Curso Laboratorio II 2020
Tarea en grupos

Compresión y descompresión de archivos usando un árbol de Huffman.

Supuestos
Se da por supuesto que el alumno está al día con las recomendaciones, ejercicios de clase y tareas domiciliarias de semanas anteriores. El curso está pensado para construir sobre las bases adquiridas en las clases anteriores.

Objetivo
Comprimir archivos usando *codificación Huffman*.

Debe respetarse rigurosamente la implementación solicitada en esta letra.

Uso
Deben escribirse dos ejecutables: huffman y huffman-d. El primero comprime y el segundo descomprime.

Ambos requieren un único argumento que será el nombre de un archivo.

`huffman archivo` comprime y genera un archivo comprimido de nombre `archivo.huf`

`huffman-d archivo.huf` descomprime y genera un archivo de nombre `archivo.ori`

Es obligatorio chequear por las terminaciones .huf y .ori, así como detectar problemas en el contenido del archivo comprimido (cabezal mal, etc.).

Estructuras para usar
Deben usarse las estructuras definidas en el archivo huffman.h del Anexo. Allí se definen algunas constantes y tipos. Ver el Anexo antes de seguir leyendo para comprender mejor.

En este ejercicio los símbolos serán bytes.

Funciones de heap
No es necesario programarlas, ya se brindan. En el servidor estará el objeto no ejecutable heap_generic.o que contiene las funciones definidas en el archivo heap_generic.h en el Anexo.

¿Cómo funciona el algoritmo de compresión?
1. Lee secuencialmente el archivo de entrada y almacena en un array de ***Symcode*** de tamaño 256, indexado por el propio símbolo, los símbolos usados (***symbol***) y su frecuencia de uso (***count***). Los campos ***mask*** y ***masklen*** quedarán en 0 hasta que se tenga la capacidad de calcularlos (paso 3).
2. La cantidad de símbolos diferentes en el archivo de entrada serán el tamaño de un nuevo array de ***Treenode*** que será convertido en un heap del tipo ***Heap***.
3. Se arma un árbol de Huffman usando el algoritmo clásico. Los nodos internos usarán símbolos “ficticios” en donde solo interesa la cantidad (frecuencia).

4. Una vez generado el árbol, a partir de la raíz, se recorre todo el árbol y se generan los campos ***mask*** y ***masklen*** en las hojas.
5. Finalmente, ya se tienen todos los datos para leer nuevamente el archivo de entrada y generar la salida comprimida. Los archivos comprimidos según este algoritmo deben contener la información de cada símbolo para el descompresor. Ver en el Anexo el formato que lleva el archivo comprimido.

Lecturas

- https://en.wikipedia.org/wiki/Huffman_coding,
- https://en.wikipedia.org/wiki/Binary_heap
- https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

ANEXOS

Archivo huffman.h

```c
#include <stdlib.h>
#include <stdint.h>
/*
 * huffman.h
 * Estructuras requeridas para resolver compresión/descompresión Huffman,
 * de acuerdo al algoritmo de Wikipedia.
 *
 * El archivo comprimido tendrá este formato:
 * - un cabezal que identifica este archivo y contiene datos del contenido posterior
 *   (8 bytes)
 * - un array de elementos, cada uno de los cuales identifica un símbolo y su bitmask
 *   (8 bytes por símbolo)
 * - un string de bits que representan el contenido del archivo original,
 *   transformado según el algoritmo de compresión.
 *   (resto del contenido del archivo)
 *
 */

#define MAXSYMBOL 256               // Valores 0 a 255 (8 bits)
#define MYMAGICNBR 0x5147           // mi número mágico para archivos ("GQ")
#define MAXFILELEN (1ul << 24)      // tamaño máximo de archivos (24 bits)

// CABEZAL DEL ARCHIVO COMPRIMIDO
typedef struct compr_header {       // 8 bytes: cabezal de archivo comprimido
    uint16_t    magic_nbr;          // número mágico (2)
    uint8_t     sym_arraylen;       // largo del array de los Symcode (1)
    uint8_t     sym_arraysize;      // tamaño de cada elemento del array anterior (1)
    uint32_t    filelen;            // largo del archivo original en bytes (4)
} Compr_header;

                                    // optimizado para aprovechar al máximo 32 bits
#define BITSFORMASK 27              // largo máximo de la máscara: 27 bits
#define MAXMASK (1ul << BITSFORMASK) // para controlar error por overflow

// REPRESENTACIÓN DE 1 SIMBOLO
typedef struct symcode {            // 8 bytes para cada símbolo existente en la entrada
    uint64_t    symbol: 8,          // 8 bits para el símbolo (byte)
                count: 24,          // 24 bits para la cantidad de ocurrencias (16Mega)
                mask: BITSFORMASK,      // definido más arriba como 27
```

```
                masklen: 32 - BITSFORMASK;  // 5 bits para representar el largo
} Symcode;

/*
 * La estructura de un nodo del árbol, que tendrá dos formas de acceso:
 * Una es un array accedido como un heap.
 * La otra es como un árbol, para lo cual están los punteros a sus hijos.
 *
 */
typedef struct treenode {
    struct symcode *code;           // los datos del símbolo están en un Symcode
    struct treenode *children[2];   // Si es interno, punteros a hijos (0=izq, 1=der)
} Treenode;

/*
 * La estructura de un heap para almacenar los nodos del árbol
 *
 */
typedef struct heap {
    int capacity;           // la capacidad máxima de elementos de este heap
    int used;               // la cantidad de elementos usados
    Treenode **nodearray;   // array mallocado, símil nodearray[capacity]
} Heap;
```

Archivo heap_generic.h

```
/*
 * heap_generic.h
 *
 * Declaraciones de las funciones definidas en heap_generic.c
 *
 * Gerardo Quincke - febrero 2020
 *
 */
#include <stddef.h>

extern void swap_heap_elem(void *heapbase, size_t heaplen, size_t size, size_t i, size_t
j);
/*
 * función que intercambia elementos i y j de un array
 * que comienza en la dirección heapbase con heaplen elementos de tamaño size
 * El array empieza en índice 0 que es lo estándar en lenguaje C.
 *
 */


extern void heap_bubbleup(void *heapbase, size_t heaplen, size_t size, size_t i,
                int (*compare)(const void *, const void *));
/*
 * función que ejecuta algoritmo estándar de inserción de UN elemento
 * se supone que el array del heap comienza en la dirección heapbase
 * con heaplen elementos de tamaño size y usa la función compare
 * para comparar elementos del array
 * El array empieza en índice 0 que es lo estándar en lenguaje C.
 *
 */


extern void heap_bubbledown(void *heapbase, size_t heaplen, size_t size, size_t i,
                int (*compare)(const void *, const void *));
/*
 * función que ejecuta algoritmo estándar de heapify o "bubbledown"
 * se supone que el array del heap comienza en la dirección heapbase
 * con heaplen elementos de tamaño size y usa la función compare
 * para comparar elementos del array
 * El array empieza en índice 0 que es lo estándar en lenguaje C.
```

```
 *
 */


extern void build_heap(void *heapbase, size_t heaplen, size_t size,
                int (*compare)(const void *, const void *));
/*
 * función que ejecuta algoritmo estándar para crear un heap
 * se supone que el array del heap comienza en la dirección heapbase
 * con heaplen elementos de tamaño size y usa la función compare
 * para comparar elementos del array
 * El array empieza en índice 0 que es lo estándar en lenguaje C.
 *
 */
```