TECNOLÓGICO DE COSTA RICA

Escuela de Ingeniería en Computación

DOCUMENTACIÓN TÉCNICA

Proyecto 3A: Compresor/Descompresor de Huffman

Algoritmo de Compresión sin Pérdida

Programación en C++ - Estructuras de Datos

Estudiante: Lee Sang Cheol (Diego)

Carné: 2024801079

Carrera: Ingeniería en Computación

Curso: Estructuras de Datos

Profesor: Prof. Víctor Manuel Garro Abarca

Índice

1 Introducción

1.1 Descripción del Proyecto

Este documento presenta la implementación del algoritmo de Huffman para compresión y descompresión de archivos, desarrollado como parte del Proyecto 3A del curso de Estructuras de Datos. El algoritmo de Huffman es un método de compresión sin pérdida que asigna códigos de longitud variable a los símbolos basándose en su frecuencia de aparición.

1.2 Objetivos

- Implementar el algoritmo de Huffman en C++
- Demostrar el uso de operadores de bits para manipulación de datos
- Comprimir y descomprimir diferentes tipos de archivos
- Analizar la efectividad de la compresión según el tipo de archivo
- Verificar la integridad de los archivos restaurados

1.3 Alcance

El proyecto incluye:

- Compresor de archivos usando códigos de Huffman
- Descompresor que restaura archivos a su estado original
- Tabla de frecuencias y códigos generados
- Pruebas con múltiples tipos de archivos (.txt, .bmp, .jpg, .exe)
- Documentación técnica completa

2 Marco Teórico

2.1 Algoritmo de Huffman

El algoritmo de Huffman, desarrollado por David A. Huffman en 1952, es un método de compresión de datos sin pérdida que utiliza códigos de longitud variable para representar símbolos. Los símbolos más frecuentes reciben códigos más cortos, mientras que los menos frecuentes reciben códigos más largos.

2.1.1. Principios Fundamentales

- **Códigos Prefix-Free:** Ningún código es prefijo de otro, garantizando decodificación única
- **Frecuencia Óptima:** Los símbolos más frecuentes tienen códigos más cortos
- **Árbol Binario:** La estructura del árbol determina los códigos asignados

2.2 Construcción del Árbol de Huffman

El árbol se construye mediante un algoritmo greedy:

1. Calcular la frecuencia de cada símbolo en el archivo
2. Crear un nodo hoja para cada símbolo con su frecuencia
3. Insertar todos los nodos en una cola de prioridad (min-heap)
4. Mientras haya más de un nodo en la cola:
   - Extraer los dos nodos con menor frecuencia
   - Crear un nodo padre con frecuencia = suma de frecuencias
   - Insertar el nodo padre en la cola
5. El último nodo es la raíz del árbol

2.3 Generación de Códigos

Los códigos se generan mediante recorrido del árbol:

- Rama izquierda = bit 0
- Rama derecha = bit 1
- Los códigos se obtienen al llegar a las hojas

3 Uso de Operadores de Bits

NOTA IMPORTANTE: Esta sección vale 20 puntos del proyecto y demuestra el uso práctico de operadores de bits en el manejo de archivos binarios.

3.1 Operador OR - Activar Bits

3.1.1. Ubicación en el Código

Archivo: Huffman.cpp, Línea 210 (función `ComprimirArchivo`)

3.1.2. Código

```
// Convertir primeros 8 bits a byte
for (int i = 0; i < 8; i++) {
    if (bufferBits[i] == '1') {
        byteComprimido |= (1 << (7 - i));
        // Operador OR para ACTIVAR bit
    }
}
```

3.1.3. Explicación

El operador OR se utiliza para **activar (poner en 1)** bits específicos. La expresión crea una máscara con un solo bit activo en la posición deseada, y el operador OR combina el byte actual con la máscara. Si el bit ya está en 1, permanece en 1. Si el bit está en 0 y la máscara tiene 1, el bit se activa.

Ejemplo Visual:

```
byte inicial:    00000000
máscara (i=0):   10000000  (1 << 7)
resultado:       10000000  (byte | máscara)

byte actual:     10000000
máscara (i=3):   00010000  (1 << 4)
resultado:       10010000  (byte | máscara)
```

3.2 Operador Left Shift - Desplazar Bits

3.2.1. Ubicación en el Código

Archivo: Huffman.cpp, Línea 210

3.2.2. Código

```
1 byteComprimido |= (1 << (7 - i));
2 // Operador Left Shift para posicionar bit
```

3.2.3. Explicación

El operador Left Shift desplaza bits hacia la izquierda. La expresión desplaza el bit 1 hacia la izquierda n posiciones, lo que equivale a multiplicar por 2 elevado a n. Se usa para crear máscaras de bits en posiciones específicas.

Ejemplo Visual:

```
1 << 0 = 00000001  (posición 0)
1 << 1 = 00000010  (posición 1)
1 << 2 = 00000100  (posición 2)
1 << 7 = 10000000  (posición 7)
```

3.3 Operador Right Shift - Extraer Bits

3.3.1. Ubicación en el Código

Archivo: Huffman.cpp, Línea 285 (función `DescomprimirArchivo`)

3.3.2. Código

```
// Leer byte bit por bit
for (int i = 7; i >= 0; i--) {
    // Extraer bit
    int bit = (byte >> i) & 1;
    // Right Shift para acceder al bit en posicion i
    codigoActual += (bit ? '1' : '0');

    // Buscar en tabla de decodificacion
    if (tablaDecode.find(codigoActual) !=
        tablaDecode.end()) {
        unsigned char simbolo = tablaDecode[codigoActual];
        fwrite(&simbolo, 1, 1, destino);
        bytesDescomprimidos++;
        codigoActual = "";
    }
}
```

3.3.3. Explicación

El operador Right Shift desplaza bits hacia la derecha. Desplaza todos los bits n posiciones a la derecha, lo que equivale a dividir por 2 elevado a n (división entera). Se usa para acceder a bits en posiciones específicas.

Ejemplo Visual:

```
byte original: 10110101

byte >> 0 = 10110101  (sin desplazamiento)
byte >> 1 = 01011010  (desplaza 1 posición)
byte >> 3 = 00010110  (desplaza 3 posiciones)
byte >> 7 = 00000001  (desplaza 7 posiciones)
```

3.4 Operador AND - Leer Bits Específicos

3.4.1. Ubicación en el Código

Archivo: Huffman.cpp, Línea 285

3.4.2. Código

```
int bit = (byte >> i) & 1;
// Operador AND para EXTRAER el bit menos significativo
```

3.4.3. Explicación

El operador AND realiza una operación lógica bit a bit. La expresión con 1 extrae solo el bit menos significativo (LSB). Actúa como una **máscara** que filtra bits no deseados. El resultado es 1 si el bit está activo, 0 si está inactivo.

Tabla de Verdad AND:

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Ejemplo Visual:

```
byte >> 3:    00010110
máscara:      00000001  (constante 1)
resultado:    00000000  (bit en posición 0 es 0)

byte >> 2:    00101101
máscara:      00000001
resultado:    00000001  (bit en posición 0 es 1)
```

3.5 Aplicación Práctica Combinada

3.5.1. Escritura de Bits (Compresión)

```
unsigned char buffer = 0;
int contadorBits = 0;

// Para cada bit del codigo Huffman
for (char c : codigoHuffman) {
    int bit = (c == '1') ? 1 : 0;

    // Desplazar buffer y anadir nuevo bit
    buffer = (buffer << 1) | bit;
    contadorBits++;

    // Cuando tengamos 8 bits, escribir al archivo
    if (contadorBits == 8) {
        fwrite(&buffer, 1, 1, archivoDestino);
        buffer = 0;
        contadorBits = 0;
    }
}
```

3.5.2. Lectura de Bits (Descompresión)

```
unsigned char byte;
string codigoActual = "";

// Leer cada byte del archivo comprimido
while (fread(&byte, 1, 1, archivoOrigen) == 1) {
    // Extraer cada bit del byte
    for (int i = 7; i >= 0; i--) {
        int bit = (byte >> i) & 1;
        codigoActual += (bit ? '1' : '0');
```

```
10
11          // Buscar codigo en tabla
12          if (estaEnTabla(codigoActual)) {
13              char simbolo = obtenerSimbolo(codigoActual);
14              escribirSimbolo(simbolo);
15              codigoActual = "";
16          }
17      }
18  }
```

3.6 Resumen de Operadores

Operador	Nombre	Uso en Huffman	Línea
|	OR	Activar bits en buffer	210
LS	Left Shift	Crear máscaras de bits	210
RS	Right Shift	Extraer bits del byte	285
&	AND	Leer bit específico	285

Cuadro 1: Operadores de bits utilizados en el proyecto

4 Implementación

4.1 Estructura del Programa

El programa está organizado en los siguientes módulos:

1. **Estructuras de Datos:** Nodo del árbol, tabla de frecuencias, tabla de códigos
2. **Cálculo de Frecuencias:** Analiza el archivo y cuenta apariciones
3. **Construcción del Árbol:** Usa cola de prioridad (min-heap)
4. **Generación de Códigos:** Recorrido recursivo del árbol
5. **Compresión:** Escribe header y datos comprimidos
6. **Descompresión:** Lee header y restaura datos originales

4.2 Formato del Archivo Comprimido

El archivo .huf tiene la siguiente estructura:

```
+---------------------------+
|  NUM_SIMBOLOS (4 bytes)   |  Cantidad de símbolos únicos
+---------------------------+
|  SIMBOLO (1 byte)         |  \
|  LONGITUD_CODIGO (4 bytes)|   | Repetido para
|  CODIGO (variable)        |  /  cada símbolo
+---------------------------+
```

```
|  SIZE (4 bytes)           |  Tamaño del archivo original
+---------------------------+
|  DATOS COMPRIMIDOS        |  Bits comprimidos
|  (variable)               |
+---------------------------+
```

5 Pruebas y Resultados

5.1 Tabla de Resultados

Archivo	Tipo	Original	Comprimido	Reducción	Obs.
prueba.txt	.txt	20	6	**70.00 %**	Excelente
documento.txt	.txt	1,824	1,108	**39.25 %**	Buena
numeros.txt	.txt	1,198	493	**58.85 %**	Excelente
imagen.bmp	.bmp	5,796	5,667	**2.23 %**	Baja
foto.jpg	.jpg	3,697	3,546	**4.08 %**	Baja
programa.exe	.exe	624,128	397,974	**36.24 %**	Buena

Cuadro 2: Resultados de compresión para diferentes tipos de archivos

5.2 Verificación de Integridad

Todos los archivos fueron verificados usando comparación binaria:

```
C:\> fc /b prueba.txt prueba.txt.restored
FC: no differences encountered
```

Resultado: 0 diferencias encontradas en todos los casos. Los archivos restaurados son **100 % idénticos** a los originales.

5.3 Análisis de Resultados

5.3.1. Archivos con Alta Compresión

- **prueba.txt (70 %):** Archivo pequeño con alta repetición de caracteres
- **numeros.txt (58.85 %):** Secuencias repetitivas de dígitos
- **documento.txt (39.25 %):** Texto con palabras repetidas
- **programa.exe (36.24 %):** Ejecutable con patrones binarios

5.3.2. Archivos con Baja Compresión

- **imagen.bmp (2.23 %):** Datos relativamente aleatorios
- **foto.jpg (4.08 %):** Ya comprimido con JPEG

6 Conclusiones

1. El algoritmo de Huffman fue implementado exitosamente en C++, logrando compresión y descompresión sin pérdida de datos.
2. Los operadores de bits son fundamentales para la manipulación eficiente de datos a nivel de bit.
3. La efectividad de la compresión depende del tipo de archivo: archivos de texto 39-70 %, archivos binarios 36 %, archivos ya comprimidos 2-4 %.
4. La verificación binaria confirmó que el proceso es lossless (sin pérdida).
5. El proyecto demostró la importancia del manejo de bits en aplicaciones de bajo nivel.

7 Referencias

1. Huffman, D. A. (1952). A Method for the Construction of Minimum-Redundancy Codes.
2. Cormen, T. H., et al. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
3. Zator. Operadores de Bits en C++. http://www.zator.com/Cpp/E4_9_3.htm
4. Garro, V. M. (2025). Material del Curso - Estructuras de Datos.