1. **均值方差**
2. **ip 地址划分，判断是否合法，构造函数中参数为 ip 地址字符串**

如 192.168.15.1 是属于 C 类 ip 地址

关键在于:

1.怎么把字符串分割成四小段:使用字符串中.find 函数找到"."的位置，使用.substr()提取"."之前的字符串并将其转换成整型，存储到数组中

2.怎么将这四小段字符串转换成整型：使用模板函数，函数中使用 istringstream流，绑定字符串，然后将其输出到整型的变量中去

3.然后分别把这四小段整型元素判断条件就可以了：也就是数组中元素是否都在0-255 之间。

若合法那么数组中元素依次判别，确定 ip 的类型：A,B,C,D,E

注意：在构造函数中传入的参数要设置成 const string& ip,而 不是 string& ip，不知道为什么不设置不对

```cpp
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

// 模板函数用来将字符串类型转换成整型，返回值就是转换的类型值
template <class T>
T fromString(const string& str) {
    // 使用 istringstream 来绑定当前的字符串 str，导入头文件 sstream
    istringstream in(str);
    T temp; // 需要转换成的临时变量
    in >> temp; // 将这个字符串输入到 T 类型的变量中，完成转换
    return temp;
}

// 创建 Ip 对象
class Ip {
public:
    // 构造函数初始化数据成员 a[4]，也就是我们要进行分割字符串了
    Ip(const string& ip) {
        getIntIp(ip); // 将分割的实现细节隐藏起来，遵循信息隐藏原则
    }

    // 测试函数:是否完成分割，也就是整型数组中是否有数字了
    void show() const {
        for (int i = 0; i < 4; i++) {
            cout << a[i] << endl;
        }
    }

    // 判断 ip 地址是否合法：也就是数组中每一位数字是否在 0 - 255 之间,返回值为 bool
    bool isLegal() const {
        for (int i = 0; i < 4; i++) {
            if (a[i] < 0 || a[i] > 255) {
                return false;
            }
        }
        return true;
    }

    // 查看是哪种类型的 ip 地址
    void type() const {
```

```
        if (isLegal()) {
            if (a[0] >= 1 && a[0] <= 127) {
                cout << "A 类地址！" << endl;
            } else if (a[0] >= 128 && a[0] <= 191) {
                cout << "B 类地址！" << endl;
            } else if (a[0] >= 192 && a[0] <= 223) {
                cout << "C 类地址！" << endl;
            } else if (a[0] >= 224 && a[0] <= 239) {
                cout << "D 类地址！" << endl;
            } else if (a[0] >= 240 && a[0] <= 255) {
                cout << "E 类地址！" << endl;
            }
        } else {
            cout << "该 IP 地址不合法，无法判断类型。" << endl;
        }
    }

private:
    // 确定数据成员
    int a[4]; // 存储分割后的数字(字符串要转换成整型)

    // 遵循信息隐藏原则，隐藏转换的实现细节
    void getIntIp(const string& ip) {
        string str = ip; // 存储分割下来的字符串，当前为原字符串
        int count = 0;   // 记录点号的数量

        for (int i = 0; i < 4; i++) {
            if (i == 3) {
                a[i] = fromString<int>(str); // 将字符串类型转换成整型，使用模板函数 fromString
            } else {
                int location = str.find(".");
                if (location == string
```

3. 赫夫曼编码:符号出现频率高的用较短的编码，反之用长的。目的就是提高通信传输的效率，压缩文件大小，使编码之后的字符串长度降低，期望值降低，达到无损压缩数据

BABACACADADABBCBABEBEDDABEEEBB

1 首先计算出每个字符出现的次数（概率）：B:10 A:8 C:3 D:4 E:5

2 把出现次数（概率）最小的两个相加，并作为左右子树，重复此过程，直到概率值为 1

第一次：将概率最低值 3 和 4 相加，组合成 7：

第二次：将最低值 5 和 7 相加，组合成 12：

第三次：将 8 和 10 相加，组合成 18：

第四次：将最低值 12 和 18 相加，结束组合：

3 将每个二叉树的左边指定为 0，右边指定为 1

4 沿二叉树顶部到每个字符路径，获得每个符号的编码

我们可以看到出现次数（概率）越多的会越在上层，编码也越短，出现频率越少的就越在下层，编码也越长。

当我们编码的时候，我们是按“bit”来编码的，解码也是通过 bit 来完成，如果我们有这样的 bitset “10111101100" 那么其解码后就是 “ABBDE”。所以，我们需要通过这个二叉树建立我们 Huffman 编码和解码的字典表。

设计：

（1）内部类 HuffNode，储存的是结点的一些属性：例如权值、左右孩子是谁(前提要有)、结点代表的字符、双亲是谁

（2）类 HuffTree：私有数据成员：创建大小为 2*128 的内部类 HuffNode 的对象数组，那么每个元素就是一个对象，也就是一个结点了，拥有内部类的属性整型数组 count 来记录客户输入字符串中每个字符出现的频率，而这个频率就是权值。ASCII 码对应的码值就是元素下标的值客户输入的字符串 str、记录一共多少个叶子结点的 m

（3）默认构造函数：初始化结点数组以及整型数组 count，默认结点 m=128 个（4）输入函数 input()：客户输入要编码的字符串，顺便统计每个字符出现的次数，也就是权值，存放在 count 数组中

（5）创建赫夫曼树函数 createTree：首先创建 i 个叶子结点，也就是 m 的值，数组 count 的值就是权值，下标就是每个结点代表的字符所有叶子结点创建完毕，就开始创建结点，也就是逐步生成二叉树，遍历结点数组，找权值最小的两个结点组成一个新的结点。

（6）找权值最小的两个结点在数组中的位置 select()：思想就是打擂台算法，先放两个最大的变量，依次遍历结点数组的权值，进行比对前提是此结点没有双亲，也就是 parent=0.否则是已经生成过的了

（7）遍历二叉树了，输出对应的二进制编码：用二维指针 char 来存储每个字符的编码。刚开始申请 m 个指针，然后每个指针指向的列根据编码的长度来申请内存从二叉树的头开始遍历，也就是结点数组的最后一个元素就是头！！，p=2*m-1

if 判断是遍历左边还是右边，若遍历了左边，用权值 weight 来=1 表示已经遍历过了，下次不用再遍历了。（注：weight 权值只是在创建最优二叉树的时候要根据权值来创建使用，此时已经不再使用了）若遍历的左边，那么对于的编码就是 0，将它存储在临时的 char 型数组中，等遍历到叶子结点，也就是末尾了，也就是结点已经没有左右孩子了，再把这个临时的编码数组赋值给二维的那个指针数组最后，也就是 p=0 的时候，循环也就结束了，将二维数组的编码值依次输出出来，也就是每个结点对应的编码

```
#ifndef _HUFFNODE_H_
#define _HUFFNODE_H_

// 1. 创建结点类: 属性有权值、左右孩子、代表的字符（注意只有叶子结点才会有代表的字符）
class HuffNode {
public:
    int weight;  // 权值
    int parent;  // 双亲
    int lchild;  // 左孩子
    int rchild;  // 右孩子
    char c;      // 结点代表的字符
};

#endif

/*
创建赫夫曼树
*/
#ifndef _HUFFTREE_H_
#define _HUFFTREE_H_

#include <iostream>
#include <string>
#include "huffNode.h"
#define MAX 128
using namespace std;

class HuffTree {
public:
    HuffTree() {
        int m = 128;  // 默认叶子结点为 128 个
        // 初始化结点数组全部属性
        for (int i = 1; i <= (2 * MAX - 1); i++) {
            huff[i].weight = 0;
            huff[i].lchild = 0;
            huff[i].rchild = 0;
            huff[i].parent = 0;
        }
        for (int j = 0; j < MAX; j++) {
```

```
            count[j] = 0;
        }
    }

    ~HuffTree() {}

    void input();
    void creatTree();
    void LeafCode();

private:
    // 创建结点数组：存储每一种字符对应的结点，从下标 1 的位置开始，固不是 2*MAX-1
    HuffNode huff[2 * MAX];
    int count[MAX];
    int m;        // 实际叶子结点数
    string str;  // 输入的字符串

    // 找最小权值的过程可以隐藏起来
    void select(int, int&, int&);
};

// 采用非递归方法遍历最优二叉树，求 n 个叶子的编码
void HuffTree::LeafCode() {
    // HC 就好比如是个二维的指针数组，每一维对应一个字符的编码
    char** Hc;
    // m 是实际的叶子结点数，但是实际的结点数是 2*m-1，因为还有很多包含没有代表字符的结点，也就是桥梁的结点存在
    // 虽然我们申请了 2*MAX 的结点数组，但是显然到目前为止我们只用到了 2*m-1 个的结点
    int i, p = 2 * m - 1, cdlen = 0;  // p 就是实际的结点数组的大小
    char code[30];                    // 每个结点对应的二进制码

    // Hc =(char**)malloc((m+ 1) * sizeof(char *)); // 确定申请多少维的内存，也就是知道了有多少个叶子结点
    Hc = new char*[m + 1];

    // 但是每一维对应的多少列是不确定的，因为编码的长度是不同的，所以要随机申请
    for (i = 1; i <= p; i++) {
        // 权值已经不再使用了，权值只是在创建最优二叉树的时候使用
        huff[i].weight = 0;  // 遍历二叉树时用作结点的状态标志
    }

    while (p)  // 挨个结点遍历最优二叉树，求 n 个叶子的编码
    {
        if (0 == huff[p].weight)  // weight=0 表明左边没有被遍历
        {
            huff[p].weight = 1;  // 标记已经开始遍历左边
            if (huff[p].lchild != 0)  // 该结点有左孩子
            {
                p = huff[p].lchild;  // 存在左孩子，
                // 关键！：此时 p 的位置就是左孩子的位置，那么下一次循环 p 从当前孩子继续向下遍历，直到叶子结点末尾
                code[cdlen++] = '0';
            }
            // 这一步是直到遍历到最后叶子结点才会做的！就是把编码的二进制输出出来
            else if (0 == huff[p].rchild)  // 叶子结点也没有右孩子，已走到叶子结点，记下该叶子结点的字符的编码
            {
                // 申请当前维的内存，也就是用了多少列
                Hc[p] = new char[cdlen + 1];
                code[cdlen] = '\0';
                strcpy(Hc[p], code);
            }
        } else if (1 == huff[p].weight)  // 向右走：此时说明左孩子已经遍历完毕，开始右孩子
        {
            huff[p].weight = 2;
```

```
            if (huff[p].rchild != 0)  // 该结点有右孩子
            {
                p = huff[p].rchild;  // 若该右孩子还有左右孩子，那么就是 p 不会是 0，循环继续走下去
                code[cdlen++] = '1';
            }
        } else  // 2 == Ht[p].weight, 回退
        {
            huff[p].weight = 0;
            p = huff[p].parent;
            cdlen--;
        }
    }  // end of while

    // 输出每个叶子结点对应的编码：
    for (i = 1; i <= m; i++) {
        cout << "字符" << huff[i].c << "的编码为" << Hc[i] << endl;
    }

    // 释放内存空间
    /*for(i=0;i<m+1;i++)
    {
        delete[] Hc[i];
    }*/
    delete[] Hc;
}

void HuffTree::select(int n, int& left, int& right) {
    // n 就是当前结点数组的结点数，left 为最小的结点的下标位置，right 为第二小结点的下标位置
    // 打擂台算法
    int min1;
    int min2;
    min1 = min2 = 32767;
    left = right = 1;
    for (int i = 1; i <= n; i++) {
        if (huff[i].parent == 0 && huff[i].weight < min1) {
            min2 = min1;
            right = left;
            min1 = huff[i].weight;
            left = i;
        } else if (huff[i].parent == 0 && huff[i].weight < min2) {
            min2 = huff[i].weight;
            right = i;
        }
    }
}

void HuffTree::creatTree() {
    int i, k, left = 0, right = 0;
    // 先分配输入的字符所对应的结点，从结点数组 1 开始到一共输入多少字符结束，以及他的权值
    // 我们从结点数组 1 的位置开始存储对应的字符，而不是 0 的位置。
    // 所以创建的结点数组会浪费一块很大的内存，因为只有存储了全部的 128 个字符的时候才会用到这么大数组
    for (i = 1, k = 0; k < MAX; k++) {
        if (count[k] != 0) {
            huff[i].c = (char)k;           // 结点所代表的字符，而 k 为下标，就是 ASCII 码值，那么转换成对应的字符。如 65-->A
            huff[i].weight = count[k];     // 分配对应的权值，也就是字符串中一共出现了几次
            i++;                           // 此时结点数组中存储一个实际的有属性的结点，
            // 比如我们输入了 AAADDDSSS 字符串，那么其实结点数组目前只用了 3 个元素，也就是目前只有三个结点
        }
    }

```

```
    // m 就是用来储存一共有多少个叶子结点，也就是带字符含义的结点的个数
    m = i - 1;  // 最后 i++了，所以实际的结点数只有 i-1 个

    // 此时我们就开始形成新的结点了（两个权值最小的结点，形成一个新的结点），
    // 注意！不是叶子结点！，那么就需要从结点数组的没有存储结点的位置开始存储没有代表字符的结点了
    // 一共 m 个叶子结点，就会对应 2*m-1 个结点，注意是包含那 m 个叶子结点的，所以我们要从 m+1 的位置开始创建新的结点
    for (k = m + 1; k <= (2 * m - 1); k++) {
        // 从结点数组中 1 至 k-1 位置，寻找两个权值最小的结点，来当做当前结点的左右孩子
        // left，right 是对应的下标位置
        select(k - 1, left, right);
        // 左右两个孩子的双亲皆是刚生成的父结点 k
        huff[left].parent = k;
        huff[right].parent = k;
        // 父结点的左右两个孩子分别就是获取到的权值最小的两个叶子结点
        huff[k].lchild = left;
        huff[k].rchild = right;
        huff[k].weight = huff[left].weight + huff[right].weight;
    }
}

void HuffTree::input() {
    cout << "输入编码的字符串：" << endl;
    cin >> str;
    // 统计输入字符串中每个字符出现的次数
    for (int i = 0; i < str.length(); i++) {
        count[str[i]]++;  // str[i] 是对应的字符，但是会自动转换成对应的十进制数值，
        // 如 A---对应 65，那么就是 count[65]++
    }
}

#endif

/*
主函数测试
*/
#include "hufftree.h"
int main() {
    // BABACACADADABBCBABEBEDDABEEEBB
    HuffTree huff;
    huff.input();        // 输入要编码的字符串
    huff.creatTree();    // 根据字符串来创建赫夫曼树
    huff.LeafCode();     // 对字符进行编码，并输出对应的二进制码
    return 0;
}
```

4. 微信类