```
//***************************************************************************
**//
// FILE:                LZW.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 <string.h>
#include "zarLib.h"


//***************************************************************************
**
// Defines:

// The maximum bit-size the code-book can grow to:
#define MAX_CODEBOOK_SIZE                       24

// The maximum nodes that the code-book can contain:
#define MAX_NODES                                       (2 << (MAX_CODEBOOK_SIZE
 -1))


// The minimum bit-size of the code-book:
#define MIN_CODEBOOK_SIZE                       16


// The minimum number of nodes, which should be allocated:
#define MIN_NODES_ALLOC                         (2 << (MIN_CODEBOOK_SIZE -1))


// The maximum number of stacks:
#define NUM_STACKS                                      (MAX_CODEBOOK_SIZE -MIN_
CODEBOOK_SIZE)


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

typedef struct NODE
{
        int             nValue;       // int could be faster than _int
        _int            nCode;        // important must be 32 bit
        struct NODE*  pChild;
        struct NODE*  pNext;
        struct NODE*  pParent;
} NODE;

typedef struct CODE
{
        _pstr   pValue;
        int             nValueSize;
} CODE;


//***************************************************************************
**
// Globals:

static int         g_nCodeSize;                 // The size of a code in bits (s
tarts at 9)
static _int        g_nNotesAtCodeSize;  // The number of notes at this codesize
static _int        g_cxNodes;                   // The count of nodes
static _int    g_nAllocNodes;           // The number of allocated nodes thus fa
r.
```



```
static _int        g_nStackPtr;

static NODE*   g_pCurNodeStack;
static NODE*   g_pNodeStacks[NUM_STACKS];

static CODE*   g_pCodeStack;
static _pstr   g_pCodeStrStack;
static _int        g_nAllocStrStack;
static _int    g_nCodeStrStackPtr;


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

static int    initCompression(void);
static int        initExtraction(void);
static NODE*  getFreeNode(void);
static _pstr  getFreeCodeStr(int nSize);
static int    updateCodeSize(CODE** ppCode, CODE** ppLastNode);
static void   compress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst);
static void   addChild(NODE* pParentNode, int nValue);
static void   addSibling(NODE* pParentNode, int nValue);
static NODE*  findValue(NODE* pNode, int nValue);
static void   extract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst);
static int        getCodeValue(int nCode);


//***************************************************************************
**
// lzwCompress()

int lzwCompress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        // Print info:
        printf("--- Starting LZW compression ---\n");
        printf("Progress: ");

        if (!initCompression())
                return 0;

        // Compress the file:
        compress(pzSrc, pzDst);

        return 1;
}


//***************************************************************************
**
// lzwCompress()

int lzwExtract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        // Print info:
        printf("--- Starting LZW extraction ---\n");
        printf("Progress: ");

        if (!initExtraction())
                return 0;

        extract(pzSrc, pzDst);

        return 1;
}



//***************************************************************************
**
```

```
// findCode()

static void compress(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        NODE* pNode, *pLastNode;
        int       nValue, i;

        nValue = zarGetByte(pzSrc);

        while (nValue != EOF)
        {
                pLastNode = g_pNodeStacks[0] +nValue;
                pNode = pLastNode;

                do
                {
                        // Read next byte:
                        nValue = zarGetByte(pzSrc);

                        // Set pointer to next node:
                        pLastNode = pNode;
                        pNode = pLastNode->pChild;

                        if (pNode == NULL)
                        {
                                zarSetBitsF(pzDst, pLastNode->nCode, g_nCodeSize
);
                                addChild(pLastNode, nValue);
                                break;
                        }

                        pLastNode = pNode;

                        // Try to find the value:
                        pNode = findValue(pNode, nValue);

                        if (pNode == NULL)
                        {
                                zarSetBitsF(pzDst, pLastNode->pParent->nCode, g_
nCodeSize);
                                addSibling(pLastNode, nValue);
                                break;
                        }
                } while (nValue != EOF);
        }

        // Free stacks
        for (i=0; i<NUM_STACKS; i++)
                free(g_pNodeStacks[i]);
}


//*****************************************************************************
**
// findValue()

static NODE* findValue(NODE* pNode, int nValue)
{
        // Go through siblings:
        while (pNode != NULL)
        {
                // Did we find a match
                if (pNode->nValue == nValue)
                        return pNode;

                pNode = pNode->pNext;
        }

        return NULL;
}
```



```
//*****************************************************************************
**
// addChild()

static void addChild(NODE* pParentNode, int nValue)
{
        NODE* pChild;

        // Add a child:
        pParentNode->pChild = getFreeNode();
        pChild = pParentNode->pChild;

        if (pChild)
        {
                pChild->nValue  = nValue;
                pChild->nCode   = g_cxNodes;
                pChild->pParent = pParentNode;
        }
}


//*****************************************************************************
**
// addSibling()

static void addSibling(NODE* pParentNode, int nValue)
{
        NODE* pCousin;

        // Add a cousin?
        pCousin = pParentNode;

        while (pCousin->pNext != NULL)  // Find last cousin
                pCousin = pCousin->pNext;

        // Get free memory-block:
        pCousin->pNext = getFreeNode();
        pCousin = pCousin->pNext;

        if (pCousin)
        {
                // Update node data:
                pCousin->nValue = nValue;
                pCousin->nCode  = g_cxNodes;
                pCousin->pParent = pCousin->pParent;
        }
}


//*****************************************************************************
**
// initCompressTree()

static int initCompression(void)
{
        int             i;

        /* Initialize the members for the stack-memory */

        g_cxNodes       = 256;  // The first 256 nodes are already initialized!
        g_nCodeSize = 9;        // The size of the first codes

        g_nNotesAtCodeSize      = 2 << (g_nCodeSize -1);        // The number of
 notes available, before code size should increase

        // Zero all stacks:
        memset(g_pNodeStacks, 0, sizeof(NODE*) *NUM_STACKS);

        // Allocate memory for notes:
```

```
        g_nAllocNodes = MIN_NODES_ALLOC;

        if (NULL == (g_pNodeStacks[0] = calloc(g_nAllocNodes, sizeof(NODE))))
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Initialize current stack:
        g_pCurNodeStack = g_pNodeStacks[0];
        g_nStackPtr = g_cxNodes;

        // Initialize the default nodes:
        for (i=0; i<256; i++)
                g_pCurNodeStack[i].nValue = g_pCurNodeStack[i].nCode = i;

        return 1;
}


//***************************************************************************
**
// initExtraction()

static int initExtraction(void)
{
        int i;

        g_cxNodes       = 256;
        g_nCodeSize = 9;

        g_nNotesAtCodeSize      = 2 << (g_nCodeSize -1);        // The number of
 codes available, before code size should increase.

        // Zero stack:
        g_pCodeStack            = NULL;
        g_pCodeStrStack     = NULL;

        // Allocate memory for codes:
        g_nAllocNodes    = MIN_NODES_ALLOC;
        g_nAllocStrStack = MIN_NODES_ALLOC;

        if (NULL == (g_pCodeStack = malloc(sizeof(CODE) *g_nAllocNodes)))
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Allocate memory for strings:
        if (NULL == (g_pCodeStrStack = malloc(sizeof(char) *g_nAllocNodes)))
                return zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);

        // Initialize stacks:
        g_nStackPtr         = g_cxNodes;
        g_nCodeStrStackPtr      = 0;


        // Initialize the default codes:
        for (i=0; i<256; i++)
        {
                g_pCodeStack[i].nValueSize = 1;
                g_pCodeStack[i].pValue     = getFreeCodeStr(1);
                *g_pCodeStack[i].pValue    = (char) i;
        }

        return 1;
}


//***************************************************************************
**
// getFreeNode()

static NODE* getFreeNode(void)
{
        // Have we reached the limit of this code size?
```



```
        if (g_cxNodes == g_nNotesAtCodeSize)
        {
                // Have we reached maximum number of codes?
                if (g_nCodeSize != MAX_CODEBOOK_SIZE)
                {
                        g_nCodeSize++;
                        g_nNotesAtCodeSize <<= 1;

                        // Should we allocate some more memory?
                        if (g_cxNodes == g_nAllocNodes)
                        {
                                int nStackIdx = g_nCodeSize -MIN_CODEBOOK_SIZE;
                                g_nStackPtr = 0;

                                // Allocate memory for the new stack:
                                g_pNodeStacks[nStackIdx] = calloc(g_nAllocNodes,
 sizeof(NODE));
                                g_pCurNodeStack = g_pNodeStacks[nStackIdx];
                                g_nAllocNodes <<= 1;

                                if (g_pCurNodeStack == NULL)
                                {
                                        zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NU
LL);
                                        exit(0);
                                }
                        }
                }
                else {
                        return NULL;    // No code added.
                }
        }

        // Return next node:
        g_cxNodes++;
        return (g_pCurNodeStack +g_nStackPtr++);
}


//***************************************************************************
**
// getFreeCodeStr()

static _pstr getFreeCodeStr(int nSize)
{
        g_nCodeStrStackPtr += nSize;

        if (g_nCodeStrStackPtr >= g_nAllocStrStack)
        {
                int i;

                char* pOld = g_pCodeStrStack;
                g_nAllocStrStack <<= 1;

                g_pCodeStrStack = realloc(g_pCodeStrStack, sizeof(char) *g_nAllo
cStrStack);

                // Reallocate value pointers:
                for (i=0; i<g_nStackPtr; i++)
                {
                        g_pCodeStack[i].pValue = g_pCodeStack[i].pValue -pOld +g
_pCodeStrStack;
                }


                if (g_pCodeStrStack == NULL)
                {
                        zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);
                        exit(0);
                }
```

```
        }

        return (g_pCodeStrStack +g_nCodeStrStackPtr -nSize);
}


//***************************************************************************
**
// extract()

static void extract(ZAR_FILE* pzSrc, ZAR_FILE* pzDst)
{
        int             nCode, nValSize;
        CODE*   pCode, *pLastCode;

        nCode = zarGetBits(pzSrc, g_nCodeSize);
        pCode = g_pCodeStack +nCode;

        while (1)
        {
                nCode = zarGetBits(pzSrc, g_nCodeSize);

                pLastCode = pCode;
                pCode     = g_pCodeStack +nCode;

                // Write value of code:
                nValSize = pLastCode->nValueSize;
                zarSetBytes(pzDst, pLastCode->pValue, nValSize);

                if (nCode == EOF)
                        break;

                // Have we reached the limit of this code size?
                if (++g_cxNodes == g_nNotesAtCodeSize)
                {
                        if (!updateCodeSize(&pCode, &pLastCode))
                                continue;
                }

                g_nStackPtr++;

                // Add code:
                g_pCodeStack[g_nStackPtr].pValue = getFreeCodeStr(nValSize +1);
                g_pCodeStack[g_nStackPtr].nValueSize = nValSize +1;
                memcpy(g_pCodeStack[g_nStackPtr].pValue, pLastCode->pValue, nVal
Size);
                g_pCodeStack[g_nStackPtr].pValue[nValSize] = g_pCodeStack[nCode]
.pValue[0];
        }

}


//***************************************************************************
**
// updateCodeSize()

static int updateCodeSize(CODE** ppCode, CODE** ppLastCode)
{
        // Have we reached maximum number of codes?
        if (g_nCodeSize != MAX_CODEBOOK_SIZE)
        {
                g_nCodeSize++;
                g_nNotesAtCodeSize <<= 1;

                // Should we allocate some more memory?
                if (g_cxNodes == g_nAllocNodes)
                {
                        CODE* pOld = g_pCodeStack;
```



```
                        g_nAllocNodes <<= 1;

                        g_pCodeStack = realloc(g_pCodeStack, sizeof(CODE) *g_nAl
locNodes);

                        *ppLastCode = *ppLastCode -pOld +g_pCodeStack;
                        *ppCode = *ppCode -pOld +g_pCodeStack;

                        if (g_pCodeStack == NULL)
                        {
                                zarError(ZAR_ERR_FAILED_MEMORY_ALLOC, NULL);
                                exit(0);
                        }
                }

                return 1;
        }

        return 0;
}
```