Parallel Data Compression using Huffman Coding

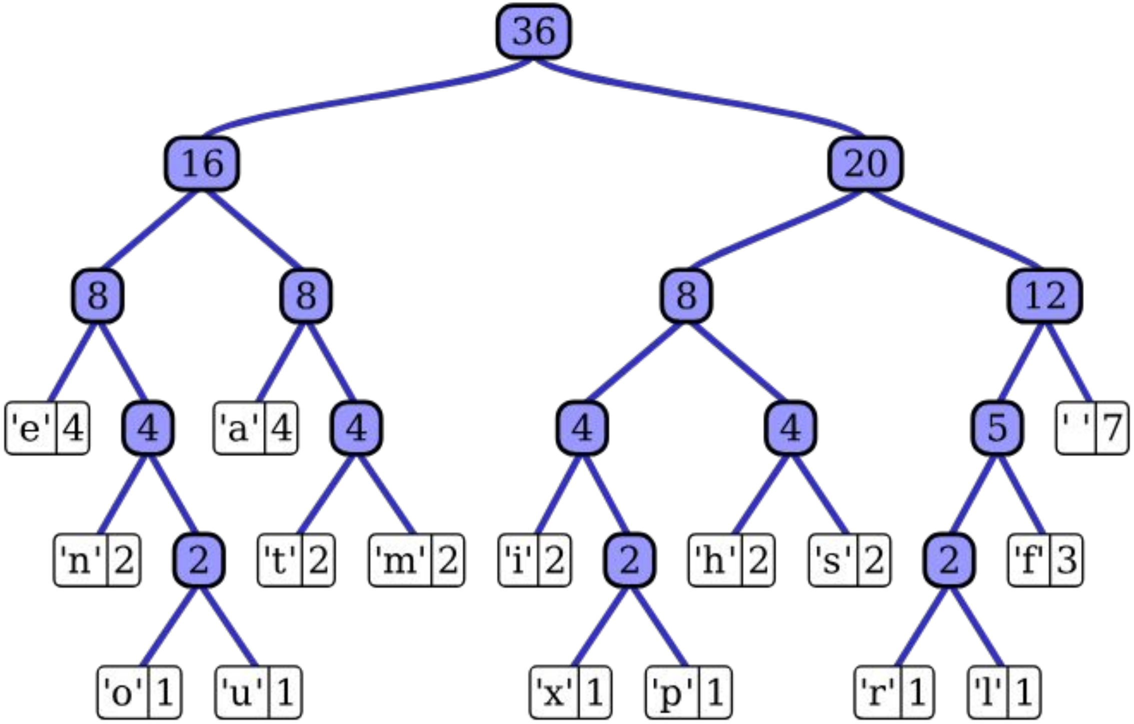
Pawin Pothasuthon

What is Huffman coding?

Basic idea

Huffman coding is a lossless data compression algorithm.

The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.



The Algorithm

Huffman

Char	Freq
H	1
u	1
f	2
m	1
a	1
n	1

1) Create a node for each character and label each with the frequency

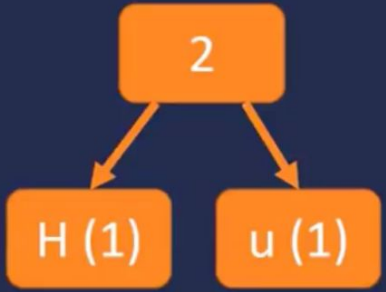
H (1) u (1) f (2) m (1) n (1) a (1)

2) Arrange these nodes in ascending frequency

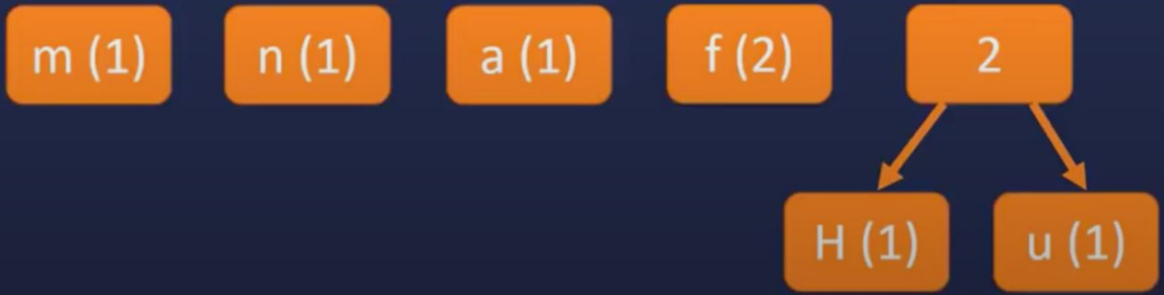
H (1) u (1) m (1) n (1) a (1) f (2)

Huffman

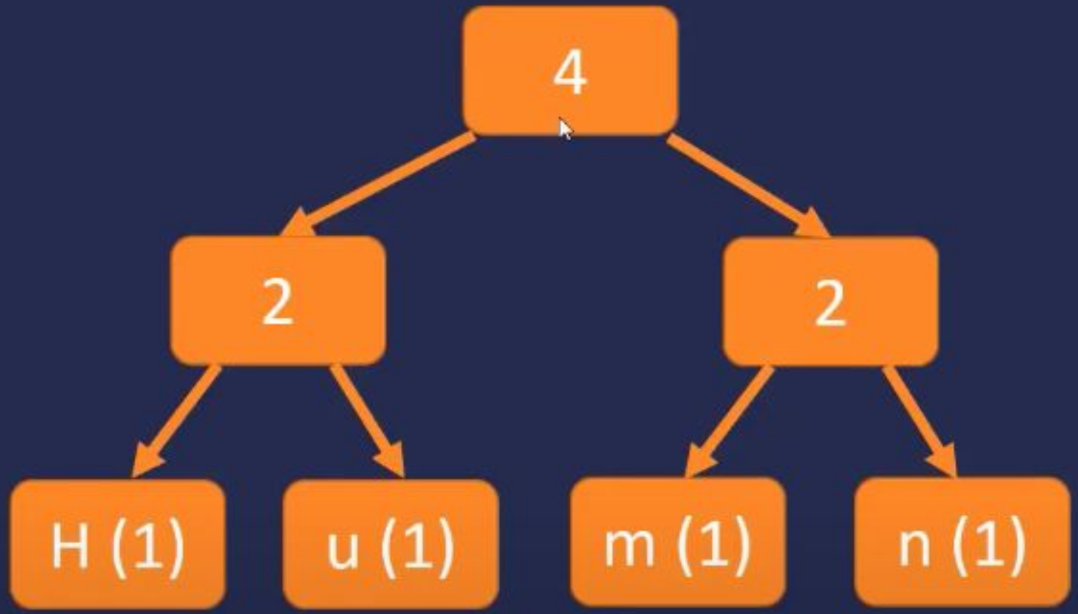
3) Take the first two nodes and join them, with the new parent node's label being the combined frequency of the two nodes



4) Insert this node with the children into the ordered list

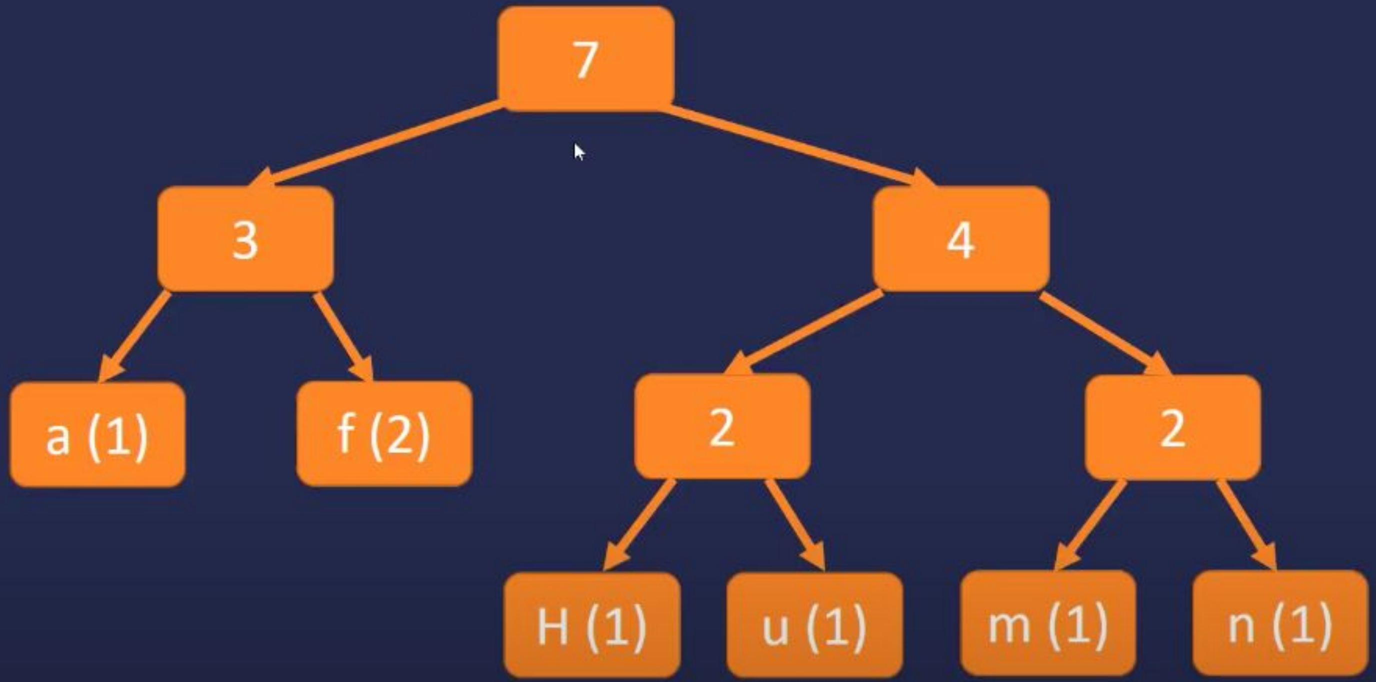


5) Repeat 3) and 4) until there is only one node



5) Repeat 3) and 4) until there is only one node

Huffman



The top node should equal the total number of original characters

The project

Project overview

Frequency Analysis

Calculate the frequency of each character in the input text.

Huffman tree construction

Build the binary tree where characters with higher frequencies are closer to the root.

Encoding and decoding

Encode the input text using the generated Huffman codes

Project structure

Main	Huffman	Node	Helpers
File operation and mode selection	Main implementation of Huffman encoding and decoding algorithm	Node definition for the Huffman tree and encoded data for serialized data storage	Utility functions

Implementation in code

```rust
#[derive(Debug, Clone, PartialEq, Eq)]
7 implementations
pub struct Node {
    pub freq: usize,                 // Frequency of the character
    pub char: Option<char>,          // Character stored in the node
    pub left: Option<Box<Node>>,     // Left child node
    pub right: Option<Box<Node>>,    // Right child node
}

impl Node {
    // Generate a new leaf node
    pub fn new_leaf(freq: usize, char: char) -> Self {
        Node {
            freq,
            char: Some(char),
            left: None,
            right: None,
        }
    }

    // Generate a new internal node with left and right children
    pub fn new_internal(freq: usize, left: Node, right: Node) -> Self {
        Node {
            freq,
            char: None,
            left: Some(Box::new(left)),
            right: Some(Box::new(right)),
        }
    }
}
```

The **Box** is one of the smart pointers. It saves an address that points to data in memory. **Box** helps us to create a Node struct with an unknown size so that we can grow the Huffman Tree

https://hackernoon.com/how-to-insert-binary-tree-in-rust

```rust
// Builds the Huffman tree
pub fn build_huffman_tree(freq_map: &HashMap<char, usize>) -> Node {
    println!("Generating Huffman tree...");

    let mut heap: std::collections::BinaryHeap<Node> = freq_map &HashMap<char, usize>
        .par_iter() Iter<char, usize>
        .map(map_op: |(&char: char, &freq: usize)| Node::new_leaf(freq, char)) Map<Iter<char, usize>, …
        .collect();

    // Build Huffman tree from frequency map
    while (heap.len() > 1) {
        let left: Node = heap.pop().unwrap();
        let right: Node = heap.pop().unwrap();
        let merged: Node = Node::new_internal(freq: left.freq + right.freq, left, right);

        heap.push(item: merged);
    }

    println!("Huffman tree built successfully.");

    heap.pop().unwrap()
}
```

Removes and returns the smallest element (node) from the heap. This node will become the left child of the new internal node.

Again removes and returns the next smallest element, which becomes the right child of the new internal node.

Building the Huffman tree takes *O(n log n)*, where n is the number of unique characters, since it involves sorting based on frequency.

```
// Compresses the contents of a file.
pub fn compress_file(input_path: &Path) -> io::Result<()> {
    println!("Starting compression...");

    // Read input file into a string
    let file_content: File = File::open(input_path)?;
    let mut reader: BufReader<File> = BufReader::new(inner: file_content);
    let mut text: String = String::new();
    reader.read_to_string(buf: &mut text)?;

    let freq_map: HashMap<char, usize> = text String
        .par_chars() Chars
        .fold(identity: HashMap::new, fold_op: |mut acc: HashMap<char, usize>, char: ch…| {
            *acc.entry(key: char).or_insert(default: 0) += 1;
            acc
        }) Fold<Chars, fn new<char, …>() -> …, …>
        .reduce(identity: || HashMap::new(), op: |mut acc: HashMap<char, usize>, map: HashMap…| {
            for (char: char, count: usize) in map {
                *acc.entry(key: char).or_insert(default: 0) += count;
            }
            acc
        });

    // Build Huffman tree from frequency map
    println!("Building Huffman tree from frequency map...");
    let huffman_tree: Node = build_huffman_tree(&freq_map);
```

```rust
// Build Huffman tree from frequency map
println!("Building Huffman tree from frequency map...");
let huffman_tree: Node = build_huffman_tree(&freq_map);

// Generate Huffman codes for each character
// Don't know any other way to do this.
let codes: Mutex<HashMap<char, String>> = Mutex::new(HashMap::new());

println!("Generating Huffman codes for each characters...");
generate_codes(node: &huffman_tree, prefix: String::new(), &codes);

// Encode text using Huffman codes
println!("Encoding text using Huffman codes...");
let encoded_text: String = encode_text(&text, codes: &codes.lock().unwrap());

// Prepare encoded data structure
let encoded_data: EncodedData = EncodedData {
    codes: codes.lock().unwrap().clone(),
    encoded_text,
};

// Generate output file path
let output_path: PathBuf = input_path.with_file_name(format!(
    "{}_compressed.txt",
    input_path.file_stem()
            .unwrap()
            .to_string_lossy()
));

println!("Writing encoded data to output file: {:?}", output_path);
write_encoded_file(&output_path, &encoded_data)?;

println!("Compression completed successfully.");
Ok(())
```

Initializes a `Mutex` that protects access to a `HashMap`. The `HashMap` will store the generated Huffman codes for each character.

Here, the program encodes the provided `text` using the generated Huffman codes. The `codes.lock().unwrap()` call acquires a lock on the `Mutex`, allowing safe access to the `HashMap` containing the codes.

This part constructs the output file path. It takes an existing `input_path`, modifies the file name to append "_compressed.txt", and retains the original file's stem (name without extension).

`to_string_lossy()` ensures safely converted to a string, even if it contains non-UTF-8 characters.

```rust
// Encodes input text.
pub fn encode_text(text: &str, codes: &HashMap<char, String>) -> String {
    println!("Encoding text...");

    let encoded_text: String = text &str
        .par_chars() Chars
        .map(map_op: |char: char| codes.get(&char).unwrap().as_str()) Map<Chars, impl Fn(char) -> ..>
        .collect::<Vec<&str>>() Vec<&str>
        .join(sep: "");

    println!("Text encoded successfully.");

    encoded_text
}
```

Encoding the text requires a lookup for each character, which is *O(m)* for m characters in the input text.

Each character in the text is processed with `map()`.

For each character, it retrieves the corresponding Huffman code from the `codes` map using `codes.get(&char)`. The `unwrap()` method is used, which will panic if the character isn't found in the map (indicating a potential issue).

So, the overall work is *O(n logn + m)*

Building the Huffman tree generally has a logarithmic depth due to the nature of binary trees, so span of O(n).

```rust
// Decodes Huffman encoded text using pre-generated Huffman codes.
pub fn decode_text(encoded_text: &str, codes: &HashMap<char, String>) -> String {
    println!("Decoding text...");
    let mut reverse_codes: HashMap<String, char> = HashMap::new();

    for (char: &char, code: &String) in codes {
        reverse_codes.insert(k: code.clone(), v: *char);
    }

    // Initialized to store the final decoded output.
    let mut decoded_text: String = String::new();
    // Build the Huffman code as we iterate through the bits of encoded_text.
    let mut current_code: String = String::new();

    // Iterate through the bits of the encoded text
    for bit: char in encoded_text.chars() {
        current_code.push(ch: bit);

        // If a match is found in the reverse_codes HashMap,
        // add the corresponding character to the decoded_text.
        if let Some(&char: char) = reverse_codes.get(&current_code) {
            decoded_text.push(ch: char);
            current_code.clear();
        }
    }

    println!("Text decoded successfully.");

    decoded_text
} fn decode_text
```

Each decoded character involves possibly traversing the height of the Huffman tree, which is *O(log n)* in the worst case.

Typically taking *O(m)* time, where *m* is the length of the encoded string.

Thus, the overall work for decoding the entire encoded string is $O(m \cdot \log n)$.

The span for decoding will be linear with respect to the length of the encoded string, leading to *O(m).*

Problem faced

The compressed file sometimes is bigger than the original file

Implementation Issues

Issues in the implementation of the compression algorithm, primarily **inefficient serialization of the compressed data.**

Encoding Overhead

The way you're encoding the output—using a text-based representation rather than binary—could also lead to larger sizes.

```
// Serialize encoded_data to JSON and write to file
serde_json::to_writer(writer: file, value: encoded_data)?;
```

This function serializes the `encoded_data` structure into JSON format.