605.202: Data Structures

Ziyu(Yvonne) Lin

Lab 3 Analysis

Lab 3 Analysis

1. Description of data structures

This lab is implementing Huffman encoding and decoding by constructing a Huffman Encoding Tree using a given frequency table. The encoding tree is then used to generate binary codes for group of characters and to encode or decode text files accordingly.

For example, the phrase “Hello World” could be encoded as a binary string - “1101101000010001111100011111101000000101100”.

The main data structure used here is a binary tree, where each node represents characters and their combined frequency. The leaves of the binary tree represent single character, and internal nodes of the binary tree represent merged groups of characters. In the Huffman tree, a left branch corresponds to binary 0, while a right branch corresponds to binary 1.

2. Justification of the data structure choices and implementation

A binary tree is an appropriate data structure for implementing the Huffman encoding and decoding logic. The tree structure naturally supports the binary decision process for encoding, where moving left represents a 0 and moving right represents a 1. This allows efficient traversal when generating code for both encoding and decoding.

The tree is built by merging the two least frequent nodes at each step, where

- The left tree represents binary 0
- The right tree represents binary 1

The Huffman code for each character can then be generated by traversing the tree from the root to its corresponding leaf. As we could see, the binary tree structure is both intuitive and efficient for the encoding process.

3. Discussion of appropriateness to the application

The tree structure is appropriate for this application due to the following reasons:

- Scalability

  The binary tree can be easily scaled to accommodate different character sets or different frequency distributions for encoding and decoding.

- Efficient Encoding and Decoding

The tree structure allows for fast traversal for both encoding and decoding. This enables a reliable and efficient mechanism for compression and decompression.

- Data compression

  As discussed in the following section, the tree-based method, where a frequency table is used to generate the Huffman tree, naturally leads to a more compressed encoded string. Instead of having a fixed number for every character, more frequent characters are given shorter codes, while less frequent characters are given longer ones. This approach ensures better data compression, especially when we have skewed character frequency distributions.

4. Data compression with the method

The hierarchical structure of the tree ensures that more frequent characters are located closer to the root, which means that they have shorter codes, and less frequent characters are located deeper in the tree, which means that they have longer codes. And this matches the data compression philosophy of Huffman encoding. This method reduces the bits needed to represent one word, especially when there is a skewed distribution of character frequencies.

Compared to ASCII way of encoding, where each character is given a fixed length of code (typically 8 bits), Huffman encoding is more efficient. For example. The phrase “Hello World” contains 10 characters, which would require 80 bits in ASCII. But in our case, Huffman encoding compresses it to just 43 bits, and this results in a compression ratio of almost 50%, which is pretty efficient in this example. By taking character frequency into account, Huffman encoding generates a more tailored encoding binary representation. And this approach also uses fewer binary bits overall and thus saves the storage space.

5. Other data structured used

Besides the binary tree structure, this implementation also uses other data structures:

- Priority Queue

  The priority queue is another important data structure I used in this implementation. In this implementation, Huffman nodes are compared using the following rules:
  1. Nodes with lower frequencies come first.
  2. If frequencies are equal, single-character nodes take precedence over multi-character nodes.
  3. If still tied, nodes are compared alphabetically.

  Because of these comparison rules, priority queue is the ideal process for the merging process. At each step, two least frequent nodes are merged to build the Huffman tree.

And the two least frequent nodes appear at the front of the queue because of its priority number.

- HashMap

  A HashMap is used to store the final binary codes for every character. In the recursive method that builds the encoding strings, every time when the node is a leaf node, the method stores the node and its corresponding encoding binary bits into the HashMap.

- StringBuilder

  I also used a StringBuilder to concatenate binary codes or decoded characters for encoding and decoding. In the HuffmanEncoderDecoder class, within the encoding method, the method loops through each character in the input string, and get its corresponding binary bits from the encoding map, and then uses the StringBuilder to append the binary codes together. Within the decoding method, the method traverses the Huffman binary tree to find the character corresponding to the current sequence of binary bits. And the method will also use the StringBuilder to concatenate the decoded characters together.

6. Impact of Using a Different Tie-Breaking Scheme

If a different scheme is used to break ties, for example, giving precedence to alphabetical ordering first, and followed by the number of letters in the key, this will result in a different tree structure because of different precedence order. Thus the encoded binary bits and decoded strings could vary slightly based on how big the tree structure changes are. But overall it does not affect the correctness of the Huffman encoding method. The core functionality remains the same; only the specific encoding and decoding mappings are affected.

7. Design Decisions and Justifications

In this implementation, I have the following classes: HuffmanNode, HuffmanTree, and HuffmanEncoderDecoder. I also have a Main class and a HuffmanTest class to run and test the implementation.

In the HuffmanNode class, it represents a single node in the Huffman tree. Each node stores the character, and its corresponding frequency, and pointers. The class also implements a comparison method that determines the precedence order of Huffman nodes. Nodes are compared based on the following rules:

1. Nodes with lower frequencies come first.
2. If frequencies are equal, single-character nodes take precedence over multi-character nodes.
3. If still tied, nodes are compared alphabetically.

The comparison logic is used later for the priority queue to determine which two nodes have the lowest frequencies and should be merged together.

In the HuffmanTree class, it firstly has one method to build the Huffman tree. Each key-value pair from the frequency table—where the key is a character and the value is its frequency—is used to create a Huffman node. And then the Huffman nodes are added to the priority queue. Since the Huffman node has comparable logic, the nodes with the lowest frequencies will appear at the front of the priority queue. In the comparison logic, When a.compareTo(b) returns -1, then a is less than b. Then a is closer to the front of the queue. And then, the method builds a tree from the priority queue. The front two nodes of the priority queue are merged together to become one parent node. And then the method links the parent node with the two child nodes. The order of linking matters: the left child should be the less frequent node, the one that was polled first from the priority queue, and the right child should be the second polled node. I once linked the nodes in the wrong order, and this gave me an incorrect tree and produced invalid encoding. Afterwards, the merged node is added to the priority queue. At last, the priority queue consists of different tree nodes. Then this method returns the root node of the tree, which is the node polled first from the priority queue.

In the HuffmanTree class, it also has one method that recursively traverses the tree to build the encoding map. The base case of the recursion is that the node is a leaf node, and then the method is going to return and put the leaf node and its corresponding encoding into an encoding map. In the recursive traversal, the method appends binary 0 to the encoding code when moving to the left child and appends binary 1 when moving to the right child. Then there is another method called buildEncodingMap that passes in the root node, an empty encoding string, and an empty encoding map, calls the recursion function to build the encoding map.

In the HuffmanEncoderDecoder class, it has one constructor that has the root node, and one encoding map that is built from the root node. It then has one method for the encoding part. For every input string, the method checks every character against the encoding map, and then constructs the encoded string character by character. The class also has one decode method. The method is checking every binary bit in the Huffman tree and when it reaches a leaf node, the method appends the current character to the decoded string and then reset the current node to the root node, and then do the check again for the next binary bit until the entire encoded string is decoded.

8. Efficiency with respect to both time and space

- Time efficiency

  Building one Huffman tree is O(nlogn) in time complexity with n characters. Encoding and decoding a string of length L is O(L) in time complexity. Since the main part is the binary tree part, overall time complexity is O(nlogn).

- Space efficiency

Storing a Huffman tree is O(n) in space complexity with n characters. Encoding map is also O(n) in space complexity with n characters. Encoded string and decoded string storage depend on the encoding map. They require space proportional to the length of the input string L. Since each character is replaced by a variable-length binary code (typically a few bits per character), the overall storage required for the encoded string can be approximated as $O(a \times L)$, where a is the average length of a Huffman code. Since a is bounded, this could be simplified as O(L). Since the main space consuming components are tree and the map, the overall space complexity is O(n).

9. What I learned

This lab is an interesting lab and also challenging at the same time. One important lesson I learned is that Huffman encoding is very sensitive to the rules used for building the tree. Initially, I struggled with generating the correct encoded strings, and after careful debugging, I realized the issue was in the priority queue logic—specifically in how I compared different Huffman nodes. Fixing the comparison logic took considerable time, but it was a valuable learning experience that helped me better understand how priority queues and custom comparators work.

Another part I found especially helpful was implementing and working with a binary tree. Thinking through the entire process—creating nodes, building the Huffman tree, and traversing it recursively—gave me a much deeper understanding of tree structures and recursion. This lab reinforced my understanding of how binary trees can be applied in real-world algorithms.

10. What I might do differently next time

In this lab, I generated the frequency table directly from the original input file. While this approach works for demonstration purposes, I realized it may not be suitable for real-world applications. If the frequency table is derived from the same file being encoded, it becomes easier for someone to decode the message by simply reconstructing the table, which is not secure. I might try generating the frequency table from a separate sample file or using a predefined dictionary-style corpus to build the Huffman tree, so that it could be a real encoding and decoding task.

11. Enhancements

One enhancement I made in this lab was adding JUnit tests—hooray! I finally learned how to do it. In previous labs, I kept writing in the "what I might do differently next time" section that I wanted to try unit testing. However, I hadn’t actually learned how to write unit tests during my earlier studies. This time, I took the initiative to learn it through online resources and was able to implement JUnit tests successfully. It felt great to finally check this off my list and strengthen my testing skills for future projects.

Another enhancement I made was generating a frequency table from the input text. In this updated version, the frequency table accounts for not only letters but also punctuation, spaces, and tabs, which makes it more comprehensive. I then used this enhanced frequency table for encoding and decoding the original input. This improvement allowed the Huffman tree to reflect a more accurate distribution of all characters, not just letters, making the encoding process more robust.