Design and Analysis of Algorithm (DAA)

Greedy Algorithms

(Huffman Coding)

Dr. Dayal Kumar Behera

School of Computer Engineering
KIIT Deemed to be University, Bhubaneswar, India

Encoding

- During transmission or storage of data each character is coded.
- A **binary code** encodes each character as a binary string or **codeword**.
- Assume we have a 1,00,000 character file, we want to store it in an efficient way.
- Suppose we are using ASCII representation for each character (7 bits per character) => Size of file = 7,00,000 Bits.

Types of Encoding

Type of coding

- Fixed-Length Coding: **each codeword has the same length.**
- Variable-Length Coding: **codewords may have different lengths**.

	a	b	c	d	e	f
Frequency(In Thousand)	**45**	**13**	**12**	**16**	**9**	**5**
Fixed Length	**000**	**001**	**010**	**011**	**100**	**101**
Variable Length	**0**	**101**	**100**	**111**	**1101**	**1100**

Types of Encoding: Example

	a	b	c	d	e	f
Frequency (in Thousands)	45	13	12	16	9	5
Fixed length codeword	000	001	010	011	100	101
Variable length codeword	0	101	100	111	1101	1100

Code for Data Compression

--- **Fixed length Code**

(45+13+12+16+9+5)*3*1000=300,000 bits

--- **Variable length Code**

(45*1+13*3+12*3+16*3+9*4+5*4)*1000=224,000 bits

A saving of 25%

Application of Variable-length Code

- Used for data compression.

Encoding

Given a code (corresponding to some alphabet L) and a message it is easy to *encode* the message. Just replace the characters by the codewords.

Example: L = {a; b; c; d}
If the code is C1{a = 00; b = 01; c = 10; d = 11}:
then **bad** is encoded into 010011

If the code is C2 {a = 0; b = 110; c = 10; d = 111}
then **bad** is encoded into 1100111

Decoding

Given an encoded message, *decoding* is the process of turning it back into the original message.
A message is *uniquely decodable* (vis-a-vis a particular code) if it can only be decoded in one way.

$$
\begin{aligned}
C_1 &= \{a = 00,\ b = 01,\ c = 10,\ d = 11\}.\\
C_2 &= \{a = 0,\ b = 110,\ c = 10,\ d = 111\}.\\
C_3 &= \{a = 1,\ b = 110,\ c = 10,\ d = 111\}
\end{aligned}
$$

For example relative to C1, 010011 is uniquely decodable to **bad**.
Relative to C2, 1100111 is uniquely decodable to **bad**.

C3 {a=0, b=01, c=11, d=1011}

Text: aabac

Encoded Binary String: 0001011

Decoded Message: aaad

Not Worked. Why?

Prefix Codes

Prefix Code: A code is called a prefix (free) code if no codeword is a prefix of another one.

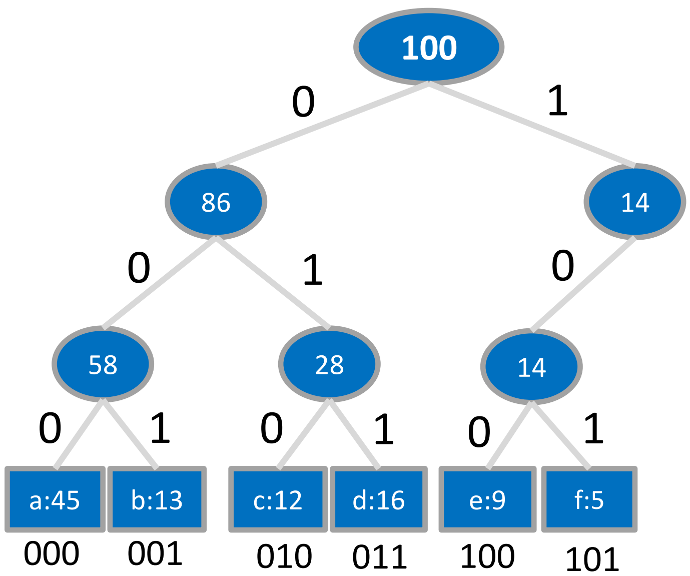
	a	b	c	d	e	f
Prefix Codes	**0**	**101**	**100**	**111**	**1101**	**1100**

- Encoding String "abc" as 0.101.100

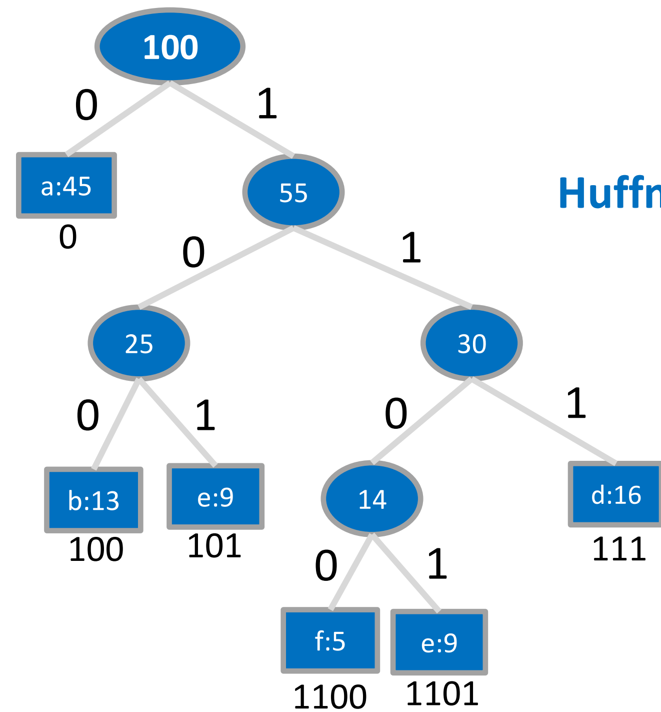
Every message encoded by a prefix free code is uniquely decipherable(Decoding without any ambiguity).

Example: 001011101 → aabe

Tree Representation



Fixed Length Coding System

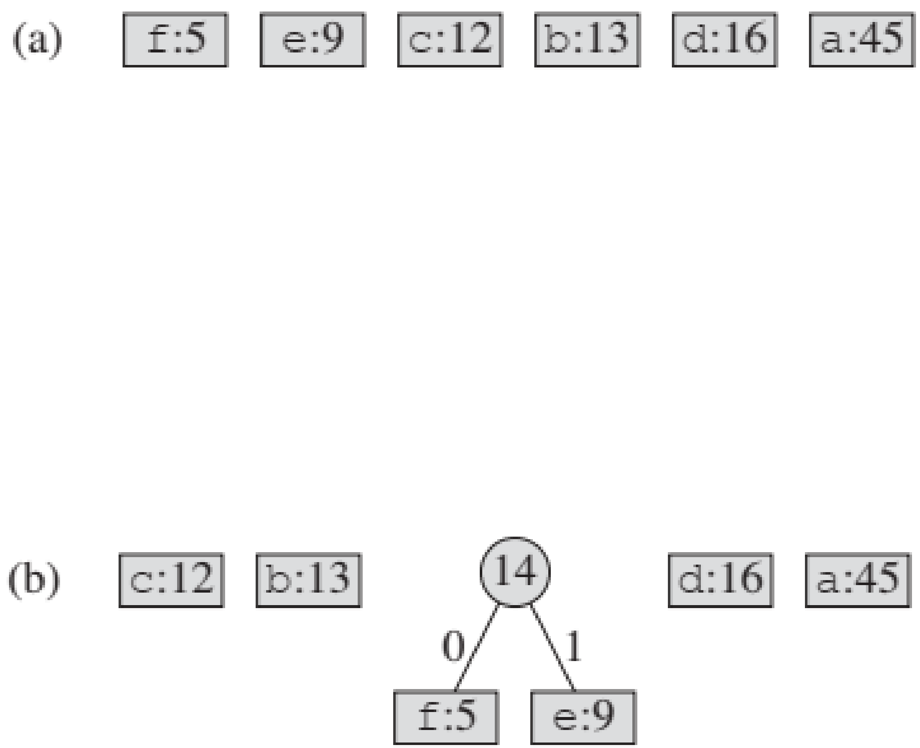


Huffman Coding

Variable Length Coding System

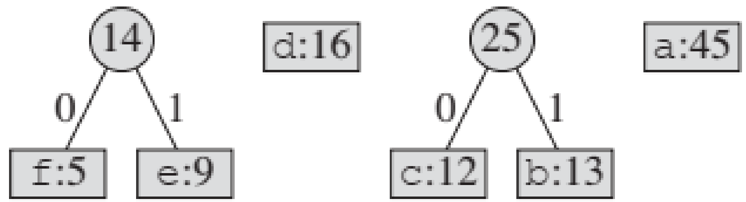
- Full Binary Tree
- N leaf nodes and N-1 Intermediate Nodes

Huffman Coding Procedure

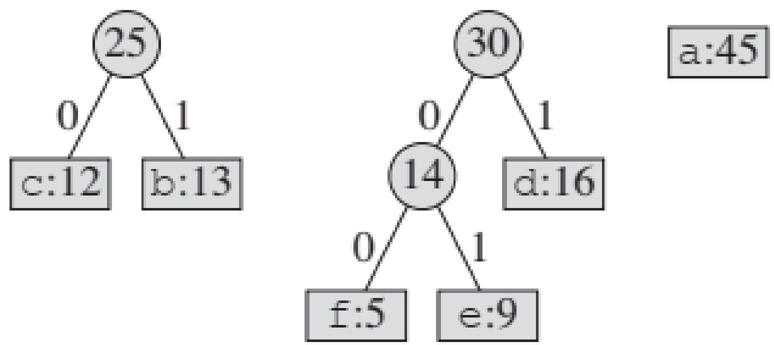


Huffman Coding Procedure

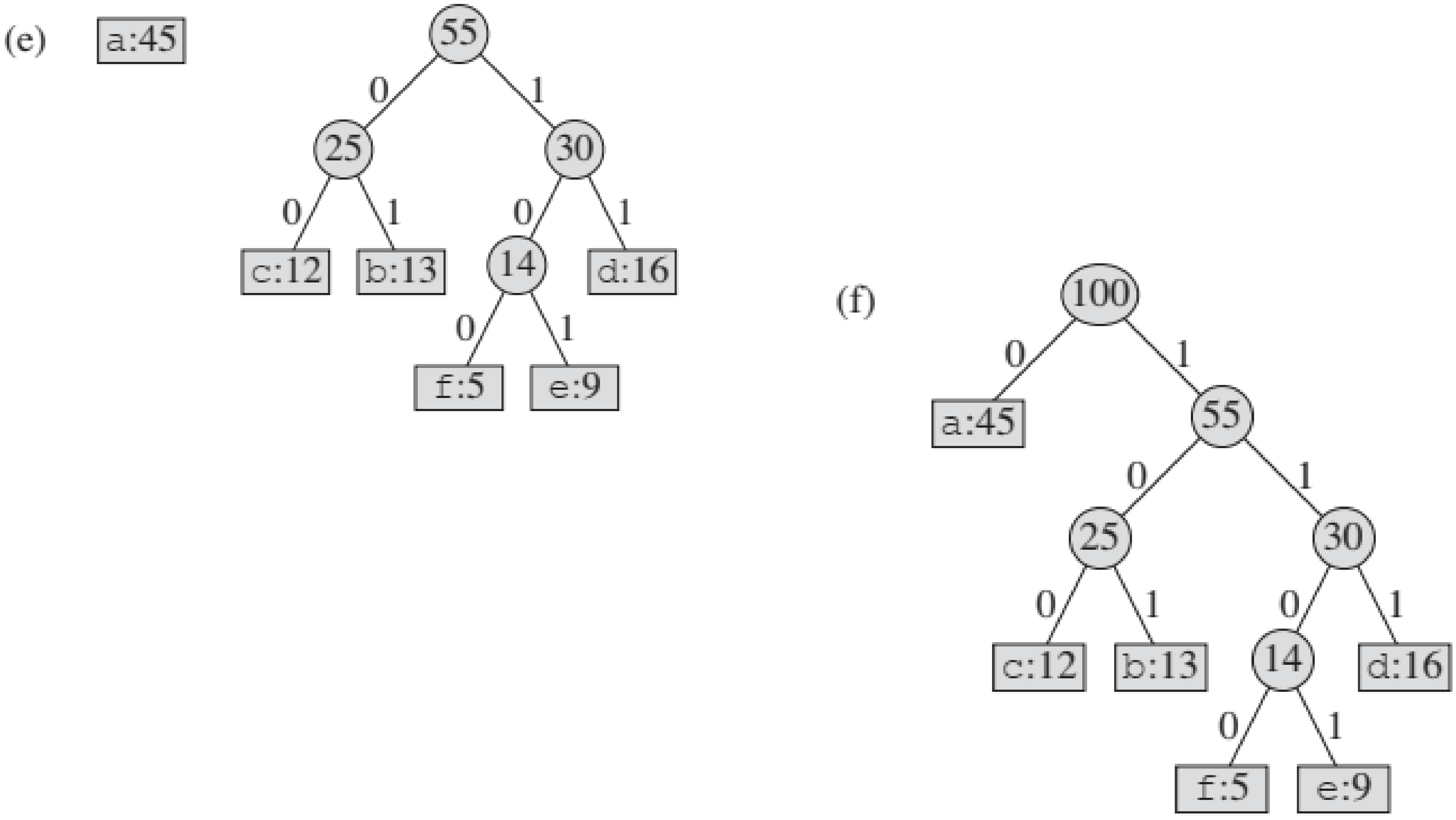
(c)



(d)



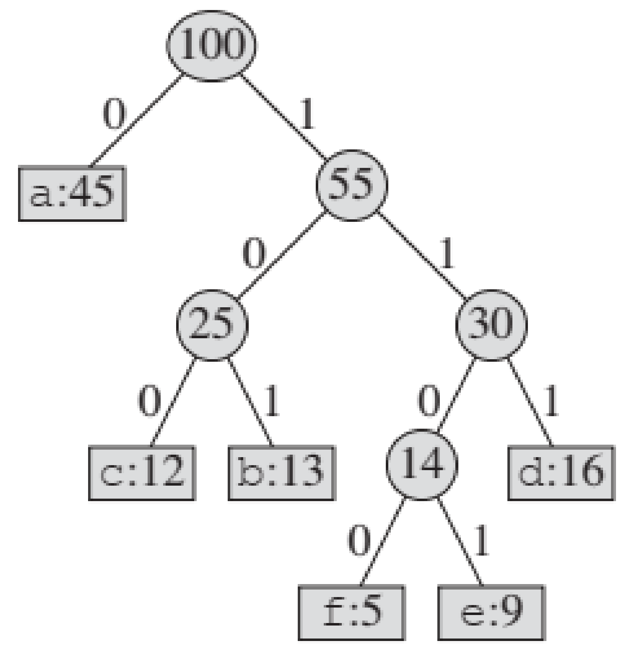
Huffman Coding Procedure



Representation of Prefix Codes

Representation is a binary tree *T*:

- One leaf for each character
- Internal nodes always have two outgoing edges, labeled 0 and 1
- Code of character: follow path to leaf and accumulate bits



Resulting Huffman Code:

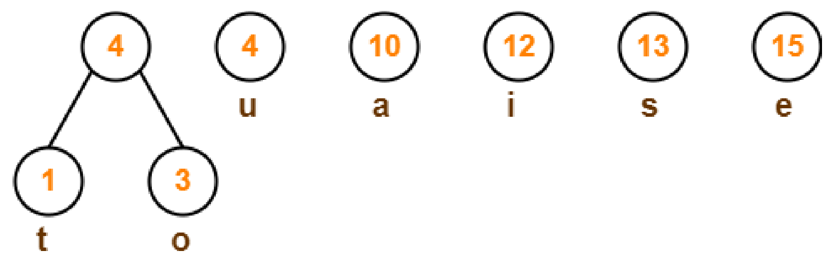
a = 0 b = 101 c = 100 d = 111 e = 1101 f = 1100

Example 2

Characters	Frequencies
a	10
e	15
i	12
o	3
u	4
s	13
t	1

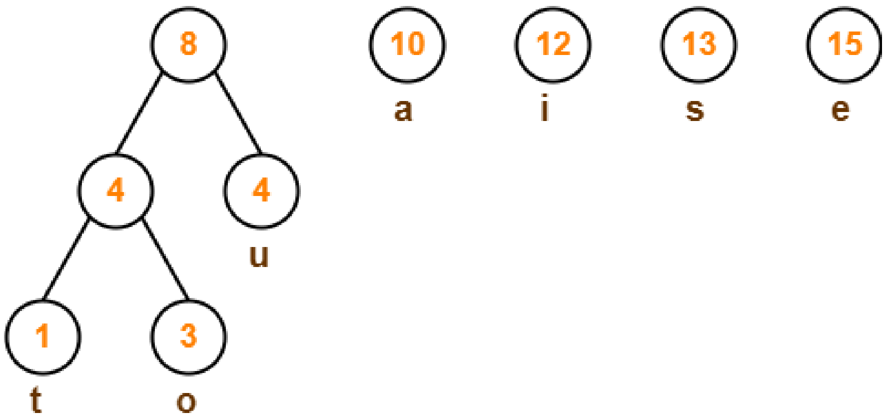
STEP 1:

STEP 2:

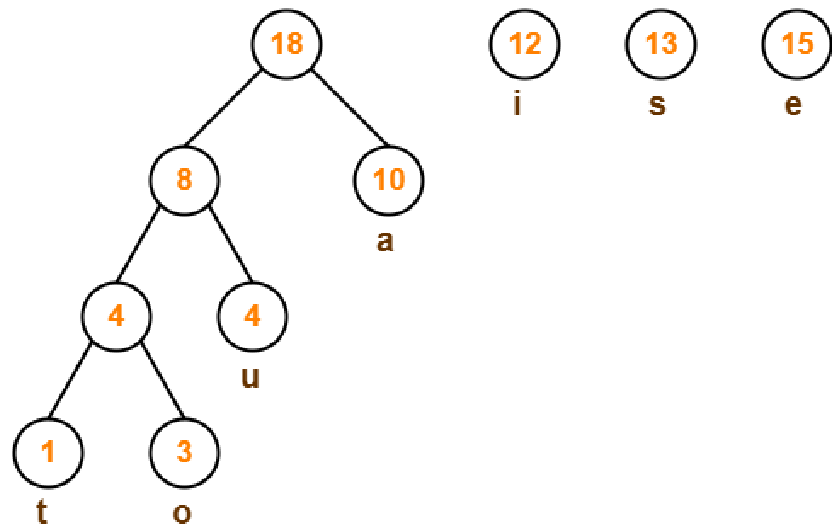


Example 2

STEP 3:

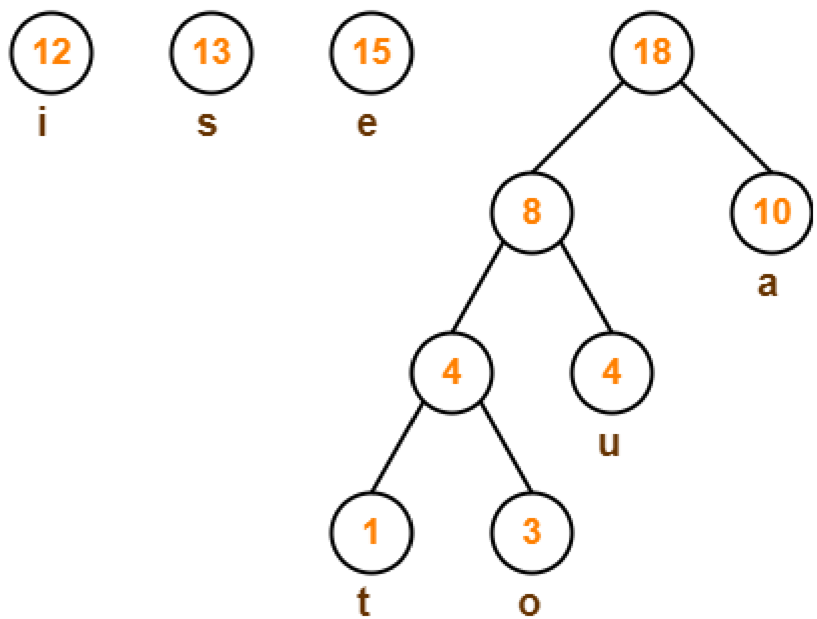


STEP 4:

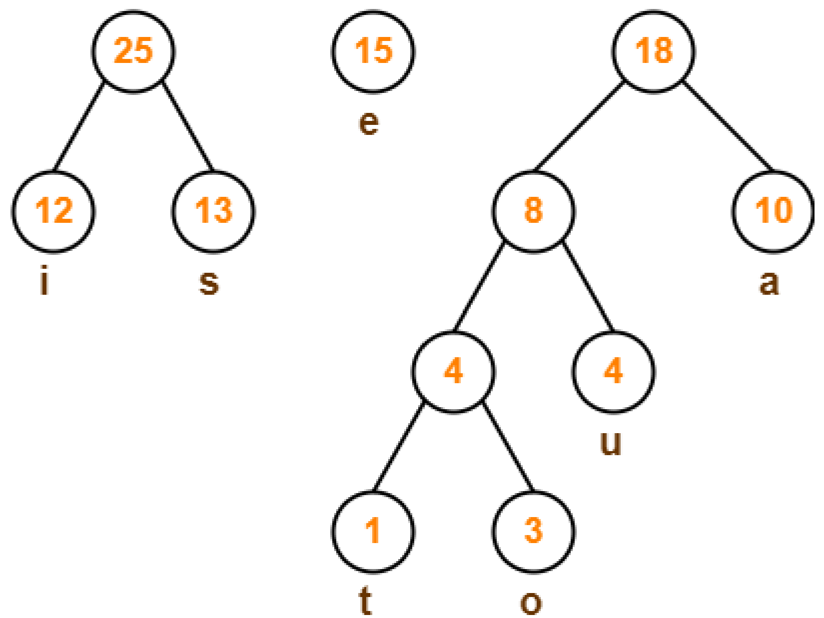


Example 2

STEP 5:

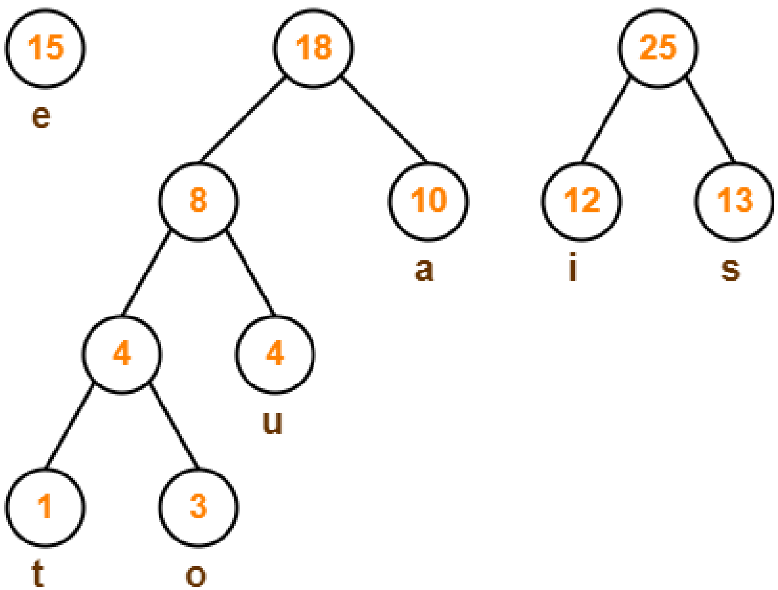


STEP 6:

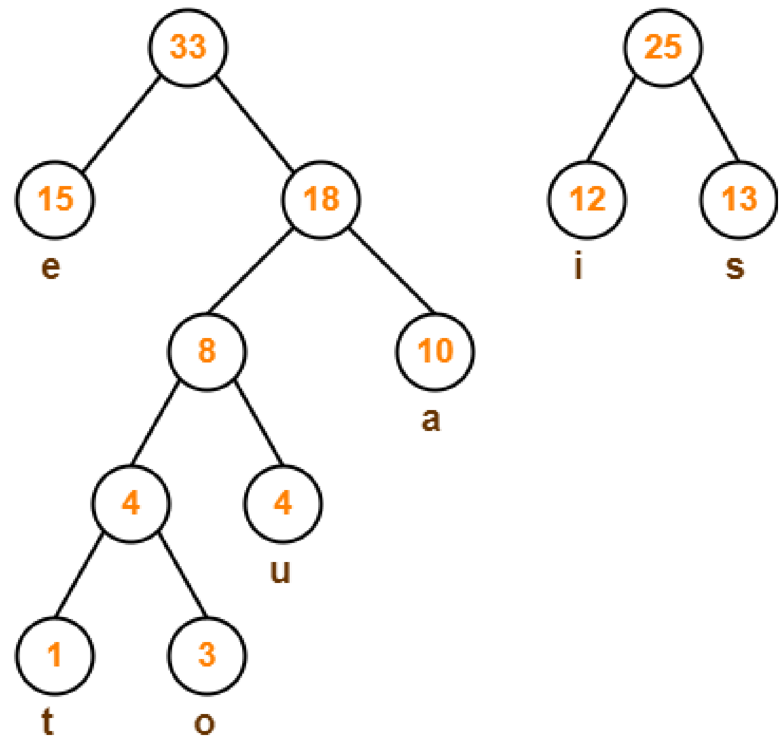


Example 2

STEP 7:

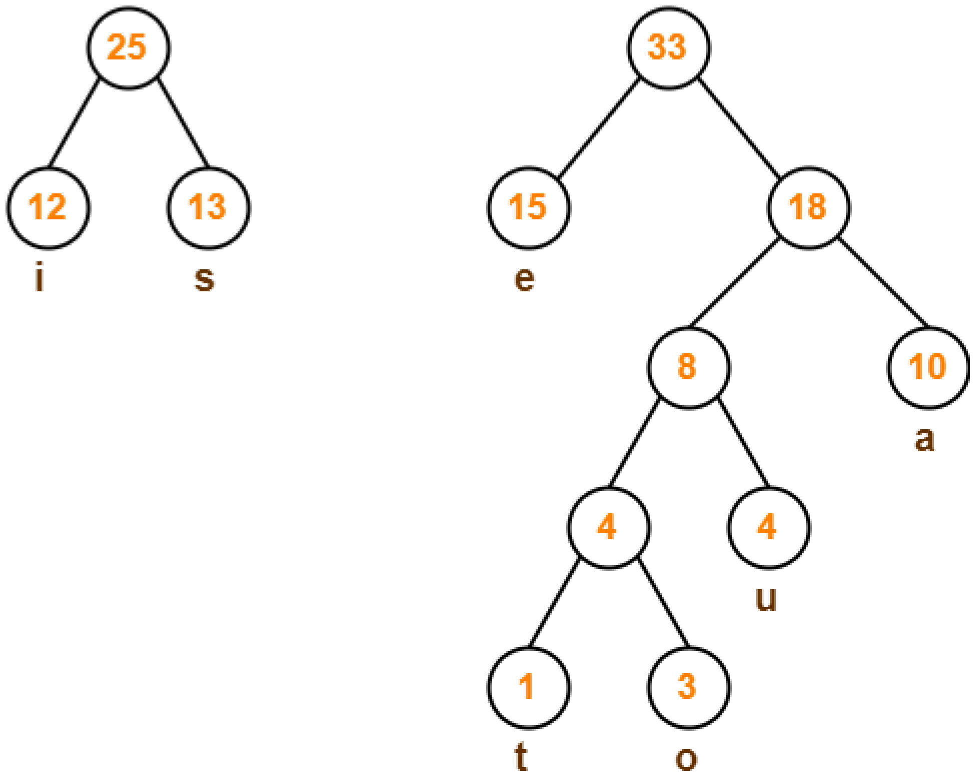


STEP 8:



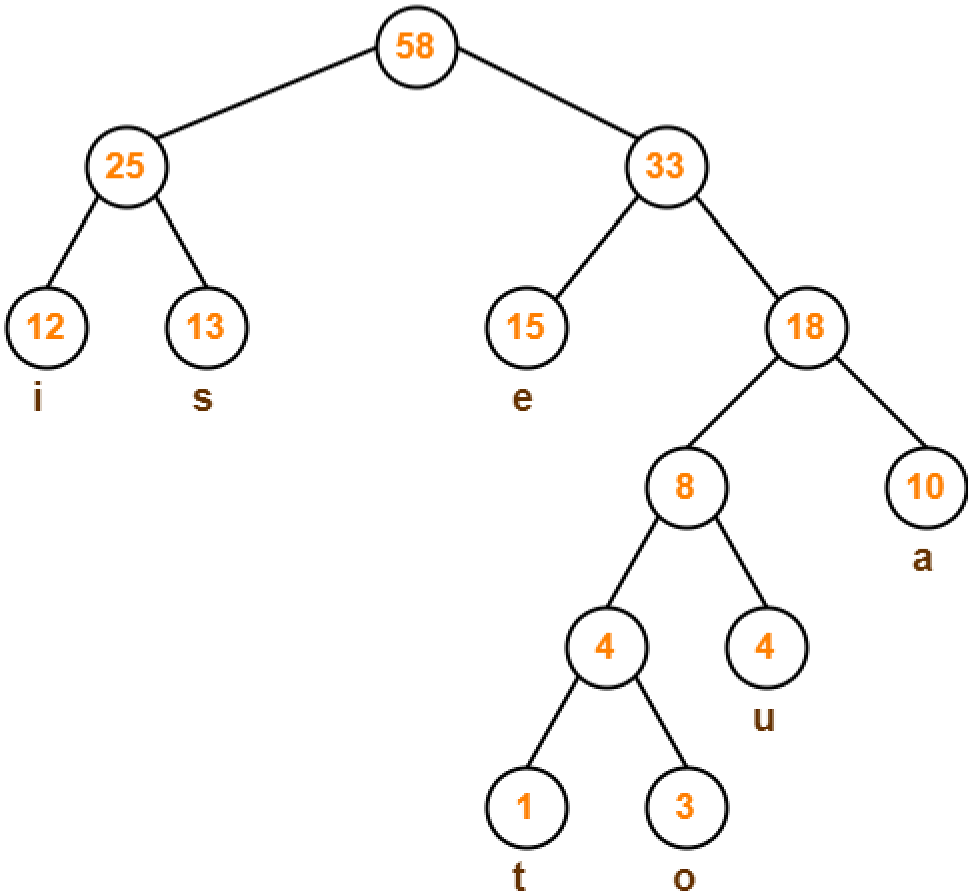
Example 2

STEP 9:



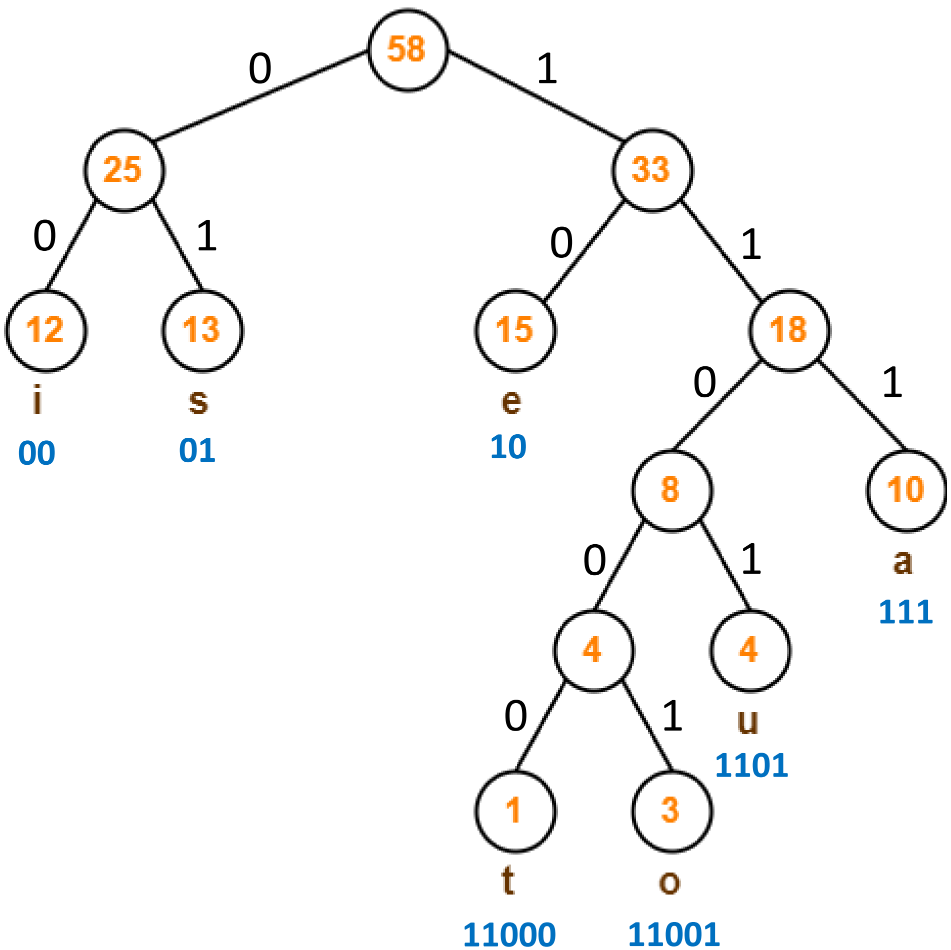
Example 2

STEP 10:



Huffman Tree

Example 2



- Label of leaf is frequency of character.
- Left edge is labeled 0; right edge is labeled 1
- Path from root to leaf is codeword associated with character.

Huffman Code

1. Huffman Code For Characters-

To write Huffman Code for any character, traverse the Huffman Tree from root node to the leaf node of that character.
Following this rule, the Huffman Code for each character is-

Characters	Frequencies	Code
a	10	111
e	15	10
i	12	00
o	3	11001
u	4	1101
s	13	01
t	1	11000

From here, we can observe
Characters occurring less frequently in the text are assigned the larger code.
Characters occurring more frequently in the text are assigned the smaller code.

Average Code Length

2. Average Code Length-

$$L_{avg} = \frac{\sum_{c \in C} f[c] \; * \; d_T[c]}{\sum_{c \in C} f[c]}$$

Using formula, we have-
Average code length

$d_T[c]$ = depth of char c in the tree T
$f[c]$ = frequency of char c

= ∑ (frequency_i x code length_i) / ∑ (frequency_i)

= { (10 x 3) + (15 x 2) + (12 x 2) + (3 x 5) + (4 x 4) + (13 x 2) + (1 x 5) } / (10 + 15 + 12 + 3 + 4 + 13 + 1)

= 2.52

3. Length of Huffman Encoded Message-

Total number of bits in Huffman encoded message

= Total number of characters in the message x Average code length per character

= 58 x 2.52

= 146.16

≅ 147 bits

OR

$$B(T) = \sum_{c \in C} f[c] * d_T[c]$$

Huffman Algorithm

```
Huffman(C)
n = |C|
Q = C                                   // Q is a Min-Priority Queue.
for i=1 to n-1
    allocate a new node z
    left[z] = x = EXTRACT-MIN(Q)
    right[z] = y = EXTRACT-MIN(Q)
    f[z] = f[x] + f[y]
    INSERT(Q, z)
return EXTRACT-MIN(Q)                   // Return the root of the tree
```

Huffman Algorithm: Analysis

Algorithm Huffman(C)

1. n = |C|
2. Q = C
3. **for** i=1 **to** n-1
4. allocate a new node z
5. left[z] = x = EXTRACT-MIN(Q) lgn
6. right[z] = y = EXTRACT-MIN(Q) lgn
7. f[z] = f[x] + f[y]
8. INSERT(Q, z) lgn
9. **return** EXTRACT-MIN(Q) lgn

Line 2: Building a min Heap : O(n)

n-1 times => O(nlgn)

Time Complexity of the Algorithm : O(nlgn)


Each of your
actions will
have an
impact on your
future.


Once you know
who is walking
with you on your path.
you will never
be afraid.

Thank you