Huffman-Based File Compression and Decompression System

1st David Boules
Email: dboules@aucegypt.edu

2nd Jana Hassan
Email: janahassan@aucegypt.edu

3rd John Kamal
Email: johnk.kamal@aucegypt.edu

4th Mariam Hassan
Email: mariam_abdo@aucegypt.edu

5th Michael Guirgis
Email: MichaelAyad@aucegypt.edu

***Abstract*—In this project, we implemented a Huffman-based file compression and decompression system. Our implementation includes custom data structures such as a HashMap, DynamicArray, MinHeap, and a bit-level I/O stream. Experimental evaluation on text samples demonstrates reductions in file size, particularly for repetitive inputs. In this report, we present the design, algorithms, data structures, experimental results of the system's performance**

I. Introduction

Data compression techniques reduce the number of bits required to store and transmit information. Huffman coding is one of the most widely used lossless compression techniques due to its efficiency and simplicity [1]. Compression formats used in every-day contexts such as ZIP and JPEG incorporate variants of Huffman coding.

This project implements a Huffman-based file compressor and decompressor in C++. The system constructs a frequency map of input characters, builds a Huffman tree using a MinHeap, generates prefix-free codes (no code is a prefix of another), and writes the compressed output using a bit-stream. Decompression reconstructs the Huffman tree and decodes the bitstream back into the original file.

II. Problem Definition

The goal of this project is to design and implement a Huffman compressor capable of handling byte values (0–255), achieving space savings for files with non-uniform symbol distributions. The system must be able to do all of the following:

- Read an input text file and compute the frequency of each character.
- Build an optimal prefix-free Huffman tree.
- Encode the file using binary codes.
- Serialize the Huffman tree (convert to a sequence of bits/bytes) and store it alongside the encoded bitstream.
- Be able to fully reconstruct the original file without information loss.

III. Methodology

To fulfil the requirements outlined in the problem definition, we split the full problem into six smaller phases to understand what data structures must be implemented at each step:

1) Frequency Analysis

A `HashMap` stores character frequencies from the input file.

2) Tree Construction

A `MinHeap` is used to extract the two least-frequent nodes and merge them into internal Huffman tree nodes.

3) Huffman Code Generation

A depth-first traversal assigns binary strings to each symbol:

$$\text{left child} \rightarrow 0, \qquad \text{right child} \rightarrow 1$$

4) File Encoding

A custom `BitStream` class writes individual bits and bytes to the compressed file.

5) Serialization

The Huffman tree is serialized in preorder:

- 0 = internal node
- 1 + byte value = leaf node

6) Decompression

The tree is reconstructed and decodes the bit-stream.

IV. Specification of Algorithms

This section describes the main algorithms used in the implementation of the Huffman compression system.

A. Huffman Tree Construction

The `buildHuffmanTree` algorithm takes a character-frequency map and constructs a prefix-free binary tree.

```
function buildHuffmanTree(freqMap):
    heap = MinHeap()

    for each character c with frequency f:
        heap.insert(new Node(c, f))

    while heap.size() > 1:
        left = heap.extractMin()
        right = heap.extractMin()
        parent = new Node(left.freq + right.freq,
            left, right)
        heap.insert(parent)

    return heap.extractMin()
```

The algorithm repeatedly extracts the two least frequent nodes from a `MinHeap` and merges them into a new parent node, ensuring that frequently occuring characters have shorter codewords, minimizing the total weighted path length. This continues until only one node remains: the root of the Huffman tree. The use of a MinHeap ensures that each merge operation selects the optimal pair of nodes, resulting in an overall $O(n \log n)$ construction time.

B. Huffman Code Generation

Once the tree is constructed, a depth-first traversal assigns binary strings to each symbol. Left edges correspond to 0 and right edges to 1. Each leaf node receives a unique bit-string, and due to the tree structure, the resulting code is prefix-free.

```
function generateCodes(node, code):
    if node is leaf:
        codes[node.data] = code
        return
    generateCodes(node.left,  code + "0")
    generateCodes(node.right, code + "1")
```

This algorithm performs a full traversal of the tree, running in $O(n)$, where n is the number of nodes. The implementation handles the single-character file case by assigning a default code 0.

C. Huffman Tree Serialization

During compression, the Huffman tree must be stored inside the output file so that decompression can reconstruct it. Serialization converts the tree into a linear bit sequence using a preorder traversal. Internal nodes are written as a single bit 0, while leaf nodes are written as 1 followed by the 8-bit character value.

```
function serializeTree(node, bitstream):
    if node is a leaf:
        bitstream.writeBit(1)
        bitstream.writeByte(node.data)
        return

    bitstream.writeBit(0)
    serializeTree(node.left,  bitstream)
    serializeTree(node.right, bitstream)
```

This representation is compact and uniquely decodable. Since every leaf writes 1 byte after its marker, the serialized structure is a prefix-free preorder encoding of the tree. Serialization runs in $O(n)$, where n is the number of nodes.

D. Huffman Tree Deserialization

During decompression, the serialized tree is read from the bitstream and reconstructed recursively. Reading a bit determines whether to create a leaf or internal node. For leaf nodes, the next byte represents the character; for internal nodes, recursive calls rebuild the left and right subtrees.

```
function deserializeTree(bitstream):
    flag = bitstream.readBit()

    if flag == 1:
        ch = bitstream.readByte()
        return new Node(ch, 0)

    left  = deserializeTree(bitstream)
    right = deserializeTree(bitstream)
    return new Node(0, left, right)
```

This algorithm guarantees the exact reconstruction of the Huffman tree used during compression. Its runtime is $O(n)$.

E. Bit-Level Encoding

File data is encoded at the bit level rather than the byte level to maximize compression efficiency. The `BitStream` class buffers bits into an 8-bit holder and writes to disk only when a full byte is accumulated.

```
function writeBit(bit):
    if bit == 1:
        byteHolder |= (1 << (7 - bitPosition))

    bitPosition += 1

    if bitPosition == 8:
        file.write(byteHolder)
        byteHolder = 0
        bitPosition = 0
```

This mechanism supports writing both bits and full bytes. The corresponding read operations reverse the process during decompression. The overall efficiency is $O(n)$, where n is the number of encoded bits.

V. Data Specifications

Experiments used three representative datasets:

- `test1.txt` — short English sample.
- `test2.txt` — long, highly repetitive sample.
- `test_input.txt` — test string “AAABBC”.

Metrics examined:

- Original file size
- Compressed size
- Compression ratio $R = \frac{\text{Compressed}}{\text{Original}}$

VI. Experimental Results

File	Original (B)	Compressed (B)	Ratio
`test1.txt`	142	87	0.613
`test2.txt`	7300	1840	0.252
`test_input.txt`	6	9	1.50

Observations

- Highly repetitive files achieve excellent compression.
- Small files may grow slightly due to header + tree serialization overhead.
- Natural English text compresses moderately.

VII. Analysis and Critique

Key observations include:

- Tree serialization dominates cost for tiny files.
- HashMap (linear probing) is efficient for English text.
- MinHeap operations align with expected $O(n \log n)$ complexity.
- BitStream provides an efficient bit-packing abstraction.

Overall, the implementation is efficient and aligned with Huffman’s theoretical performance.

VIII. Conclusions

This project successfully implements a complete Huffman compression system with custom data structures. Experiments validate the approach, showing strong performance for repetitive data.

Acknowledgements

We thank the course instructors and teaching assistants for their support and valuable feedback during milestone discussions.

References

[1] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

Appendix

A. BitStream.cpp

```cpp


#include "BitStream.h"



// Initializes the bit stream with the given file and mode (read/write).
BitStream::BitStream(std::fstream* fileStream, bool mode) {
    file = fileStream;
    writingMode = mode;
    byteHolder = 0;
    bitPosition = 0;
}

BitStream::~BitStream() {
    if (writingMode && bitPosition > 0) {
        pushRemainingBits();
    }
}

/*
  writeBit:
  Writes a single bit (0 or 1) into the byteHolder.
  When 8 bits are collected, writes the byte to the file.
 */
void BitStream::writeBit(bool bitValue) {
    if (bitValue) {
        byteHolder = byteHolder | (1 << (7 - bitPosition));  // set bit at correct position
    }

    bitPosition++;

    if (bitPosition == 8) {
        file->put(byteHolder);
        byteHolder = 0;
        bitPosition = 0;
    }
}


// writeByte function: writes 8 bits (a full byte) one by one.

void BitStream::writeByte(unsigned char value) {
    for (int i = 7; i >= 0; i--) {
        bool bitValue = ((value >> i) % 2);
        writeBit(bitValue);

    }
}


// pushRemainingBits:writes any leftover bits (less than 8) to the file.

void BitStream::pushRemainingBits() {
    if (bitPosition > 0) {
        file->put(byteHolder);
        byteHolder = 0;
        bitPosition = 0;
    }
}

/*
 readBit:
 Reads a single bit from the file.
 Loads a new byte when the current one is done.
 */
bool BitStream::readBit() {
    if (bitPosition == 0) {
        if (file->eof()) {
            return false;  // Prevent reading past EOF
        }
        byteHolder = file->get();
    }

    bool bit = (byteHolder >> (7 - bitPosition)) & 1;
    bitPosition++;
    if (bitPosition == 8) {
        bitPosition = 0;
    }

    return bit;
}


// readByte: Reads 8 bits (1 byte) by calling readBit repeatedly.
unsigned char BitStream::readByte() {
    unsigned char value = 0;
    for (int i = 0; i < 8; i++) {
        value = (value << 1) | readBit();
    }
    return value;
}

// hasMoreBits:Checks if the file still has bits left to read.

bool BitStream::hasMoreBits() {
    return !file->eof();
}



// resetStream:Resets the buffer and bit position.

void BitStream::resetStream() {
    bitPosition = 0;
    byteHolder=0;
}
```

B. DynamicArray.cpp

```cpp

#include "DynamicArray.h"
#include "HuffmanNode.h"
```

```

// Method Implementations
template <typename T>
DynamicArray<T>::DynamicArray(size_t cap)
{
    capacity = (cap > 0) ? cap : 1; // check if the capacity is a valid positive number
    size = 0;
    array = new T[capacity];
}

template <typename T>
DynamicArray<T>::~DynamicArray()
{
    delete[] array;
}

template <typename T>
void DynamicArray<T>::resize()
{
    size_t newCap = (capacity == 0) ? 1 : capacity * 2; // scale factor is subject to change as doubling is not very effective for large capacities
    reserve(newCap);
}

template <typename T>
void DynamicArray<T>::reserve(size_t newCapacity) // A function for adjusting the capacity
{
    if (newCapacity <= capacity)
        return;
    T *newArray = new T[newCapacity];
    for (size_t i = 0; i < size; ++i)
        newArray[i] = array[i];
    delete[] array;
    array = newArray;
    capacity = newCapacity;
}

template <typename T>
void DynamicArray<T>::shrinkToFit()
{
    if (size == capacity)
        return;
    size_t newCap = (size > 0) ? size : 1; // shrinking the capacity to free memory
    T *newArray = new T[newCap];
    for (size_t i = 0; i < size; ++i)
        newArray[i] = array[i];
    delete[] array;
    array = newArray;
    capacity = newCap;
}

template <typename T>
void DynamicArray<T>::pushBack(const T &element)
{
    if (size == capacity) // to avoid overflow
        resize();
    array[size++] = element;
}

template<typename T>
void DynamicArray<T>::popBack() {
    if (isEmpty())
        throw std::out_of_range("DynamicArray::popBack: array is empty");
    --size; // Only decrement once
}


template <typename T>
T &DynamicArray<T>::operator[](size_t index)
{
    assert(index < size);
    return array[index];
}

template <typename T>
const T &DynamicArray<T>::operator[](size_t index) const
{
    assert(index < size);
    return array[index];
}

template <typename T>
T &DynamicArray<T>::at(size_t index)
{
    if (index >= size)
        throw std::out_of_range("DynamicArray::at: index out of range");
    return array[index];
}

template <typename T>
const T &DynamicArray<T>::at(size_t index) const
{
    if (index >= size)
        throw std::out_of_range("DynamicArray::at: index out of range");
    return array[index];
}

template <typename T>
size_t DynamicArray<T>::getSize() const
{
    return size;
}

template <typename T>
size_t DynamicArray<T>::getCapacity() const
{
    return capacity;
}

template <typename T>
bool DynamicArray<T>::isEmpty() const
{
    return size == 0;
}

template <typename T>
void DynamicArray<T>::clear()
{
    size = 0;
}

template <typename T>
T *DynamicArray<T>::getData()
{
    return array; // return pointer to elements
}

template <typename T>
const T *DynamicArray<T>::getData() const
{
    return array;
}

template <typename T>
T *DynamicArray<T>::begin()
{
    return array; // return pointer to the first element
```

```
}

template <typename T>
T *DynamicArray<T>::end()
{
    return array + size; // return pointer to the last element
}

template <typename T>
const T *DynamicArray<T>::begin() const
{
    return array; // const method to get the first element
}

template <typename T>
const T *DynamicArray<T>::end() const
{
    return array + size; // const method to get the last element
}

template <typename T>
DynamicArray<T>::DynamicArray(const DynamicArray<T> &other) // A function to copy another array
{
    capacity = other.capacity;
    size = other.size;
    array = new T[capacity];
    for (size_t i = 0; i < size; ++i)
        array[i] = other.array[i];
}

template <typename T>
DynamicArray<T> &DynamicArray<T>::operator=(const DynamicArray<T> &other) // A function to overload the = operator
{
    if (this == &other)
        return *this;
    T *newArray = new T[other.capacity];
    for (size_t i = 0; i < other.size; ++i)
        newArray[i] = other.array[i];
    delete[] array;
    array = newArray;
    capacity = other.capacity;
    size = other.size;
    return *this;
}

template <typename T>
DynamicArray<T>::DynamicArray(DynamicArray<T> &&other) noexcept
{
    array = other.array;
    capacity = other.capacity;
    size = other.size;
    other.array = nullptr;
    other.capacity = 0;
    other.size = 0;
}

template <typename T>
DynamicArray<T> &DynamicArray<T>::operator=(DynamicArray<T> &&other) noexcept
{
    if (this == &other)
        return *this;
    delete[] array;
    array = other.array;
    capacity = other.capacity;
    size = other.size;
    other.array = nullptr;
    other.capacity = 0;
    other.size = 0;
    return *this;
}


// Explicit instantiations for types you use: int, char, HuffmanNode*
template class DynamicArray<int>;
template class DynamicArray<char>;
template class DynamicArray<HuffmanNode*>;
```

C. *HashMap.cpp*

```

#include "HashMap.h"

HashMap::HashMap(int cap) {
    capacity = cap;
    size = 0;
    table = new HashNode[capacity];
}

HashMap::~HashMap() {
    delete[] table;
}

int HashMap::hash(char key) {

    return (int)key % capacity;
}

int HashMap::findIndex(char key) {
    int index = hash(key);
    int start = index;

    while (table[index].occupied) {
        if (!table[index].deleted && table[index].key == key)
            return index;
        index = (index + 1) % capacity;
        if (index == start)
            break; // table full
    }
    return -1;
}

void HashMap::insert(char key, int value) {
    int index = hash(key);

    while (table[index].occupied && !table[index].deleted && table[index].key != key) {
        index = (index + 1) % capacity;
    }

    if (!table[index].occupied || table[index].deleted) {
        table[index].key = key;
        table[index].value = value;
        table[index].occupied = true;
        table[index].deleted = false;
        size++;
    } else {
        table[index].value = value; // update
    }
}

bool HashMap::find(char key, int& value) {
    int index = findIndex(key);
    if (index != -1) {
        value = table[index].value;
        return true;
    }
```

```
    return false;
}

bool HashMap::remove(char key) {
    int index = findIndex(key);
    if (index != -1) {
        table[index].deleted = true;
        size--;
        return true;
    }
    return false;
}

bool HashMap::contains(char key) {
    int dummy;
    return find(key, dummy);
}

int HashMap::getSize() const {
    return size;
}

void HashMap::print() {
    cout << "HashMap contents:\n";
    for (int i = 0; i < capacity; i++) {
        if (table[i].occupied && !table[i].deleted)
            {
            cout << table[i].key << " : " << table[i
                ].value << endl;
        }
    }
}
```

D. *HuffmanNode.cpp*

```

#include "HuffmanNode.h"

/**
 * Constructor for leaf nodes or default
     initialization
 * Handles all byte values 0-255
 */
HuffmanNode::HuffmanNode(unsigned char ch, unsigned
    int freq) {
    data = ch;
    frequency = freq;
    left = nullptr;
    right = nullptr;
}

/**
 * Constructor for internal nodes with children
 * Used when merging two nodes in Huffman tree
     construction
 */
HuffmanNode::HuffmanNode(unsigned int freq,
    HuffmanNode* l, HuffmanNode* r) {
    data = 0;
    frequency = freq;
    left = l;
    right = r;
}

/**
 * Check if this is a leaf node (contains actual
     data)
 */
bool HuffmanNode::isLeaf() const {
    return (left == nullptr && right == nullptr);
}

/**
 * Recursively delete all child nodes
 */
HuffmanNode::~HuffmanNode() {
    delete left;
    delete right;
}
```

E. *HuffmanZipper.cpp*

```

#include "HuffmanZipper.h"
#include "MiniHeap.h"
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

/**
 * Build frequency map using HashMap
 */
void buildFrequencyMap(const string& filename,
    HashMap& freqMap) {
    ifstream file(filename, ios::binary);
    if (!file.is_open()) {
        cerr << "Error: Cannot open file " <<
            filename << endl;
        return;
    }

    unsigned char ch;
    while (file.get((char&)ch)) {
        int currentFreq = 0;
        char charKey = (char)ch;  // Explicit cast
        if (freqMap.find(charKey, currentFreq)) {
            freqMap.insert(charKey, currentFreq + 1)
                ;
        } else {
            freqMap.insert(charKey, 1);
        }
    }

    file.close();
}

/**
 * Build Huffman tree from HashMap
 */
HuffmanNode* buildHuffmanTree(HashMap& freqMap) {
    MinHeap heap;

    // Iterate through all possible byte values
    for (int i = 0; i < 256; i++) {
        int freq;
        if (freqMap.find((unsigned char)i, freq) &&
            freq > 0) {
            HuffmanNode* node = new HuffmanNode((
                unsigned char)i, freq);
            heap.insert(node);
        }
    }

    // Handle edge cases
    if (heap.empty()) {
        return nullptr;
    }

    if (heap.size() == 1) {
        HuffmanNode* single = heap.extractMin();
        // Create a dummy right node for symmetry
        HuffmanNode* dummy = new HuffmanNode(0, 0);
        HuffmanNode* root = new HuffmanNode(single->
            frequency, single, dummy);
```

```
        return root;
    }


    // Build tree
    while (heap.size() > 1) {
        HuffmanNode* left = heap.extractMin();
        HuffmanNode* right = heap.extractMin();
        HuffmanNode* parent = new HuffmanNode(
            left->frequency + right->frequency,
            left,
            right
        );
        heap.insert(parent);
    }

    return heap.extractMin();
}

/**
 * Generate Huffman codes for each character
 * Traverse tree and assign binary codes
 */
void generateCodes(HuffmanNode* root, string code,
    string codes[256]) {
    if (!root) return;

    // Leaf node - store the code
    if (root->isLeaf()) {
        codes[root->data] = code.empty() ? "0" :
            code; // Handle single character case
        return;
    }

    // Traverse left (add '0') and right (add '1')
    generateCodes(root->left, code + "0", codes);
    generateCodes(root->right, code + "1", codes);
}

/**
 * Serialize Huffman tree to compressed file
 * Write tree structure using pre-order traversal
 * Format: 0 for internal node, 1 followed by byte
    value for leaf
 */
void serializeTree(HuffmanNode* root, BitStream& bs)
    {
    if (!root) return;

    if (root->isLeaf()) {
        bs.writeBit(true);  // 1 indicates leaf node
        bs.writeByte(root->data);  // Write the
            character
    } else {
        bs.writeBit(false); // 0 indicates internal
            node
        serializeTree(root->left, bs);
        serializeTree(root->right, bs);
    }
}

/**
 * Deserialize Huffman tree from compressed file
 * Reconstruct tree from pre-order traversal
 */
HuffmanNode* deserializeTree(BitStream& bs) {
    bool isLeaf = bs.readBit();

    if (isLeaf) {
        unsigned char ch = bs.readByte();
        return new HuffmanNode(ch, 0);
    } else {
        HuffmanNode* left = deserializeTree(bs);
        HuffmanNode* right = deserializeTree(bs);
        return new HuffmanNode(0, left, right);
    }
}

/**
 * Compress a file using Huffman encoding
 */
void compressFile(const string& inputFile, const
    string& outputFile) {
    cout << "Compressing " << inputFile << "..." <<
        endl;

    // Step 1: Build frequency map using HashMap
    HashMap freqMap(256);
    buildFrequencyMap(inputFile, freqMap);

    // Step 2: Build Huffman tree
    HuffmanNode* root = buildHuffmanTree(freqMap);
    if (!root) {
        cerr << "Error: Empty file or cannot build
            tree" << endl;
        return;
    }

    // Step 3: Generate codes (keep array for fast
        lookup during encoding)
    string codes[256];
    for (int i = 0; i < 256; i++) {
        codes[i] = "";
    }
    generateCodes(root, "", codes);

    // Step 4: Write compressed file
    fstream outFile(outputFile, ios::out | ios::
        binary);
    if (!outFile.is_open()) {
        cerr << "Error: Cannot create output file"
            << endl;
        delete root;
        return;
    }

    BitStream bs(&outFile, true); // Write mode

    // Write original file size (for decompression
        verification)
    ifstream inFile(inputFile, ios::binary);
    inFile.seekg(0, ios::end);
    unsigned int fileSize = inFile.tellg();
    inFile.seekg(0, ios::beg);

    // Write file size as 4 bytes
    for (int i = 3; i >= 0; i--) {
        bs.writeByte((fileSize >> (i * 8)) & 0xFF);
    }

    // Serialize tree structure
    serializeTree(root, bs);

    // Encode file content
    unsigned char ch;
    while (inFile.get((char&)ch)) {
        string code = codes[ch];
        for (char bit : code) {
            bs.writeBit(bit == '1');
        }
    }

    bs.pushRemainingBits(); // Flush remaining bits

    inFile.close();
    outFile.close();

    // Cleanup
```

```
    delete root;

    cout << "Compression␣complete!␣Output:␣" <<
        outputFile << endl;
}

/**
 * Decompress a Huffman-encoded file
 */
void decompressFile(const string& inputFile, const
    string& outputFile) {
    cout << "Decompressing␣" << inputFile << "..."
        << endl;

    fstream inFile(inputFile, ios::in | ios::binary)
        ;
    if (!inFile.is_open()) {
        cerr << "Error:␣Cannot␣open␣compressed␣file"
             << endl;
        return;
    }

    BitStream bs(&inFile, false); // Read mode

    // Read original file size
    unsigned int fileSize = 0;
    for (int i = 0; i < 4; i++) {
        fileSize = (fileSize << 8) | bs.readByte();
    }

    // Deserialize tree
    HuffmanNode* root = deserializeTree(bs);
    if (!root) {
        cerr << "Error:␣Cannot␣reconstruct␣tree" <<
            endl;
        inFile.close();
        return;
    }

    // Decode content
    ofstream outFile(outputFile, ios::binary);
    if (!outFile.is_open()) {
        cerr << "Error:␣Cannot␣create␣output␣file"
             << endl;
        delete root;
        inFile.close();
        return;
    }

    HuffmanNode* current = root;
    unsigned int bytesWritten = 0;

    while (bytesWritten < fileSize && bs.hasMoreBits
        ()) {
        bool bit = bs.readBit();

        // Traverse tree with null check
        if (bit) {
            if (current->right) {
                current = current->right;
            } else {
                cerr << "Error:␣Invalid␣tree␣
                    traversal␣(right)" << endl;
                break;
            }
        } else {
            if (current->left) {
                current = current->left;
            } else {
                cerr << "Error:␣Invalid␣tree␣
                    traversal␣(left)" << endl;
                break;
            }
        }

        // Reached leaf node
        if (current && current->isLeaf()) {
            outFile.put(current->data);
            bytesWritten++;
            current = root; // Reset to root for
                next character
        }
    }


    outFile.close();
    inFile.close();

    // Cleanup
    delete root;

    cout << "Decompression␣complete!␣Output:␣" <<
        outputFile << endl;
}

/**
 * Display menu
 */
void displayMenu() {
    cout << "\n
        ======================================" <<
         endl;
    cout << "␣␣␣␣Huffman␣File␣Zipper" << endl;
    cout << "
        ======================================" <<
         endl;
    cout << "1.␣Compress␣a␣file" << endl;
    cout << "2.␣Decompress␣a␣file" << endl;
    cout << "3.␣Exit" << endl;
    cout << "
        ======================================" <<
         endl;
    cout << "Enter␣your␣choice:␣";
}
```

F. MiniHeap.cpp

```

#include "MiniHeap.h"

static const int initialCapacity = 16; // choosing a
     small starting capacity to keep it light

// Constructor
MinHeap::MinHeap() {
    size_ = 0;
    capacity_ = initialCapacity;
    data_ = new HuffmanNode*[capacity_];
}

// Destructor
MinHeap::~MinHeap() {
    // Do not delete HuffmanNode*s (ownershp is
        external)
    // Huffman tree destructor will free them later
    delete[] data_; // only internal array is freed
}

void MinHeap::ensureCapacity() {
    if (size_ < capacity_) return; // capacity not
        reached, can exit safely

    int newCap = (capacity_ == 0) ? initialCapacity
        : capacity_ * 2; // double array size
    HuffmanNode** newData = new HuffmanNode*[newCap
        ];

```

```
    for (int i = 0; i < size_; ++i) {
        newData[i] = data_[i];
    }

    delete[] data_;
    data_ = newData;
    capacity_ = newCap;
}

void MinHeap::insert(HuffmanNode* node) {

    if (!node) return;                  // ignore nulls
    ensureCapacity();

    // insertion process:
    data_[size_] = node; // place at the end
    siftUp(size_);       // sift up if needed
    ++size_;             // increment size
}

HuffmanNode* MinHeap::extractMin() {
    if (size_ == 0) return 0;           // nullptr if empty

    HuffmanNode* minNode = data_[0];  // root is the smallest
    data_[0] = data_[size_ - 1];       // move last element to the root
    --size_;                           // decrement size

    if (size_ > 0) siftDown(0);        // restore heap property
    return minNode;                    // caller owns this pointer
}

HuffmanNode* MinHeap::top() const {
    if (size_ == 0) return 0;
    return data_[0];
}

void MinHeap::clear() {
    // nodes are not deleted, ownership is external (Huffman tree will clean up)
    // simply set size = 0
    size_ = 0;
}

void MinHeap::siftUp(int i) {
    // while node is smaller than parent, swap upward
    while (i > 0) {
        int p = parent(i);
        if (less(data_[i], data_[p])) {
            // swap pointers
            HuffmanNode* tmp = data_[i];
            data_[i] = data_[p];
            data_[p] = tmp;

            i = p; // continue from parent's index

        } else {
            break; // heap property satisfied
        }
    }
}

void MinHeap::siftDown(int i) {
    // swap down with smaller child until heap property holds
    while (true) {
        int l = left(i);
        int r = right(i);
        int smallest = i;

        if (l < size_ && less(data_[l], data_[smallest])) smallest = l;
        if (r < size_ && less(data_[r], data_[smallest])) smallest = r;

        if (smallest == i) break;

        // swap with smallest child
        HuffmanNode* tmp = data_[i];
        data_[i] = data_[smallest];
        data_[smallest] = tmp;

        i = smallest; // continue from child swapped with
    }
}
```

G. *main.cpp*

```
#include "HuffmanZipper.h"
#include "HuffmanNode.h"
#include "MiniHeap.h"
#include "BitStream.h"
#include "HashMap.h"
#include "DynamicArray.h"
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>


using namespace std;

int main() {
    int choice;
    string inputFile, outputFile;

    while (true) {
        displayMenu();
        cin >> choice;
        cin.ignore(); // Clear newline

        switch (choice) {
        case 1:
            cout << "Enter input file name: ";
            getline(cin, inputFile);
            cout << "Enter output file name: ";
            getline(cin, outputFile);
            compressFile(inputFile, outputFile);
            break;

        case 2:
            cout << "Enter compressed file name: ";
            getline(cin, inputFile);
            cout << "Enter output file name: ";
            getline(cin, outputFile);
            decompressFile(inputFile, outputFile);
            break;

        case 3:
            cout << "Exiting program. Goodbye!" << endl;
            return 0;

        default:
            cout << "Invalid choice. Please try again." << endl;
        }
    }

```

```
50      return 0;
51  }
```