

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:

left child → 0, right child → 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,⁹ ensuring that frequently occurring 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,¹ and due to the tree structure, the resulting code is prefix-free.³

```

1 function generateCodes(node, code):           5
2     if node is leaf:                         6
3         codes[node.data] = code              7
4         return                                8
5     generateCodes(node.left, code + "0")      9
6     generateCodes(node.right, code + "1")     10

```

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.

```

1 function serializeTree(node, bitstream):          5
2     if node is a leaf:                           6
3         bitstream.writeBit(1)                   7
4         bitstream.writeByte(node.data)          8
5         return                                 9
6
7     bitstream.writeBit(0)                        10
8     serializeTree(node.left, bitstream)          11
9     serializeTree(node.right, bitstream)         12

```

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.

```

1 function deserializeTree(bitstream):           5
2     flag = bitstream.readBit()                  6
3
4     if flag == 1:                            7
5         ch = bitstream.readByte()            8
6         return new Node(ch, 0)               9
7

```

```

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

```

1
2
3 #include "BitStream.h"
4
5
6 // Initializes the bit stream with the given file
7 // and mode (read/write).
8 BitStream::BitStream(std::fstream* fileStream, bool
9     mode) {
10    file = fileStream;
11    writingMode = mode;
12    byteHolder = 0;
13    bitPosition = 0;
14 }
15 BitStream::~BitStream() {
16    if (writingMode && bitPosition > 0) {
17        pushRemainingBits();
18    }
19 }
20 /*
21 writeBit:
22 Writes a single bit (0 or 1) into the byteHolder.
23 When 8 bits are collected, writes the byte to the
24 file.
25 */
26 void BitStream::writeBit(bool bitValue) {
27    if (bitValue) {
28        byteHolder = byteHolder | (1 << (7 -
29            bitPosition)); // set bit at correct
30            position
31    }
32    bitPosition++;
33    if (bitPosition == 8) {
34        file->put(byteHolder);
35        byteHolder = 0;
36        bitPosition = 0;
37    }
38 }
39
40 // writeByte function: writes 8 bits (a full byte)
41 // one by one.
42 void BitStream::writeByte(unsigned char value) {
43    for (int i = 7; i >= 0; i--) {
44
45        bool bitValue = ((value >> i) % 2);
46        writeBit(bitValue);
47    }
48 }
49
50 // pushRemainingBits:writes any leftover bits (less
51 // than 8) to the file.
52
53 void BitStream::pushRemainingBits() {
54    if (bitPosition > 0) {
55        file->put(byteHolder);
56        byteHolder = 0;
57        bitPosition = 0;
58    }
59 }
60
61 /*
62 readBit:
63 Reads a single bit from the file.
64 Loads a new byte when the current one is done.
65 */
66 bool BitStream::readBit() {
67    if (bitPosition == 0) {
68        if (file->eof()) {
69            return false; // Prevent reading past
70            EOF
71        }
72        byteHolder = file->get();
73    }
74
75    bool bit = (byteHolder >> (7 - bitPosition)) &
76        1;
77    bitPosition++;
78    if (bitPosition == 8) {
79        bitPosition = 0;
80    }
81
82    return bit;
83 }
84
85 // readByte: Reads 8 bits (1 byte) by calling
86 // readBit repeatedly.
87 unsigned char BitStream::readByte() {
88    unsigned char value = 0;
89    for (int i = 0; i < 8; i++) {
90        value = (value << 1) | readBit();
91    }
92    return value;
93 }
94
95 // hasMoreBits:Checks if the file still has bits
96 // left to read.
97 bool BitStream::hasMoreBits() {
98    return !file->eof();
99 }
100
101 // resetStream:Resets the buffer and bit position.
102 void BitStream::resetStream() {
103    bitPosition = 0;
104    byteHolder=0;
105 }
```

B. DynamicArray.cpp

```

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

```

4
5
6 // Method Implementations
7 template <typename T>
8 DynamicArray<T>::DynamicArray(size_t cap)
9 {
10     capacity = (cap > 0) ? cap : 1; // check if the
11     // capacity is a valid positive number
12     size = 0;
13     array = new T[capacity];
14 }
15
16 template <typename T>
17 DynamicArray<T>::~DynamicArray()
18 {
19     delete[] array;
20 }
21
22 template <typename T>
23 void DynamicArray<T>::resize()
24 {
25     size_t newCap = (capacity == 0) ? 1 : capacity * 2;
26     // scale factor is subject to change as doubling is not very effective for large capacities
27     reserve(newCap);
28 }
29
30 template <typename T>
31 void DynamicArray<T>::reserve(size_t newCapacity) // A function for adjusting the capacity
32 {
33     if (newCapacity <= capacity)
34         return;
35     T *newArray = new T[newCapacity];
36     for (size_t i = 0; i < size; ++i)
37         newArray[i] = array[i];
38     delete[] array;
39     array = newArray;
40     capacity = newCapacity;
41 }
42
43 template <typename T>
44 void DynamicArray<T>::shrinkToFit()
45 {
46     if (size == capacity)
47         return;
48     size_t newCap = (size > 0) ? size : 1; // shrinking the capacity to free memory
49     T *newArray = new T[newCap];
50     for (size_t i = 0; i < size; ++i)
51         newArray[i] = array[i];
52     delete[] array;
53     array = newArray;
54     capacity = newCap;
55 }
56
57 template <typename T>
58 void DynamicArray<T>::pushBack(const T &element)
59 {
60     if (size == capacity) // to avoid overflow
61         resize();
62     array[size++] = element;
63 }
64
65 template<typename T>
66 void DynamicArray<T>::popBack() {
67     if (isEmpty())
68         throw std::out_of_range("DynamicArray::popBack(): array is empty");
69     --size; // Only decrement once
70 }
71
72 template <typename T>
73 T &DynamicArray<T>::operator[](size_t index)
74 {
75     assert(index < size);
76     return array[index];
77 }
78
79 template <typename T>
80 const T &DynamicArray<T>::operator[](size_t index) const
81 {
82     assert(index < size);
83     return array[index];
84 }
85
86 template <typename T>
87 T &DynamicArray<T>::at(size_t index)
88 {
89     if (index >= size)
90         throw std::out_of_range("DynamicArray::at(): index_out_of_range");
91     return array[index];
92 }
93
94 template <typename T>
95 const T &DynamicArray<T>::at(size_t index) const
96 {
97     if (index >= size)
98         throw std::out_of_range("DynamicArray::at(): index_out_of_range");
99     return array[index];
100 }
101
102 template <typename T>
103 size_t DynamicArray<T>::getSize() const
104 {
105     return size;
106 }
107
108 template <typename T>
109 size_t DynamicArray<T>::getCapacity() const
110 {
111     return capacity;
112 }
113
114 template <typename T>
115 bool DynamicArray<T>::isEmpty() const
116 {
117     return size == 0;
118 }
119
120 template <typename T>
121 void DynamicArray<T>::clear()
122 {
123     size = 0;
124 }
125
126 template <typename T>
127 T *DynamicArray<T>::getData()
128 {
129     return array; // return pointer to elements
130 }
131
132 template <typename T>
133 const T *DynamicArray<T>::getData() const
134 {
135     return array;
136 }
137
138 template <typename T>
139 T *DynamicArray<T>::begin()
140 {
141     return array; // return pointer to the first element

```

```
141 }
142
143 template <typename T>
144 T *DynamicArray<T>::end()
145 {
146     return array + size; // return pointer to the
147     last element
148 }
149
150 template <typename T>
151 const T *DynamicArray<T>::begin() const
152 {
153     return array; // const method to get the first
154     element
155 }
156
157 template <typename T>
158 const T *DynamicArray<T>::end() const
159 {
160     return array + size; // const method to get the
161     last element
162 }
163
164 template <typename T>
165 DynamicArray<T>::DynamicArray(const DynamicArray<T>
166     &other) // A function to copy another array
167 {
168     capacity = other.capacity;
169     size = other.size;
170     array = new T[capacity];
171     for (size_t i = 0; i < size; ++i)
172         array[i] = other.array[i];
173 }
174
175 template <typename T>
176 DynamicArray<T> &DynamicArray<T>::operator=(const
177     DynamicArray<T> &other) // A function to
178     overload the = operator
179 {
180     if (this == &other)
181         return *this;
182     T *newArray = new T[other.capacity];
183     for (size_t i = 0; i < other.size; ++i)
184         newArray[i] = other.array[i];
185     delete[] array;
186     array = newArray;
187     capacity = other.capacity;
188     size = other.size;
189     return *this;
190 }
191
192 template <typename T>
193 DynamicArray<T> &DynamicArray<T>::operator=(DynamicArray<T> &&
194     other) noexcept
195 {
196     array = other.array;
197     capacity = other.capacity;
198     size = other.size;
199     other.array = nullptr;
200     other.capacity = 0;
201     other.size = 0;
202 }
203
204 template <typename T>
205 DynamicArray<T> &DynamicArray<T>::operator=(const
206     DynamicArray<T> &other) noexcept
207 {
208     if (this == &other)
209         return *this;
210     delete[] array;
211     array = other.array;
212     capacity = other.capacity;
213     size = other.size;
214     other.array = nullptr;
```

```
207     other.capacity = 0;
208     other.size = 0;
209     return *this;
210 }
211
212
213 // Explicit instantiations for types you use: int,
214 // char, HuffmanNode*
215 template class DynamicArray<int>;
216 template class DynamicArray<char>;
217 template class DynamicArray<HuffmanNode*>;
```

C. *HashMap.cpp*

```
1 #include "HashMap.h"
2
3 e
4 HashMap::HashMap(int cap) {
5     capacity = cap;
6     size = 0;
7     table = new HashNode[capacity];
8 }
9
10 > HashMap::~HashMap() {
11     delete[] table;
12 }
13
14 int HashMap::hash(char key) {
15
16     return (int)key % capacity;
17 }
18
19 int HashMap::findIndex(char key) {
20     int index = hash(key);
21     int start = index;
22
23     while (table[index].occupied) {
24         if (!table[index].deleted && table[index].key == key)
25             return index;
26         index = (index + 1) % capacity;
27         if (index == start)
28             break; // table full
29     }
30     return -1;
31 }
32
33 void HashMap::insert(char key, int value) {
34     int index = hash(key);
35
36     while (table[index].occupied && !table[index].deleted && table[index].key != key) {
37         index = (index + 1) % capacity;
38     }
39
40     if (!table[index].occupied || table[index].deleted) {
41         table[index].key = key;
42         table[index].value = value;
43         table[index].occupied = true;
44         table[index].deleted = false;
45         size++;
46     } else {
47         table[index].value = value; // update
48     }
49 }
50
51 bool HashMap::find(char key, int& value) {
52     int index = findIndex(key);
53     if (index != -1) {
54         value = table[index].value;
55         return true;
56     }
57 }
```

```

57     return false;
58 }
59
60 bool HashMap::remove(char key) {
61     int index = findIndex(key);
62     if (index != -1) {
63         table[index].deleted = true;
64         size--;
65         return true;
66     }
67     return false;
68 }
69
70 bool HashMap::contains(char key) {
71     int dummy;
72     return find(key, dummy);
73 }
74
75 int HashMap::getSize() const {
76     return size;
77 }
78
79 void HashMap::print() {
80     cout << "HashMap_contents:\n";
81     for (int i = 0; i < capacity; i++) {
82         if (table[i].occupied && !table[i].deleted)
83             {
84                 cout << table[i].key << ":" << table[i]
85                     .value << endl;
86             }
87     }
88 }

```

D. HuffmanNode.cpp

```

1 #include "HuffmanNode.h"
2
3 /**
4  * Constructor for leaf nodes or default
5  * initialization
6  * Handles all byte values 0-255
7  */
8 HuffmanNode::HuffmanNode(unsigned char ch, unsigned
9     int freq) {
10    data = ch;
11    frequency = freq;
12    left = nullptr;
13    right = nullptr;
14 }
15
16 /**
17  * Constructor for internal nodes with children
18  * Used when merging two nodes in Huffman tree
19  * construction
20  */
21 HuffmanNode::HuffmanNode(unsigned int freq,
22     HuffmanNode* l, HuffmanNode* r) {
23     data = 0;
24     frequency = freq;
25     left = l;
26     right = r;
27 }
28
29 /**
30  * Check if this is a leaf node (contains actual
31  * data)
32  */
33 bool HuffmanNode::isLeaf() const {
34     return (left == nullptr && right == nullptr);
35 }
36
37 /**
38  * Insert a character into the Huffman tree
39  * @param ch Character to insert
40  * @param freq Frequency of the character
41  */
42 void HuffmanNode::insert(char ch, int freq) {
43     if (frequency > 0) {
44         HuffmanNode* node = new HuffmanNode(
45             ch, freq);
46         if (left == nullptr) {
47             left = node;
48         } else {
49             right = node;
50         }
51     }
52 }
53
54 /**
55  * Get the frequency of a character
56  * @param ch Character to get frequency for
57  */
58 int HuffmanNode::getFrequency(char ch) const {
59     if (ch == data) {
60         return frequency;
61     }
62     if (left != nullptr) {
63         return left->getFrequency(ch);
64     }
65     if (right != nullptr) {
66         return right->getFrequency(ch);
67     }
68     return 0;
69 }
70
71 /**
72  * Print the Huffman tree
73  */
74 void HuffmanNode::print() const {
75     cout << data << ":" << frequency << endl;
76     if (left != nullptr) {
77         left->print();
78     }
79     if (right != nullptr) {
80         right->print();
81     }
82 }
83
84 /**
85  * Delete the Huffman tree
86  */
87 void HuffmanNode::deleteTree() {
88     delete left;
89     delete right;
90 }

```

```

34     * Recursively delete all child nodes
35     */
36 HuffmanNode::~HuffmanNode() {
37     delete left;
38     delete right;
39 }

```

E. HuffmanZipper.cpp

```

1 #include "HuffmanZipper.h"
2 #include "MiniHeap.h"
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6
7 using namespace std;
8
9 /**
10  * Build frequency map using HashMap
11  */
12 void buildFrequencyMap(const string& filename,
13     HashMap& freqMap) {
14     ifstream file(filename, ios::binary);
15     if (!file.is_open()) {
16         cerr << "Error: Cannot open file" <<
17             filename << endl;
18     }
19
20     unsigned char ch;
21     while (file.get((char&)ch)) {
22         int currentFreq = 0;
23         char charKey = (char)ch; // Explicit cast
24         if (freqMap.find(charKey, currentFreq)) {
25             freqMap.insert(charKey, currentFreq + 1)
26                 ;
27         } else {
28             freqMap.insert(charKey, 1);
29         }
30     }
31
32     file.close();
33 }
34
35 /**
36  * Build Huffman tree from HashMap
37  */
38 HuffmanNode* buildHuffmanTree(HashMap& freqMap) {
39     MinHeap heap;
40
41     // Iterate through all possible byte values
42     for (int i = 0; i < 256; i++) {
43         int freq;
44         if (freqMap.find((unsigned char)i, freq) &&
45             freq > 0) {
46             HuffmanNode* node = new HuffmanNode((
47                 unsigned char)i, freq);
48             heap.insert(node);
49         }
50     }
51
52     // Handle edge cases
53     if (heap.empty()) {
54         return nullptr;
55     }
56
57     if (heap.size() == 1) {
58         HuffmanNode* single = heap.extractMin();
59         // Create a dummy right node for symmetry
60         HuffmanNode* dummy = new HuffmanNode(0, 0);
61         HuffmanNode* root = new HuffmanNode(single->
62             frequency, single, dummy);
63
64         return root;
65     }
66
67     HuffmanNode* left = heap.extractMin();
68     HuffmanNode* right = heap.extractMin();
69
70     HuffmanNode* parent = new HuffmanNode(left->
71         frequency, left, right);
72
73     heap.insert(parent);
74
75     return parent;
76 }

```

```

59         return root;
60     }
61
62     // Build tree
63     while (heap.size() > 1) {
64         HuffmanNode* left = heap.extractMin();
65         HuffmanNode* right = heap.extractMin();
66         HuffmanNode* parent = new HuffmanNode(
67             left->frequency + right->frequency,
68             left,
69             right
70         );
71         heap.insert(parent);
72     }
73
74     return heap.extractMin();
75 }
76
77 /**
78 * Generate Huffman codes for each character
79 * Traverse tree and assign binary codes
80 */
81 void generateCodes(HuffmanNode* root, string code,
82     string codes[256]) {
83     if (!root) return;
84
85     // Leaf node - store the code
86     if (root->isLeaf()) {
87         codes[root->data] = code.empty() ? "0" :
88             code; // Handle single character case
89         return;
90     }
91
92     // Traverse left (add '0') and right (add '1')
93     generateCodes(root->left, code + "0", codes);
94     generateCodes(root->right, code + "1", codes);
95 }
96
97 /**
98 * Serialize Huffman tree to compressed file
99 * Write tree structure using pre-order traversal
100 * Format: 0 for internal node, 1 followed by byte
101 *          value for leaf
102 */
103 void serializeTree(HuffmanNode* root, BitStream& bs)
104 {
105     if (!root) return;
106
107     if (root->isLeaf()) {
108         bs.writeBit(true); // 1 indicates leaf node
109         bs.writeByte(root->data); // Write the
110             character
111     } else {
112         bs.writeBit(false); // 0 indicates internal
113             node
114         serializeTree(root->left, bs);
115         serializeTree(root->right, bs);
116     }
117 }
118
119 /**
120 * Deserialize Huffman tree from compressed file
121 * Reconstruct tree from pre-order traversal
122 */
123 HuffmanNode* deserializeTree(BitStream& bs) {
124     bool isLeaf = bs.readBit();
125
126     if (isLeaf) {
127         unsigned char ch = bs.readByte();
128         return new HuffmanNode(ch, 0);
129     } else {
130         HuffmanNode* left = deserializeTree(bs);
131         HuffmanNode* right = deserializeTree(bs);
132
133         return new HuffmanNode(0, left, right);
134     }
135 }
136
137 /**
138 * Compress a file using Huffman encoding
139 */
140 void compressFile(const string& inputFile, const
141 string& outputFile) {
142     cout << "Compressing " << inputFile << "..." <<
143         endl;
144
145     // Step 1: Build frequency map using HashMap
146     HashMap freqMap(256);
147     buildFrequencyMap(inputFile, freqMap);
148
149     // Step 2: Build Huffman tree
150     HuffmanNode* root = buildHuffmanTree(freqMap);
151     if (!root) {
152         cerr << "Error: Empty file or cannot build
153             tree" << endl;
154         return;
155     }
156
157     // Step 3: Generate codes (keep array for fast
158     //          lookup during encoding)
159     string codes[256];
160     for (int i = 0; i < 256; i++) {
161         codes[i] = "";
162     }
163     generateCodes(root, "", codes);
164
165     // Step 4: Write compressed file
166     fstream outFile(outputFile, ios::out | ios::
167         binary);
168     if (!outFile.is_open()) {
169         cerr << "Error: Cannot create output file"
170             << endl;
171         delete root;
172         return;
173     }
174
175     BitStream bs(&outFile, true); // Write mode
176
177     // Write original file size (for decompression
178     //          verification)
179     ifstream inFile(inputFile, ios::binary);
180     inFile.seekg(0, ios::end);
181     unsigned int fileSize = inFile.tellg();
182     inFile.seekg(0, ios::beg);
183
184     // Write file size as 4 bytes
185     for (int i = 3; i >= 0; i--) {
186         bs.writeByte((fileSize >> (i * 8)) & 0xFF);
187     }
188
189     // Serialize tree structure
190     serializeTree(root, bs);
191
192     // Encode file content
193     unsigned char ch;
194     while (inFile.get((char&)ch)) {
195         string code = codes[ch];
196         for (char bit : code) {
197             bs.writeBit(bit == '1');
198         }
199     }
200
201     bs.pushRemainingBits(); // Flush remaining bits
202
203     inFile.close();
204     outFile.close();
205
206     // Cleanup

```

```

194     delete root;
195
196     cout << "Compression_complete!_Output:_" <<
197         outputFile << endl;
198 }
199 /**
200 * Decompress a Huffman-encoded file
201 */
202 void decompressFile(const string& inputFile, const
203     string& outputFile) {
204     cout << "Decompressing_" << inputFile << "..." <<
205         endl;
206
207     fstream inFile(inputFile, ios::in | ios::binary);
208     ;
209     if (!inFile.is_open()) {
210         cerr << "Error:_Cannot_open_compressed_file" <<
211             endl;
212         return;
213     }
214
215     BitStream bs(&inFile, false); // Read mode
216
217     // Read original file size
218     unsigned int fileSize = 0;
219     for (int i = 0; i < 4; i++) {
220         fileSize = (fileSize << 8) | bs.readByte();
221     }
222
223     // Deserialize tree
224     HuffmanNode* root = deserializeTree(bs);
225     if (!root) {
226         cerr << "Error:_Cannot_reconstruct_tree" <<
227             endl;
228         inFile.close();
229         return;
230     }
231
232     // Decode content
233     ofstream outFile(outputFile, ios::binary);
234     if (!outFile.is_open()) {
235         cerr << "Error:_Cannot_create_output_file" <<
236             endl;
237         delete root;
238         inFile.close();
239         return;
240     }
241
242     HuffmanNode* current = root;
243     unsigned int bytesWritten = 0;
244
245     while (bytesWritten < fileSize && bs.hasMoreBits())
246     {
247         bool bit = bs.readBit();
248
249         // Traverse tree with null check
250         if (bit) {
251             if (current->right) {
252                 current = current->right;
253             } else {
254                 cerr << "Error:_Invalid_tree_"
255                     traversal_(right)" << endl;
256                 break;
257             }
258         } else {
259             if (current->left) {
260                 current = current->left;
261             } else {
262                 cerr << "Error:_Invalid_tree_"
263                     traversal_(left)" << endl;
264                 break;
265             }
266         }
267     }
268
269     // Reached leaf node
270     if (current && current->isLeaf()) {
271         outFile.put(current->data);
272         bytesWritten++;
273         current = root; // Reset to root for
274             next character
275     }
276
277     outFile.close();
278     inFile.close();
279
280     // Cleanup
281     delete root;
282
283     cout << "Decompression_complete!_Output:_" <<
284         outputFile << endl;
285 }
286
287 /**
288 * Display menu
289 */
290 void displayMenu() {
291     cout << "\n"
292         =====
293     endl;
294     cout << "_____Huffman_Zipper" << endl;
295     cout << "
296         =====
297     endl;
298     cout << "1._Compress_a_file" << endl;
299     cout << "2._Decompress_a_file" << endl;
300     cout << "3._Exit" << endl;
301     cout << "
302         =====
303     endl;
304     cout << "Enter_your_choice:_";
305 }

```

F. MiniHeap.cpp

```

1 #include "MiniHeap.h"
2
3 static const int initialCapacity = 16; // choosing a
4             small starting capacity to keep it light
5
6 // Constructor
7 MinHeap::MinHeap() {
8     size_ = 0;
9     capacity_ = initialCapacity;
10    data_ = new HuffmanNode*[capacity_];
11}
12
13 // Destructor
14 MinHeap::~MinHeap() {
15    // Do not delete HuffmanNode*s (ownership is
16        external)
17    // Huffman tree destructor will free them later
18    delete[] data_; // only internal array is freed
19}
20
21 void MinHeap::ensureCapacity() {
22    if (size_ < capacity_) return; // capacity not
23        reached, can exit safely
24
25    int newCap = (capacity_ == 0) ? initialCapacity
26        : capacity_ * 2; // double array size
27    HuffmanNode** newData = new HuffmanNode*[newCap];
28
29}

```

```

26     for (int i = 0; i < size_; ++i) {
27         newData[i] = data_[i];
28     }
29
30     delete[] data_;
31     data_ = newData;
32     capacity_ = newCap;
33 }
34
35 void MinHeap::insert(HuffmanNode* node) {
36
37     if (!node) return; // ignore nulls
38     ensureCapacity();
39
40     // insertion process:
41     data_[size_] = node; // place at the end
42     siftUp(size_); // sift up if needed
43     ++size_; // increment size
44 }
45
46 HuffmanNode* MinHeap::extractMin() {
47     if (size_ == 0) return 0; // nullptr if empty
48
49     HuffmanNode* minNode = data_[0]; // root is the smallest
50     data_[0] = data_[size_ - 1]; // move last element to the root
51     --size_; // decrement size
52
53     if (size_ > 0) siftDown(0); // restore heap property
54     return minNode; // caller owns this pointer
55 }
56
57 HuffmanNode* MinHeap::top() const {
58     if (size_ == 0) return 0;
59     return data_[0];
60 }
61
62 void MinHeap::clear() {
63     // nodes are not deleted, ownership is external
64     // (Huffman tree will clean up)
65     // simply set size = 0
66     size_ = 0;
67 }
68
69 void MinHeap::siftUp(int i) {
70     // while node is smaller than parent, swap upward
71     while (i > 0) {
72         int p = parent(i);
73         if (less(data_[i], data_[p])) {
74             // swap pointers
75             HuffmanNode* tmp = data_[i];
76             data_[i] = data_[p];
77             data_[p] = tmp;
78
79             i = p; // continue from parent's index
80
81         } else {
82             break; // heap property satisfied
83         }
84     }
85
86 void MinHeap::siftDown(int i) {
87     // swap down with smaller child until heap property holds
88     while (true) {
89         int l = left(i);
90
91         int r = right(i);
92         int smallest = i;
93
94         if (l < size_ && less(data_[l], data_[smallest])) smallest = l;
95         if (r < size_ && less(data_[r], data_[smallest])) smallest = r;
96
97         if (smallest == i) break;
98
99         // swap with smallest child
100        HuffmanNode* tmp = data_[i];
101        data_[i] = data_[smallest];
102        data_[smallest] = tmp;
103
104        i = smallest; // continue from child swapped with
105    }
}

```

G. main.cpp

```

1 #include "HuffmanZipper.h"
2 #include "HuffmanNode.h"
3 #include "MinHeap.h"
4 #include "BitStream.h"
5 #include "HashMap.h"
6 #include "DynamicArray.h"
7 #include <iostream>
8 #include <fstream>
9 #include <string>
10 #include <cstring>
11
12 using namespace std;
13
14 int main() {
15     int choice;
16     string inputFile, outputFile;
17
18     while (true) {
19         displayMenu();
20         cin >> choice;
21         cin.ignore(); // Clear newline
22
23         switch (choice) {
24             case 1:
25                 cout << "Enter input file name: ";
26                 getline(cin, inputFile);
27                 cout << "Enter output file name: ";
28                 getline(cin, outputFile);
29                 compressFile(inputFile, outputFile);
30                 break;
31
32             case 2:
33                 cout << "Enter compressed file name: ";
34                 getline(cin, inputFile);
35                 cout << "Enter output file name: ";
36                 getline(cin, outputFile);
37                 decompressFile(inputFile, outputFile);
38                 break;
39
40             case 3:
41                 cout << "Exiting program. Goodbye! " << endl;
42                 return 0;
43
44             default:
45                 cout << "Invalid choice. Please try again." << endl;
46         }
47     }
48
49

```

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