DATA COMPRESSION THROUGH HUFFMAN ENCODING

November 21, 2021

Amit Kumar (2020CSB1070) ,
Ankit Sharma (2020CSB1072) ,
Tanish Goyal (2020CSB1133)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Ms. Kirandeep Kaur

Summary: Every information in computer science is encoded as binary digits i.e. 1s and 0s. The motive of transmitting information is to use the fewest number of bits in such a way that every encoding is unique and unambiguous.
This is where 'Huffman Encoding' comes into play! It uses the Greedy Algorithm and is an example of variable length encoding which uses variable number of bits for encoding the characters depending on their frequency in the given text. The character with maximum frequency gets the least bit code, while the opposite happens for the characters with the least frequency. This results in the minimum data loss possible, during encoding.
Being concise, we start with building a Huffman tree using the input characters and frequencies for each character. Priority Queue is used for building the Huffman tree such that nodes with lowest frequency have the highest priority. A Min Heap data structure can be used to implement the functionality of a priority queue. At the starting, all nodes are leaf nodes containing the character itself along with the frequency of that character. As the tree builds, internal nodes contain weight and links to two child nodes. Finally, we assign the binary codes to each character by traversing Huffman tree (Convention: 0 for left child and 1 for right child)

---

1. Introduction

Have you ever wondered if your precious data can be saved and stored in a minimum space possible without losing it? In this fast growing world, the storage management is the need of the hour and we shouldn't be surprised by the fact that we are able to develop numerous such techniques of compressing the data to store it in the least space, and one of these techniques is Huffman Encoding. It is one of the earliest compression techniques developed by David Huffman in 1951. This technique is the basis for all data compression and encoding schemes used even today.
We would be thoroughly discussing this technique using tables, figures and pseudo algorithms, so that the reader can get the grasp of the topic in a concise manner.
At the end, we would be emphasizing on the fact that it is still useful in modern world and conclude the topic.

2. Tables And Algorithms

The best way to visualize how the Huffman algorithm works, and is used in lossless data compression is by using an example, and working alongside that example. Thus from this section, we would be defining an example and explain the working of our algorithms on them simultaneously, in steps.

2.1. Tables

The program for converting a text into Huffman encoding demands for 2 kinds of data: characters present in the text and corresponding frequency of each character.

Let's try to create Huffman Tree from the following characters along with their frequencies using the Huffman algorithm:

TEXT: **aaeeeaeeuuaaeeeeeauutoooaaeeeassssaiiieeiiissssisisisisisi**

CHARACTER FREQUENCY TABLE

S.No.	Character	Frequency
1	u	4
2	a	10
3	o	3
4	t	1
5	s	13
6	i	12
7	e	15

Table 1: Listing the frequencies of characters from the given text

After enlisting frequencies of all the characters, we sort them in an ascending order and store them in a priority queue (implemented using MIN HEAP), and all of them are made the leaf nodes of the Huffman tree.

2.2. Algorithms

Initial program performs job of just reading through the input text file i.e. input.txt, and stores frequency of characters at their ASCII value place. Huffman algorithm gets a heap, with leaf nodes containing 4 information, character stored, it's frequency, it's left and right pointer, which both are pointed to null initially 2.

Algorithm 1 Making Huffman Tree

Building a Huffman tree using the input set of symbols and weight/ frequency for each symbol
Assigning the binary codes to each symbol by traversing Huffman tree:Generally, bit '0' represents the left child and bit '1' represents the right child.
{
Structure of Leaf Node:
1. Character
2. Frequency
3. left pointer
4. Right pointer
}
while $heapSize > 1$ **do**
  ASSIGN $temp1 = heap[0]$
  ASSIGN $temp2 = heap[1]$
  ASSIGN $temp3 = MakeLeaf(NULL, heap[0] -> freq + heap[1] -> freq)$
  ASSIGN $temp3 -> left = temp1$
  ASSIGN $temp3 -> right = temp2$
  ASSIGN $heap[0] = NULL$
  ASSIGN $heap[1] = NULL$
  INSERT *temp3* at the end of **heap** array
  {Since first 2 nodes now point to **NULL**, we shift each non-NULL node, left by two nodes.}
  {i.e move all the heap element starting from index 2 two times to it's left}
  for integer i =0 ; i < heapSize +1 ; i = i+1 **do**
    ASSIGN leafNode* ptr = heap[i];
    heap[i] = heap[i - 2];
    heap[i - 2] = ptr;
  end for
  ASSIGN $heapSize = heapSize - 1$
  $heap = sortHeap(heap)$
end while
ASSIGN root = heap[0];

Algorithm 2 Huffman tree traversal

ASSIGN size = ceil(log2((float)total frequency / minimum frequency)) - 1;
ASSIGN Top=0
INIT *int arr*[*size*];
DEF ENCODER(struct leafNode *root, int arr[], intTop, char characters[], string Codings[])
if root->left ! = NULL **then**
  arr[Top] = 0
  recursively call *ENCODER* for (root->left, arr, top + 1, characters, Codings)
end if
if $root -> right != NULL$ **then**
  arr[Top] = 1;
  recursively call *ENCODER* for (root->right, arr, top + 1, characters, Codings)
end if
if root=leaf **then**
  print the encoding of that leaf node
end if

3. Procedure For Huffman Tree Construction

We would take the example of Table 1 only:
Create a leaf node for each character along with its frequency and build a priority queue which we'll implement using the MINHEAP for all the nodes (The frequency value is used to compare two nodes in min heap)

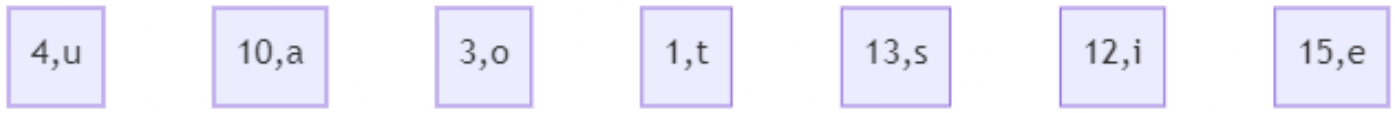

Figure 1: Step 1

Sort the leaf nodes according to their frequencies.

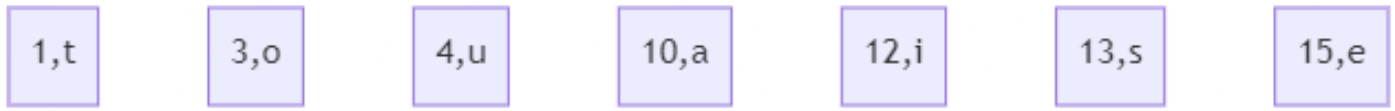

Figure 2: Step 2

Combine the first two nodes and make a new node whose frequency is sum of the first two i.e. 4 $(1+3)$ and it's left,right children are the first two nodes of the heap. We then sort the heap again.

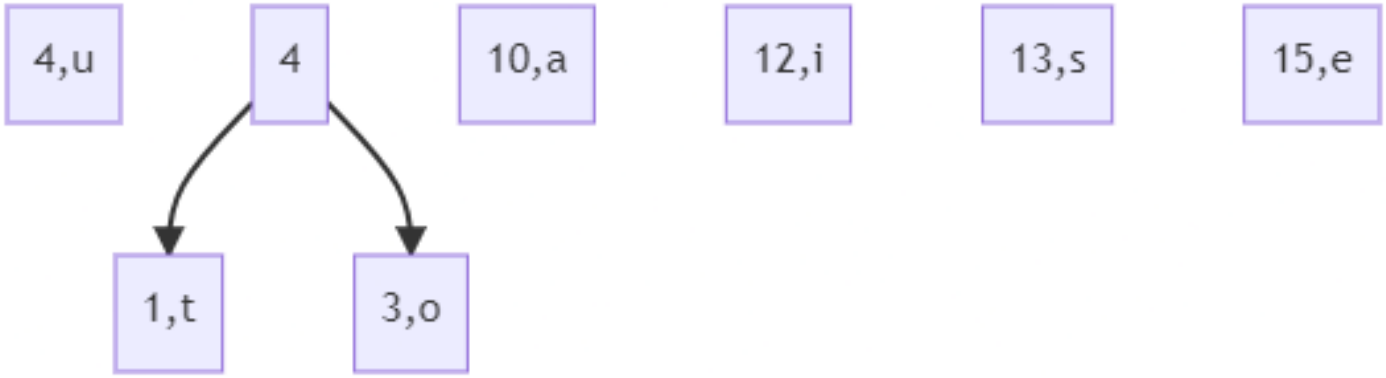

Figure 3: Step 3

Repeat the above step again and again till we get a single node whose key value would be the total number of characters in the text (that becomes the root node)

Nodes with frequency 4 and 4 are the first two values in heap, so we get a new node whose frequency is 8 $(4+4)$

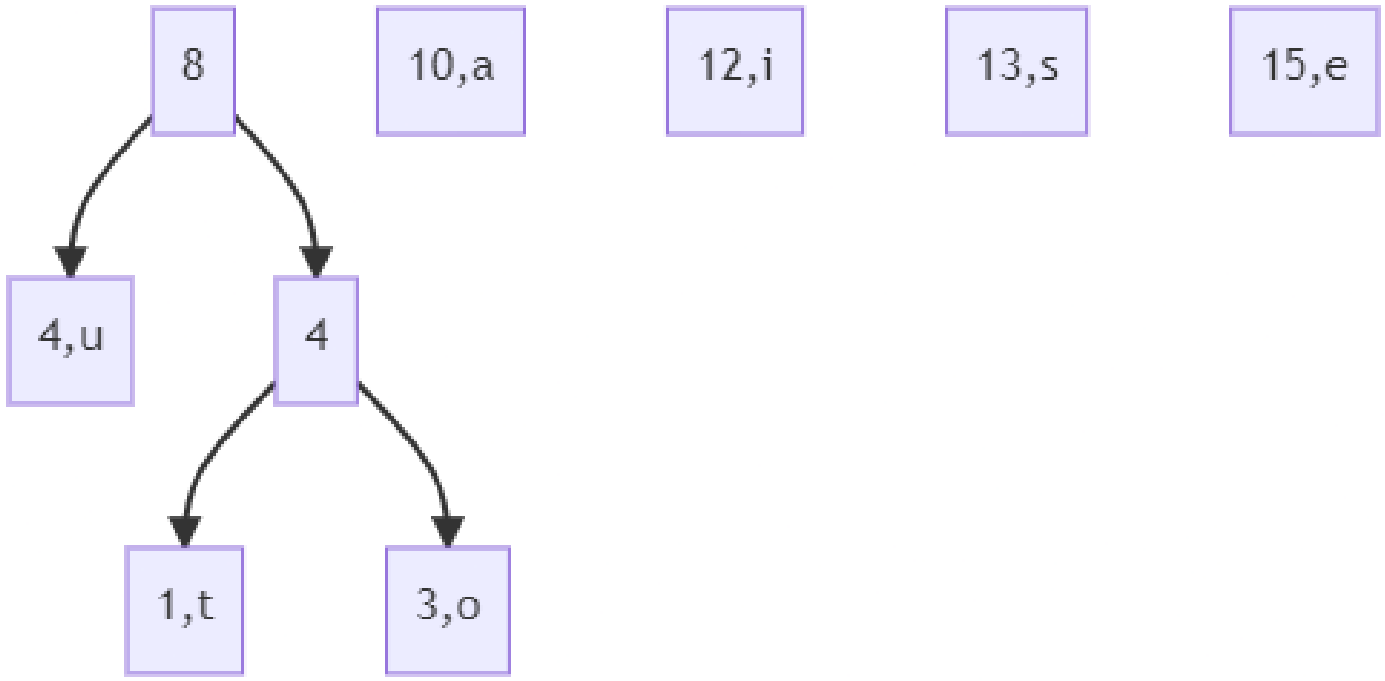

Figure 4: Step 4

Now, 8 and 10 are the frequencies of first two heap values so we get a new node whose frequency is 18 $(8 + 10)$

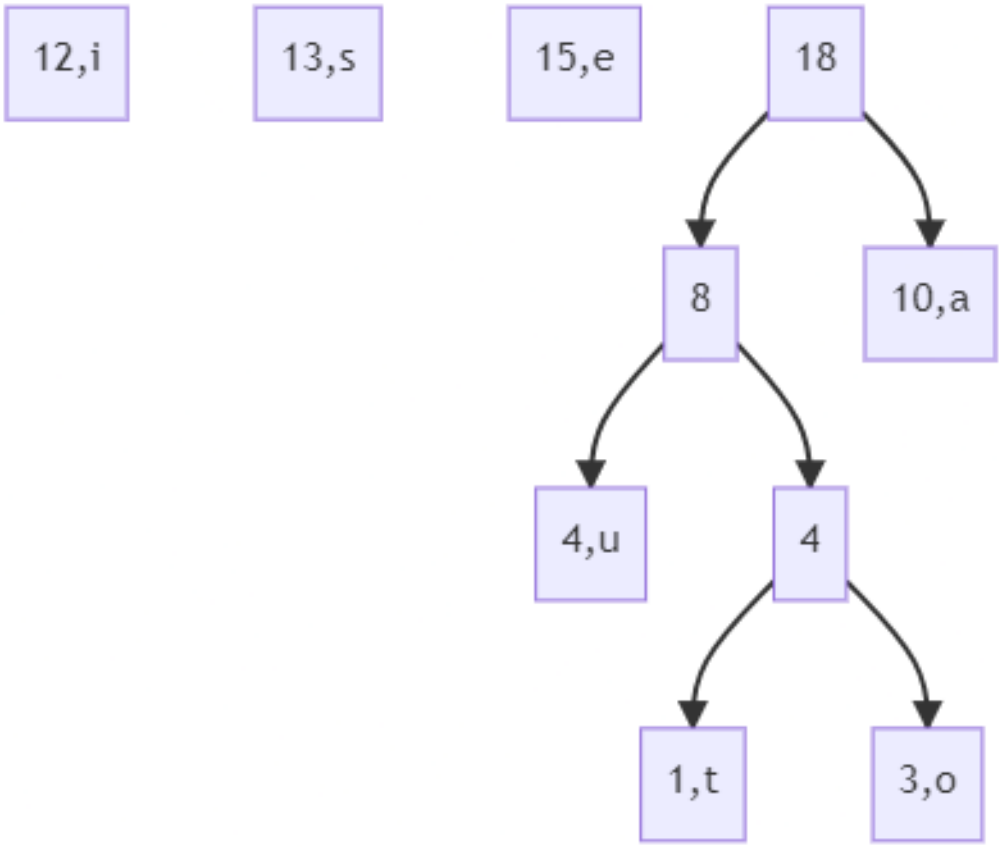

Figure 5: Step 5

Again, 12 and 13 frequencies would come at the starting so we get a new node whose frequency is 25 $(12 + 13)$

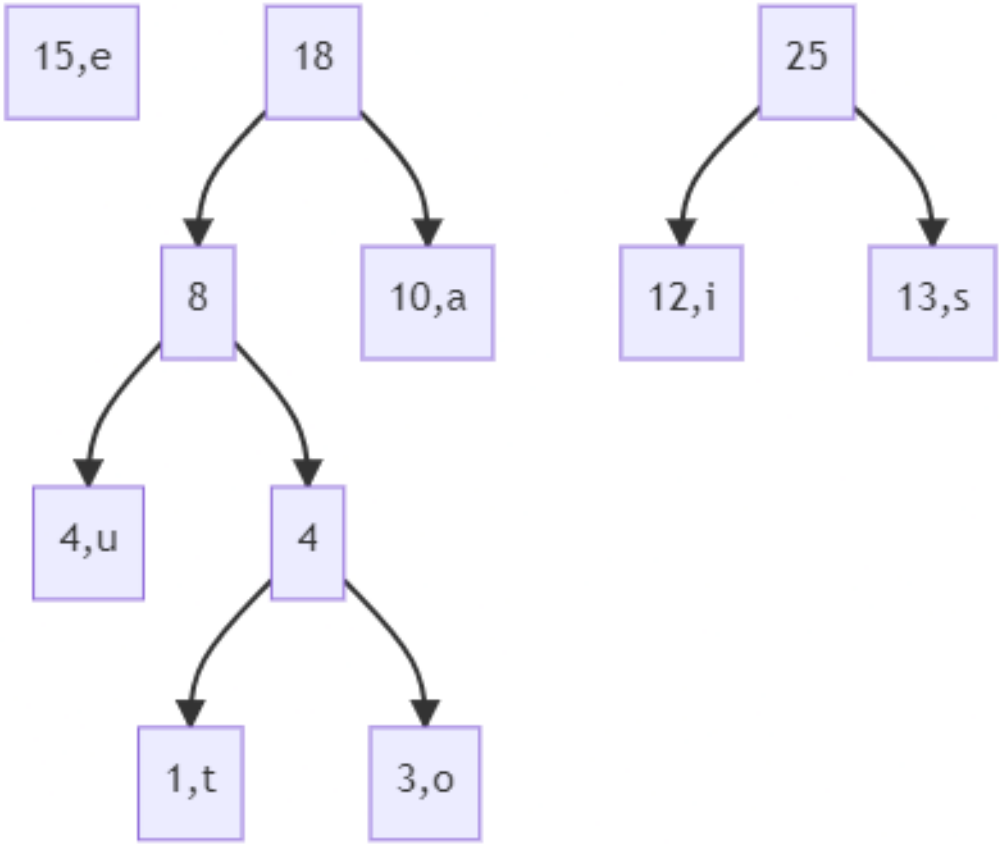

Figure 6: Step 6

Now, we have nodes with frequencies 15 and 18 at the starting so we get a new node whose frequency is 33 $(15 + 18)$

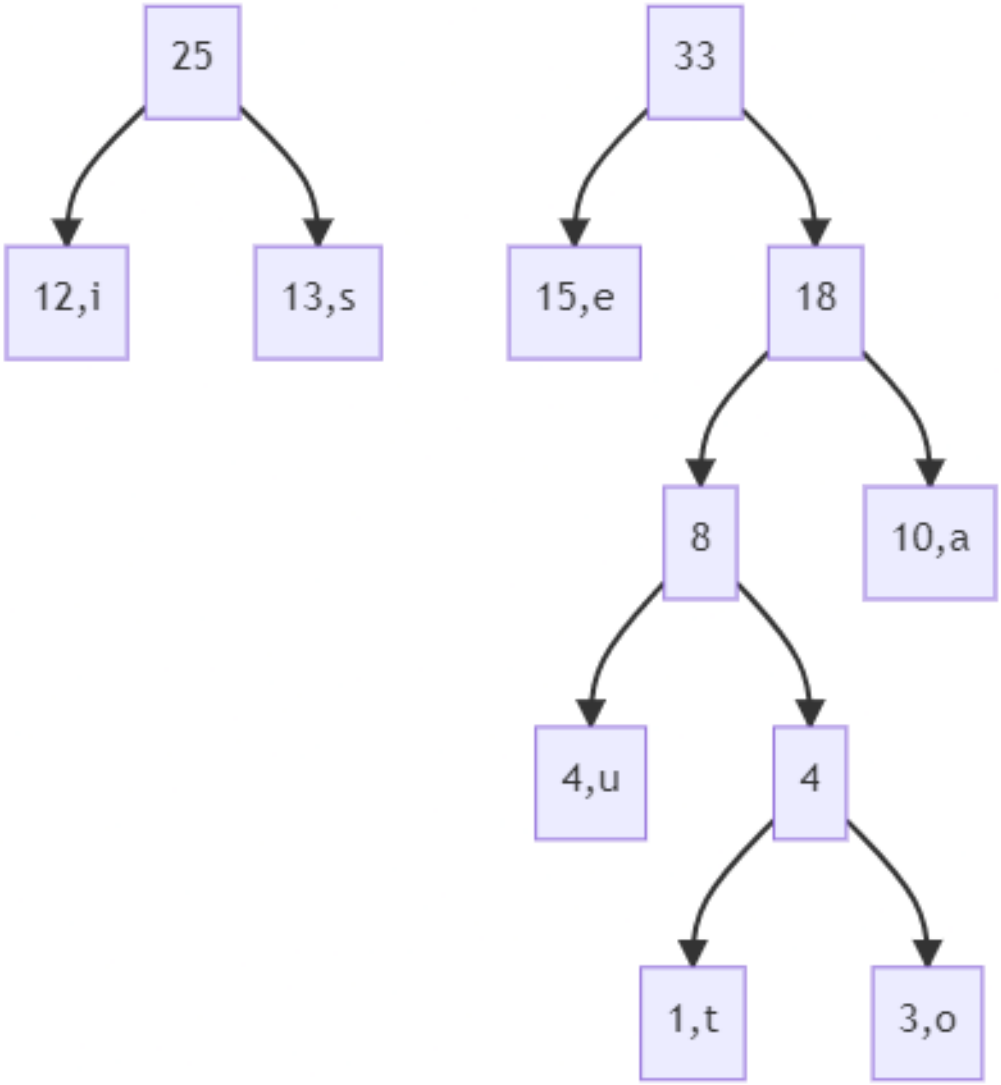

Figure 7: Step 7

Finally, two nodes are left with frequencies 25 and 33 and we get the final root node whose frequency is 58 $(25 + 33)$ **This is our final Huffman tree. Our algorithm stops when there is only one non-NULL element left**

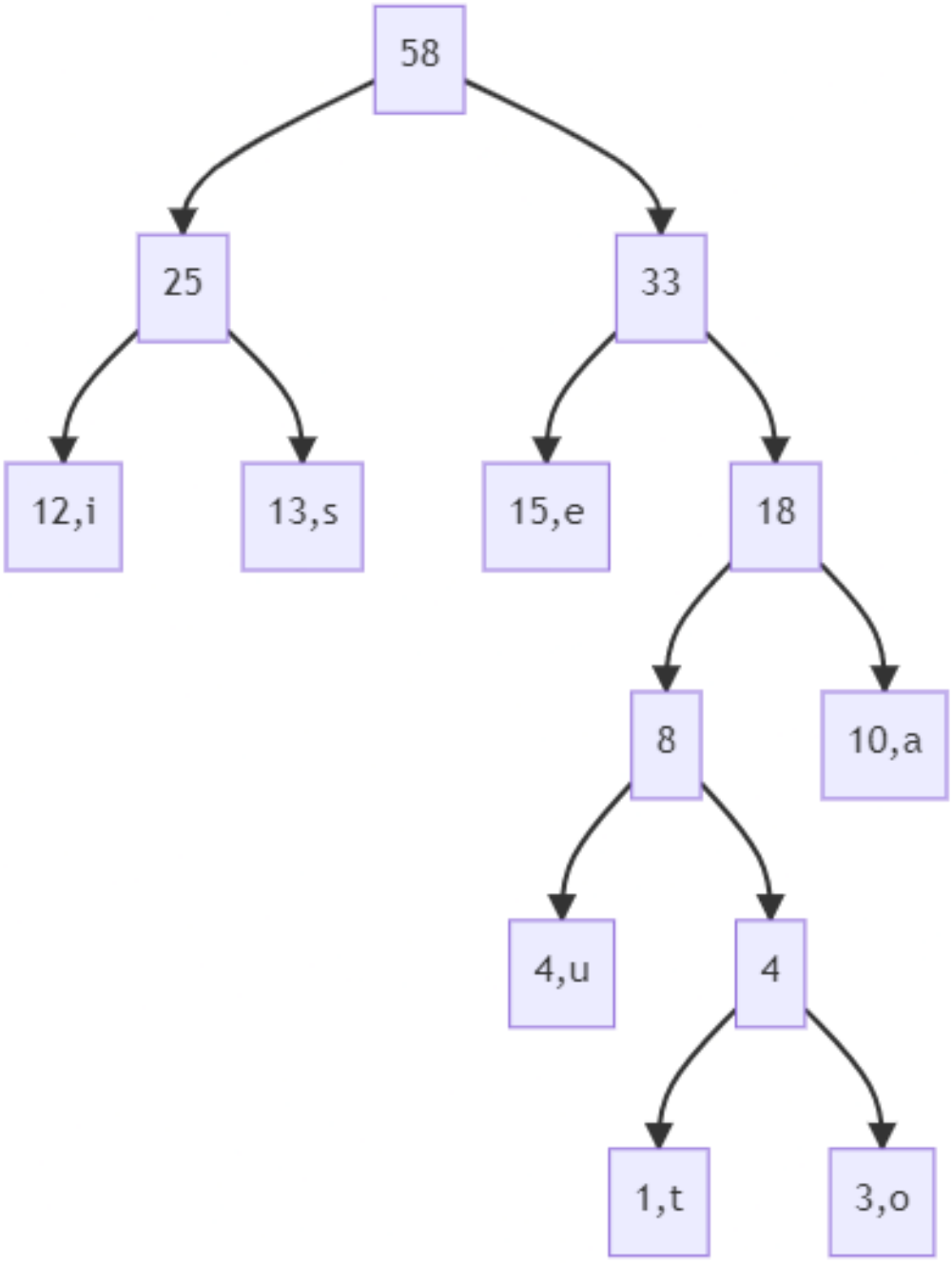

Figure 8: Step 8 , **Huffman tree**

4. Theorems and Proofs

We have the following theorems:

Theorem 4.1. ***In an optimum code, symbols that occur more frequently (have a higher probability of occurrence) will have shorter codewords than symbols that occur less frequently.***

Proof.
The proof statement is self explanatory! If more frequent symbols will have smaller codes and vice versa, then the total space occupied that is sum of products of lengths of encode and character itself, would be optimised. Therefore, a code that assigns longer codewords to symbols that occur more frequently cannot be optimum. Hence, Huffman algorithm is optimum.

Theorem 4.2. ***In an optimum code, the two symbols that occur least frequently will have the same length.***

Proof.
We are proving this point using principle of contradiction.
Suppose an optimum code exists in which the two codewords corresponding to the two least probable symbols are different . So, let's say, the longer codeword (Code2) is k bits longer than the shorter codeword (Code1):
Code1:01001....
Code2:01000...
Because this is a prefix free code, the shorter codeword cannot be a prefix of the longer codeword. This means that even if we drop the last k bits of the longer codeword, the two codewords would still be distinct.
Code1:01001....
Code2:01000...
As these codewords correspond to the least probable symbols in the alphabet, no other codeword can be longer than these codewords; therefore, there is no danger that the shortened codeword would become the prefix of some other codeword. Furthermore, by dropping these k bits we obtain a new code that has a shorter average length than previous one. This contradicts our initial assumption. Therefore, for an optimal code the second observation also holds true.
Also, Huffman Encode follows the same theorem as we are using MINHEAP in it so we get the sorted order of their frequencies(probabilities) and then we are using the first two values to build our Tree further, and as the tree builds, the codes get longer. And the last two values with the least frequency (probability) get the longest code .

5. Conclusions

As we collaborated on project, we learned alot about Huffman Encoding. While studying about it, we got to know more about the applications of greedy algorithms and various data structures and explored about various other compression techniques. We learned how Huffman compression algorithm is still used and implemented in various modern compression techniques even today.
We also observed the fact that compression ratio always comes out to be more than 1, that means our algorithm is able to compress the input text. As we worked through, we also got to know about how to collaborate using the Overleaf (latex),Git and Github.

6. Bibliography and citations

Handbook of Data Structures and Applications by Dinesh P. Mehta, Sartaj Sahni
Introduction to algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

7. References

`https://en.wikipedia.org/wiki/Huffman_coding`
`https://www.studytonight.com/data-structures/huffman-coding`
`https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Huffman.html`