl;
}
```

## 8. 读入 f1 文件，全部编码后存入另一个文件 f2 (StoreAfterCoding)

核心思想：利用先前求好的 HuffmanCode，存入另一个文件的时候加个空格方便译码时读出

核心代码：

```
fstream f1, f2;
f1.open(f1_Name, ios::in);
f2.open(f2_Name, ios::out);
if (!f1) {
    cout << "f1.txt文件打开失败!" << endl;
    return;
}
if (!f2) {
    cout << "f2.txt文件打开失败!" << endl;
    return;
}
string str;
while (getline(f1, str)) {
    int len = str.length();
    for (int i = 0; i < len; ++i) {
        f2 << HuffmanCode[str[i]] << ' '; //存的时候加个空格方便译码
    }
}
f1.close();
f2.close();
```

## 9. 读入 f2 文件，全部解码后存入另一个文件 f3 (GeneratedAfterDecoding)

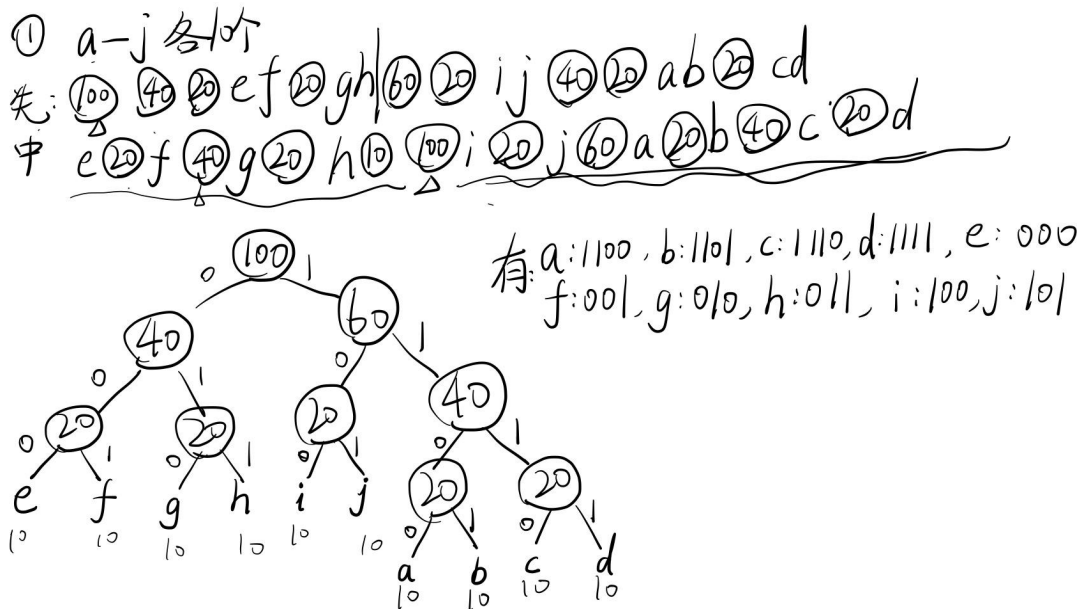
核心思想：利用先前求好的 DeHuffmanCode，存入另一个文件的时候加个空格方便译码时读出

核心代码：

```
fstream f2, f3;
f2.open(f2_Name, ios::in);
f3.open(f3_Name, ios::out);
if (!f2) {
    cout << "f2.txt文件打开失败!" << endl;
    return;
}
if (!f3) {
    cout << "f3.txt文件打开失败!" << endl;
    return;
}
string str;
while (f2 >> str) {
    f3 << DeHuffmanCode[str];
}
f2.close();
```

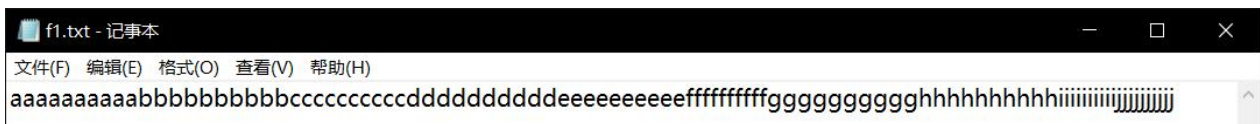
f3.close();

### (3) 测试数据



### (5) 运行结果

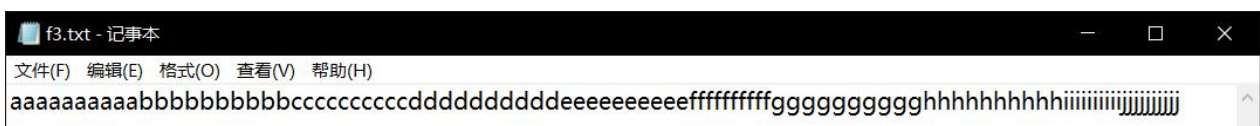
f1 文件



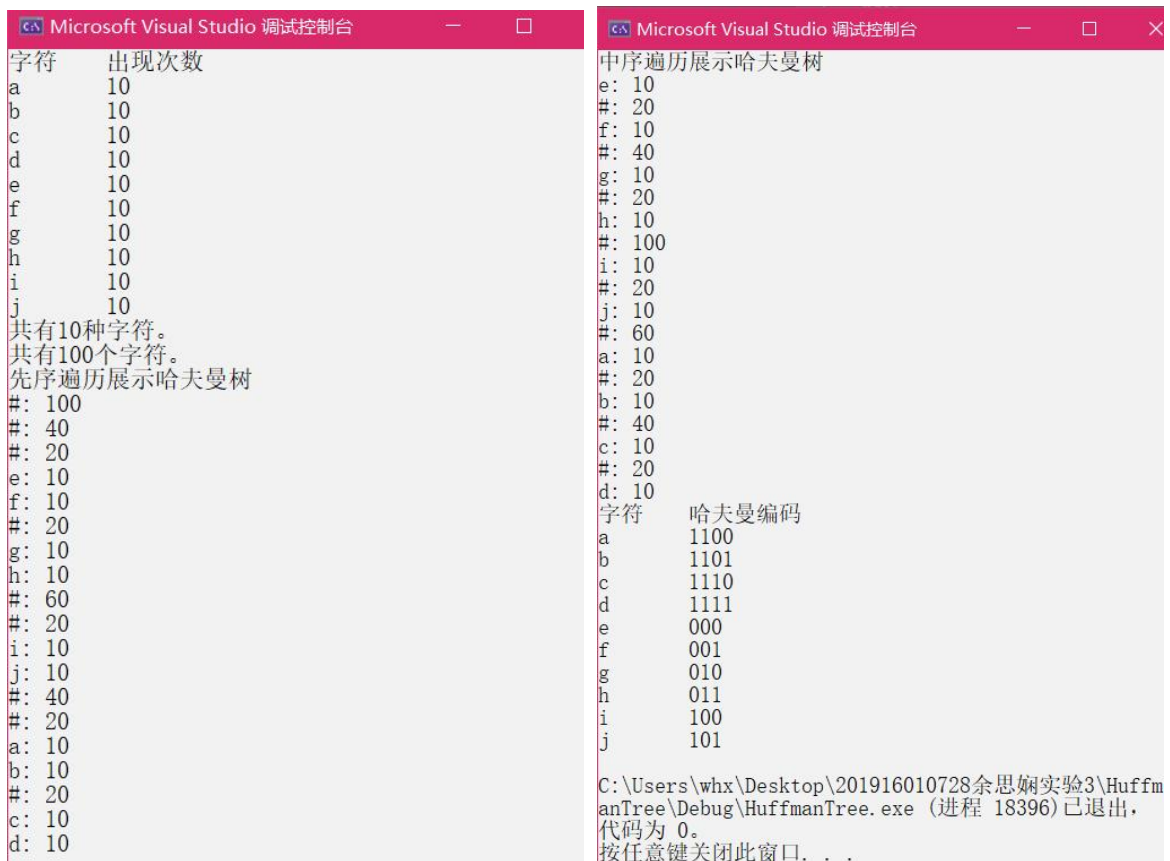
f2 文件

```
1100 1100 1100 1100 1100 1100 1100 1100 1100 1100 1101 1101 1101 1101 1101 1101 1101 1101 1101
1110 1110 1110 1110 1110 1110 1110 1110 1110 1110 1111 1111 1111 1111 1111 1111 1111 1111 1111
000 000 000 000 000 000 000 000 000 000 001 001 001 001 001 001 001 001 001 010 010 010 010
010 010 010 010 010 010 011 011 011 011 011 011 011 011 011 011 011 100 100 100 100 100 100
100 100 101 101 101 101 101 101 101 101 101 101
```

f3 文件



## 输出结果



```
Microsoft Visual Studio 调试控制台
字符    出现次数
a       10
b       10
c       10
d       10
e       10
f       10
g       10
h       10
i       10
j       10
共有10种字符。
共有100个字符。
先序遍历展示哈夫曼树
#: 100
#: 40
#: 20
e: 10
f: 10
#: 20
g: 10
h: 10
#: 60
#: 20
i: 10
j: 10
#: 40
#: 20
a: 10
b: 10
#: 20
c: 10
d: 10

Microsoft Visual Studio 调试控制台
中序遍历展示哈夫曼树
e: 10
#: 20
f: 10
#: 40
g: 10
#: 20
h: 10
#: 100
i: 10
#: 20
j: 10
#: 60
a: 10
#: 20
b: 10
#: 40
c: 10
#: 20
d: 10
字符    哈夫曼编码
a       1100
b       1101
c       1110
d       1111
e       000
f       001
g       010
h       011
i       100
j       101

C:\Users\whx\Desktop\201916010728余思娴实验3\HuffmanTree\Debug\HuffmanTree.exe (进程 18396) 已退出，
代码为 0。
按任意键关闭此窗口。 . . .
```

## 四 调试情况、设计技巧及体会

此次实验相对前几次来说难度提升不小，第一题的完整先序序列重建二叉树等基本操作里遇到了几个 bug，比如传空指针导致的最后什么也没有修改，所以传了个二级指针避免此类情况。本次实验的第二题难度略大，为了方便在哈夫曼树节点中增加了指向父结点的指针，并且运用了很多 STL 中的 map 进行键值对存储，方便编码译码，顺便复习了下 c++ 的文件流操作。所有代码均在 VS2019，gcc9.2.0 下调试通过。