# Advanced Algorithms Analysis and Design

By

Nazir Ahmad Zafar

# Lecture No 26

## Huffman Coding

# Today Covered

- Huffman Problem
- Problem Analysis
  - Binary coding techniques
  - Prefix codes
- Algorithm of Huffman Coding Problem
- Time Complexity
- Road Trip Problem
  - Analysis and Greedy Algorithm
- Conclusion

# Huffman Coding

# Using ASCII Code: Text Encoding

- Our objective is to develop a code that represents a given text as compactly as possible.
- A standard encoding is ASCII, which represents every character using 7 bits

Example

Represent "An English sentence" using ASCII code

– 1000001 (A) 1101110 (n) 0100000 ( ) 1000101 (E) 1101110 (n) 1100111 (g) 1101100 (l) 1101001 (i) 1110011 (s) 1101000 (h) 0100000 ( ) 1110011 (s) 1100101 (e) 1101110 (n) 1110100 (t) 1100101 (e) 1101110 (n) 1100011 (c) 1100101 (e)

= 133 bits ≈ 17 bytes

# Refinement in Text Encoding

- Now a better code is given by the following encoding:

  ‹space› = 000, A = 0010, E = 0011, s = 010,
  c = 0110, g = 0111, h = 1000, i = 1001,
  l = 1010, t = 1011, e = 110, n = 111

- Then we encode the phrase as

  0010 (A) 111 (n) 000 ( ) 0011 (E) 111 (n) 0111 (g)
  1010 (l) 1001 (i) 010 (s) 1000 (h) 000 ( ) 010 (s)
  110 (e) 111 (n) 1011 (t) 110 (e) 111 (n) 0110 (c)
  110 (e)

- This requires 65 bits ≈ 9 bytes. Much improvement.
- The technique behind this improvement, i.e., Huffman coding which we will discuss later on.

# Major Types of Binary Coding

There are many ways to represent a file of information.

**Binary Character Code** (or C**ode**)

- each character represented by a unique binary string.

- **Fixed-Length Code**
  - If $\Sigma = \{0, 1\}$ then
  - All possible combinations of two bit strings

    $$\Sigma \times \Sigma = \{00, 01, 10, 11\}$$

  - If there are less than four characters then two bit strings enough
  - If there are less than three characters then two bit strings not economical

# Fixed Length Code

- **Fixed-Length Code**
  - All possible combinations of three bit strings

    $\Sigma \times \Sigma \times \Sigma = \{000, 001, 010, 011, 100, 101, 110, 111\}$
  - If there are less than nine characters then three bit strings enough
  - If there are less than five characters then three bit strings not economical and can be considered two bit strings
  - If there are six characters then needs 3 bits to represent, following could be one representation.

    a = 000, b = 001, c = 010,
    d = 011, e = 100, f = 101

# Variable Length Code

- **Variable-Length Code**
  - better than a fixed-length code
  - It gives short code-words for frequent characters and
  - long code-words for infrequent characters
- Assigning variable code requires some skill
- Before we use variable codes we have to discuss prefix codes to assign variable codes to set of given characters

# Prefix Code (Variable Length Code)

- A prefix code is a code typically a variable length code, with the "prefix property"
- Prefix property is defined as no codeword is a prefix of any other code word in the set.

Examples

1. Code words {0,10,11} has prefix property
2. A code consisting of {0, 1, 10, 11} does not have, because "1" is a prefix of both "10" and "11".

Other names

- Prefix codes are also known as prefix-free codes, prefix condition codes, comma-free codes, and instantaneous codes etc.

# Why are prefix codes?

- Encoding simple for any binary character code;
- Decoding also easy in prefix codes. This is because no codeword is a prefix of any other.

Example 1

- If a = 0, b = 101, and c = 100 in prefix code then the string: 0101100 is coded as 0·101·100

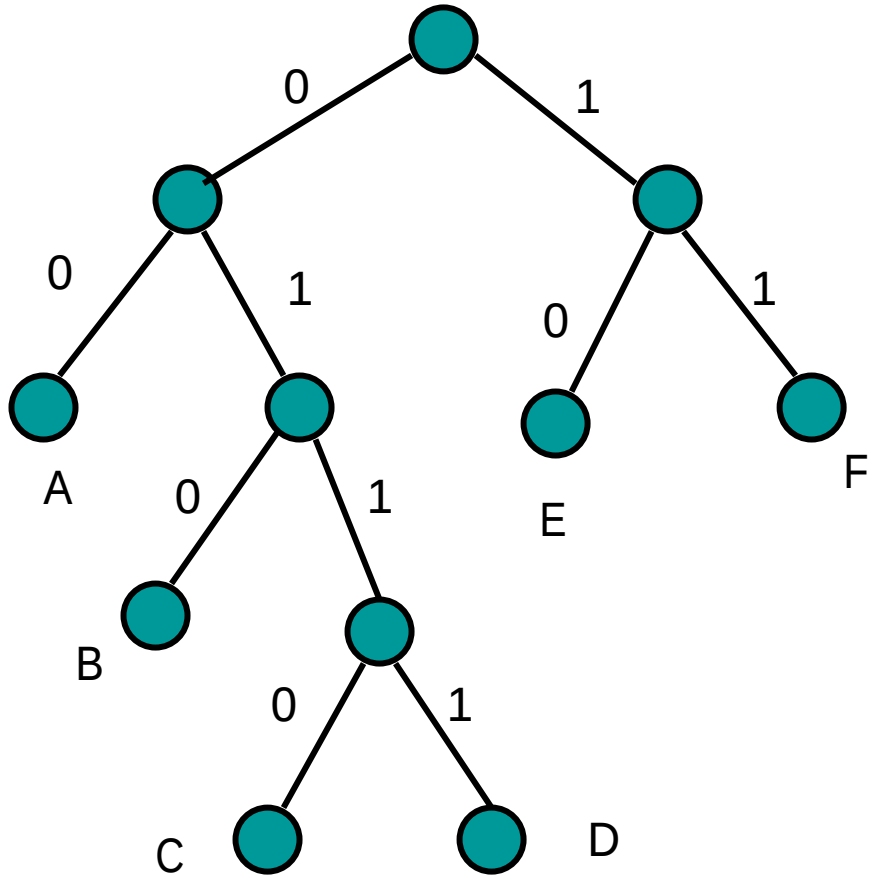
Example 2

- In code words: {0, 1, 10, 11}, receiver reading “1” at the start of a code word would not know whether
  - that was complete code word “1”, or
  - prefix of the code word “10” or of “11”

# Prefix codes and binary trees

- Tree representation of prefix codes

| | |
|---|---|
| A | 00 |
| B | 010 |
| C | 0110 |
| D | 0111 |
| E | 10 |
| F | 11 |



# Huffman Codes

- In Huffman coding, variable length code is used
- Data considered to be a sequence of characters.
- Huffman codes are a widely used and very effective technique for compressing data
  - Savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.
- Huffman's greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.
- Now let us see an example to understand the concepts used in Huffman coding

# Example: Huffman Codes

| | 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 |

A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated above.

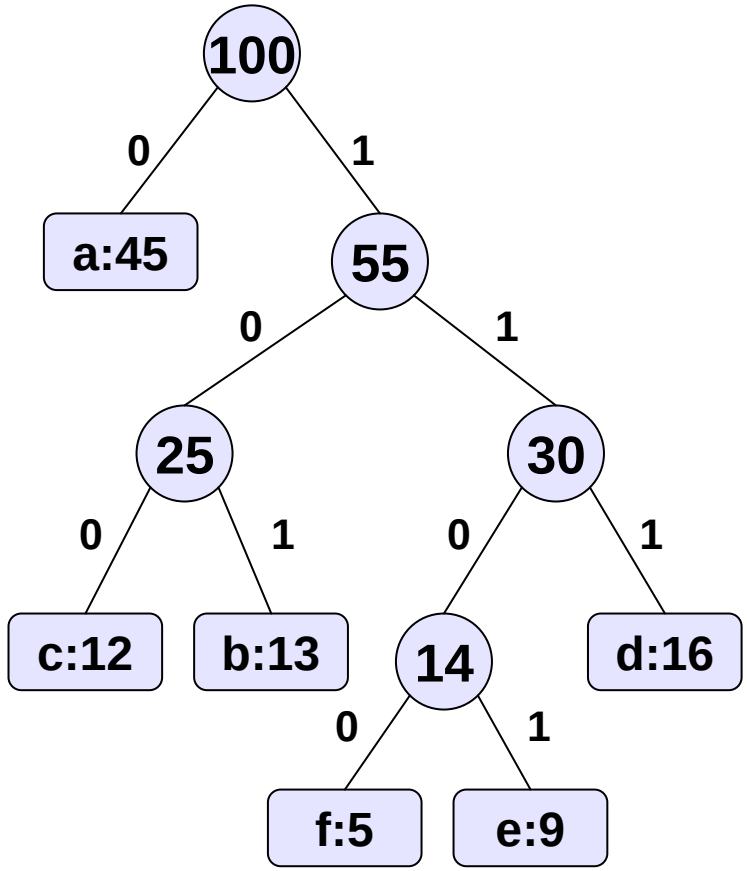
- If each character is assigned a 3-bit **fixed-length codeword**, the file can be encoded in 300,000 bits.
- Using the **variable-length code**

  $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000$ bits

  which shows a savings of approximately 25%.

# Binary Tree: Variable Length Codeword

| | Frequency (in thousands) | Variable-length codeword |
|---|---|---|
| a | 45 | 0 |
| b | 13 | 101 |
| c | 12 | 100 |
| d | 16 | 111 |
| e | 9 | 1101 |
| f | 5 | 1100 |




**The tree corresponding to the variable-length code is shown for the data in table.**

# Cost of Tree Corresponding to Prefix Code

- Given a tree $T$ corresponding to a prefix code. For each character $c$ in the alphabet $C$,
  - let $f(c)$ denote the frequency of $c$ in the file and
  - let $d_T(c)$ denote the depth of $c$'s leaf in the tree.
  - $d_T(c)$ is also the length of the codeword for character $c$.
  - The number of bits required to encode a file is

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

  - which we define as the ***cost*** of the tree $T$.

# Algorithm: Constructing a Huffman Codes

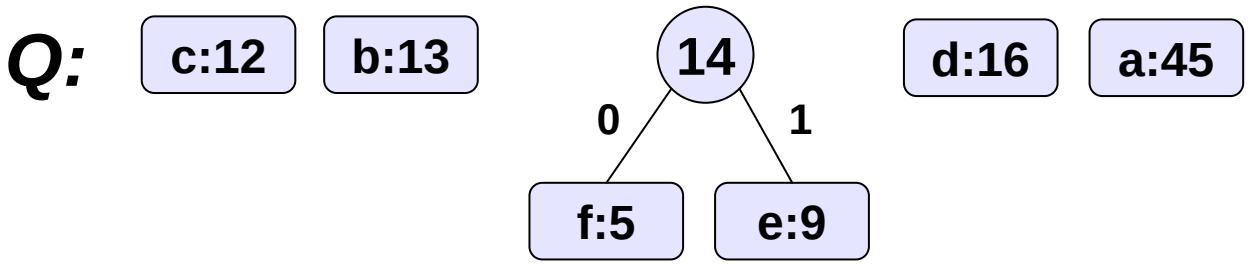
```
Huffman (C)
1   n ← |C|
2   Q ← C
3   for i ← 1 to n - 1
4       do allocate a new node z
5          left[z] ← x ← Extract-Min (Q)
6          right[z] ← y ← Extract-Min (Q)
7          f [z] ← f [x] + f [y]
8          Insert (Q, z)
9   return Extract-Min(Q)  ▷ Return root of the tree.
```

# Example: Constructing a Huffman Codes

***Q:*** **f:5** **e:9** **c:12** **b:13** **d:16** **a:45**

The initial set of *n* = 6 nodes, one for each letter.
Number of iterations of loop are 1 to n-1 (6-1 = 5)

# Constructing a Huffman Codes




for i ← 1

Allocate a new node z

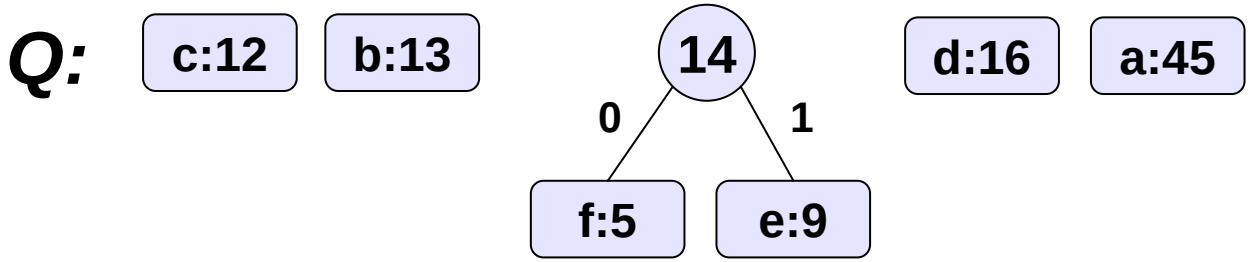
*left*[z] ← *x* ← Extract-Min (*Q*) = f:5

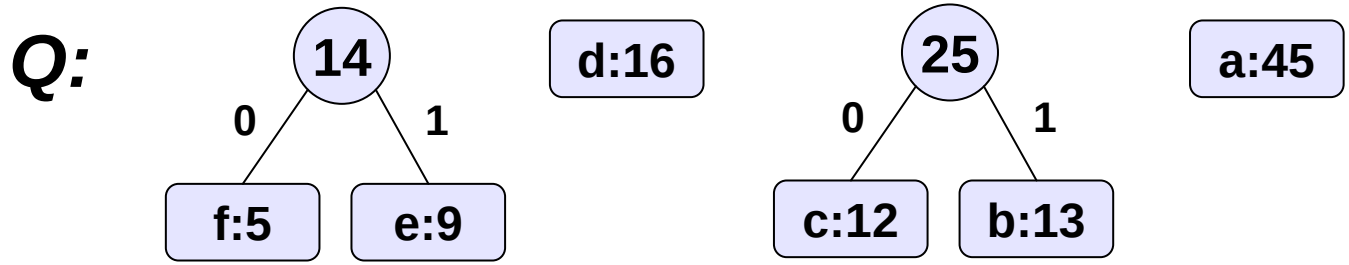
*right*[z] ← *y* ← Extract-Min (*Q*) = e:9

*f* [z] ← *f* [*x*] + *f* [*y*] (5 + 9 = 14)

Insert (*Q*, z)

# Constructing a Huffman Codes




for i ← 2

Allocate a new node z

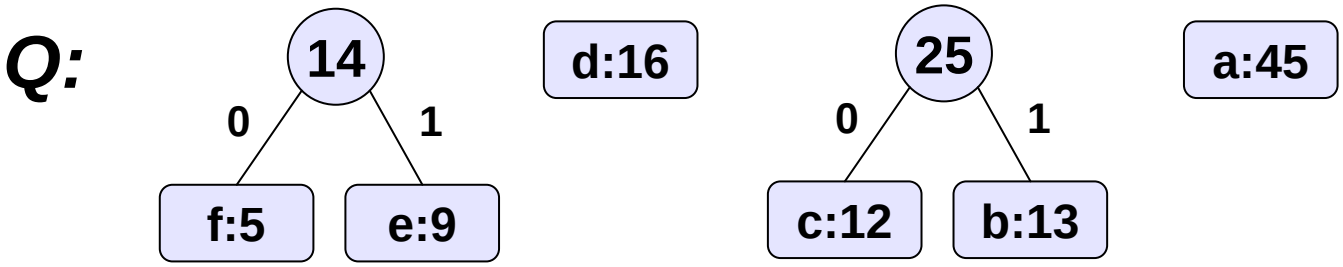
*left*[z] ← *x* ← Extract-Min (*Q*) = c:12

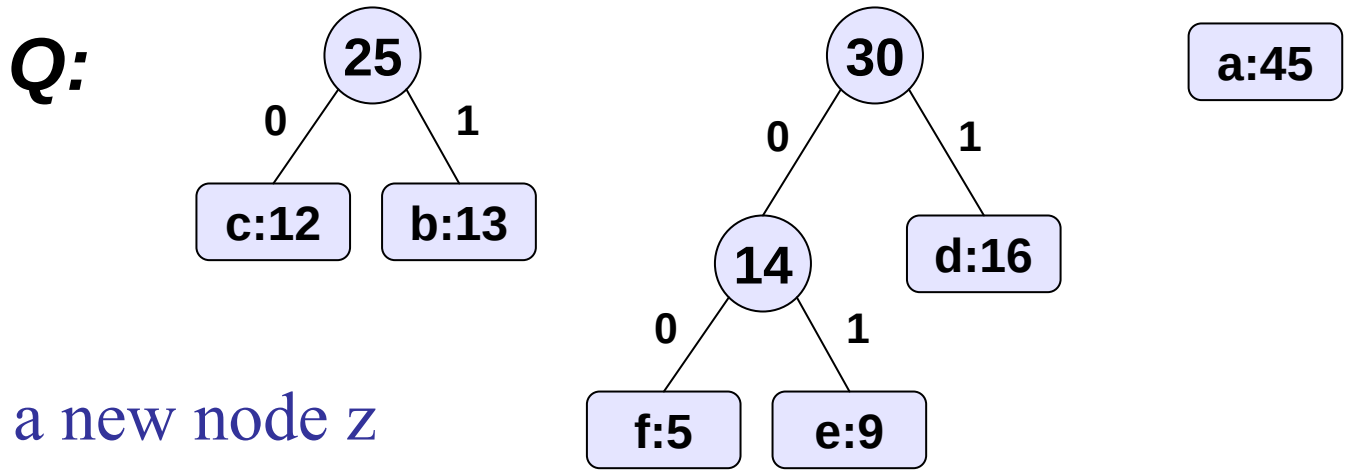
*right*[z] ← *y* ← Extract-Min (*Q*) = b:13

*f* [z] ← *f* [*x*] + *f* [*y*] (12 + 13 = 25)

Insert (*Q*, z)

# Constructing a Huffman Codes




for i ← 3

Allocate a new node z

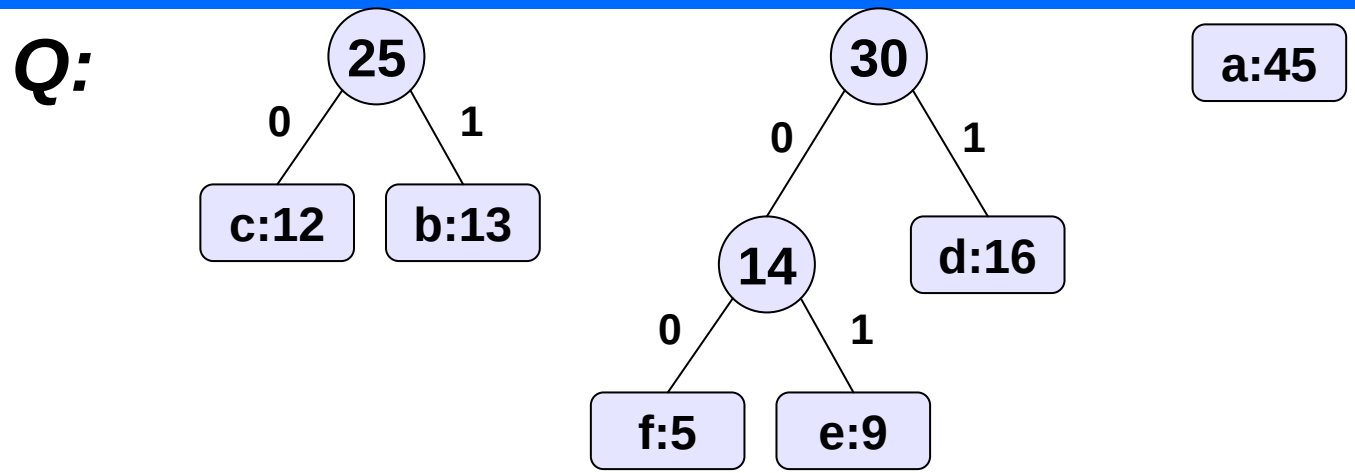
*left*[z] ← *x* ← Extract-Min (*Q*) = z:14

*right*[z] ← *y* ← Extract-Min (*Q*) = d:16

*f* [z] ← *f* [*x*] + *f* [*y*] (14 + 16 = 30)

Insert (*Q*, *z*)

# Constructing a Huffman Codes




for i ← 4

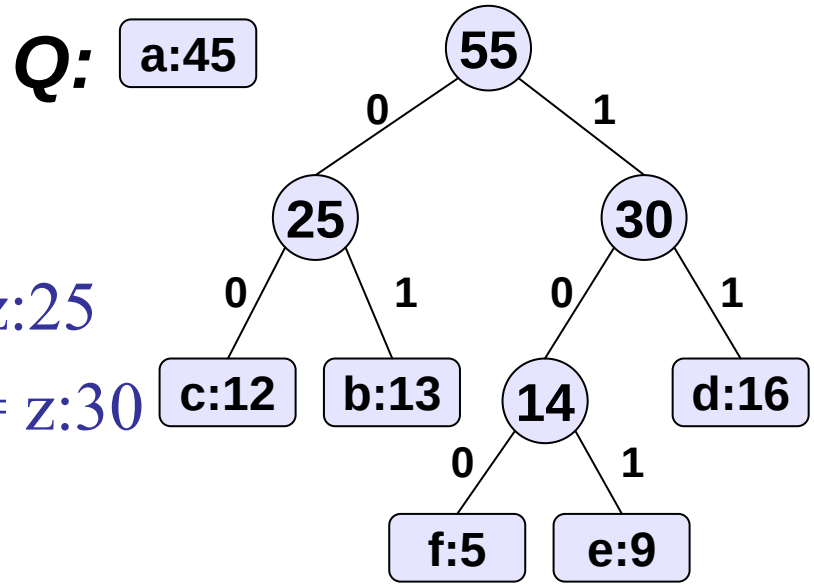
Allocate a new node z

*left*[z] ← *x* ← Extract-Min (*Q*) = z:25

*right*[z] ← *y* ← Extract-Min (*Q*) = z:30

*f* [z] ← *f* [*x*] + *f* [*y*] (25 + 30 = 55)

Insert (*Q*, *z*)



# Constructing a Huffman Codes

for i ← 5

Allocate a new node z

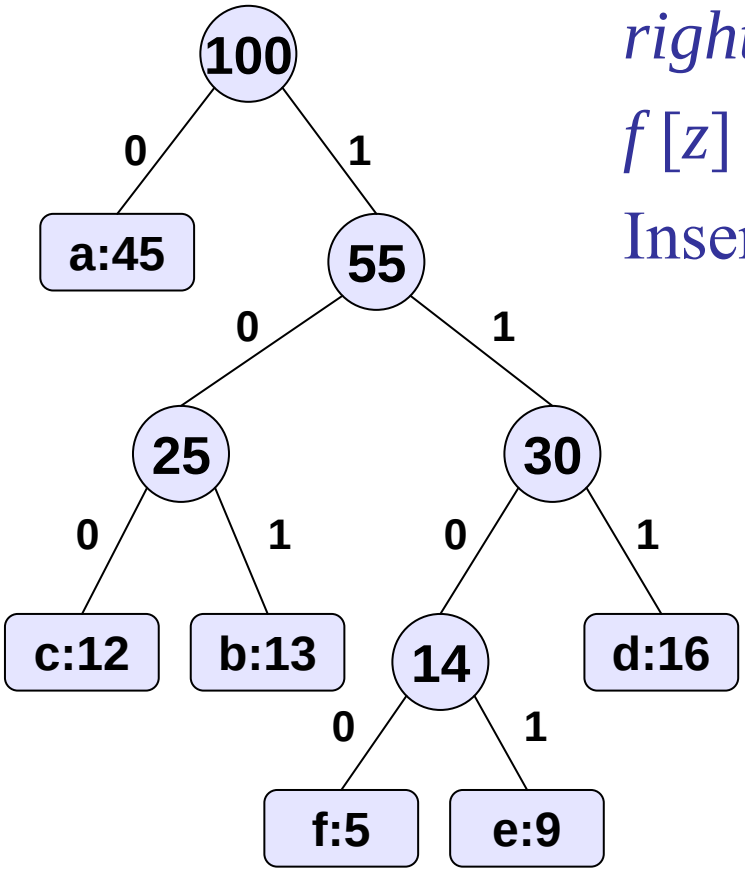
*left*[z] ← *x* ← Extract-Min (*Q*) = a:45

*right*[z] ← *y* ← Extract-Min (*Q*) = z:55

*f* [z] ← *f* [*x*] + *f* [*y*] (45 + 55 = 100)

Insert (*Q*, *z*)

***Q:***

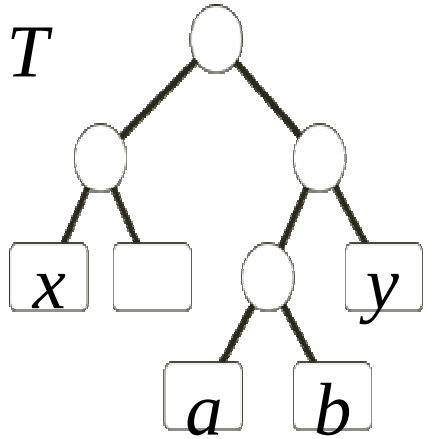


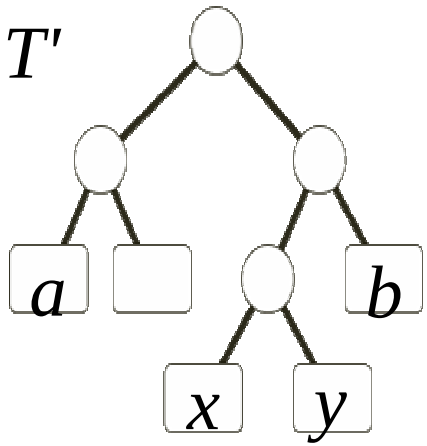
# Lemma 1: Greedy Choice

*There exists an optimal prefix code such that the two characters with smallest frequency are siblings and have maximal depth in T.*

Proof:

- Let $x$ and $y$ be two such characters, and let $T$ be a tree representing an optimal prefix code.
- Let $a$ and $b$ be two sibling leaves of maximal depth in $T$, and assume with out loss of generality that $f(x) \leq f(y)$ and $f(a) \leq f(b)$.
- This implies that $f(x) \leq f(a)$ and $f(y) \leq f(b)$.
- Let $T'$ be the tree obtained by exchanging $a$ and $x$ and $b$ and $y$.



# Contd..

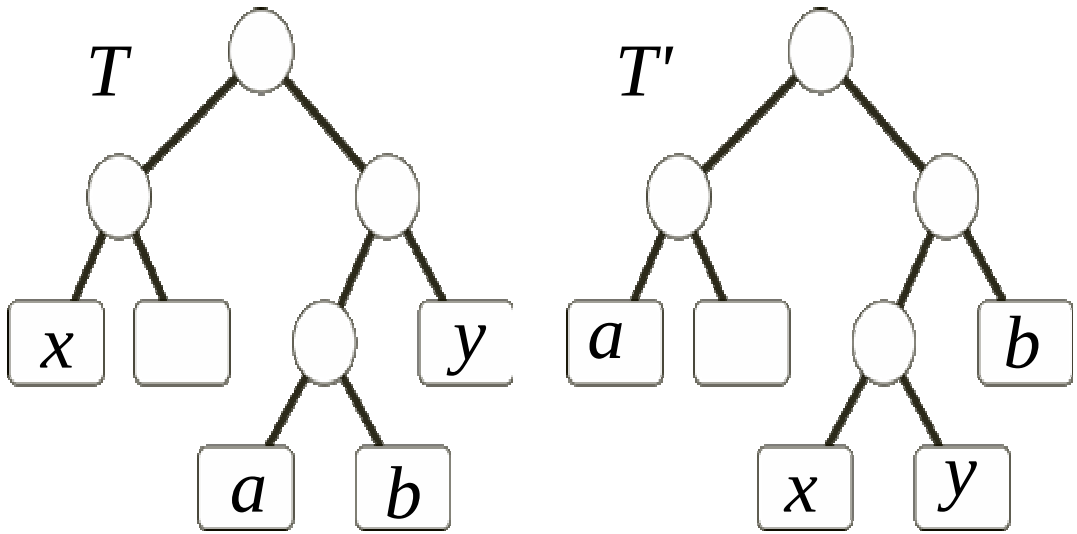
The cost difference between trees $T$ and $T'$ is

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x) d_T(x) + f(a) d_T(a) - f(x) d_{T'}(x) - f(a) d_{T'}(a) + \\
&\quad f(y) d_T(y) + f(b) d_T(b) - f(y) d_{T'}(y) - f(b) d_{T'}(b) \\
&= (f(a) - f(x))(d_T(a) - d_T(x)) + (f(b) - f(y))(d_T(b) - d_T(y)) \\
&\geq 0
\end{aligned}
$$

Hence B(T') ≤ B(T)

Since B(T) ≤ B(T')

Hence B(T) = B(T')



# Conclusion

- Huffman Algorithm is analyzed
- Design of algorithm is discussed
- Computational time is given
- Applications can be observed in various domains e.g. data compression, defining unique codes, etc. generalized algorithm can used for other optimization problems as well