Huffman Algorithm

By

Parsa Besharat

A handout submitted as part of the requirements

for the lecture, Algorithmics, of MSc Mathematics of Data and Resources Sciences

at the Technische Universität Bergakademie Freiberg

July, 2024

Supervisor: Dr. Jan Kurkofka

Abstract

Huffman's algorithms are well-known in data compression that serves the purpose of generating Huffman codes. Initially, as a prefix-free code the codes will be distinct from one another. The Huffman algorithm will evaluate frequencies of symbols after it completes the pass over data that populates symbols. It creates a structure that can efficiently manage symbols in an order such as a priority queue. The Huffman algorithm starts off by initializing data with symbols that each has a single node. Next, it would merge these symbols into a priority queue by associating merged symbols with the priority and introduces a decision to the new node that has branches to both symbols. Eventually the process will continue to extract the two symbols of lowest priority, merge, and introduce them back to the priority queue. The process is completed after all of the symbols are conforming to a binary tree. Then the tree that is created from would form a hierarchical tree to allow for quicker access to frequently used symbols. Huffman Algorithm table, then use backward to indicate directions to the root for a symbol. The Huffman Algorithm table designed to guarantee shortest directions to the most frequent symbols.

The Huffman algorithm produces an efficient data or maximized compression, since the most frequent symbols will have a shortened representation by the Binary Tree. Progressing further, the Huffman algorithm has a wide array of usage that continues to develop as there are specific applications in information theory, file encoding, and file compression standards, like ZIP. In addition, with the rise of multimedia, Huffman encoding is increasingly used with additional protocols, like network protocols.

Contents

Fundamentals

1.1 Definition of Greedy Algorithms

Algorithms that are considered greedy make decisions that are locally optimal, with the goal of reaching a global optimum or close to it. This method involves choosing what seems to be the best option at the present time without taking into account the future implications. Typically, this approach entails a series of steps or decisions, where each decision is based on the current state of the problem and the best choice is determined by a specific rule related to the problem domain.

For example, supposing we have a classroom and want to hold as many classes here as possible according to the Figure 1.1. As shown in the Figure 1.1, we would like to arrange the class of Art to be from 9AM until 10AM, English be from 9:30AM to 10:30AM, Math class be from 10AM to 11AM, Computer Science be from 10:30AM to 11:30 and at the end we have the Music class from 11AM to 12PM.

Art	9 AM	10 AM
English	9:30 AM	10:30 AM
Math	10 AM	11 AM
Computer science	10:30 AM	11:30 AM
Musics	11 AM	12 PM

Figure 1.1, Table of charts

However, we have problem in arranging the classes. As shown in the Figure 1.2, we cannot hold all of these classes in there, because some of them overlap. Since we may interfere with other classes as well. We cannot have both English and Math at the same time, in the same class. The main idea is to choose the best option available at each step without worrying about the overall solution.

Art	9 AM	10 AM
English	9:30 AM	10:30 AM
Math	10 AM	11 AM
Computer science	10:30 AM	11:30 AM
Musics	11 AM	12 PM

Figure 1.2, Greedy approach

1.2 Characteristics of Greedy Algorithms in Huffman Algorithm

In the context of Huffman coding, a greedy algorithm efficiently constructs an optimal prefix-free code by always merging the two least frequent symbols or subtrees. This characteristic ensures that the algorithm consistently makes locally optimal choices favoring the smallest frequencies first—without reconsidering previous mergers. By iteratively combining symbols or subtrees based solely on their immediate frequencies, the algorithm builds a Huffman tree that minimizes the overall bit length required to represent the input data. This greedy strategy leads to an efficient solution, though it does not guarantee the absolute optimal solution in every scenario, particularly when initial frequency distributions vary significantly.

Introduction of Huffman Algorithm

2.1 Overview of the Algorithm

In the field of computer science and Information Technology (IT), the Huffman algorithm stands as an algorithm, renowned for its efficiency in generating optimal prefix codes to minimize the storage or transmission size of textual or binary data. The algorithm begins with a frequency analysis of symbols within the input data, constructing a binary tree where each leaf node represents a symbol and its associated frequency. Through a systematic process of merging the two least frequent nodes into a new internal node, the tree progressively condenses until all nodes are interconnected into a single hierarchical structure. The resulting structure ensures that no codeword is a prefix of another, guaranteeing unambiguous decoding. By virtue of its elegant simplicity and robust theoretical underpinnings in information theory, Huffman coding has found widespread application in diverse fields such as data compression standards (like ZIP files), telecommunications, and multimedia compression formats. Its enduring relevance underscores not only its foundational role in efficient data handling but also its influence in shaping subsequent advancements in algorithmic efficiency and information processing methodologies.

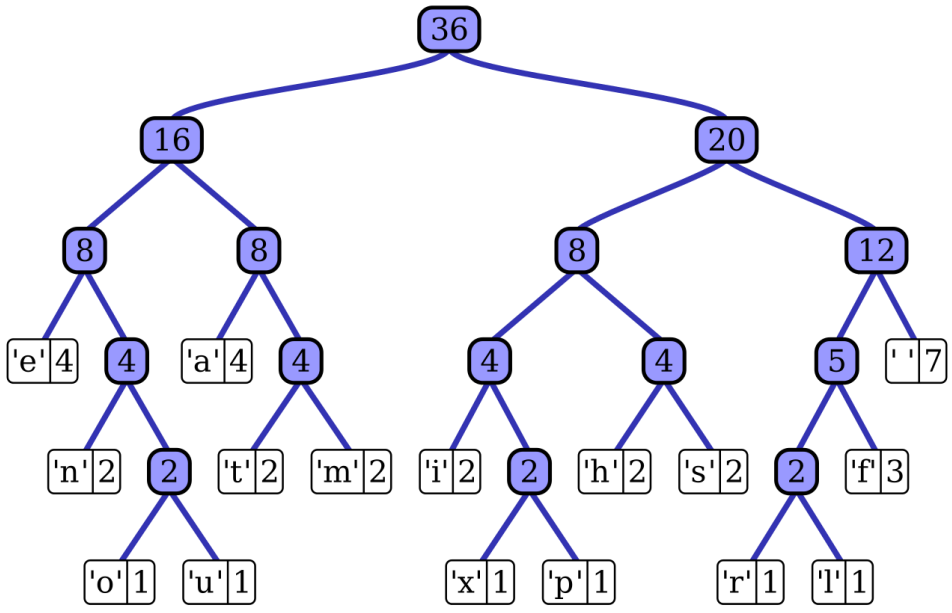


Figure 2.1. Huffman Tree Structure

2.2 History

In 1952, David A. Huffman created the Huffman algorithm to solve the challenge of efficiently encoding data using variable-length codes. While studying at MIT, Huffman was assigned the task of developing a method to minimize the average code length for messages sent over a noisy channel. Drawing on Claude Shannon's work in information theory, Huffman sought to design a coding system that would give shorter codes to symbols that appeared more frequently, thereby improving data compression.

Huffman made a significant discovery with a greedy algorithm that constructs an optimal prefix-free code in a step-by-step manner. The algorithm begins by constructing a binary tree of symbols based on their frequencies. It then systematically merges the two least frequent symbols into a single node until all symbols are integrated into a single tree. This approach guarantees that the resulting code is prefix-free, enabling unambiguous decoding, and has minimal redundancy in terms of its average code length.

In 1952, Huffman's algorithm gained widespread acclaim for its elegance and efficiency after the publication of his influential paper "A Method for the Construction of Minimum-Redundancy Codes". It presented a viable solution to the data compression problem and established the groundwork for subsequent progress in the field. Today, Huffman coding continues to be a fundamental technique in various applications, such as ZIP file compression formats, telecommunications protocols, and multimedia encoding schemes, demonstrating its enduring influence on information theory and computer science.

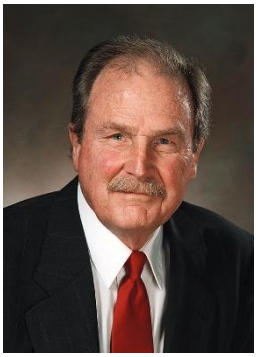


Figure 2.2, David A. Huffman

Huffman Algorithm

3.1 Implementation

The implementation of the Huffman algorithm involves 3 main steps:

1. Calculating the symbol frequencies;
2. Constructing the Huffman tree;
3. Assigning Huffman codes.

We have an example sentence and we would like to implement these steps:

Figure 3.1.1.1, Text in order to implement the algorithm.

3.1.1 Calculating the symbol frequencies:

a) In order to calculate the the symbol frequencies of our text, first we need to separate each character and see what characters are in our text. We started a traversal by iterating through the sentence. Because the first letter in text is **E**, then we must start with that.

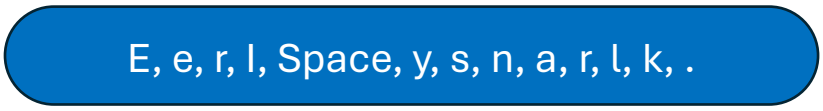


Figure 3.1.1.2, Each unique character in text.

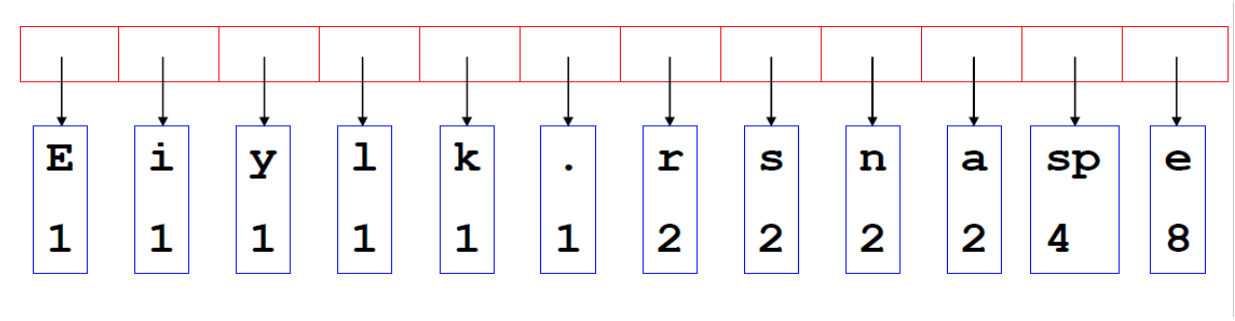
b) After that, we need to calculate how many times that character occurs in the text. For example, we have.

E	e	r	i	Space	y	s	n	a	l	k	.
1	8	2	1	4	1	2	2	2	1	1	1

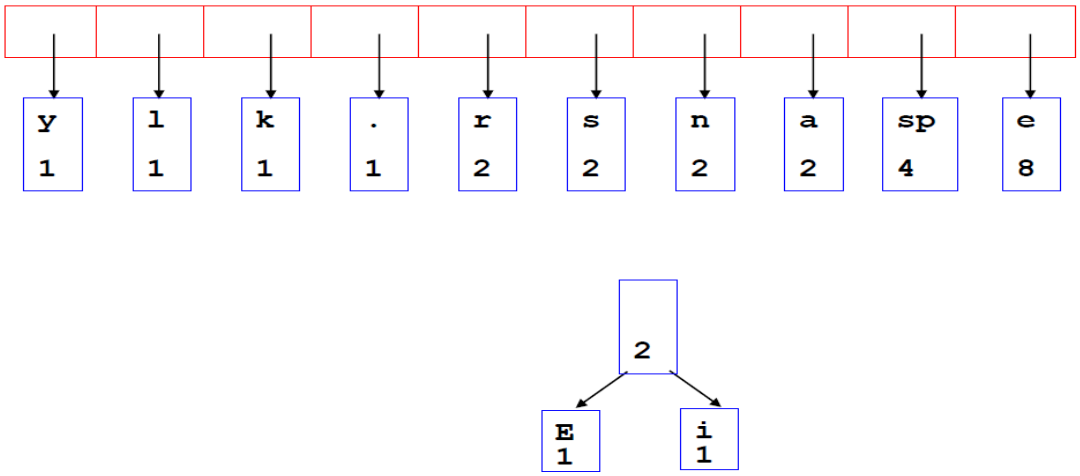
Figure 3.1.1.3, Each unique character in text. In our text even the character **Space** counts as well which we have 4 times this character in our text.

3.1.2 Constructing the Huffman tree

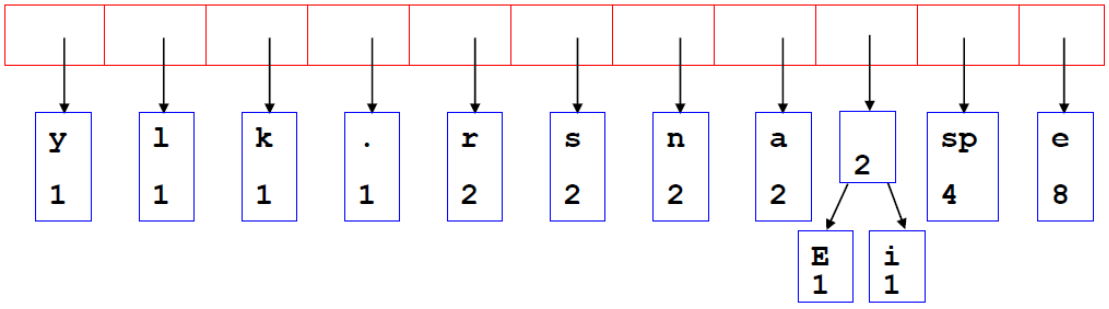
The construction of the Huffman tree involves repeatedly merging the two nodes with the lowest frequencies into a single node until only one tree remains, representing the optimal prefix-free code. Therefore, by iteratively merging the least frequent nodes, we could construct our node. The Figure 3.1.2.1 shows the main process of constructing the tree.



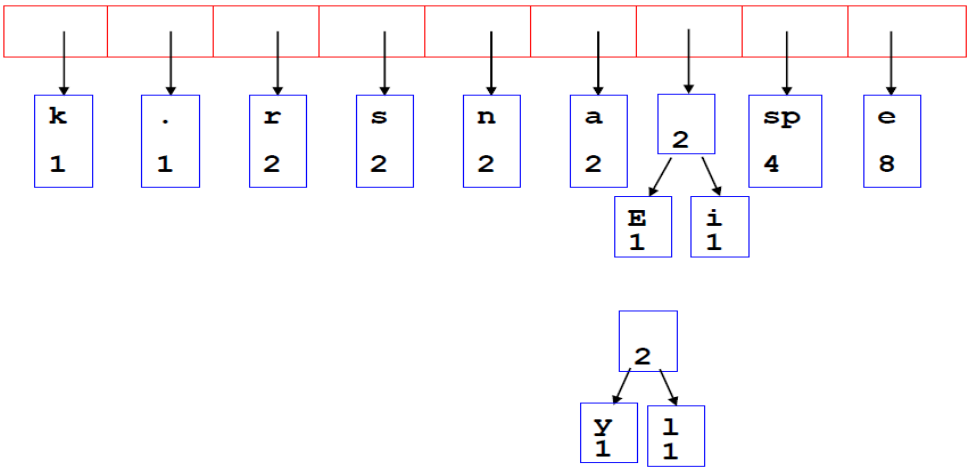
(a)



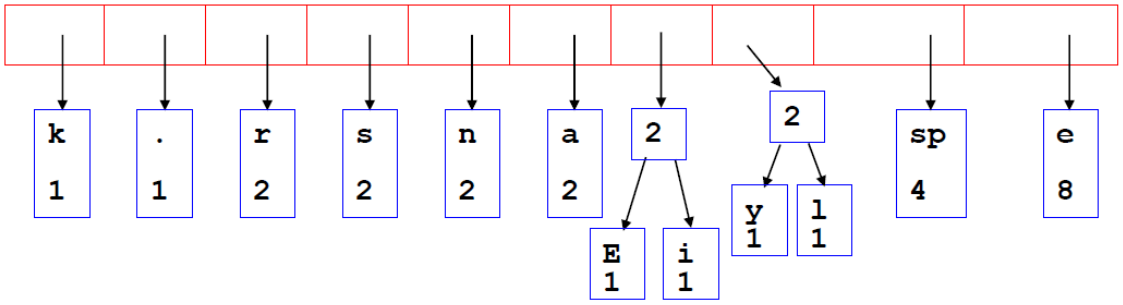
(b)



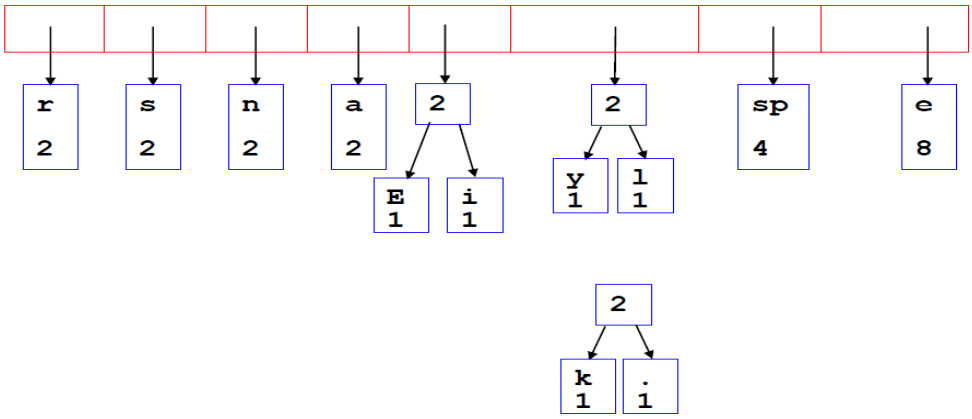
(c)


k 1
. 1
r 2
s 2
n 2
a 2
2
sp 4
e 8
E 1
i 1
2
y 1
l 1

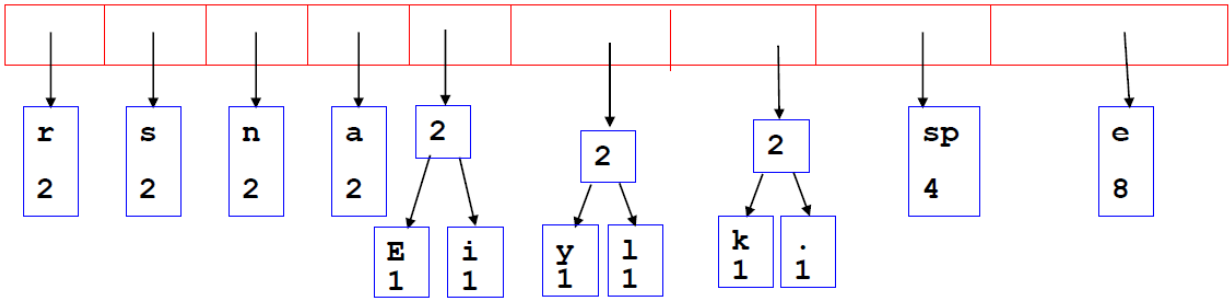
(d)


k 1
. 1
r 2
s 2
n 2
a 2
2
2
sp 4
e 8
E 1
i 1
y 1
l 1

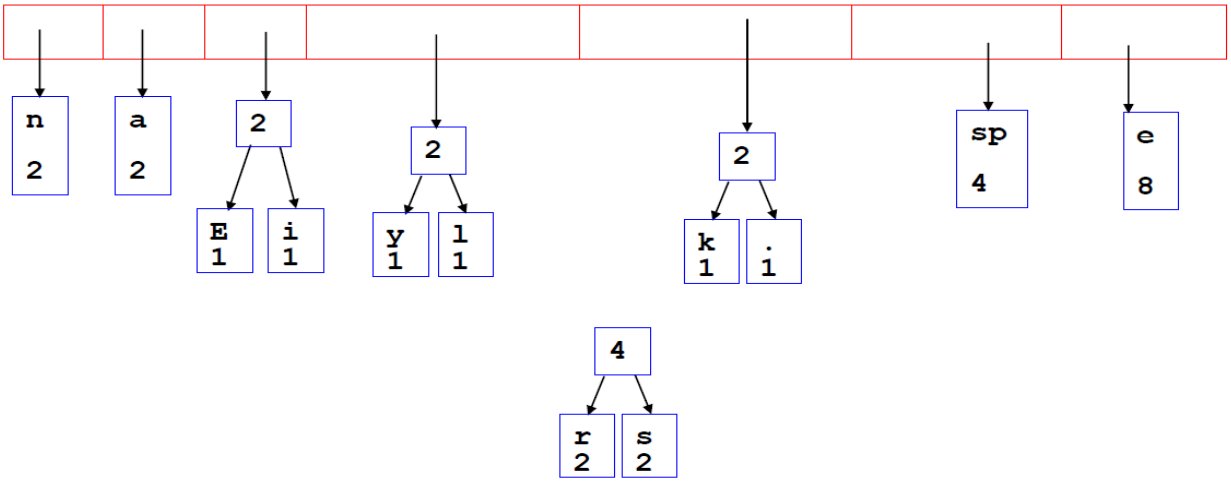
(e)


r 2
s 2
n 2
a 2
2
2
sp 4
e 8
E 1
i 1
y 1
l 1
2
k 1
. 1

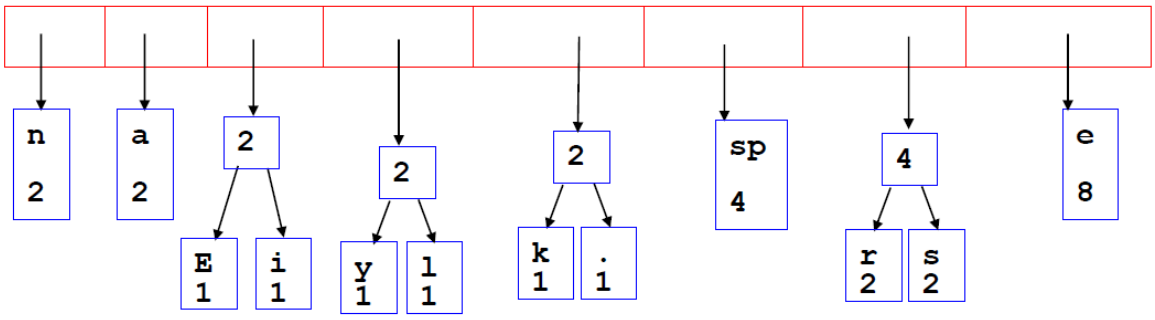
(f)


r 2
s 2
n 2
a 2
2
2
2
sp 4
e 8
E 1
i 1
y 1
l 1
k 1
. 1

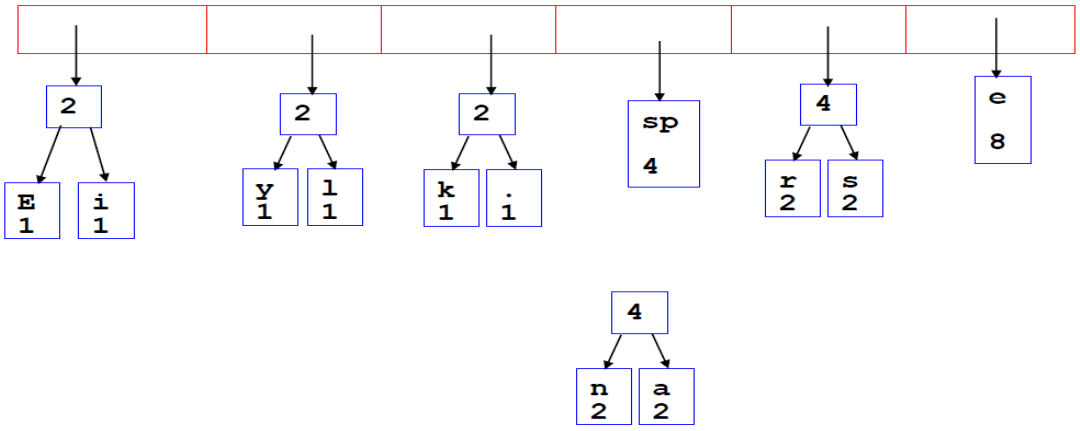
(g)


n 2
a 2
2
E 1
i 1
2
y 1
l 1
2
k 1
. 1
sp 4
e 8
4
r 2
s 2

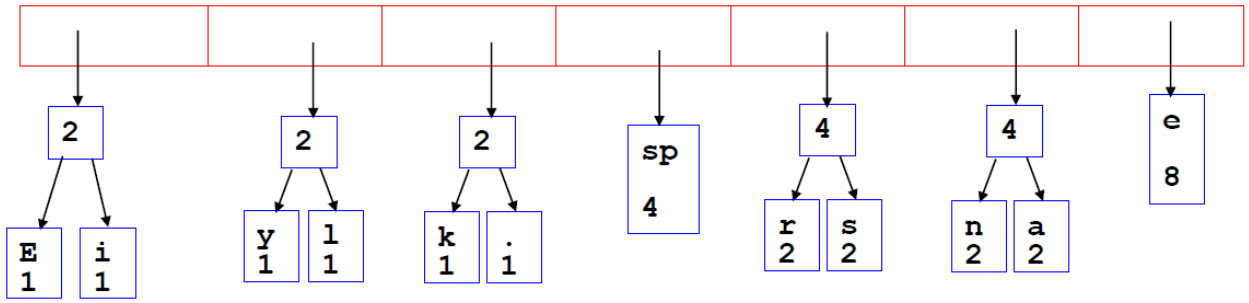
(h)


n 2
a 2
2
E 1
i 1
2
y 1
l 1
2
k 1
. 1
sp 4
4
r 2
s 2
e 8

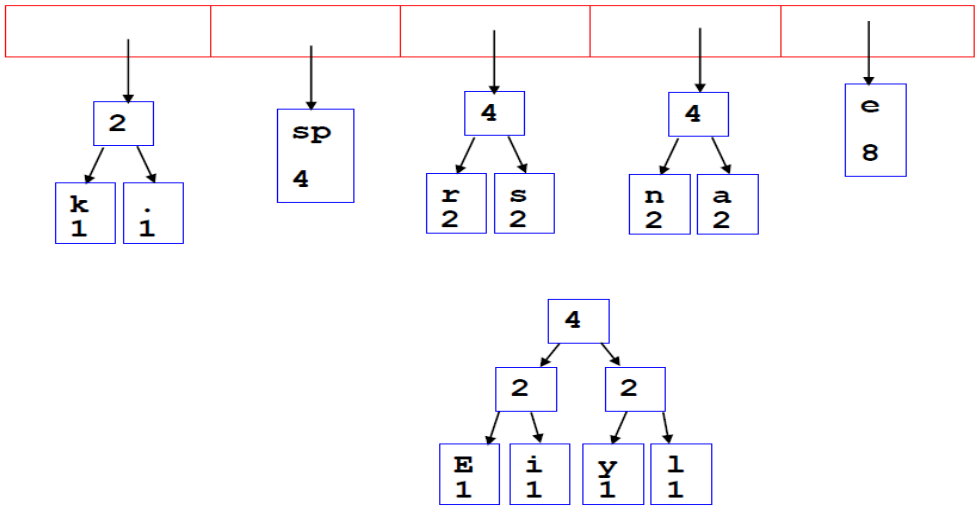
(i)


2
E 1
i 1
2
y 1
l 1
2
k 1
. 1
sp 4
4
r 2
s 2
e 8
4
n 2
a 2

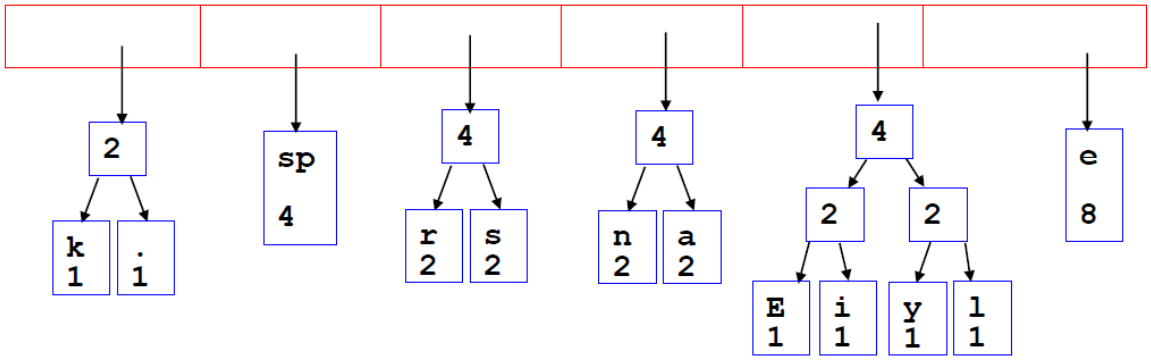
(j)


2
E 1
i 1
2
y 1
l 1
2
k 1
. 1
sp 4
4
r 2
s 2
4
n 2
a 2
e 8

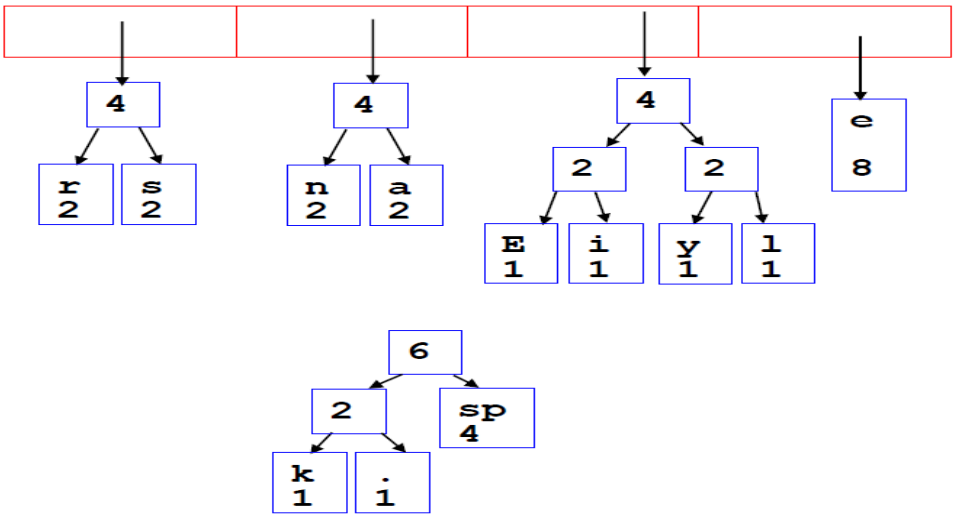
(k)


2
k 1
. i
sp 4
4
r 2
s 2
4
n 2
a 2
e 8
4
2
2
E 1
i 1
y 1
l 1

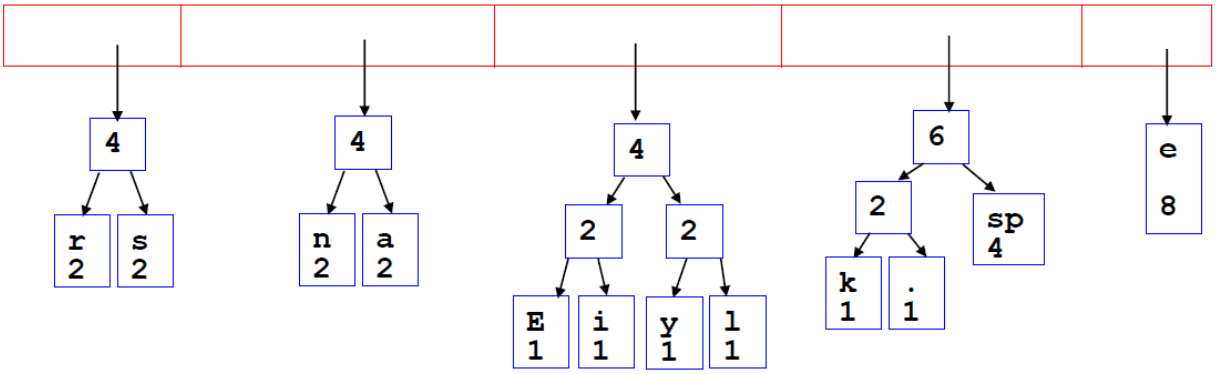
(L)


2
k 1
. 1
sp 4
4
r 2
s 2
4
n 2
a 2
4
2
2
E 1
i 1
y 1
l 1
e 8

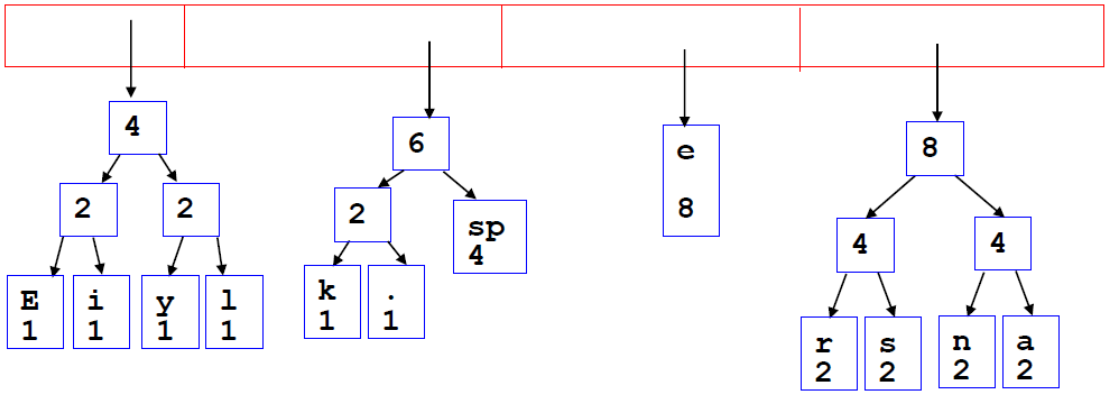
(m)


4
r 2
s 2
4
n 2
a 2
4
2
2
E 1
i 1
y 1
l 1
e 8
6
2
sp 4
k 1
. 1

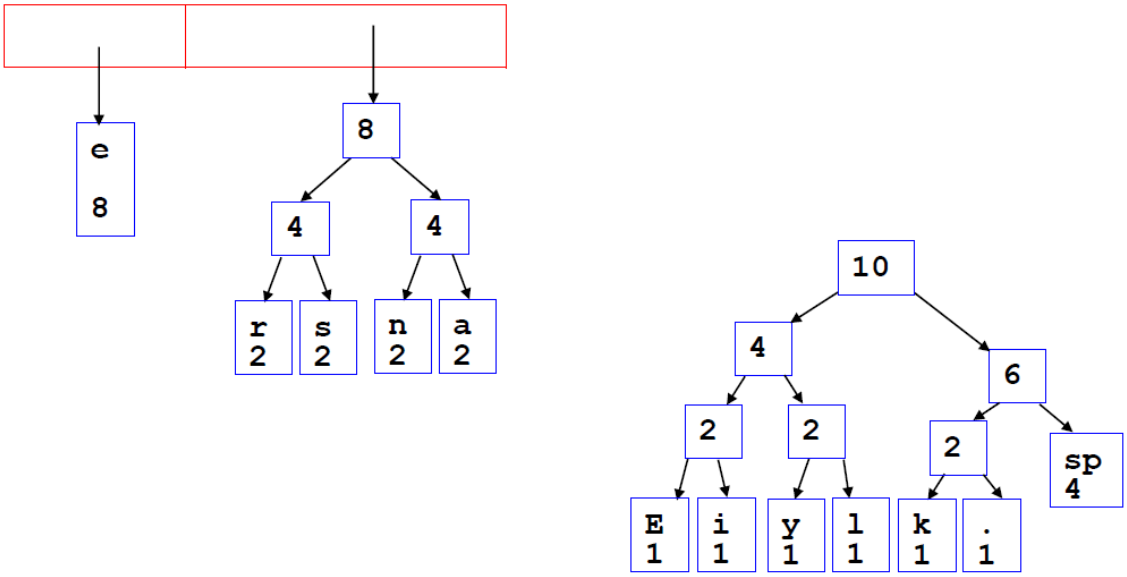
(n)


4
r 2
s 2
4
n 2
a 2
4
2
2
E 1
i 1
y 1
l 1
6
2
sp 4
k 1
. 1
e 8

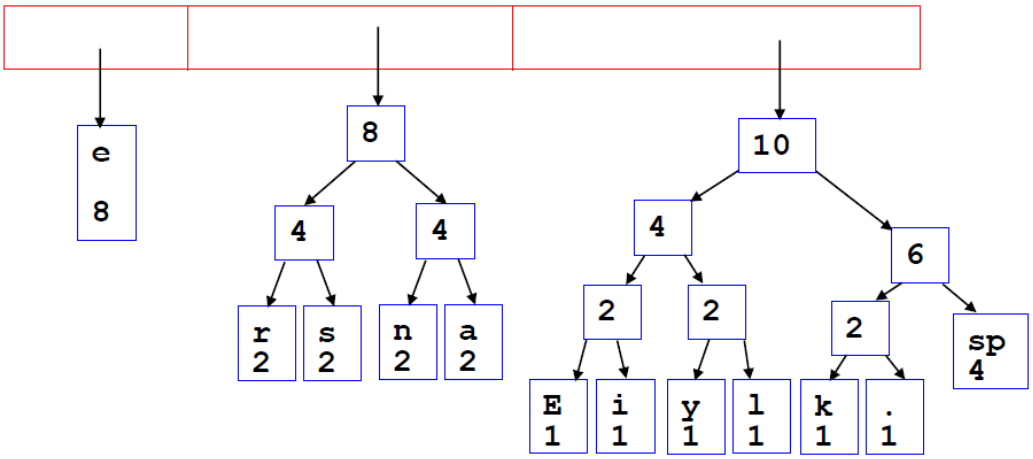
(o)


4
2
2
E 1
i 1
y 1
l 1
6
2
sp 4
k 1
. 1
e 8
8
4
4
r 2
s 2
n 2
a 2

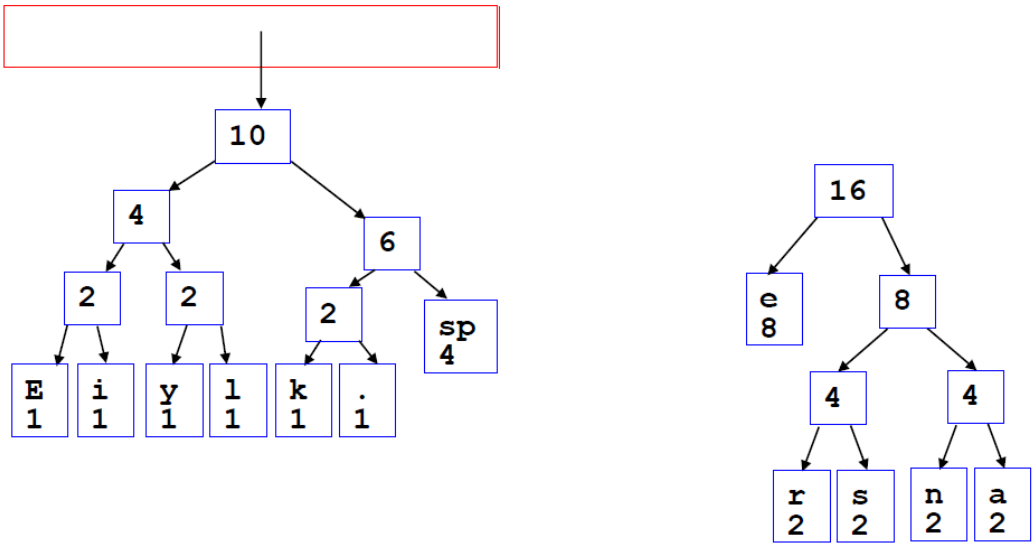
(p)


e
8
8
4
4
r
2
s
2
n
2
a
2
10
4
6
2
2
2
sp
4
E
1
i
1
y
1
l
1
k
1
.
1

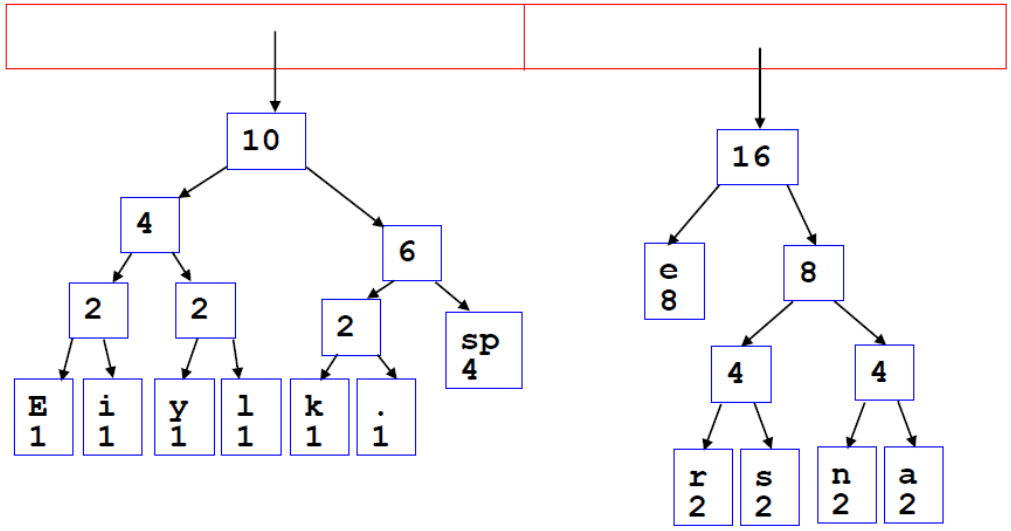
(q)


e
8
8
4
4
r
2
s
2
n
2
a
2
10
4
6
2
2
2
sp
4
E
1
i
1
y
1
l
1
k
1
.
1

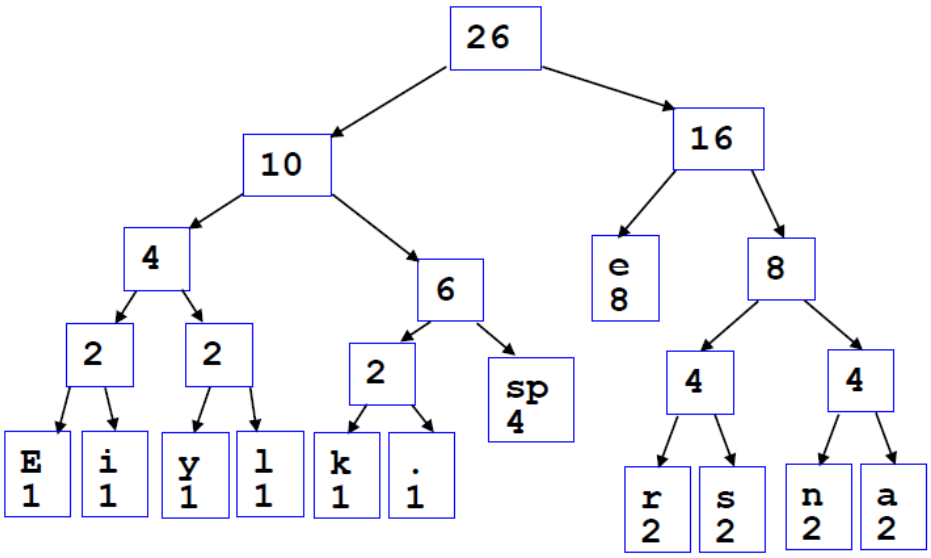
(r)


10
4
6
2
2
2
sp
4
E
1
i
1
y
1
l
1
k
1
.
1
16
e
8
8
4
4
r
2
s
2
n
2
a
2

(s)


10
4
6
2
2
2
sp
4
E
1
i
1
y
1
l
1
k
1
.
1
16
e
8
8
4
4
r
2
s
2
n
2
a
2

(t)



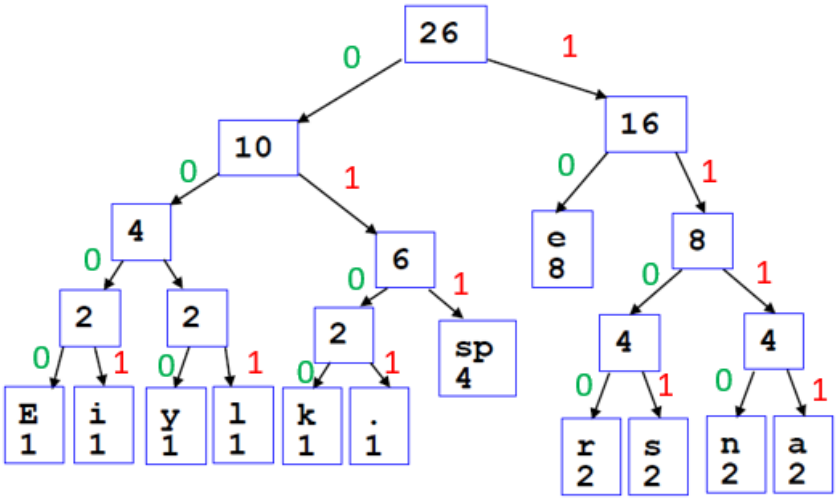
(u)

Figure 3.1.2.1, for constructing the Huffman tree, in figure (a), first we must put our words with their repetition in the text together. In figure (b) we sum the repetition of the first lowest amount together and create a small tree in the case that the sequence be followed. In figure (c), since we have the amount 2, we put our tree in the last step of the number 2. Since character a was the last element, we put are tree on our sequence in a way that it could be the last number 2. In figure (d), we calculate again the sum of our smallest numbers which is y and l. In figure (e), we repeat the step-in figure (c) and put our tree to be the last number 2. In figure (f), again we calculate the smallest numbers and create a tree again (figure (g)). In figure(h) since it is 4, we put our tree in the last side of the numbers between character e and **space** (figure (i)). In figure (j), by calculating the smallest numbers again we achieve number 4 tree with 2 leaf nodes, n and a, and put it to the last node of number 4 (figure (k)). In figure (L), again we sum the smallest numbers together and create a tree (figure (m)) then we must put it again in the last number 4. In figure (n), we sum the number 2 with our number 4 which is the **space** character. Then we will put it in sequence after number 4 because it is number 6 and bigger than 4 and smaller than 8 which is our character e (figure (o)). In figure (p), with sum of the first smallest numbers, we get our tree and put it in the sequence. Since it is 8, it is the biggest number so we put it at last. In the figure (q), we would calculate our first smallest numbers again which is 10 and put it in the last section (figure (r)). In the figure (s), by sum of the character e and the number 8, we get a tree again with number 16, and put it again the last section of our sequence (figure (t)). In the figure (u), we have created our tree fully.

3.1.3 Assigning Huffman codes

At the final step we must assign codes to it. For each leaf in the right section and right edges we assign code **1**, and for left edges we put number **0** as shown in the figure 3.1.3.1. At the end we finalize our text with the codes we assinged and turn it into hufman result text.

E	e	r	l	Sp	y	s	n	a	i	k	.
0000	10	1100	0011	011	0011	1101	1110	1111	0001	0100	0101



Eerie eyes seen near lake.

The encrypted text:

0000101100000110011

1000101011011010011

1110101111110001100

Figure 3.1.3.1, from the root which is 26, we assign the right edges the number 1 and left edges we assign number 0. As shown, the character **E** is **0000**, character **e** is **10** and so on. To get these codes we need to find the characters which is in the last nodes. Then we must follow the edges from the root and append each code to it (not sum). By calculating each number, we could get our text in the Huffman codes.

3.2 Encoding

The encoding process using the Huffman algorithm involves converting the input data into a compressed binary format based on the Huffman tree. Each character in the input is replaced by its corresponding Huffman code, which is a unique sequence of bits derived from the path from the root to the character's leaf node in the Huffman tree. This results in a compressed bitstream that efficiently represents the original data, minimizing the number of bits required. The previous process was an encoding process.

3.3 Decoding

The decoding process in Huffman coding is a systematic method to reconstruct the original data from its compressed bitstream by using the Huffman tree, which was constructed during the encoding phase. This process begins at the root of the Huffman tree and involves reading the compressed bitstream one bit at a time. Each bit directs the traversal through the tree: a '0' indicates a move to the left child node, while a '1' signifies a move to the right child node. The decoder continues this traversal until it reaches a leaf node, which represents a specific character in the original data. Once a leaf node is reached, the character stored in that node is added to the output sequence. After recording the character, the decoder resets to the root node and resumes reading the next bit from the bitstream. This process of traversal, character identification, and reset is repeated for each bit in the compressed data, ensuring that every segment of the bitstream is correctly translated back into its corresponding character.

For instance, if the bitstream begins with the sequence '101', the decoder starts at the root, moves right for '1', left for '0', and right again for '1', arriving at a leaf node that corresponds to a specific character. This character is then added to the output, and the process restarts from the root for the next bits. The decoder repeats this procedure until the entire bitstream is processed, faithfully reconstructing the original data as intended by the Huffman algorithm's design.

Algorithmic Details

4.1 Pseudocode of the Algorithm

This part contains multiple functions in our code:

```
input: a string of characters
output: a frequency table

frequency_table = {}
for each character in input_string:
    if character in frequency_table:
        frequency_table[character] += 1
    else:
        frequency_table[character] = 1
```

(a)

```
input: frequency table
output: Huffman tree

create a priority queue (min-heap) and insert all characters with their frequencies
for each character, frequency in frequency_table:
    create a new node with the character and frequency
    insert the node into the priority queue

while priority queue has more than one node:
    extract the two nodes with the lowest frequencies
    create a new node with these two nodes as children and with frequency equal to the sum of their frequencies
    insert the new node back into the priority queue

the remaining node in the priority queue is the root of the Huffman tree
```

(b)

```
input: Huffman tree
output: a table of Huffman codes

huffman_codes = {}

function generate_codes(node, current_code):
    if node is a leaf:
        huffman_codes[node.character] = current_code
    else:
        generate_codes(node.left, current_code + "0")
        generate_codes(node.right, current_code + "1")

generate_codes(root of Huffman tree, "")
```

(c)

```
input: input_string, huffman_codes
output: encoded string

encoded_string = ""
for each character in input_string:
    encoded_string += huffman_codes[character]
```

(d)

```
input: encoded_string, Huffman tree
output: decoded string

decoded_string = ""
current_node = root of Huffman tree
for each bit in encoded_string:
    if bit is "0":
        current_node = current_node.left
    else:
        current_node = current_node.right

    if current_node is a leaf:
        decoded_string += current_node.character
        current_node = root of Huffman tree
```

(e)

Figure 4.1.2.1, the pseudocode for constructing the Huffman tree begins with the creation of a frequency table from the input data (a). Each character's frequency is stored. In step (b), a priority queue (min-heap) is initialized, where each character and its frequency are inserted as nodes. The queue ensures that nodes with lower frequencies have higher priority. In (c), the algorithm iteratively extracts the two nodes with the lowest frequencies from the priority queue, combines them into a new node with a frequency which is sum of their frequencies, and reinserts this node back into the queue. This process continues until only one node remains in the queue, representing the root of the Huffman tree (d). Once the Huffman tree is constructed, the next step (e) involves recursively traversing the tree to generate Huffman codes for each character. Starting from the root, a **0** is appended for a left branch and a **1** for a right branch until a leaf node is reached, at which point the path traced forms the Huffman code for that character. Finally, in (f), the input data is encoded using the generated Huffman codes, replacing each character with its respective code to produce the compressed output.

4.2. Analysis of Runtime Efficiency

The runtime efficiency of the Huffman algorithm primarily hinges on two key operations: constructing the Huffman tree and encoding/decoding the data. Constructing the Huffman tree involves building a priority queue (min-heap) of characters based on their frequencies, which has a time complexity of $\boldsymbol{O}\ (\boldsymbol{n}\log \boldsymbol{n})$ where $\boldsymbol{n}$ is the number of unique characters. The combination of nodes in the tree also operates due to the min-heap structure, ensuring that nodes with the lowest frequencies are consistently prioritized. Encoding and decoding operations traverse the Huffman tree in $\boldsymbol{O}(\boldsymbol{m})$ time per character, where $\boldsymbol{m}$ is the length of the input string.

5. Applications

Huffman coding finds practical applications across various domains:

- PNG Images: Huffman coding is integral to the DEFLATE algorithm used in this type image compression, reducing file sizes while maintaining image quality.
- PDF Files: It compresses text and images in PDF documents, enabling smaller file sizes and faster document sharing.
- HTTP Data Compression: Enhances web performance by compressing their responses, speeding up webpage loading times for improved user experience.
- Database Storage: Huffman coding reduces the storage space required for large databases by compressing text and binary data, optimizing database performance and efficiency.

6. Conclusion

To sum up, Huffman coding plays a vital role in efficient data compression by using optimal prefix-free codes to reduce file sizes through assigning shorter codes to more frequent symbols. Its simplicity and effectiveness are essential in many applications in today's computing and telecommunications.

References

[1] Grokking Algorithms . An illustrated guide for programmers and other curious people (2016, Manning Publications - Aditya Bhargava)

[2] Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein