数据结构课程设计



班级： 1618403

学号： 161840227

姓名： 韦 鑫

指导教师：孙 涵

目录

1.采用的数据结构 ………………………………………… 3

2.算法设计思想 …………………………………………… 3

3.关键代码 ………………………………………………… 4

4.测试数据和结果 ………………………………………… 12

5.算法的时间复杂度及其改进方法 ……………………… 13

6.结束语 …………………………………………………… 13

一、采用的数据结构

使用数组存储权值、父节点和左、右孩子位置以及字符值模拟链表

//Huffman树节点结构

typedef struct

{

int w, p, lc, rc;//权值，父节点位置，左孩子位置，右孩子位置

char c;//字符

}HTNode,\*HuffmanTree;

二、算法设计思想

①建立Huffman树：先统计文本各个字符出现的次数（频率），每次以两个频率最小的节点形成中间节点，以此形成Huffman树

②字符编码：字符编码倒序存储。对Huffman树数组进行正序遍历，若某节点为其父节点的左孩子，则向字符编码数组加入‘0’，若为右孩子，则加入‘1’，以此直至其父节点位置指向0，表示该节点字符编码完成。

字符编码数组完成后，扫描文本对各个字符进行编码，得到编码文本。

③字符解码：Huffman树数组倒序，即从根节点开始，若碰到‘0’则向下指向它的左孩子，若碰到‘1’，则向下指向它的右孩子，直到指向叶子节点，则这一01串表示的就是该叶子节点的字符值，重复以上操作。

④位运算实现压缩：建立One数组，里边共有八个元素，分别是0b10000000、0b01000000……首先开辟一个空的字符数组，扫描01串文本，以8为周期，记录位置（模8），记录已压缩字节数（除8），若碰到1，则用字符数组（下标等于记录已压缩的字节数）与One数组（下标为记录位置）进行与（&）运算，反复上述操作。解压类似。

三、关键代码

//运行环境：Visual Studio 2017

#include<iostream>

#include<cstdio>

#include<cstring>

#include<cstdlib>

#include<algorithm>

#pragma warning(disable:4996)

using namespace std;

const int maxn = 200000;//最大需要压缩的空间大小(bytes)

const int maxc = 200000;//01串的最大空间大小

//Huffman树节点结构

typedef struct

{

int w, p, lc, rc;//权值，父节点位置，左孩子位置，右孩子位置

char c;//字符

}HTNode,\*HuffmanTree;

//Huffman编码字符数组

typedef char \*\* HuffmanCode;

//统计文本字符后得到的原始数组，含有字符以及它出现的频率（出现次数）

typedef struct

{

int f;

char c;

}Original;

//在原始数组内选出两个p = 0最小节点位置s1,s2

void Select(HuffmanTree HT,int len, int & s1, int & s2)

{

s1 = s2 = 0;

for (int i = 1; i <= len; i++)

{

if (HT[i].p == 0 && HT[i].w < HT[s1].w)s1 = i;

}

for (int i = 1; i <= len; i++)

{

if (HT[i].p == 0 && HT[i].w < HT[s2].w && i != s1)s2 = i;

}

}

//建立Huffman树

void CreatHT(HuffmanTree & HT,Original A[] , int n)

{

if (n <= 1)return;

int m = 2 \* n - 1;//节点n + 中间节点n-1

HT = (HuffmanTree)malloc((m + 1) \* sizeof(HTNode));//0号位不用

int i;

HT[0].w = 1 << 30 - 1;//设置一个无穷大的数

for (i = 1; i <= n; i++)

{

HT[i].p = HT[i].lc = HT[i].rc = 0;

HT[i].w = A[i].f;

HT[i].c = A[i].c;

}

for (; i <= m; i++)

{

HT[i].p = HT[i].lc = HT[i].rc = HT[i].w = HT[i].c = 0;

}

for (i = n + 1; i <= m; i++)

{

int s1, s2;

Select(HT, i - 1, s1, s2);

HT[s1].p = HT[s2].p = i;

HT[i].lc = s1;

HT[i].rc = s2;

HT[i].w = HT[s1].w + HT[s2].w;

}

}

//字符编码

void Encoding(HuffmanTree HT,HuffmanCode & HC, int n)

{

HC = (HuffmanCode)malloc((n + 1) \* sizeof(char \*));

char \*cd = (char \*)malloc(n \* sizeof(char));//最坏的编码长度为n

cd[n - 1] = '\0';//编码结束符

int i;

for (i = 1; i <= n; i++)

{

int start = n - 1;

for (int c = i, f = HT[i].p; f != 0; c = f, f = HT[f].p)

{

if (HT[f].lc == c)cd[--start] = '0';//左0，右1

else cd[--start] = '1';

}

HC[i] = (char \*)malloc((n - start) \* sizeof(char));

strcpy(HC[i], &cd[start]);

}

free(cd);

//写入字符出现的次数及编码

FILE \*fp;

if ((fp = fopen("Huffman.txt", "w")) == NULL)

{

printf("打开文件失败！\n");

exit(0);

}

char ch;

for (int i = 1; i <= n; i++)

{

fprintf(fp, "%c:%d %s\n", HT[i].c, HT[i].w, HC[i]);//字符值，出现次数，编码结果

}

fclose(fp);

}

//输出编码测试

void PrintCode(Original A[],HuffmanCode HC, int n)

{

for (int i = 1; i <= n; i++)

{

printf("%c : %s\n", A[i].c, HC[i]);

}

}

//按位压缩与解压

//将地址address byte字节后的第pos位设位0

void Zero(char \*address, int bits)

{

char zero[8] = { 0b01111111,0b10111111,0b11011111,0b11101111,0b11110111,0b11111011,0b11111101,0b11111110 };

char \*addr = address;

int byte = bits / 8; //要往下移动的字节数

int pos = bits % 8; //移动到准确byte字节后的目标位

addr += byte;

\*addr = \*addr & zero[pos];

}

//与上类似，设为1

void One(char \*address, int bits)

{

char one[8] = { 0b10000000,0b01000000,0b00100000,0b00010000,0b00001000,0b00000100,0b00000010,0b00000001 };

char \*addr = address;

int byte = bits / 8; //要往下移动的字节数

int pos = bits % 8; //移动到准确字节后的目标位

addr += byte;

\*addr = \*addr | one[pos];

}

//压缩(01串—>按位存储)

void Compress(char encode[])

{

FILE \*bitencode;

if ((bitencode = fopen("code.dat", "wb")) == NULL)

{

printf("打开压缩文件失败！\n");

exit(0);

}

char tempStr[maxn] = { 0 }; //存储压缩后的数据

int bits = 0; //第bits个位的位置

while (encode[bits] != '\0')

{

if (encode[bits] == '1')One(tempStr, bits);

else if (encode[bits] == '0')Zero(tempStr, bits);

bits++;

}

int morebits = bits % 8; //多出的不够8位的bit数

int morebyte, bytes; //morebyte是否需要多一个字节

if (bits % 8 == 0)morebyte = 0;

else morebyte = 1;

bytes = bits / 8 + morebyte; //bytes总共需要的字节数

//记录morebits

fwrite(&morebits, 4, 1, bitencode);

//按字节逐一写入文件

for (int i = 0; i < bytes; i++)

{

fwrite(tempStr + i, 1, 1, bitencode);

}

fclose(bitencode);

}

//解压(二进制文件还原为01串,为还原为文本文件做准备)

void Decompress(char encode[])//encode[]存储01串

{

int morebits;

int bytes = 0;

char tempStr[maxn]; //存储待解压的数据

FILE \* bitencode;

if ((bitencode = fopen("code.dat", "rb")) == NULL)

{

printf("打开压缩文件失败！\n");

exit(0);

}

fread(&morebits, 4, 1, bitencode);

while (fread(tempStr + bytes, 1, 1, bitencode) == 1)bytes++;

fclose(bitencode);

//将待压缩的数据还原为01串

int bits = 0;

char one[8] = { 0b10000000,0b01000000,0b00100000,0b00010000,0b00001000,0b00000100,0b00000010,0b00000001 };

for (int i = 0; i < bytes - 1; i++)//暂时不处理最后一个字节数据(因为可能不足8位)

{

for (int bit = 0; bit < 8; bit++)

{

if ((tempStr[i] & one[bit]) == one[bit])encode[bits++] = '1';

else encode[bits++] = '0';

}

}

for (int bit = 0; bit < morebits; bit++)//处理那不足8位的bit位

{

if ((tempStr[bytes - 1] & one[bit]) == one[bit])encode[bits++] = '1';

else encode[bits++] = '0';

}

encode[bits] = '\0';

}

//对文本文件编码

void EncodeText(HuffmanTree HT, HuffmanCode HC, int n)

{

char encode[maxc]; //encode[]存储编码后的文本

FILE \*data;

if ((data = fopen("source.txt", "r")) == NULL)

{

printf("文件打开失败！\n");

exit(0);

}

int t = 0;

while (!feof(data))

{

char ch = fgetc(data);

for (int i = 1; i <= n; i++)

{

if (HT[i].c == ch)

{

for (int j = 0; HC[i][j] != '\0'; j++)

{

encode[t++] = HC[i][j];

}

}

}

}

encode[t] = '\0';

fclose(data);

//压缩

Compress(encode);

}

//对二进制文件译码

void DecodeText(HuffmanTree HT, int n)//读取文件字符及出现的次数再构建Huffman树再译码更具有一般性。

{

char encode[maxc], decode[maxc]; //encode[]存储待译码01串，decode[]存储译码后的文本

Decompress(encode);

FILE \*fp;

if ((fp = fopen("recode.txt", "w")) == NULL)

{

printf("文件打开失败！\n");

exit(0);

}

int len = strlen(encode);

int root = 2 \* n - 1,t = 0;

for (int i = 0,j = root; i < len; i++)

{

if (encode[i] == '0')j = HT[j].lc;

else if(encode[i] == '1')j = HT[j].rc;

//说明已经遍历的叶子节点

if (HT[j].lc == 0 && HT[j].rc == 0)

{

decode[t++] = HT[j].c;

j = root;

}

}

decode[t] = '\0';

fputs(decode, fp);

fclose(fp);

}

//校对源文件与译码文件

bool Comp()

{

FILE \*data, \*decode;

if ((data = fopen("source.txt", "r")) == NULL)

{

printf("打开文件失败！\n");

exit(0);

}

if ((decode = fopen("recode.txt", "r")) == NULL)

{

printf("打开文件失败！\n");

exit(0);

}

char ch1, ch2;

//防止比较结束字符导致出错

ch1 = fgetc(data);

ch2 = fgetc(decode);

while (!feof(data) && !feof(decode))

{

if (ch1 != ch2)

{

fclose(data);

fclose(decode);

return false;

}

ch1 = fgetc(data);

ch2 = fgetc(decode);

}

fclose(data);

fclose(decode);

return true;

}

//统计文本中字符出现的频率生成原始数组(Original A[])

void Frequency(Original A[],int & n)

{

//临时存储ASCII码0-255的字符出现的次数

int temp[256] = {0};

FILE \*data;

if ((data = fopen("source.txt","r")) == NULL)

{

printf("打开源文件失败！\n");

exit(0);

}

char ch;

ch = fgetc(data);

while (!feof(data))

{

temp[(int)ch]++;

ch = fgetc(data);

}

//n为A数组下标

n = 0;

for (int i = 0; i < 256; i++)

{

if (temp[i] > 0)

{

n++;

A[n].c = (char)i;

A[n].f = temp[i];

}

}

fclose(data);

}

int main()

{

int n;

Original A[257];//ASCII 0-255,下标0不用

HuffmanTree HT;

HuffmanCode HC;

Frequency(A, n); //统计文本出现的字符及次数

CreatHT(HT, A, n); //建立Huffman树

Encoding(HT, HC, n); //对字符进行编码

//PrintCode(A, HC, n); //test

EncodeText(HT, HC, n); //对文本文件压缩编码并写入code.dat文件

DecodeText(HT,n); //对二进制文件解码并写入recode.txt文件

if (Comp())

{

printf("校对结果：源文件与译码文件完全一致！\n");

}

else printf("校对结果：源文件与译码文件不一致！\n");

system("pause");

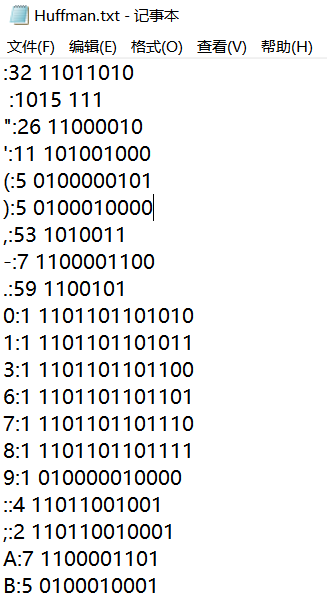
return 0;

}

四、测试数据和结果



字符编码(部分)结果：



位运算压缩结果：





五、算法的时间复杂度即改进方法

建立Huffman树：O(n\*n)

六、结束语

代码共362行

本实验除了完成课本给出的内容外，我另外利用位运算实现了对文本的Huffman编码压缩以及解压译码的过程。在此之前我们使用压缩包这类的软件对其内部实现无太多概念，而学习了Huffman编码之后顿时豁然开朗：原来这样可以实现数据的压缩。但是在实现位运算压缩的过程中，会遇到不少问题，比如压缩存储的数据是乱码的，难以检测其正确性，另外为运算是对一个bit进行运算，并不能实时输出以检验其正确性，我认为这个问题是该实验最大的难点。