作业 PA4 实验报告

姓名：钱子贤 学号：2054170 日期：2021 年 11 月 18 日

实验报告格式按照模板来即可，对字体大小、缩进、颜色等不做要求

实验报告要求在文字简洁的同时将内容表示清楚

1. 涉及数据结构和相关背景

哈夫曼树:
（1）树的路径长度： 指从树根到树中每一个结点的路径长度之和。
（2）在结点数目相同的二叉树中，路径长度最短的是： 完全二叉树。
（3）结点的权： 在一些应用中，赋予树中结点的一个有某种意义的实数。
（4）带权路径长度（树的代价）：结点到树根的路径 乘以 该结点上的权，
通常记为:
注：完全二叉树：除最后一层可能不满以外，其他各层都达到该层节点的最大数，最后一层如果不满，该层所有节点都全部靠左排；
满二叉树：所有层的节点数都达到最大。
（5）哈夫曼树的原理:
在构造哈夫曼树的过程中，每次都是选取两棵最小权值的二叉树进行合并，因此使用的是贪心算法。
（6）给定结点序列<ci，pi>（ci 为编码字符，pi 为 ci 的频度），哈夫曼编码的过程如下:
.用字符 ci 作为叶子，pi 作为 ci 的权，构造一棵哈夫曼树，并将树中的左分支和右分支分别标记为 0 和 1；将从根到叶子的路径上的标号依次相连，作为该叶子所表示字符的编码。该编码为最优前缀码。.求哈夫曼编码的具体实现过程是依次以叶子结点 c[i]为出发点，向上回溯至根为止。
.由于生成的编码与要求的编码反序，将生成的编码先从后往前依次存放在一个临时串中，并设一个指针 start 指示编码在该串中的首位置。.当某字符编码完成时，从临时串的 start 处将编码复制到该字符相应的位串 bits 中即可。
.因为字符集大小为 n，故变长编码的长度不会超过 n，加上一个结束符“\0“，bits 的大小应为 n+1。
（7）前缀码：（不存在一个序列是另一个序列的前缀的情况的）一个序列的集合。
（8）最优的前缀码：平均码长或文件总长最小的前缀码。
（9）对文件的压缩效果最佳的编码：最优的前缀码。
（10）哈夫曼树的应用：可以优化通信管道的利用率。

2. 实验内容

PA4 二叉树的应用实验报告 > 二叉树的应用实验报告
哈夫曼编码和译码
实验目的 ：
理解最优二叉树，即哈夫曼树(Huffman tree)的概念，熟悉它的构造过程。
实验内容:
实现对 ASCII 字符文本进行 Huffman 压缩，并且能够进行解压。

将给定的文本文件使用哈夫曼树进行压缩，并解压。
基本要求为：能将文本文件压缩、打印压缩后编码、解压压缩后的文件。对实际使用的具体数据结构除必须使用二叉树外不做要求。
参考信息 ：
用一个二叉树表示哈夫曼树，因为 ASCII 表一共只有 127 个字符，可以直接使用数组来构造 Huffman 树。
leftChild 和 rightChild 分别表示当前结点左儿子和右儿子的下标。因为 1 到 127 分别表示与 ASCII 表的符号相对应，所以这里无需记录每个结点代表什么符号。

```
struct Node{
    int leftChild;
    int rightChild;
} tree[256];
```

而在解压的时候，由于不知道每个结点实际表示什么符号，所以需要在树上记录下。

```
struct NodeV{
    int leftChild;
    int rightChild;
    char c;
} tree[256];
```

实验要求 ：
(1) 程序要添加适当的注释，程序的书写要采用缩进格式 。
(2) 程序要具在一定的健壮性，即当输入数据非法时，程序也能适当地做出反应，如插入删除时指定的位置不对等等。
(3) 程序要做到界面友好，在程序运行时用户可以根据相应的提示信息进行操作。
(4) 根据实验报告模板详细书写实验报告，在实验报告中给出主要算法的复杂度分析。

```
};
typedef struct HTNode    //huffman树结构
{
    int count;
    WeightGkm w;
    string code;
    HTNode* lchild;
    HTNode* rchild;
}HTNode, * HTree;

class HuffmanGKM
{
private:
    HTree T;    //构造Huffman树；
    string huffCode[CHNUM];        //256个字符的Huffman编码；

    unsigned long weight[CHNUM];
    unsigned long long file_size;
```

```
    string original_file;
    string compress_file;
    string decompress_file;

    void QuickSortHT(HTree ht[], int left, int right);        //快速排序
    int Partition(HTree ht[], int left, int right);        //快速排序中的“分半”
    void SelectInsert(HTree ht[], HTree t, int left, int right);    //按序插入
public:

    HuffmanGKM(string originalFile, string compressFile, string decompressFile);

    int ReadFile();
    int BuildHuffTree();
    int CreateHuffCode();
    int CompressFile();
    int DecompressFile();    //根据huffman树解压缩huffman编码的压缩文件

    ~HuffmanGKM();
};

struct WeightGkm        //字符频度结构，包含频度和字符值
{
    unsigned long w;
    char c;


HuffmanGKM::HuffmanGKM(string originalFile, string compressFile, string decompressFile)
{
    for (int i = 0; i < CHNUM; i++)
        weight[i] = 0;

    file_size = 0;
    original_file = originalFile;
    compress_file = compressFile;
    decompress_file = decompressFile;

}

int HuffmanGKM::ReadFile()
{
    ifstream read;
    read.open(original_file);
    if (read.fail())
```

```
    {
        cout << "The original file open failed when read file!!";
        return 0;
    }

    char next;
    read.get(next);
    while (!read.eof())//统计频度。
    {
        weight[next + PLUS]++;
        read.get(next);
        file_size++;
    }

    read.close();
    return 0;
}

void HuffmanGKM::QuickSortHT(HTree htt[], int left, int right)
{
    int pivot;
    if (left < right)  // 肯定为真的条件
    {
        pivot = Partition(htt, left, right);
        QuickSortHT(htt, left, pivot - 1);
        QuickSortHT(htt, pivot + 1, right);
    }
```

给定结点序列<ci, pi> (ci 为编码字符, pi 为 ci 的频度), 哈夫曼编码的过程如下:
.用字符 ci 作为叶子, pi 作为 ci 的权, 构造一棵哈夫曼树, 并将树中的左分支和右分支分别标记为 0 和 1; 将从根到叶子的路径上的标号依次相连, 作为该叶子所表示字符的编码。该编码为最优前缀码。.求哈夫曼编码的具体实现过程是依次以叶子结点 c[i]为出发点, 向上回溯至根为止。

```
}
//快速排序的patition算法
int HuffmanGKM::Partition(HTree htt[], int left, int right)  //这是左大右小的排序
{
    HTree HTPivot = htt[left];            //这叫“虚左以待”

    while (left < right)
    {
        while (right > left && htt[right]->w.w >= HTPivot->w.w)
            right--;
        htt[left] = htt[right];
```

```
        while (left < right && htt[left]->w.w <= HTPivot->w.w)
            left++;
        htt[right] = htt[left];
    }

    htt[left] = HTPivot;

    return left;  //最后left=right，所以返回哪个都一样

}

void HuffmanGKM::SelectInsert(HTree htt[], HTree p, int left, int right)//left是第一个要比较的元素
{
    for (; left <= right; left++)
    {
        if (p->w.w > htt[left]->w.w)
            htt[left - 1] = htt[left];//左移小元素。
        else
            break;
    }
    htt[left - 1] = p;
}
```

.由于生成的编码与要求的编码反序，将生成的编码先从后往前依次存放在一个临时串中，并设一个指针 start 指示编码在该串中的首位置。.当某字符编码完成时，从临时串的 start 处将编码复制到该字符相应的位串 bits 中即可。

.因为字符集大小为 n，故变长编码的长度不会超过 n，加上一个结束符"\0"，bits 的大小应为 n+1。

```
int HuffmanGKM::BuildHuffTree()
{
    int left = 0, right = CHNUM - 1;
    HTree ht[CHNUM];    //树结点的排序数组

    for (int i = 0; i < CHNUM; i++)    //初始化huffman树结点
    {
        ht[i] = new HTNode;
        ht[i]->w.w = weight[i];    //字符频度
        ht[i]->count = 1;            //树中结点个数,仅做测试用。
        ht[i]->w.c = i - PLUS;        //字符值
        ht[i]->lchild = 0;
        ht[i]->rchild = 0;
    }
```

```
    QuickSortHT(ht, left, right);    //先把各结点字符按频度升序排序。

    HTree parent;
    while (left < right)    //建树
    {
        ht[left]->code = "1";
        ht[left + 1]->code = "0";
        parent = new HTNode;
        parent->lchild = ht[left];
        parent->rchild = ht[left + 1];

        parent->w.c = 0;
        parent->w.w = parent->lchild->w.w + parent->rchild->w.w;
        parent->count = parent->lchild->count + parent->rchild->count + 1;
        SelectInsert(ht, parent, left + 2, right);
        left++;
    }
    T = parent;    //T为建好的huffman树。
    return 0;
}

int HuffmanGKM::CreateHuffCode()
{
    //非递归后序遍历二叉树，访问叶子结点
    HTree stack[CHNUM];
    int sign[CHNUM] = { 0 };
    HTree p = T;
    int top = 0;

    while (p || top)
    {
        if (p)
        {
            stack[top] = p;
            sign[top] = 1;
            top++;
            p = p->lchild;
        }
        else // p为空指针，循环出栈
            while (top != 0)   //后序遍历中，当访问完一个结点时，则以该结点为根的树都访
问完，所以下一步应该继续出栈，
            {
                top--;
```

```
                p = stack[top];

                if (sign[top] == 2) //表示p的左右子树都已走过
                {
                    if (p->lchild == 0 && p->rchild == 0)
                        for (int i = 1; i <= top; i++)
                            huffCode[p->w.c + PLUS] += stack[i]->code;
                }
                else //表示仅走过T的左子树 ，右子树必定是第一次遇到，
                {
                    stack[top] = p;
                    sign[top] = 2;
                    top++;
                    p = p->rchild;
                    break;
                }//else if
            } //while ( !IsEmpty )

        if (top == 0)
            break;
    }//while
    return 0;
}
```

之后，统计输入文本中每个字符的频率。由于nodeArray中的元素只是指针，我们需要对没有创建的节点进行创建（即，申请内存）。

```
int HuffmanGKM::CompressFile()
{
    ifstream read;
    read.open(original_file);
    if (read.fail())
    {
        cout << "The original file open failed when compress! ! ! ";
        return 1;
    }
    ofstream write;
    write.open(compress_file, ios::binary);
    if (write.fail())
    {
        cout << "The compress files open failed when compress! ! ! ";
        return 1;
    }
    char next;
    unsigned char buff = 0;
    int count = 0;
```

```
    read.get(next);
    while (!read.eof())
    {
        for (string::size_type i = 0; i < huffCode[next + PLUS].size(); i++)
        {
            if (huffCode[next + PLUS][i] == '0')
                buff = (buff << 1);
            else
                if (huffCode[next + PLUS][i] == '1')
                    buff = (buff << 1) | 1;
            count++;
            if (count == 8)
            {
                write << buff;
                count = 0;
            }
        }
        read.get(next);
    }
    if (count != 0)
        for (; count != 8; count++)
            buff = (buff << 1);
    write << buff;
    read.close();
    write.close();

    return 0;
}
```

简单的说就是对文本进行重新编码，因为不同的字符出现频率不同，为了节约空间，我们完全可以把经常出现的设置编码比较短，很少出现的设置比较长，而不像很多编码，所有字符的长度都一样。所以这实际上需要先对文本中字符频率进行统计，然后根据统计结果生成哈夫曼树。因为哈夫曼树特有的性质，避免了前缀问题，也就说不会因为某个编码是另一个编码的前缀而又二义性，哈夫曼树的具体实现不再赘述。

```
HuffmanGKM::DecompressFile()
{
    ifstream read;
    read.open(compress_file, ios::binary);
    if (read.fail())
    {
        cout << "The compress file pen failed when decompress!!" << endl;
        return 0;
    }
```

```
    ofstream write;
    write.open(decompress_file);
    if (write.fail())
    {
        cout << "The decompress file open failed when decompress!!" << endl;
        return 0;
    }
    HTree p = T;

    char next;
    read.get(next);
    unsigned long long countSize = 0;
    while (1)
    {
        bitset<8>b(next);
        read.get(next);
        for (int i = b.size() - 1; i >= 0; i--)
        {
            if (b.test(i))
                p = p->lchild;
            else
                p = p->rchild;

            if (p->lchild == 0 && p->rchild == 0)
            {
                write << p->w.c;
                p = T;
                countSize++;
            }
            if (countSize >= file_size)
                break;
        }
        if (countSize >= file_size)
            break;
    }

    read.close();
    write.close();
    return 0;
}
```

过程：对于一段由编码0/1构成的二进制文件，在同个树而言，重第一个开始，遇到0则访问左子树，遇到1则访问右子树。直到该节点没有了左右节点为止，即这段01字符即可解码为该叶子节点所对应的字符。

HuffmanGKM::~HuffmanGKM()

```
{

    HTree stack[CHNUM];
    int sign[CHNUM] = { 0 };

    HTree p = T;
    int top = 0;
    while (p || top)
    {
        if (p)
        {
            stack[top] = p;
            sign[top] = 1;
            top++;
            p = p->lchild;
        }
        else // p为空指针，循环出栈
            while (top != 0)   //后序遍历中，当访问完一个结点时，则以该结点为根的树都访问完，所以下一步应该继续出栈，
            {
                top--;
                p = stack[top];

                if (sign[top] == 2) //表示p的左右子树都已走过 ,后序遍历，释放所有结点
                    delete(p);

                else //表示仅走过T的左子树 ，右子树必定是第一次遇到，
                {
                    stack[top] = p;
                    sign[top] = 2;
                    top++;
                    p = p->rchild;
                    break;
                }//else if
            } //while ( !IsEmpty )

        if (top == 0)
            break;
    }//while
}
```

最后的结果呈现（省略了最后的 main 函数）：

3. 实验总结

读入文本文件的内容，并按照每一个不同字符出现的频率赋予相应的权重。这里因为我需要读入汉字（注意汉字及汉语标点符号在计算机中的存储都是两位字符类型大小），所以这里我统计每个汉字或标点出现的次数的方法是建立一个结构体数组（设不同的汉字由 NUM 种，宏定义可手动修改，当然也可改为自动添加），每读入一个新的汉字就从结构体数组的开头开始检测是否出现过，出现过则会在检测到结尾之前匹配成功，该汉字数目加一，不成功则在结束检测时把末尾结构体元素设为此汉字，以此类推直到读入到文件末尾。

解压的实现：

1、先读取第一个字节，即编码个数，确定了我们需要读多少组数据。

2、开始正式读取键值对信息。读取 key 值，读取 01 码对应的字节数，读取补 0 情况，再读取表示 01 串的字节数据，去掉之前补的 0，还原回 0 和 1 表示的字符串，即字符对应的哈夫曼编码，把读到的字符和哈夫曼编码保存在一个新建的 HashMap 中，需要注意的是

此处 key 值存储为哈夫曼编码，value 值存储为字符的信息。以此类推，直到读完所有键值对信息。

3、读整个文件补 0 个数，读取文件字节数据，去掉补的 0，得到之前存入的哈夫曼编码 01 字符串。

4、确定希望解压的文件目录。逐位读取 01 字符串，将读到的位累加在一个临时字符串中，每读一位都拿这个临时字符串和 HashMap 进行对照，如果有对应 key 值，则获取对应字符信息写入流，把字符串置空，继续循环累加新的 01 串。最终读完后，解压目录中便得到了我们解压后的文件。

若读英文字母文本则简单的多，因为英文所有字符的 ascii 码分布都在 0 - 255 之间，所以我们只需要建立一个大小为 256 的数组即可以英文字符的 ascii 码做下标存储并记录字符出现次数，这样会大大减小时间复杂度!!

读汉字时，新读入的一个汉字必须经过和之前已经存在的汉字一个一个的比较才能确定是否出现过。那么在最坏情况下，若文本文件中的 n 个汉字均各不相同，那么其时间复杂度将会达到 O（n^2）的大小，当文件大小过大时这是无法想象的。

要通过哈夫曼编码真正做到压缩文件，需要用到位操作，即：把八个二进制数整合到一个字节当中，然后通过整数形式存储到目标文件，这样才能最终达到压缩文件大小的目的。当然还需要存储的是每一个不同的字符到底是什么还有它们分别的权重，这样在后续译码过程中才能正常进行。