```
//***************************************************************************
**
// FILE:                ShannonFano.c
// PROJECT:             zarLib  - ver 1.00 -
// COPYRIGHTS:  The famous group 4 of "Roskilde Universites Center"
//***************************************************************************
***/
// Headers:
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <search.h>
#include <string.h>
#include "zarLib.h"
#include "main.h"


//***************************************************************************
**
// Constants:

#define END_OF_STREAM   256


//***************************************************************************
**
// Typedefs:

typedef struct
{
        _dword  dwCount;
        _int    ubChar;
        _dword  dwBitCode;
        _dword  dwNumBits;
} CODE;

typedef struct _NODE
{
        CODE*                   pCode;
        struct _NODE*   pChild0;
        struct _NODE*   pChild1;
} NODE;

typedef struct
{
        NODE  nodes[600];
        NODE* pCurNode;
        int       cxNodes;
} TREE;


//***************************************************************************
**
// Function prototypes:

static void freqToCode(const ZAR_CHAR_FREQ* pzCharFreq, CODE* nodes);
static void buildCompressCodes(CODE* pCode, int nNumCodes);
static void buildExtractTree(CODE* codes, int nNumCodes, TREE* tree);
static void sortCodes(CODE* codes);
static int  saveHeader(const CODE* codes, ZAR_FILE* pzFile);
static int  loadHeader(ZAR_CHAR_FREQ* zCharFreq, ZAR_FILE* pzFile);

static int  codesCompare(const void* pA, const void* pB);


//***************************************************************************
**
// ShannonFanoExtract()

int ShannonFanoExtract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
```



```
{
        int                             i, nEos;
        NODE*                   pNode;
        CODE                    codes[256 +1];
        TREE                    tree;
        ZAR_CHAR_FREQ*  pzCharFreq;

        // Print info:
        printf("- - - Starting Shannon-Fano extraction - - -\n");
        printf("Progress: ");

        // Allocate memory:
        pzCharFreq = (ZAR_CHAR_FREQ*) malloc(sizeof(ZAR_CHAR_FREQ));

        if (pzCharFreq == NULL)
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Get the frequence of characters from the file header:
        zarInitCharFreq(pzCharFreq);
        loadHeader(pzCharFreq, pzSrc);
        zarSortCharFreq(pzCharFreq);

        // Convert to nodes:
        freqToCode(pzCharFreq, codes);
        free(pzCharFreq);               // Get rid of unused memory.

        // Remove nodes with a zero-count:
        for (i=0; codes[i].dwCount != 0 && i<256; i++);         // Don´t need an
y body!

        nEos = i++;
        codes[nEos].dwCount = 1;


        // Build the tree:
        memset(&tree, 0, sizeof(TREE));
        tree.pCurNode = tree.nodes;

        buildExtractTree(codes, i, &tree);

        // Extract the file:
        pNode = tree.nodes;

        while (1)
        {
                // Get next bit:
                i = zarGetBit(pzSrc);

                if (i)
                // Go left:
                        pNode = pNode->pChild0;
                else
                // Go right:
                        pNode = pNode->pChild1;

                // Are we finished?
                if (pNode->pCode != NULL)
                {
                        // Have we reached end of stream?
                        if (pNode->pCode == (codes +nEos))
                                break;

                        zarSetByte(pzDst, pNode->pCode->ubChar);

                        // Reset pos:
                        pNode = tree.nodes;
                        continue;
                }
        }
```

```
        return 1;
}

//***************************************************************************
**
// ShannonFanoCompress()

int ShannonFanoCompress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        int                             i, nEos;
        CODE                    codes[256 +1];
        ZAR_CHAR_FREQ*  pzCharFreq;

        // Print info:
        printf("- - - Starting Shannon-Fano compression - - -\n");
        printf("Progress: ");

        // Allocate memory:
        pzCharFreq = (ZAR_CHAR_FREQ*) malloc(sizeof(ZAR_CHAR_FREQ));

        if (pzCharFreq == NULL)
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Count the frequence of characters in the file:
        zarInitCharFreq(pzCharFreq);
        zarCountCharFreq(pzSrc, pzCharFreq);

        zarSortCharFreq(pzCharFreq);    // faster than qsort!

        // Convert to nodes:
        freqToCode(pzCharFreq, codes);
        free(pzCharFreq);                               // Get rid of unused mem
ory.


        // Remove nodes with a zero-count:
        for (i=0; codes[i].dwCount != 0 && i<256; i++);         // Don´t need an
y body!

        // Add end off stream (eos) symbol:
        nEos = i++;
        codes[nEos].dwCount = 1;

        // Build the compression tree!
        buildCompressCodes(codes, i);

        // Save end of stream code:
        codes[END_OF_STREAM].dwNumBits = codes[nEos].dwNumBits;
        codes[END_OF_STREAM].dwBitCode = codes[nEos].dwBitCode;
        codes[nEos].dwCount = 0;        // We don´t want to include this in the
saved header!


        // Sort the nodes after code:
        sortCodes(codes);

        // Write header:
        if (!saveHeader(codes, pzDst))
                return zarError(ERR_OUTPUT_FILE_NOT_WRITEABLE, pzDst);

        // Write the compressed output:
        while (EOF != (i = zarGetByte(pzSrc)))
                zarSetBits(pzDst, codes[i].dwBitCode, codes[i].dwNumBits);


        // Write end of stream bits:
        zarSetBits(pzDst, codes[END_OF_STREAM].dwBitCode, codes[END_OF_STREAM].d
wNumBits);
        zarSetBits(pzDst, 0, 8);        // Flush the last bits:
```



```
        return 1;
}


//***************************************************************************
**
// buildExtractTree()

static void buildExtractTree(CODE* pCode, int nNumCodes, TREE* pTree)
{
        int             i, j;
        NODE*   pNode = pTree->pCurNode;
        _dword  dwSum, dwSplitSum;

        // Insert node:
        pNode->pChild0 = pTree->nodes + (++(pTree->cxNodes));
        pNode->pChild1 = pTree->nodes + (++(pTree->cxNodes));

        // Left leaf:
        if (nNumCodes == 1)
        {
                // The left leaf:
                pCode->dwNumBits++;
                pCode->dwBitCode <<= 1;
                pCode->dwBitCode |=  1; // Add 1 to the end of the bits.

                // Insert left node:
                pNode->pChild0->pCode = pCode;
                return;
        }


        // Left and right leaf:
        if (nNumCodes == 2)
        {
                // The left leaf (add 1):
                pCode[0].dwNumBits++;
                pCode[0].dwBitCode <<= 1;
                pCode[0].dwBitCode |=  1; //// Add 1 to the end of the bits.
                pNode->pChild0->pCode = pCode;

                // The right leaf (add 0):
                pCode[1].dwNumBits++;
                pCode[1].dwBitCode <<= 1;
                pNode->pChild1->pCode = pCode +1;

                return;
        }


        // Get the sum of the char counts:
        for (i=0, dwSum=0; i<nNumCodes; i++)
                dwSum += pCode[i].dwCount;

        // The value for the split between left and right brance:
        dwSplitSum = dwSum /2;

        // Find the place to spilt:
        for (i=0, dwSum=0; ; i++)
        {
                dwSum += pCode[i].dwCount;

                // Add 1 bit:
                pCode[i].dwNumBits++;
                pCode[i].dwBitCode <<= 1;
                pCode[i].dwBitCode |= 1;

                // Have we reached the spilt point?
                if (dwSum >= dwSplitSum)
                {
```

```
                        // Walk down left brance:
                        i++;
                        pTree->pCurNode = pNode->pChild0;
                        buildExtractTree(pCode, i, pTree);

                        // Add 0 bit to all right brances:
                        for (j=i; j<nNumCodes; j++)
                        {
                                pCode[j].dwNumBits++;
                                pCode[j].dwBitCode <<= 1;
                        }

                        // Walk down right brance:
                        pTree->pCurNode = pNode->pChild1;
                        buildExtractTree(pCode +i, nNumCodes -i, pTree);

                        // Quit loop:
                        break;
                }
        }
}


//***************************************************************************
**
// buildTree()

static void buildCompressCodes(CODE* pCode, int nNumCodes)
{
        int             i, j;
        _dword  dwSum, dwSplitSum;

        // Left leaf:
        if (nNumCodes == 1)
        {
                // The left leaf:
                pCode->dwNumBits++;
                pCode->dwBitCode <<= 1;
                pCode->dwBitCode |=  1;         // Add 1 to the end of the bits.
                return;
        }


        // Left and right leaf:
        if (nNumCodes == 2)
        {
                // The left leaf (add 1):
                pCode[0].dwNumBits++;
                pCode[0].dwBitCode <<= 1;
                pCode[0].dwBitCode |=  1;       // Add 1 to the end of the bits.

                // The right leaf (add 0):
                pCode[1].dwNumBits++;
                pCode[1].dwBitCode <<= 1;
                return;
        }


        // Get the sum of the char counts:
        for (i=0, dwSum=0; i<nNumCodes; i++)
                dwSum += pCode[i].dwCount;

        // The value for the split between left and right brance:
        dwSplitSum = dwSum /2;

        // Find the place to spilt:
        for (i=0, dwSum=0; ; i++)
        {
                dwSum += pCode[i].dwCount;
```



```
                // Add 1 bit:
                pCode[i].dwNumBits++;
                pCode[i].dwBitCode <<= 1;
                pCode[i].dwBitCode |=  1;

                // Have we reached the spilt point?
                if (dwSum >= dwSplitSum)
                {
                        // Walk down left brance:
                        i++;
                        buildCompressCodes(pCode, i);

                        // Add 0 bit to all right brances:
                        for (j=i; j<nNumCodes; j++)
                        {
                                pCode[j].dwNumBits++;
                                pCode[j].dwBitCode <<= 1;
                        }

                        // Walk down right brance:
                        buildCompressCodes(pCode +i, nNumCodes -i);

                        // Quit loop:
                        break;
                }
        }
}

//***************************************************************************
**
// saveHeader()

static int saveHeader(const CODE* codes, ZAR_FILE* pzFile)
{
        int             j, i=0, nOk=1;

        // Start to write the file header:
        while (nOk)
        {
                // Find first entry to write::
                while (codes[i].dwCount == 0)   // Find non-zero!
                {
                        if (++i == 255)
                        {
                                i = 255;
                                break;
                        }
                }

                // Write the index of the found entry:
                fputc(i, pzFile->pFile);

                // Find the end of the entry (3 zeros):
                j = i +1;

                do
                {
                        if (++j >= 256)
                        {
                                j   = 256;
                                nOk = 0;        // end loop
                                break;
                        }
                }
                while (codes[j].dwCount != 0 || codes[j +1].dwCount != 0 || code
s[j +2].dwCount != 0);


                // Store the ending index:
                fputc(j -1, pzFile->pFile);
```

```
                // Save the values between the indexs:
                if (i != 255)
                {
                        for (; i<j; i++)
                                fputc(codes[i].dwCount, pzFile->pFile);
                }
        }

        return 1;
}


//***************************************************************************
**
// loadHeader()

static int loadHeader(ZAR_CHAR_FREQ* pzCharFreq, ZAR_FILE* pzFile)
{
        int i, from, to;
        ZAR_FREQ* pFreq = pzCharFreq->freq;

        // Extract the frequence from the header:
        while (1)
        {
                from = fgetc(pzFile->pFile);
                to   = fgetc(pzFile->pFile);

                if (from == 255)
                        break;

                // Fill the range up to i with zeros:
                for (i=from; i<=to; i++)
                        pFreq[i].cxChar = fgetc(pzFile->pFile);

                if (to == 255)
                        break;
        }

        // Read next byte:
        pzFile->nCurByte = zarGetByte(pzFile);
        pzFile->nBitMask = 0x80;                // Make sure not to read next by
te!

        return 1;
}


//***************************************************************************
**
// sortNodes()

static void sortCodes(CODE* codes)
{
        qsort(codes, 256, sizeof(CODE), codesCompare);
}


//***************************************************************************
**
// freqToCode()

static void freqToCode(const ZAR_CHAR_FREQ* pzCharFreq, CODE* codes)
{
        int i;
        const ZAR_FREQ* pFreq = pzCharFreq->freq;

        memset(codes, 0, sizeof(CODE) *257);

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



```
        {
                codes[i].ubChar  = (_int) pFreq[i].ubChar;
                codes[i].dwCount = pFreq[i].cxChar;
        }
}


//***************************************************************************
**
// nodesCompare()

static int codesCompare(const void* pA, const void* pB)
{
        return (((const CODE*) pA)->ubChar -((const CODE*) pB)->ubChar);
}
```