Mini-Project Report

Abdelwahab Hassan 202201281

Adaptive Huffman Coding

Src Files/

```
1  public class Node {
 1      public int weight;
 2      public int symbol;
 3      public Node parent;
 4      public Node leftChild;
 5      public Node rightChild;
 6      public int orderNumber;
 7
 8      public Node(int weight, int orderNumber) {
 9          this.weight = weight;
10          this.symbol = -1;
11          this.orderNumber = orderNumber;
12          this.parent = null;
13          this.leftChild = null;
14          this.rightChild = null;
15      }
16
17      public Node(int weight, int symbol, int orderNumber) {
18          this.weight = weight;
19          this.symbol = symbol;
20          this.orderNumber = orderNumber;
21          this.parent = null;
22          this.leftChild = null;
23          this.rightChild = null;
24      }
25
26      public boolean isLeaf() {
27          return leftChild == null && rightChild == null;
28      }
29
30      public boolean isRoot() {
31          return parent == null;
32      }
33
34      @Override
35      public String toString() {
36          if (isLeaf()) {
37              return "Leaf[symbol=" + (char)symbol + ", weight=" + weight + ", order=" + orderNumber + "]";
38          } else {
39              return "Node[weight=" + weight + ", order=" + orderNumber + "]";
40          }
41      }
42  }
~
```

Node class:

- weight: The frequency of the symbol or node.
- symbol: The symbol represented by the node, or -1 for internal nodes.
- parent: The parent node of this node.
- leftChild: The left child of this node.
- rightChild: The right child of this node.
- orderNumber: The order number assigned to the node for ordering purposes.

Constructors:

- Node(int weight, int orderNumber):
  - Creates an internal node with the given weight and order number, with no symbol assigned.

- Node(int weight, int symbol, int orderNumber):
  - Creates a leaf node with the given weight, symbol, and order number.

Methods:

- isLeaf():
  - Returns `true` if the node is a leaf (has no children), otherwise `false`.

- isRoot():
  - Returns `true` if the node is the root (has no parent), otherwise `false`.

- toString():
  - Returns a string representation of the node, indicating whether it is a leaf or internal node, and its attributes (symbol, weight, order number).

```
public class HuffmanTree {
    private Node root;
    private Node NYT;
    private int nextOrderNumber;
    private Map<Integer, Node> symbolToNode;
    private List<Node> nodeSwaps;

    public HuffmanTree() {
        nextOrderNumber = 512;
        symbolToNode = new HashMap<>();
        nodeSwaps = new ArrayList<>();

        NYT = new Node(0, -1, nextOrderNumber--);
        root = NYT;
    }

    public Node getRoot() {
        return root;
    }

    public Node getNYT() {
        return NYT;
    }

    public List<Node> getNodeSwaps() {
        return nodeSwaps;
    }

    public void clearNodeSwaps() {
        nodeSwaps.clear();
    }

    public boolean contains(int symbol) {
        return symbolToNode.containsKey(symbol);
    }

    public Node getNode(int symbol) {
        return symbolToNode.get(symbol);
    }

    public void update(int symbol) {
        nodeSwaps.clear();

        if (contains(symbol)) {
            updateExistingSymbol(symbol);
        } else {
            addNewSymbol(symbol);
        }
    }
```

```
private void addNewSymbol(int symbol) {
    Node oldNYT = NYT;

    Node internalNode = new Node(0, nextOrderNumber--);

    Node symbolNode = new Node(0, symbol, nextOrderNumber-

    Node newNYT = new Node(0, -1, nextOrderNumber--);

    internalNode.leftChild = newNYT;
    internalNode.rightChild = symbolNode;
    newNYT.parent = internalNode;
    symbolNode.parent = internalNode;

    if (oldNYT == root) {
        root = internalNode;
        internalNode.parent = null;
    } else {
        if (oldNYT.parent.leftChild == oldNYT) {
            oldNYT.parent.leftChild = internalNode;
        } else {
            oldNYT.parent.rightChild = internalNode;
        }
        internalNode.parent = oldNYT.parent;
    }

    NYT = newNYT;

    symbolToNode.put(symbol, symbolNode);

    incrementWeight(symbolNode);
}

private void updateExistingSymbol(int symbol) {
    Node node = symbolToNode.get(symbol);

    incrementWeight(node);
}
```

```
    private void incrementWeight(Node node) {
        Node highestNode = findHighestNodeWithSameWeight(node);

        if (highestNode != null && highestNode != node && !isAncestor(node, highestNode)) {
            nodeSwaps.add(node);
            nodeSwaps.add(highestNode);

            swapNodes(node, highestNode);
        }

        node.weight++;

        if (node.parent != null) {
            incrementWeight(node.parent);
        }
    }

    private boolean isAncestor(Node potential, Node node) {
        Node current = node.parent;
        while (current != null) {
            if (current == potential) return true;
            current = current.parent;
        }
        return false;
    }

    private Node findHighestNodeWithSameWeight(Node node) {
        Node result = null;
        int highestOrder = -1;

        Node current = findHighestOrderNode();
        while (current != null) {
            if (current.weight == node.weight &&
                current.orderNumber > node.orderNumber &&
                current != node &&
                current != root) {

                if (!areSiblings(current, node)) {
                    if (result == null || current.orderNumber > highestOrder) {
                        result = current;
                        highestOrder = current.orderNumber;
                    }
                }
            }

            current = findNextHighestOrderNode(current);
        }

        return result;
    }

    private boolean areSiblings(Node n1, Node n2) {
        return n1.parent != null && n1.parent == n2.parent;
    }
```

```
private void swapNodes(Node a, Node b) {
    if (a.parent == b || b.parent == a) return;

    Node aParent = a.parent;
    Node bParent = b.parent;

    boolean aIsLeftChild = aParent != null && aParent.leftChild == a;
    boolean bIsLeftChild = bParent != null && bParent.leftChild == b;

    if (aParent != null) {
        if (aIsLeftChild) {
            aParent.leftChild = b;
        } else {
            aParent.rightChild = b;
        }
    }

    if (bParent != null) {
        if (bIsLeftChild) {
            bParent.leftChild = a;
        } else {
            bParent.rightChild = a;
        }
    }

    a.parent = bParent;
    b.parent = aParent;

    if (a == root) root = b;
    else if (b == root) root = a;

    int temp = a.orderNumber;
    a.orderNumber = b.orderNumber;
    b.orderNumber = temp;
}

public String getPathToNode(Node node) {
    StringBuilder path = new StringBuilder();
    Node current = node;

    while (current != root) {
        if (current.parent.leftChild == current) {
            path.append('0');
        } else {
            path.append('1');
        }
        current = current.parent;
    }

    return path.reverse().toString();
}
```

HuffmanTree Class:

- root: The root node of the Huffman tree.
- NYT: The "Not Yet Transmitted" (NYT) node.
- nextOrderNumber: A counter for assigning unique order numbers to nodes, starting from 512.
- symbolToNode: A map that associates symbols to their corresponding nodes.
- nodeSwaps: A list tracking nodes that have been swapped during updates.

Constructors:
- HuffmanTree():
  - Initializes `nextOrderNumber`, `symbolToNode`, and `nodeSwaps`.
  - Creates the NYT node and sets it as the root.

Methods:
- getRoot():
  - Returns the root node of the tree.

- getNYT():
  - Returns the NYT node.

- getNodeSwaps():
  - Returns the list of node swaps.

- clearNodeSwaps():
  - Clears the node swaps list.

- contains(int symbol):
  - Returns `true` if the symbol is in the tree, otherwise `false`.

- getNode(int symbol):
  - Returns the node associated with the symbol.

- update(int symbol):
  - Updates the tree: either increments the weight of an existing symbol or adds a new symbol.

- addNewSymbol(int symbol):
  - Adds a new symbol to the tree, creating necessary internal and symbol nodes.

- updateExistingSymbol(int symbol):
  - Increments the weight of an existing symbol.

- incrementWeight(Node node):
  - Increments the weight of a node and swaps nodes if necessary.

- findHighestNodeWithSameWeight(Node node):
  - Finds the node with the same weight but a higher order number than the given node.

- swapNodes(Node a, Node b):
  - Swaps two nodes in the tree, adjusting their parent-child relationships.

- getPathToNode(Node node):
  - Returns the binary path from the root to the specified node.

- getPathToNYT():
  - Returns the binary path to the NYT node.

- toString():
  - Returns a string representation of the Huffman tree.

```
49 public class Encoder implements AutoCloseable {
48     private HuffmanTree tree;
47     private StringBuilder encodedOutput;
46     private BitOutputStream output;
45
44     public Encoder(String outputFileName) throws IOException {
43         tree = new HuffmanTree();
42         encodedOutput = new StringBuilder();
41         output = new BitOutputStream(new FileOutputStream(outputFileName));
40     }
39
38     public Encoder(BitOutputStream outputStream) {
37         tree = new HuffmanTree();
36         encodedOutput = new StringBuilder();
35         output = outputStream;
34     }
33
32     public void encodeSymbol(int symbol) throws IOException {
31         if (tree.contains(symbol)) {
30             String path = tree.getPathToNode(tree.getNode(symbol));
29             writeStringAsPath(path);
28         } else {
27             String nytPath = tree.getPathToNYT();
26             writeStringAsPath(nytPath);
25
24             writeASCIIBits(symbol);
23         }
22
21         tree.update(symbol);
20     }
19
18     private void writeASCIIBits(int symbol) throws IOException {
17         for (int i = 7; i >= 0; i--) {
16             output.writeBit((symbol >> i) & 1);
15         }
14     }
13
12     public void encodeString(String text) throws IOException {
11         for (int i = 0; i < text.length(); i++) {
10             encodeSymbol(text.charAt(i));
 9         }
 8     }
 7
 6     private void writeStringAsPath(String path) throws IOException {
 5         for (int i = 0; i < path.length(); i++) {
 4             output.writeBit(path.charAt(i) == '1' ? 1 : 0);
 3         }
 2     }
 1
52     public String getEncodedBitString() {
 1         return encodedOutput.toString();
 2     }
 3
```

Encoder Class:

- tree: An instance of the HuffmanTree class, used to manage the Huffman encoding tree.

- encodedOutput: A StringBuilder that stores the encoded output as a string of bits.
- output: An instance of BitOutputStream that writes the encoded bits to a file or an output stream.

Constructors:
- Encoder(String outputFileName):
  - Initializes a new HuffmanTree.
  - Creates a StringBuilder for storing the encoded output.
  - Sets up a BitOutputStream for writing to the specified file.

- Encoder(BitOutputStream outputStream):
  - Initializes a new HuffmanTree.
  - Creates a StringBuilder for storing the encoded output.
  - Uses the provided BitOutputStream for writing the encoded bits.

Methods:
- encodeSymbol(int symbol):
  - Encodes a single symbol by checking if it exists in the tree.
  - If the symbol is in the tree, it finds the path to the corresponding node and writes it to the output stream.
  - If the symbol is not in the tree, it finds the path to the "Not Yet Transmitted" (NYT) node, writes it, and then writes the ASCII bits of the symbol.

- writeASCIIBits(int symbol):
  - Converts the symbol (an integer) to its ASCII binary representation and writes each bit to the output stream.

- encodeString(String text):
  - Encodes an entire string by calling encodeSymbol for each character in the string.

- writeStringAsPath(String path):
  - Writes the binary path (represented as a string of '0's and '1's) to the output stream, where each '1' or '0' is written as a bit.

- getEncodedBitString():
  - Returns the encoded output as a string.

```java
public class Decoder implements AutoCloseable {
    private HuffmanTree tree;
    private BitInputStream input;
    private StringBuilder decodedOutput;

    public Decoder(String inputFileName) throws IOException {
        tree = new HuffmanTree();
        input = new BitInputStream(new FileInputStream(inputFileName));
        decodedOutput = new StringBuilder();
    }

    public Decoder(BitInputStream inputStream) {
        tree = new HuffmanTree();
        input = inputStream;
        decodedOutput = new StringBuilder();
    }

    public int decodeSymbol() throws IOException {
        Node currentNode = tree.getRoot();

        while (!currentNode.isLeaf() && currentNode != tree.getNYT()) {
            int bit = input.readBit();
            if (bit == -1) return -1;

            if (bit == 0) {
                currentNode = currentNode.leftChild;
            } else {
                currentNode = currentNode.rightChild;
            }
        }

        int symbol;
        if (currentNode == tree.getNYT()) {
            symbol = readASCIIBits();
            if (symbol == -1) return -1;
        } else {
            symbol = currentNode.symbol;
        }

        tree.update(symbol);
        return symbol;
    }

    private int readASCIIBits() throws IOException {
        int symbol = 0;
        for (int i = 0; i < 8; i++) {
            int bit = input.readBit();
            if (bit == -1) return -1;
            symbol = (symbol << 1) | bit;
        }
        return symbol;
    }
```

Decoder Class:

- tree: An instance of the HuffmanTree class, used to manage the Huffman encoding tree.

- input: A BitInputStream for reading the encoded bits from an input stream or file.
- decodedOutput: A StringBuilder to store the decoded output as a string.

Constructors:
- Decoder(String inputFileName) throws IOException:
  - Initializes a new HuffmanTree.
  - Creates a BitInputStream from the specified input file.
  - Creates a StringBuilder for storing decoded output.

- Decoder(BitInputStream inputStream):
  - Initializes a new HuffmanTree.
  - Uses the provided BitInputStream for reading the encoded bits.
  - Creates a StringBuilder for storing decoded output.

Methods:
- decodeSymbol() throws IOException:
  - Decodes a single symbol by traversing the Huffman tree based on the input bits.
  - If the NYT node is encountered, it reads the next 8 bits as the symbol.
  - Returns the decoded symbol.

- readASCIIBits() throws IOException:
  - Reads 8 bits from the input stream to form a symbol (ASCII value).

- decode() throws IOException:
  - Decodes an entire sequence of symbols, appending the characters to the decoded output.

- close() throws IOException:
  - Closes the input stream.

Nested Class:
- BitInputStream:
  - A class that handles reading bits from an input stream.

- readBit():
  - Reads a single bit from the input stream.

- close():
  - Closes the input stream used by the BitInputStream.

```
public class AdaptiveHuffman {
    public static void compress(String inputFileName, String outputFileName) throws IOException {
        try (BufferedReader reader = new BufferedReader(new FileReader(inputFileName));
             Encoder encoder = new Encoder(outputFileName)) {

            int c;
            while ((c = reader.read()) != -1) {
                encoder.encodeSymbol(c);
            }
        }
    }

    public static void decompress(String inputFileName, String outputFileName) throws IOException {
        try (Decoder decoder = new Decoder(inputFileName);
             BufferedWriter writer = new BufferedWriter(new FileWriter(outputFileName))) {

            int symbol;
            while ((symbol = decoder.decodeSymbol()) != -1) {
                writer.write((char) symbol);
            }
        }
    }

    public static double calculateCompressionRatio(String originalFile, String compressedFile) throws IOException {
        File original = new File(originalFile);
        File compressed = new File(compressedFile);

        if (!original.exists() || !compressed.exists()) {
            throw new FileNotFoundException("One or both files not found");
        }

        long originalSize = original.length();
        long compressedSize = compressed.length();

        return (double) compressedSize / originalSize;
    }
```

```
    public static void main(String[] args) {
        if (args.length < 3) {
            System.out.println("Usage: java AdaptiveHuffman [compress|decompress|analyze] inputFile outputFile");
            return;
        }

        String operation = args[0].toLowerCase();
        String inputFile = args[1];
        String outputFile = args[2];

        try {
            if (operation.equals("compress")) {
                System.out.println("Compressing " + inputFile + " to " + outputFile);
                compress(inputFile, outputFile);
                System.out.println("Compression complete.");

                // Calculate and display compression statistics
                double ratio = calculateCompressionRatio(inputFile, outputFile);
                System.out.printf("Compression ratio: %.2f (%.2f%%)\n", ratio, ratio * 100);
                System.out.println("Original size: " + new File(inputFile).length() + " bytes");
                System.out.println("Compressed size: " + new File(outputFile).length() + " bytes");

            } else if (operation.equals("decompress")) {
                System.out.println("Decompressing " + inputFile + " to " + outputFile);
                decompress(inputFile, outputFile);
                System.out.println("Decompression complete.");

            } else if (operation.equals("analyze")) {
                // Calculate and display compression statistics
                double ratio = calculateCompressionRatio(inputFile, outputFile);
                System.out.printf("Compression ratio: %.2f (%.2f%%)\n", ratio, ratio * 100);
                System.out.println("Original size: " + new File(inputFile).length() + " bytes");
                System.out.println("Compressed size: " + new File(outputFile).length() + " bytes");

            } else {
                System.out.println("Unknown operation: " + operation);
                System.out.println("Usage: java AdaptiveHuffman [compress|decompress|analyze] inputFile outputFile");
            }
        } catch (IOException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

AdaptiveHuffaman Class:

- compress(String inputFileName, String outputFileName) throws IOException:

- Compresses the input file using the Encoder class and saves the output to the specified file.

- decompress(String inputFileName, String outputFileName) throws IOException:
  - Decompresses the encoded input file using the Decoder class and writes the decoded content to the specified output file.

- calculateCompressionRatio(String originalFile, String compressedFile) throws IOException:
  - Calculates and returns the compression ratio between the original and compressed files.

- main(String[] args):
  - The main method that processes command-line arguments for compression, decompression, or analysis.
  - Displays compression ratio and file size statistics if the operation is "compress" or "analyze".
  - Handles exceptions related to file operations.

Test/

AdaptiveHuffmanTest/

First test/

```
public static void testEncodeDecodeSimpleString() throws IOException {
    String testString = "ABRACADABRA";

    File inputFile = File.createTempFile("test_input", ".txt");
    try (FileWriter writer = new FileWriter(inputFile)) {
        writer.write(testString);
    }

    File compressedFile = File.createTempFile("test_compressed", ".bin");

    File decompressedFile = File.createTempFile("test_decompressed", ".txt");

    System.out.println("Compressing...");
    AdaptiveHuffman.compress(inputFile.getAbsolutePath(), compressedFile.getAbsolutePath());

    System.out.println("Decompressing...");
    AdaptiveHuffman.decompress(compressedFile.getAbsolutePath(), decompressedFile.getAbsolutePath());
```

Second test/

```
    public static void testEncodeDecodeRepeatingPatterns() throws IOException {
        String testString = "AAABBBCCCAAABBBCCC";

        File inputFile = File.createTempFile("test_input_patterns", ".txt");
        try (FileWriter writer = new FileWriter(inputFile)) {
            writer.write(testString);
        }

        File compressedFile = File.createTempFile("test_compressed_patterns", ".bin");

        File decompressedFile = File.createTempFile("test_decompressed_patterns", ".txt");

        System.out.println("Compressing...");
        AdaptiveHuffman.compress(inputFile.getAbsolutePath(), compressedFile.getAbsolutePath());

        System.out.println("Decompressing...");
        AdaptiveHuffman.decompress(compressedFile.getAbsolutePath(), decompressedFile.getAbsolutePath());
```

Third Test/

```
public static void testEncodeDecodeLectureExample() throws IOException {
    String testString = "ABCCCAAAA";

    File inputFile = File.createTempFile("test_input", ".txt");
    try (FileWriter writer = new FileWriter(inputFile)) {
        writer.write(testString);
    }

    File compressedFile = File.createTempFile("test_compressed", ".bin");

    File decompressedFile = File.createTempFile("test_decompressed", ".txt");

    System.out.println("Compressing...");
    AdaptiveHuffman.compress(inputFile.getAbsolutePath(), compressedFile.getAbsolutePath());

    System.out.println("Decompressing...");
    AdaptiveHuffman.decompress(compressedFile.getAbsolutePath(), decompressedFile.getAbsolutePath());
```

Fourth Test/

```
public static void testLargerTextFile() throws IOException {
    StringBuilder largeText = new StringBuilder();
    for (int i = 0; i < 20; i++) {
        largeText.append("Line ").append(i).append(": The quick brown fox jumps over the lazy dog. ");
        largeText.append(i % 10).append(i % 5).append(i % 3).append("\n");
    }

    File inputFile = File.createTempFile("test_input_large", ".txt");
    try (FileWriter writer = new FileWriter(inputFile)) {
        writer.write(largeText.toString());
    }

    File compressedFile = File.createTempFile("test_compressed_large", ".bin");

    File decompressedFile = File.createTempFile("test_decompressed_large", ".txt");

    System.out.println("Compressing...");
    AdaptiveHuffman.compress(inputFile.getAbsolutePath(), compressedFile.getAbsolutePath());

    System.out.println("Decompressing...");
    AdaptiveHuffman.decompress(compressedFile.getAbsolutePath(), decompressedFile.getAbsolutePath());
```

Result/

```
❯ java -cp build AdaptiveHuffmanTest
Running Adaptive Huffman Coding tests...


=== Test 1: Simple String ===
Compressing...
Decompressing...
Original: ABRACADABRA
Decompressed: ABRACADABRA
Test passed: Decompressed output matches original input
Test case: Simple string
Original size: 11.0 bytes
Compressed size: 8.0 bytes
Compression ratio: 0.7272727272727273

=== Test 2: Repeating Patterns ===
Compressing...
Decompressing...
Original: AAABBBCCCAAABBBCCC
Decompressed: AAABBBCCCAAABBBCCC
Test passed: Decompressed output matches original input
Test case: Repeating patterns
Original size: 18.0 bytes
Compressed size: 8.0 bytes
Compression ratio: 0.4444444444444444

=== Test 3: Lecture Text File ===
Compressing...
Decompressing...
Original: ABCCCAAAA
Decompressed: ABCCCAAAA
Test passed: Decompressed output matches original input
Test case: Simple string
Original size: 9.0 bytes
Compressed size: 5.0 bytes
Compression ratio: 0.5555555555555556

=== Test 4: Larger Text File ===
Compressing...
Decompressing...
Content match: Yes
Test passed: Decompressed output matches original input
Test case: Larger text file
Original size: 1150.0 bytes
Compressed size: 737.0 bytes
Compression ratio: 0.6408695652173914

All tests completed.
```

Visualization/