#define N\_CHAR (0x80 - 0x20) //仅以可打印字符为例

struct HuffChar { //Huffman（超）字符

char ch; int weight; //字符、频率

HuffChar ( char c = '^', int w = 0 ) : ch ( c ), weight ( w ) {};

// 比较器、判等器（各列其一，其余自行补充）

bool operator< ( HuffChar const& hc ) { return weight > hc.weight; } //此处故意大小颠倒

bool operator== ( HuffChar const& hc ) { return weight == hc.weight; }

};

int\* statistics ( char\* sample\_text\_file ) { //统计字符出现频率

int\* freq = new int[N\_CHAR]; //以下统计需随机访问，故以数组记录各字符出现次数

memset ( freq, 0, sizeof ( int ) \* N\_CHAR ); //清零

FILE\* fp = fopen ( sample\_text\_file, "r" ); //assert: 文件存在且可正确打开

for ( char ch; 0 < fscanf ( fp, "%c", &ch ); ) //逐个扫描样本文件中的每个字符

if ( ch >= 0x20 ) freq[ch - 0x20]++; //累计对应的出现次数

fclose ( fp ); return freq;

}

HuffForest\* initForest ( int\* freq ) { //根据频率统计表，为每个字符创建一棵树

HuffForest\* forest = new HuffForest; //以List实现的Huffman森林

for ( int i = 0; i < N\_CHAR; i++ ) { //为每个字符

forest->insertAsLast ( new HuffTree ); //生成一棵树，并将字符及其频率

forest->last()->data->insertAsRoot ( HuffChar ( 0x20 + i, freq[i] ) ); //存入其中

}

return forest;

}

HuffTree\* minHChar ( HuffForest\* forest ) { //在Huffman森林中找出权重最小的（超）字符

ListNodePosi ( HuffTree\* ) p = forest->first(); //从首节点出发查找

ListNodePosi ( HuffTree\* ) minChar = p; //最小Huffman树所在的节点位置

int minWeight = p->data->root()->data.weight; //目前的最小权重

while ( forest->valid ( p = p->succ ) ) //遍历所有节点

if ( minWeight > p->data->root()->data.weight ) //若当前节点所含树更小，则

{ minWeight = p->data->root()->data.weight; minChar = p; } //更新记录

return forest->remove ( minChar ); //将挑选出的Huffman树从森林中摘除，并返回

}

HuffTree\* generateTree ( HuffForest\* forest ) { //Huffman编码算法

while ( 1 < forest->size() ) {

HuffTree\* T1 = minHChar ( forest ); HuffTree\* T2 = minHChar ( forest );

HuffTree\* S = new HuffTree(); /\*DSA\*/printf ( "\n################\nMerging " ); print ( T1->root()->data ); printf ( " with " ); print ( T2->root()->data ); printf ( " ...\n" );

S->insertAsRoot ( HuffChar ( '^', T1->root()->data.weight + T2->root()->data.weight ) );

S->attachAsLC ( S->root(), T1 ); S->attachAsRC ( S->root(), T2 );

forest->insertAsLast ( S ); /\*DSA\*/ //print(forest);

} //assert: 循环结束时，森林中唯一（列表首节点中）的那棵树即Huffman编码树

return forest->first()->data;

}

static void //通过遍历获取各字符的编码

generateCT ( Bitmap\* code, int length, HuffTable\* table, BinNodePosi ( HuffChar ) v ) {

if ( IsLeaf ( \*v ) ) //若是叶节点（还有多种方法可以判断）

{ table->put ( v->data.ch, code->bits2string ( length ) ); return; }

if ( HasLChild ( \*v ) ) //Left = 0

{ code->clear ( length ); generateCT ( code, length + 1, table, v->lc ); }

if ( HasRChild ( \*v ) ) //Right = 1

{ code->set ( length ); generateCT ( code, length + 1, table, v->rc ); }

}

HuffTable\* generateTable ( HuffTree\* tree ) { //将各字符编码统一存入以散列表实现的编码表中

HuffTable\* table = new HuffTable; Bitmap\* code = new Bitmap;

generateCT ( code, 0, table, tree->root() ); release ( code ); return table;

};

int encode ( HuffTable\* table, Bitmap\* codeString, char\* s ) { //按照编码表对Bitmap串编码

int n = 0; //待返回的编码串总长n

for ( size\_t m = strlen ( s ), i = 0; i < m; i++ ) { //对于明文中的每个字符

char\*\* pCharCode = table->get ( s[i] ); //取出其对应的编码串

if ( !pCharCode ) pCharCode = table->get ( s[i] + 'A' - 'a' ); //小写字母转为大写

if ( !pCharCode ) pCharCode = table->get ( ' ' ); //无法识别的字符统一视作空格

printf ( "%s", \*pCharCode ); //输出当前字符的编码

for ( size\_t m = strlen ( \*pCharCode ), j = 0; j < m; j++ ) //将当前字符的编码接入编码串

'1' == \* ( \*pCharCode + j ) ? codeString->set ( n++ ) : codeString->clear ( n++ );

}

printf ( "\n" ); return n;

} //二进制编码串记录于位图codeString中

// 根据编码树对长为n的Bitmap串做Huffman解码

void decode ( HuffTree\* tree, Bitmap\* code, int n ) {

BinNodePosi ( HuffChar ) x = tree->root();

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

x = code->test ( i ) ? x->rc : x->lc;

if ( IsLeaf ( \*x ) ) { printf ( "%c", x->data.ch ); x = tree->root(); }

}

/\*DSA\*/if ( x != tree->root() ) printf ( "..." ); printf ( "\n" );

} //解出的明码，在此直接打印输出；实用中可改为根据需要返回上层调用者