# COMP S265F Unit 2: Greedy Heuristics: Huffman Codes

Dr. Keith Lee

*School of Science and Technology*

*The Open University of Hong Kong*

# Overview

- **Greedy Heuristics**
- **Encoding problem**
  - binary character code
  - average character length
  - Find an encoding to minimize avg. character len.
- ***Code tree:*** a binary tree
- Depth of a leaf (character) = Its character code length
- Observation 1: Swapping argument
- Observation 2: Any leaf must have a sibling
- **Huffman code algorithm**
- ***Time complexity:*** Simple implementation / Using *min-Heap*
- ***Proof of correctness:*** Tree transformation

# Greedy heuristics

- A basic algorithm design technique

- **How to be greedy?**
  - At every step, make the best move you can make.
  - Keep going until you're done.

- **Advantages**
  - Don't need to pay much effort at each step.
  - Usually find a solution very quickly.
  - The solutions are usually not bad.

# Encoding problem: Definitions

- A binary character code (or simply code) **C** for a set Σ of characters assigns, for each character in Σ, a unique binary string.

**Example:**

Σ = {a, b, c, d}.

**C**: a $\rightarrow$ 1, b $\rightarrow$ 01, c $\rightarrow$ 001, d $\rightarrow$ 0001

- Encoding a sequence **S** of characters by **C**:
  - aabaacda $\rightarrow$ 11011100100011

# Encoding problem: Definitions (cont')

- Suppose we are given a sequence **S** of characters in $\Sigma$.
- For each character $\mathbf{x} \in \Sigma$, let
  - $f(\mathbf{x})$ be the total number of occurrence of **x** in **S**, and
  - $len(\mathbf{x})$ be the number of bits needed to encode **x** according to C.

- Define average character length (of C on S) of S to be

$$L_C(S) = \frac{\sum_{\mathbf{x} \in \Sigma} f(\mathbf{x}) \times len(\mathbf{x})}{\sum_{\mathbf{x} \in \Sigma} f(\mathbf{x})}$$

The total number of bits for encoding **S**.

$|\mathbf{S}|$

- In other words, average character length of **S** is the average number of bits used for encoding a character in **S**.

# Encoding problem: Definitions (cont')

Recall that

$\Sigma$ = {a, b, c, d}.

**C**: a $\rightarrow$ 1, b $\rightarrow$ 01, c $\rightarrow$ 001, d $\rightarrow$ 0001

- Define average character length (of C on S) of S to be

$$L_C(S) = \frac{\sum_{x \in \Sigma} f(x) \times len(x)}{\sum_{x \in \Sigma} f(x)}$$

The total number of bits for encoding **S**.

$|S|$

- **Example:** aabaacda $\rightarrow$ 11011100100011

$L_C(S) = (5\times1 + 1\times2 + 1\times3 + 1\times4) / 8 = 14/8 = 1.75$

(i.e., on average, **C** uses 1.75 bits to encode a character in **S**).

# Encoding problem

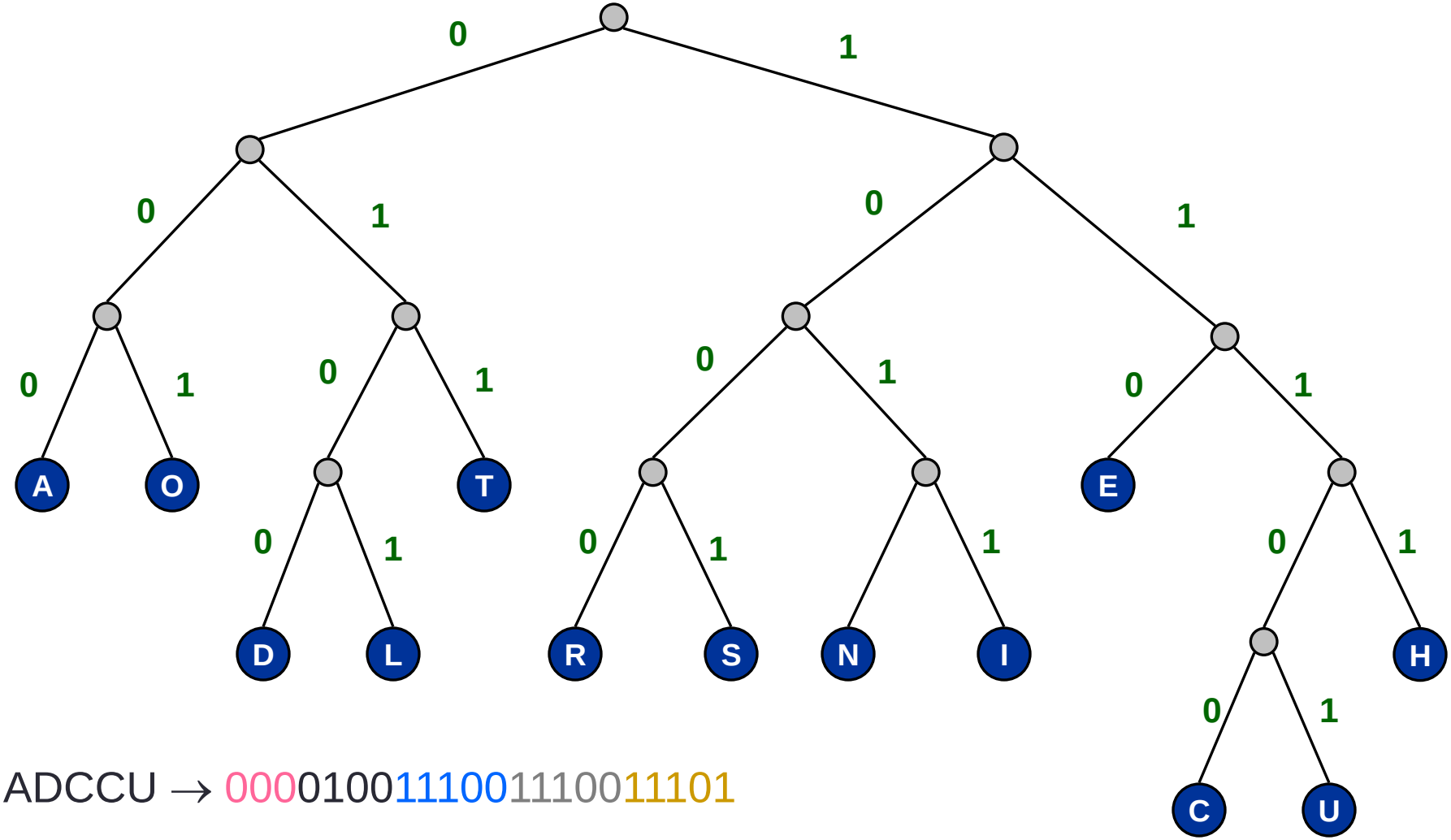
- Find an encoding C for S with the minimum $L_C(S)$.

- **Huffman code** is an encoding C with the minimum **$L_C(S)$.**
- Note that different string **S** has different minimum **$L_C(S)$**.

- The algorithm constructing Huffman code is based on a **greedy heuristics**.

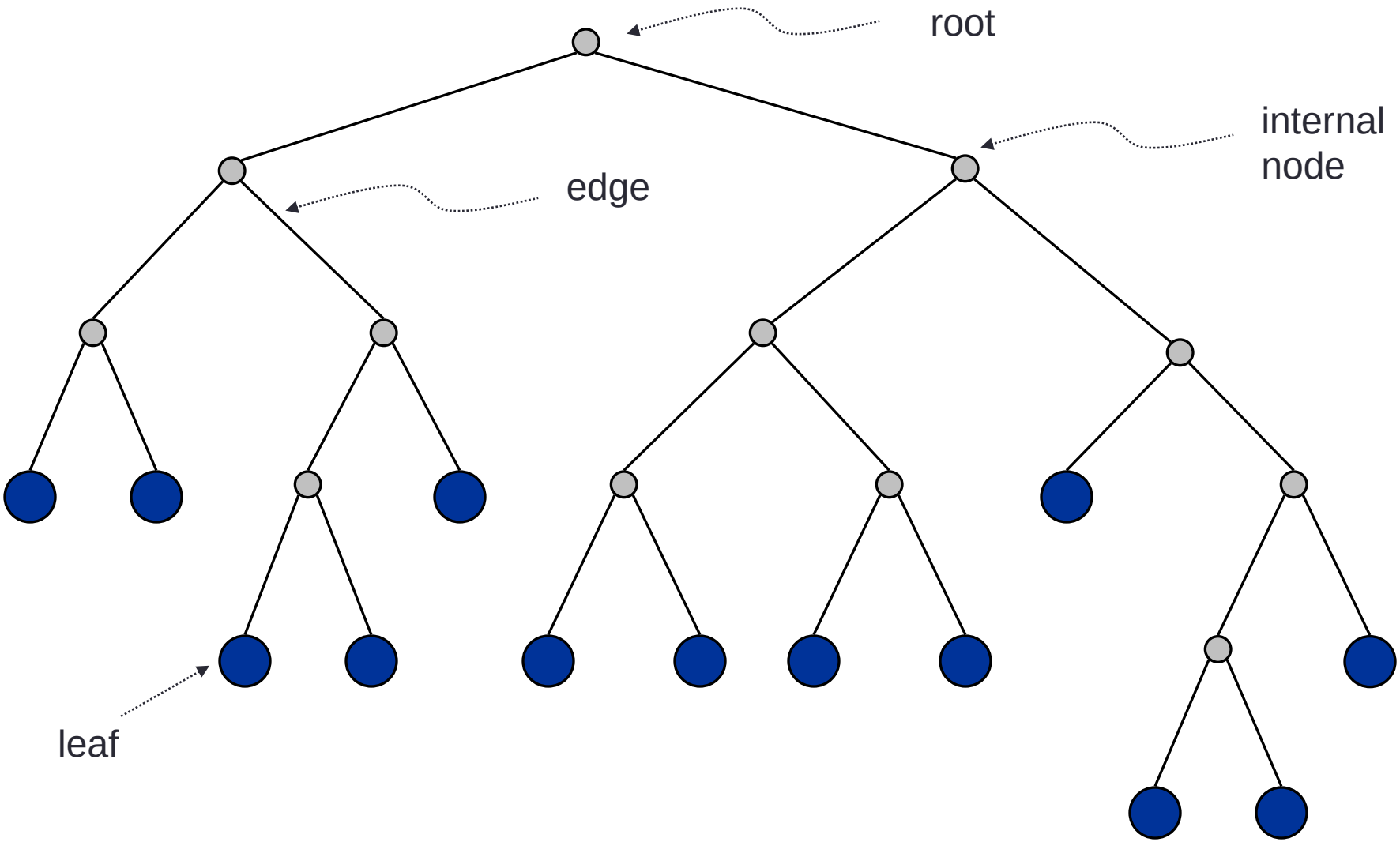
| Char | Freq | Fixed | Huffman |
|---|---|---|---|
| E | 125 | 0000 | 110 |
| T | 93 | 0001 | 000 |
| A | 80 | 0010 | 001 |
| O | 76 | 0011 | 011 |
| I | 73 | 0100 | 1011 |
| N | 71 | 0101 | 1010 |
| S | 65 | 0110 | 1001 |
| R | 61 | 0111 | 1000 |
| H | 55 | 1000 | 1111 |
| L | 41 | 1001 | 0101 |
| D | 40 | 1010 | 0100 |
| C | 31 | 1011 | 11100 |
| U | 27 | 1100 | 11101 |
| Total | 838 | **4.00** | **3.62** |

# Code tree: A graphical representation of code

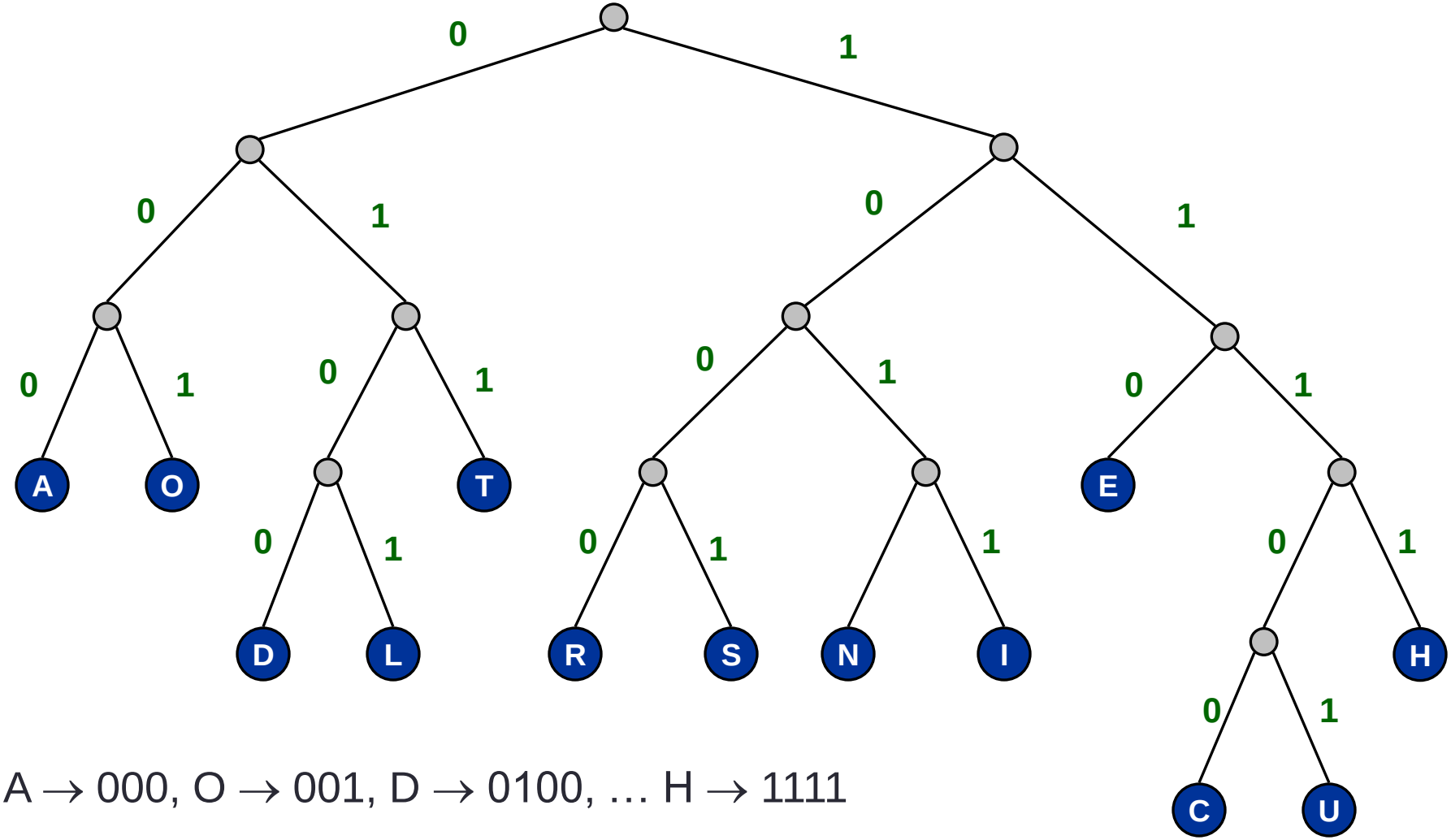



ADCCU → 000010011110011100011101

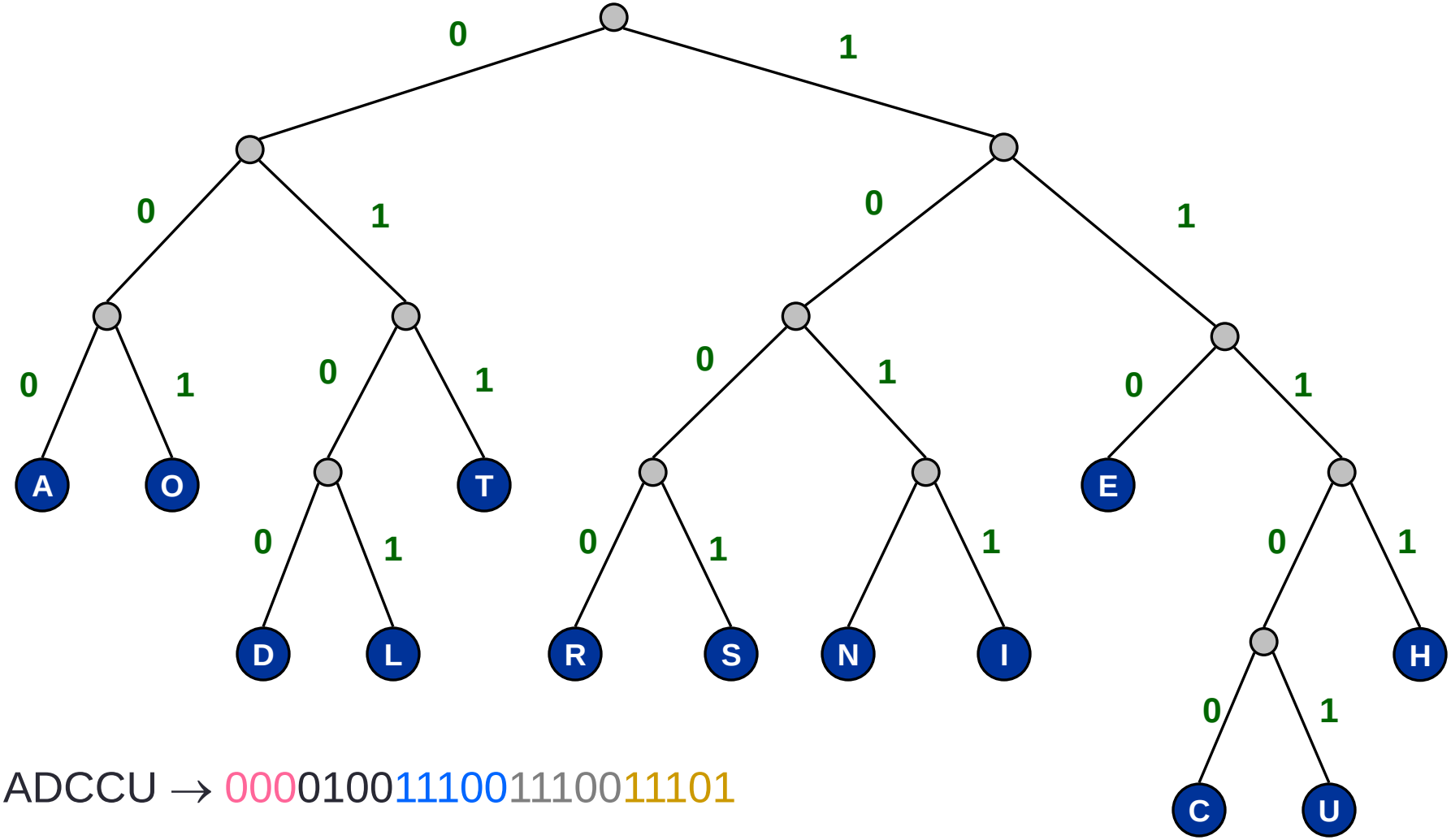
# Binary tree



# Code tree




A → 000, O → 001, D → 0100, … H → 1111

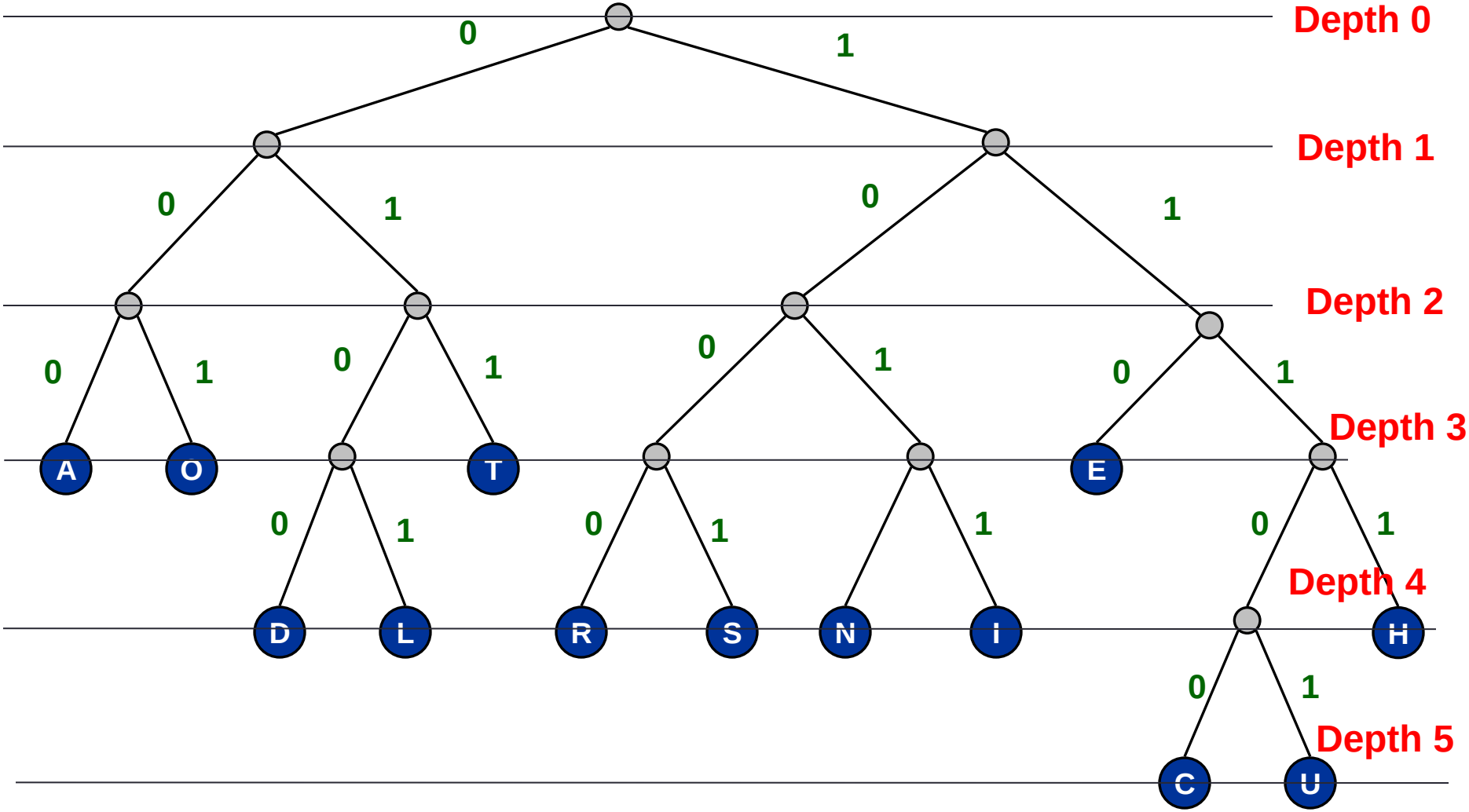


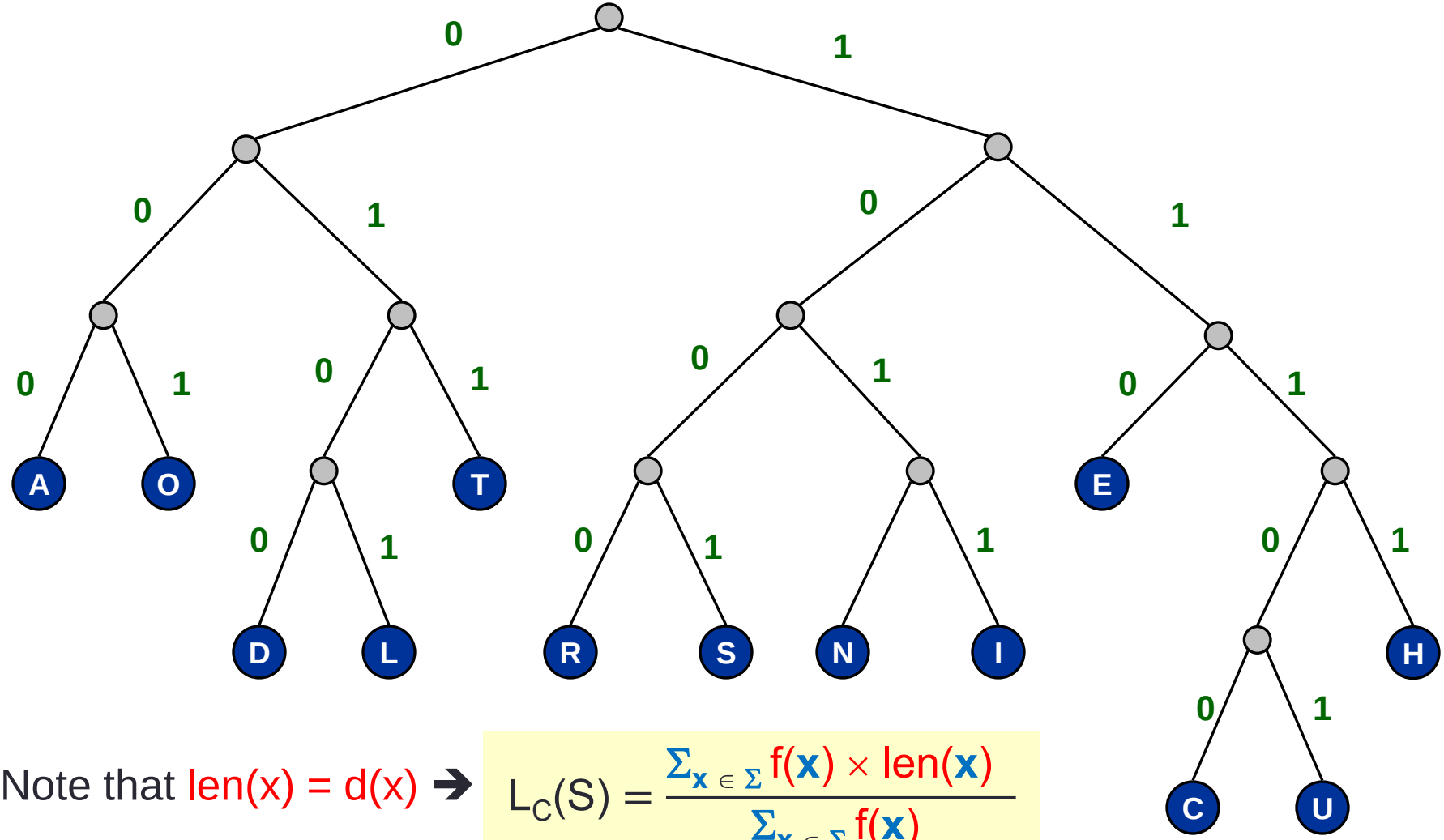

ADCCU → 000010011110011100011101

# Fact

- Given a code tree **T** for code **C**, it is a simple matter to compute the average character length of sequence **S** encoded by **C**.
- First, we need the notion of the depth of nodes in **T**.

# Depth of a node x: d(x)






Note that len(x) = d(x) ➔

$$L_C(S) = \frac{\sum_{x \in \Sigma} f(x) \times len(x)}{\sum_{x \in \Sigma} f(x)}$$

- To find a code that gives minimum average character length for input text **S**, we can construct a code tree that minimizes

$$L_C(S) = \frac{\sum_{x \in \Sigma} f(x) \times len(x)}{\sum_{x \in \Sigma} f(x)}$$

fixed; equal to |S|.

- Or equivalently, minimizes

$$\sum_{x \in \Sigma} f(x) \times len(x)$$

This is given by the input.

This is determined by the tree we construct.

# Some observations

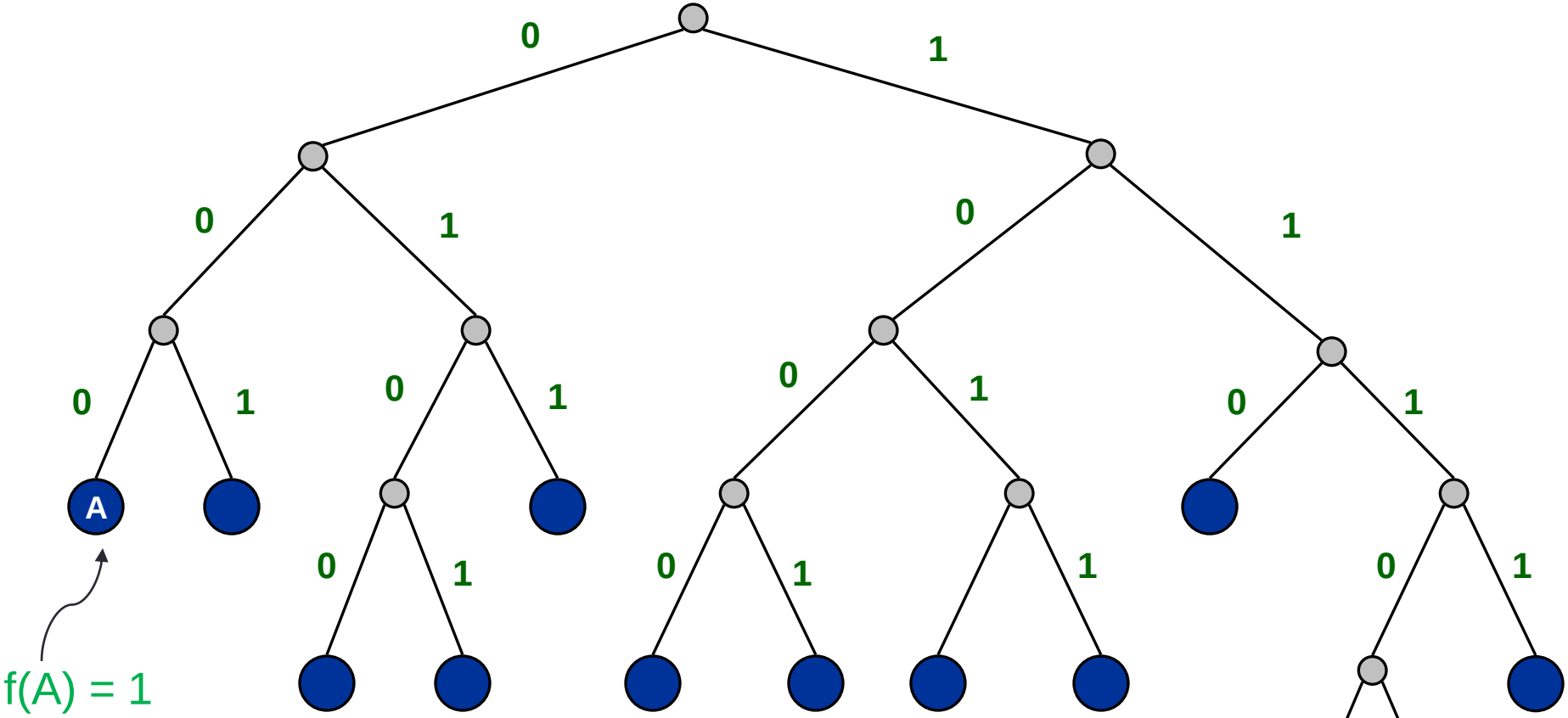
- Consider an optimal code tree **T** for **S** (i.e., one that gives minimum average character length for **S**).

  Suppose that **T** has **n** leaves $s_1, s_2, \ldots, s_n$ where
  $f(s_1) \geq f(s_2) \geq \ldots \geq f(s_n)$.
  Then,
  $d(s_1) \leq d(s_2) \leq \ldots \leq d(s_n)$.

- Why?

# Observation 1

- This code tree has two leaves X and A where ***(1)*** f(X) > f(A), but ***(2)*** d(X) > d(A).
- It is not optimal. Why?




f(A) = 1

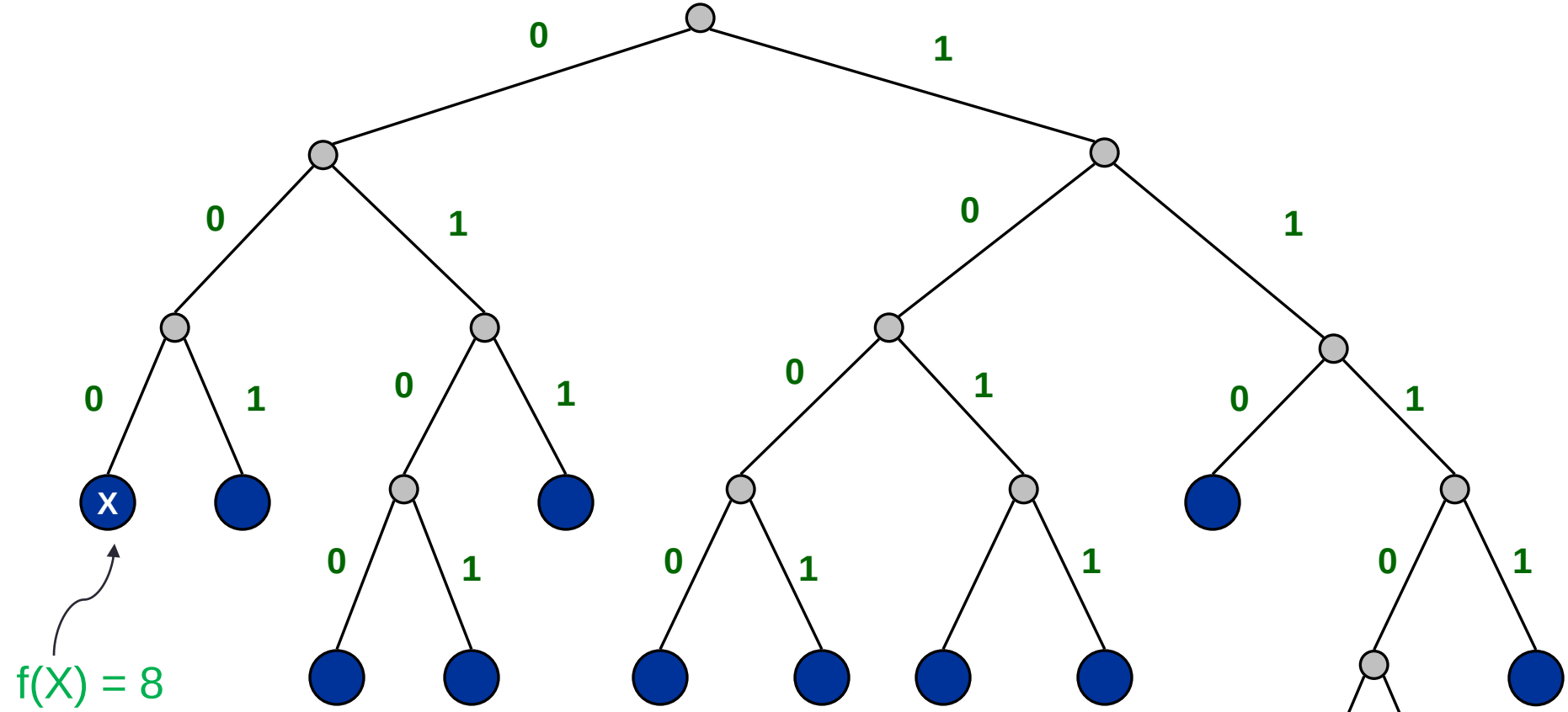
⇒ f(A)d(A)=3

f(X) = 8

⇒ f(X)d(X)=40

# Observation 1

- Because swapping A & X gives us a better tree



# Observation 1: Formal proof

- Suppose, for the sake of contradiction, that in an optimal tree, there exists X and A such that **f(X) > f(A)** but **d(X) > d(A)**.
- Before the swap: $\sum_{x \notin \{A,X\}} f(x)d(x) + f(A)d(A) + f(X)d(X)$
- After the swap: $\sum_{x \notin \{A,X\}} f(x)d(x) + f(A)d(X) + f(X)d(A)$
- The difference between the new tree and the old tree is

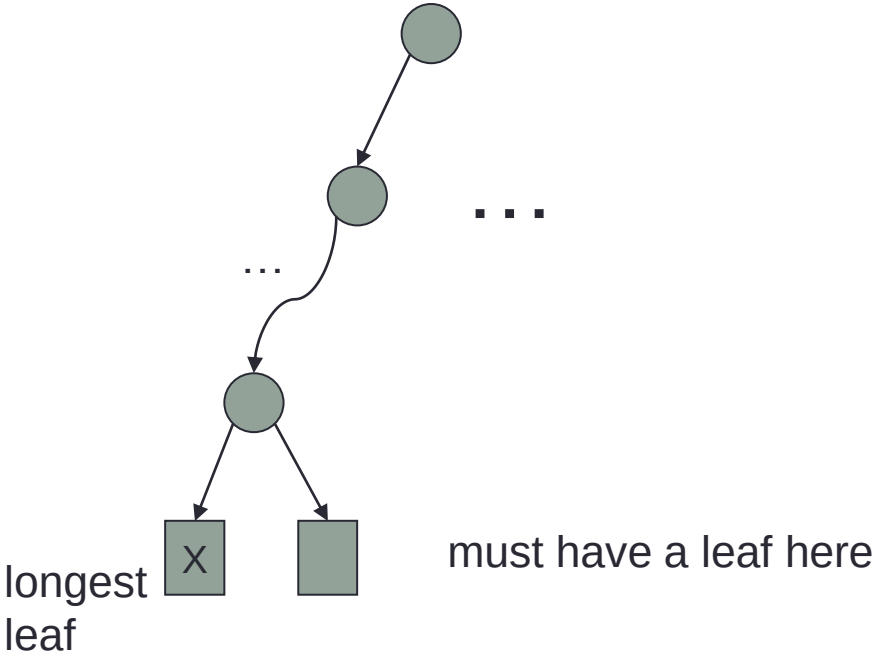
$$\begin{aligned} & f(A)d(X) + f(X)d(A) - f(A)d(A) - f(X)d(X) \\ = \; & f(A)\big(d(X) - d(A)\big) + f(X)\big(d(A) - d(X)\big) \\ = \; & \underbrace{\big(f(A) - f(X)\big)}_{<0}\underbrace{\big(d(X) - d(A)\big)}_{>0} \end{aligned}$$

$\Rightarrow$ <0

- Thus, the new tree has **smaller** average character length, which contradicts that the old tree is an optimal tree.

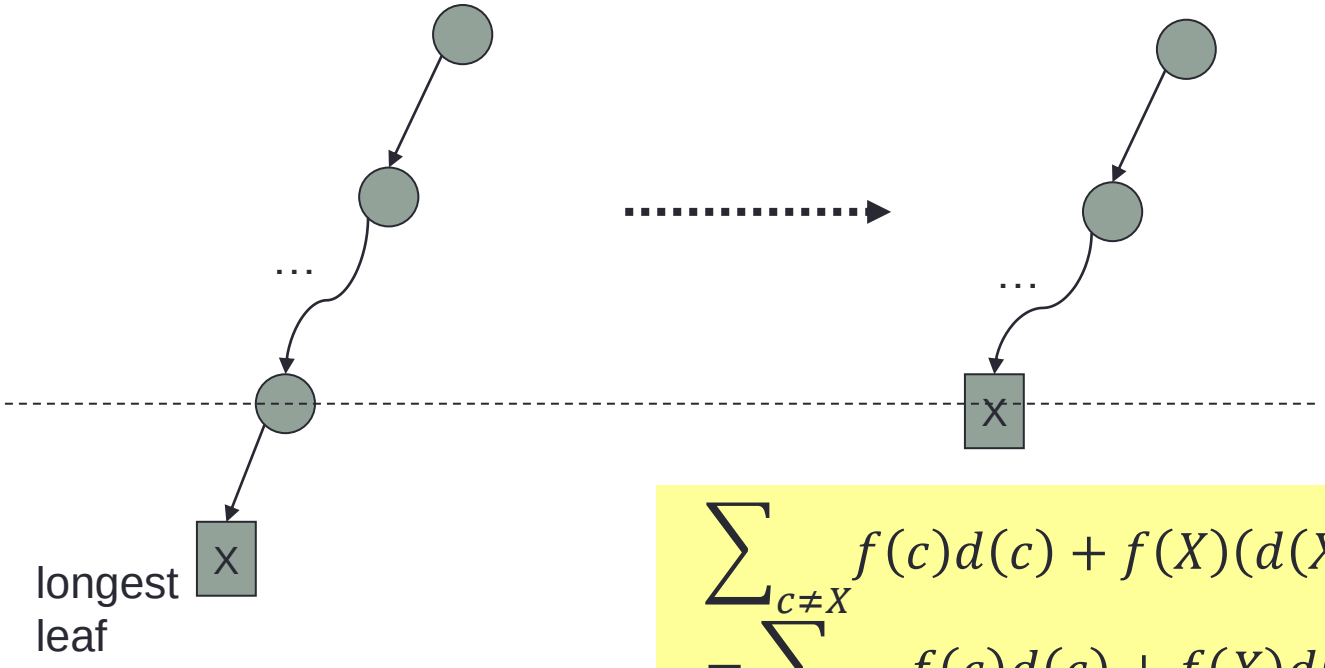
# Observation 2

- We also observe that in **T**, we can find two longest leaves joining together (i.e., have the same parent).



# Observation 2 (cont')

- If there is no leaf, then




$$\sum_{c \neq X} f(c)d(c) + f(X)d(X)$$

$$\sum_{c \neq X} f(c)d(c) + f(X)(d(X) - 1)$$
$$= \sum_{c \neq X} f(c)d(c) + f(X)d(X) - f(X)$$

This is smaller!

# Conclusion

In an optimal code tree, for every character x,

- the larger the depth d(x), the smaller the frequency f(x); and
- the leave **u** with the largest depth must have a sibling **v**.

What can we say about **u**, v?

- **u**, **v** have the largest depth

$\Rightarrow$ **u**, **v** have the smallest frequency.

# Conclusion

In an optimal code tree, for every character x,

- the larger the depth d(x), the smaller the frequency f(x); and
- the leave **u** with the largest depth must have a sibling **v**.

What can we say about **u**, v?

- **u**, **v** have the largest depth (with d(u) = d(v))

$\Rightarrow$ **u**, **v** have the smallest frequency.

$$\sum_{x\in\Sigma} f(x)d(x) = \sum_{x\notin\{u,v\}} f(x)d(x) + f(u)d(u) + f(v)d(v)$$

$$= \sum_{x\notin\{u,v\}} f(x)d(x) + \underbrace{(f(u)+f(v))}_{\text{as if it is a leaf with frequency f(u)+f(v)}}\underbrace{(d(u)-1)}_{\text{depth of the new leaf}} + (f(u)+f(v))$$

# Conclusion

In an optimal code tree, for every character x,

- the larger the depth d(x), the smaller the frequency f(x); and
- the leave **u** with the largest depth must have a sibling **v**.

What can we say about **u**, v?

- **u**, **v** have the largest depth (with d(u) = d(v))

$\Rightarrow$ **u**, **v** have the smallest frequency.

$$\sum_{x\in\Sigma} f(x)d(x) = \sum_{x\notin\{u,v\}} f(x)d(x) + f(u)d(u) + f(v)d(v)$$

$$= \sum_{x\notin\{u,v\}} f(x)d(x) + \underbrace{(f(u)+f(v))}_{\text{fixed}}\underbrace{(d(u)-1)}_{\text{fixed}} + \underbrace{(f(u)+f(v))}_{\text{fixed}}$$

As if it has n-1 leaves, with the old leaves u, v replaced by a new one with frequency f(u)+f(v). This sum is minimized for these n-1 leaves.

# Conclusion

In an optimal code tree, for every character x,

- the larger the depth d(x), the smaller the frequency f(x); and
- the leave **u** with the largest depth must have a sibling **v**.

What can we say about **u**, v?

- **u**, **v** have the largest depth (with d(u) = d(v))
  $\Rightarrow$ **u**, **v** have the smallest frequency.

Idea for constructing the optimal code tree:

> Construct the code tree from bottom to top, adding to the tree those characters with lowest frequency first.

- This idea leads to the Huffman code.

# Huffman code

The algorithm that constructs the Huffman code for a set **C** of characters:

1. Build the code tree in a bottom-up manner.
2. It begins with a set of |**C**| leaves and performs a sequence of |**C**|-1 "merging" operations to create the final tree.
3. At each merging step, the two least-frequent objects are merged together, and the result of this merging is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

**Time complexity:**

- Simple implementation: $O(|C|^2)$ time.
- Using a min-Heap: $O(|C| \log |C|)$ time.

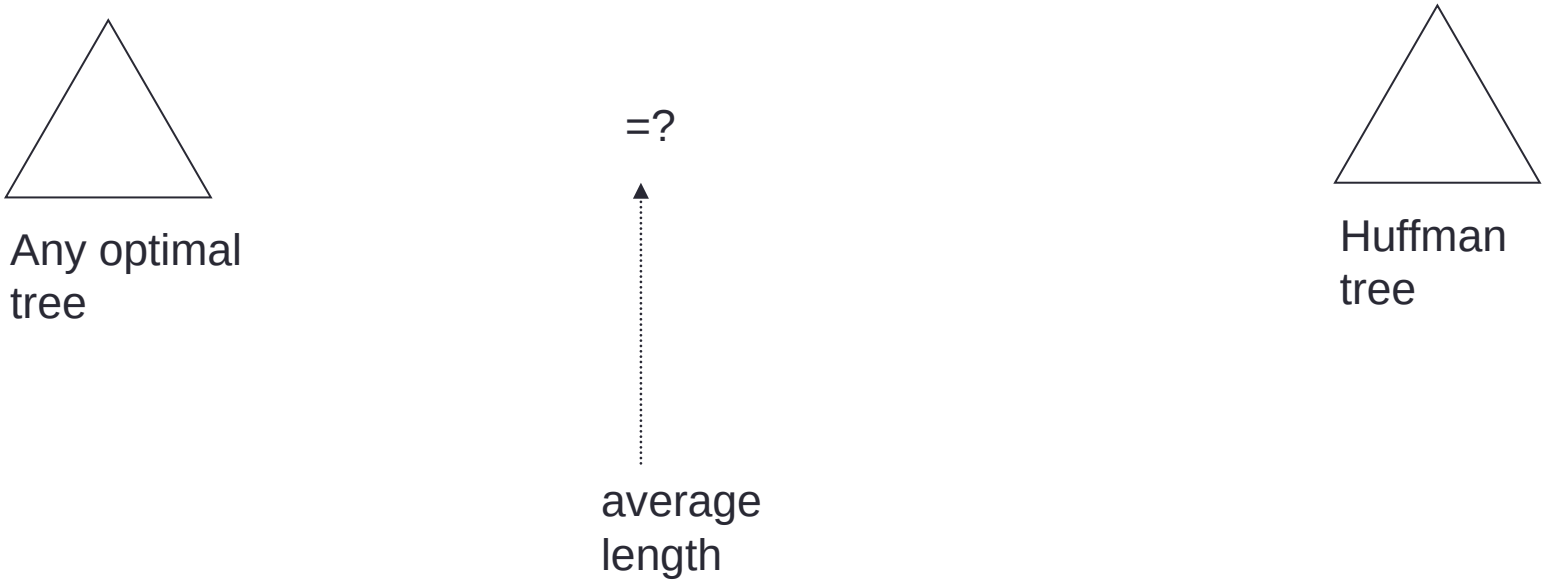
# Min-Heap

- A min-Heap is a data structure that maintains a set S of numbers.

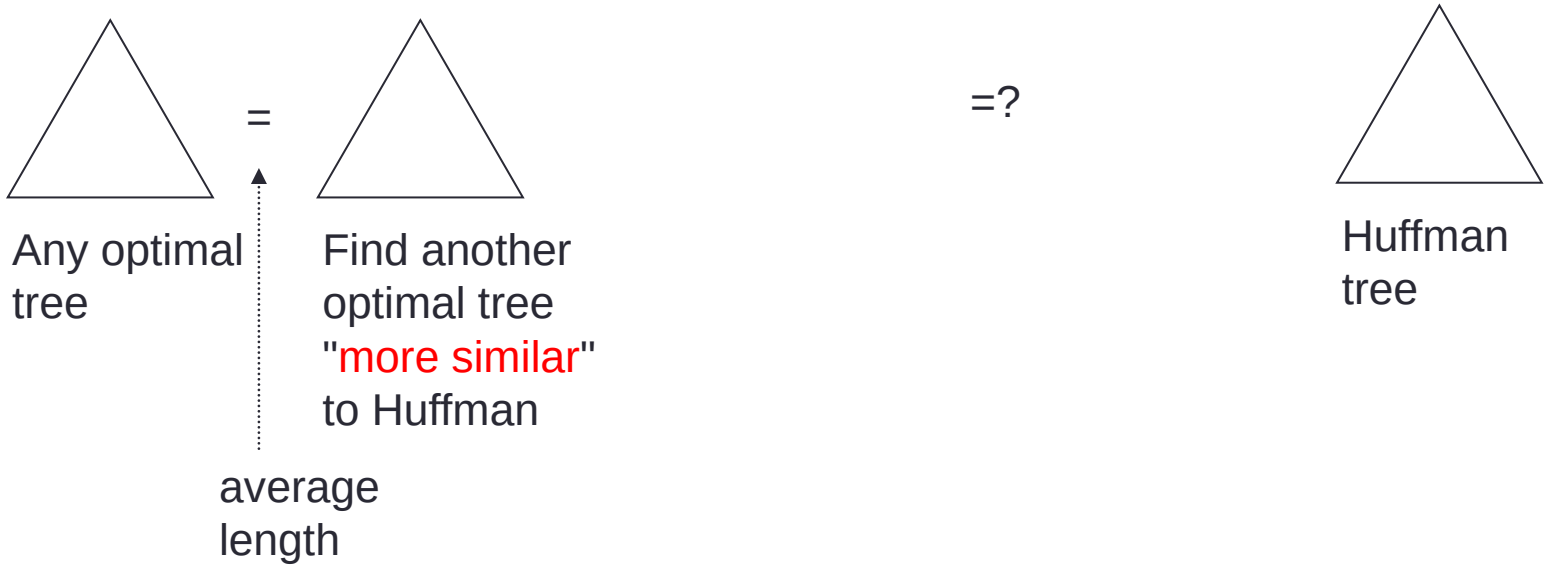
- A min-Heap supports the following operations in **O(log |S|) time**:
  - Find-Min(S): return the minimum number in S;
  - Delete-Min(S): delete the minimum number in S;
  - Insert(x, S): insert a new number to S.

- Furthermore, given any set S of n numbers, we can store a min-Heap of S in **O(n) time**.

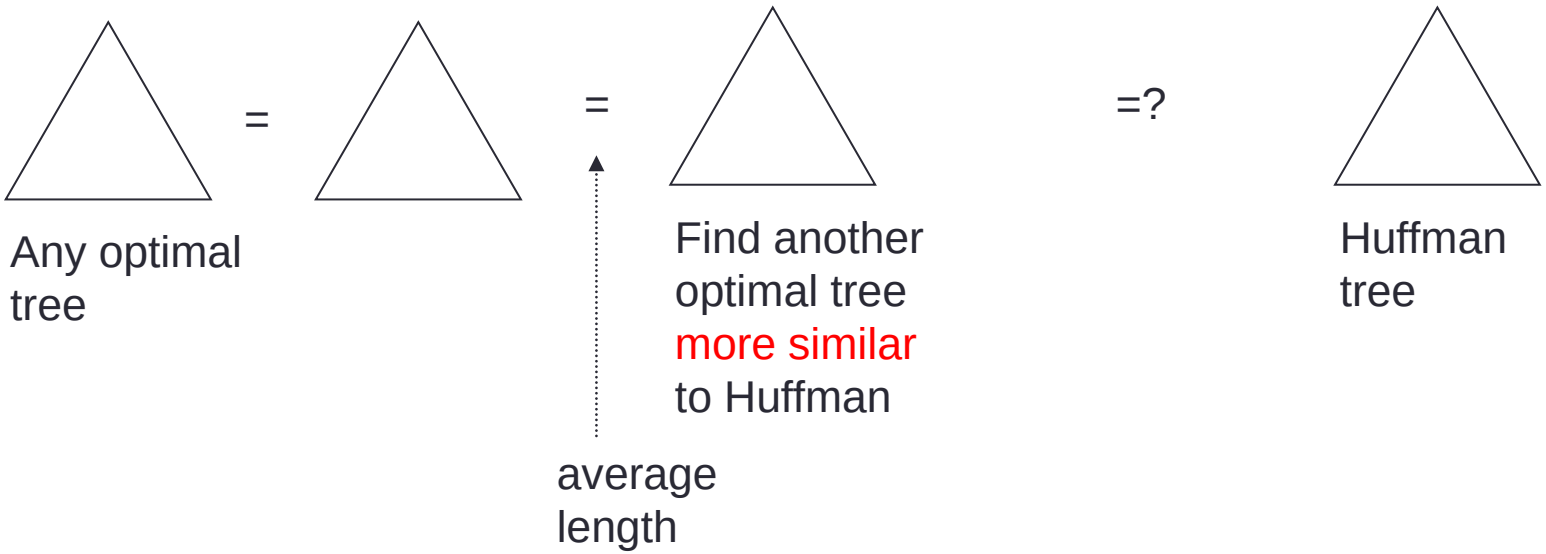
# Proof of correctness (rough ideas)



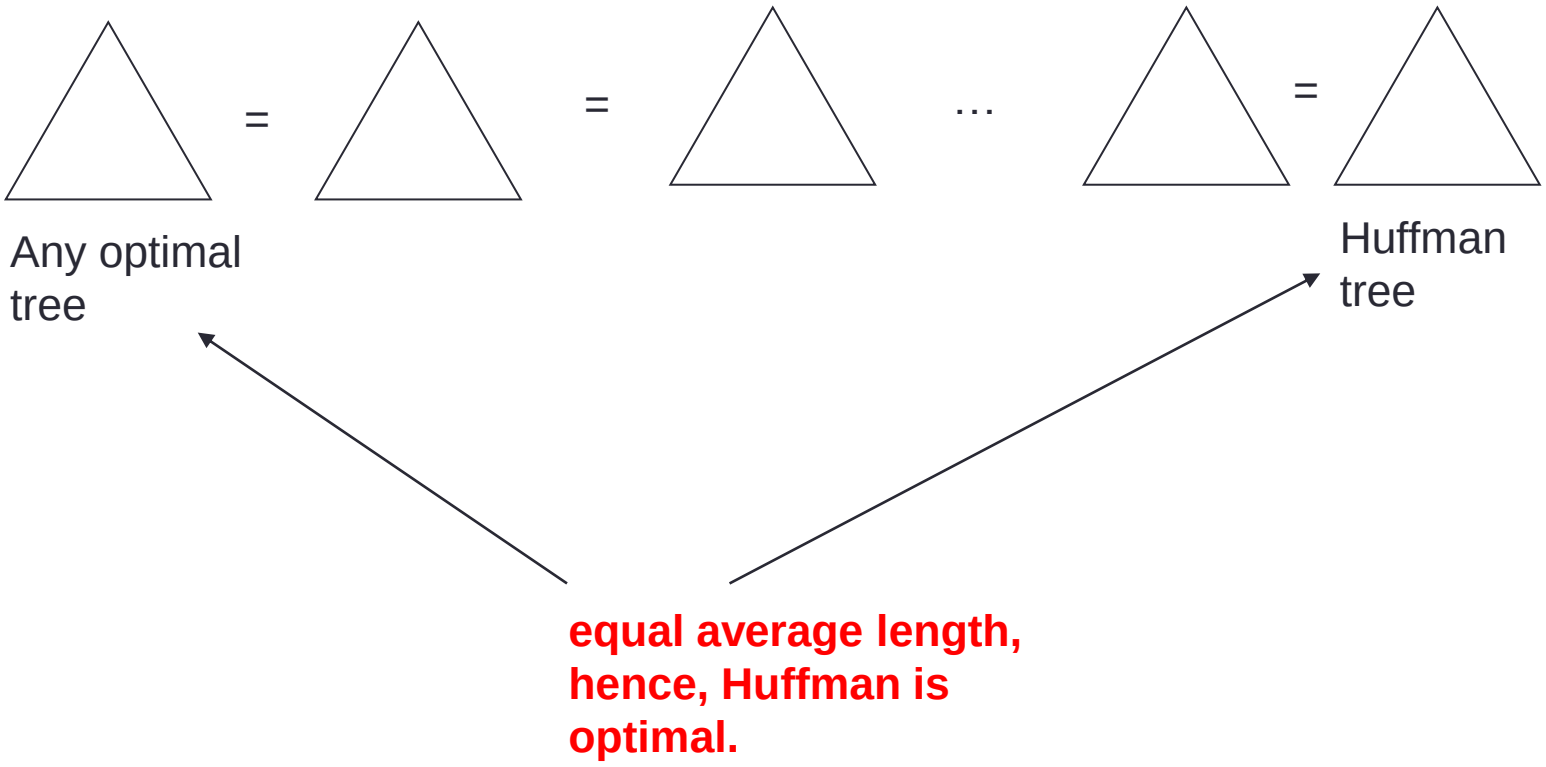
# Proof of correctness (rough ideas)



# Proof of correctness (rough ideas)

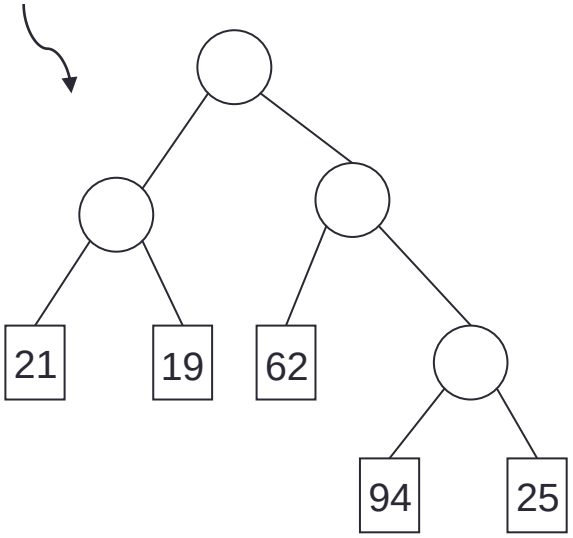


# Proof of correctness (rough ideas)



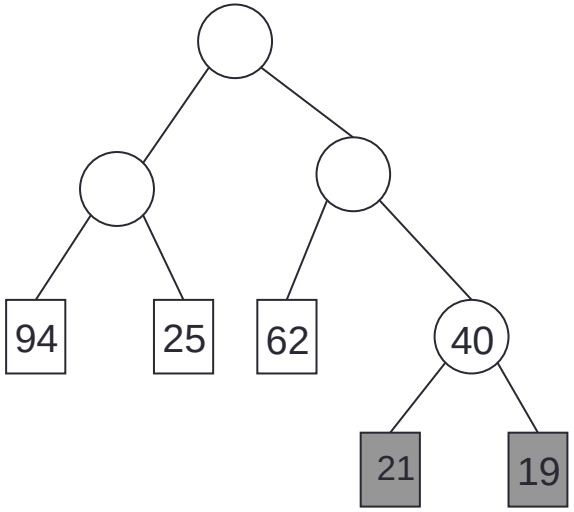
# Basic step for the transformation

Any tree



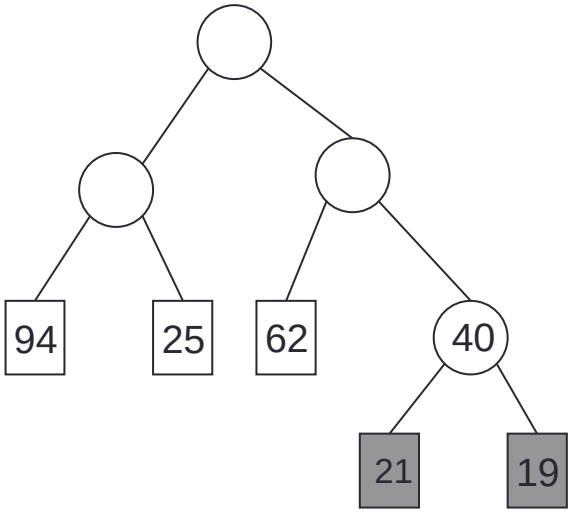

In the Huffman tree, the two leaves with the lowest frequency appear as sibling leaves with maximum depth

The change will not increase average length

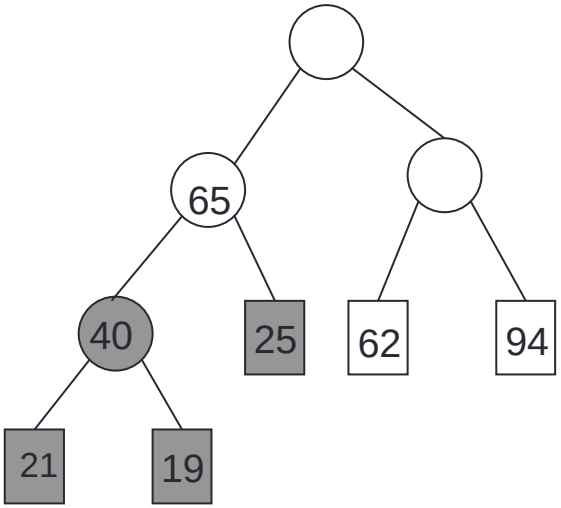



This tree has the same set of “leaves” as the one we get after the first step of the algorithm.

# Basic step for the transformation (cont')

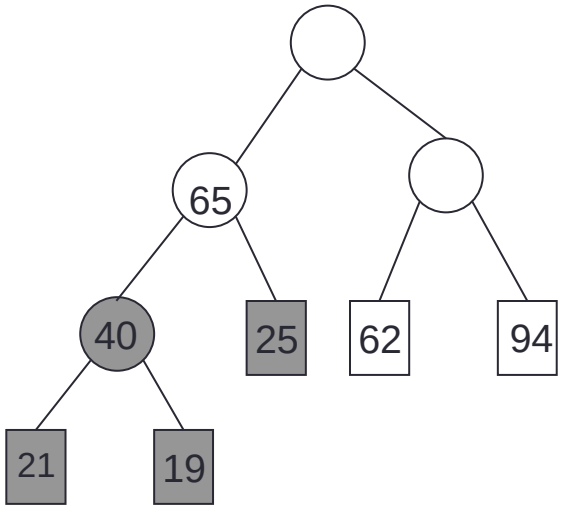


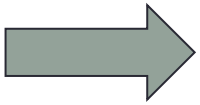

The change will not increase avg. length



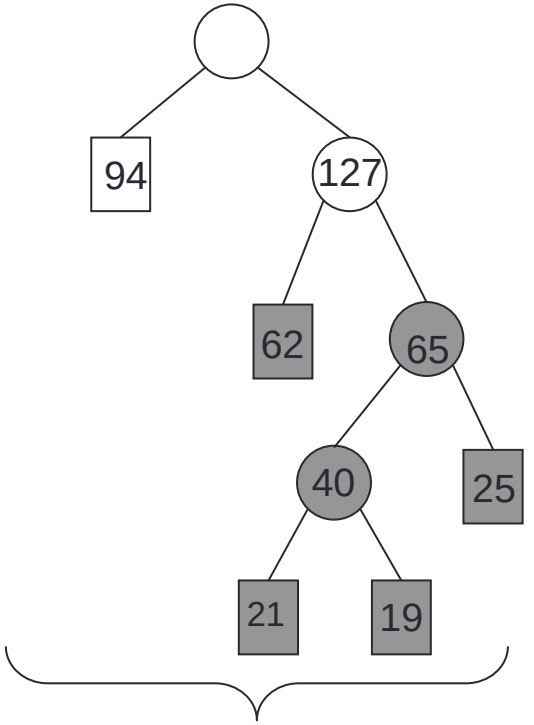

This tree has the same set of "leaves" as the one we get after the second step of the algorithm.

# Basic step for the transformation (cont')



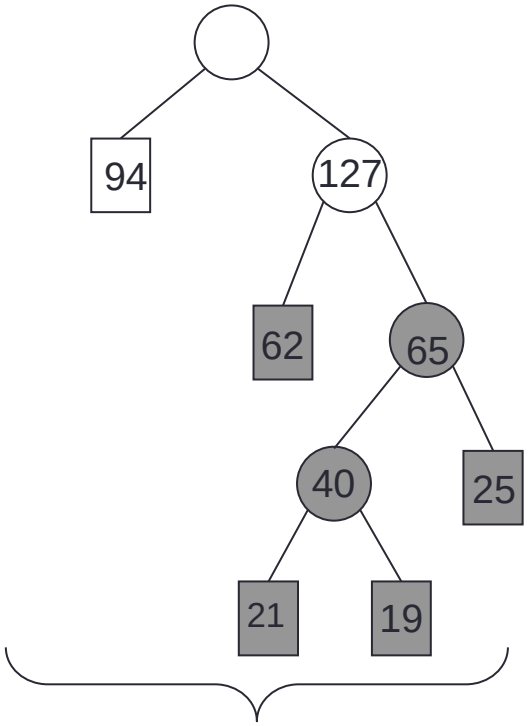
The change will not increase avg. length




This tree has the same set of "leaves" as the one we get after the second step of the algorithm.

# What do we get?

- Start from any code tree T.
- We show how to transform it to the Huffman tree such that every step of our transform does not increase avg. len.
- The average length of Huffman tree is no greater than that of any code tree.
- Huffman code has the minimum average character length.




This is a Huffman Tree.