COMP S265F(W) & 2650SEF & 8650SEF Unit 3: Greedy Heuristics (Huffman Codes)

Dr. WANG DAN, Debby

dwang@hkmu.edu.hk

School of Science and Technology

Hong Kong Metropolitan University

Many thanks to Dr. Keith Lee for providing the original course materials.

Overview

- **Greedy Heuristics**
- **Encoding problem**
  - Definition and key parameter
  - *Goal*: finding an encoding to minimize average character length
- ***Code tree***
  - Characteristics
  - Some observations
- **Huffman code algorithm**
  - *Time complexity:* Simple implementation / Using *min-Heap*
  - *Proof of correctness:* Tree transformation

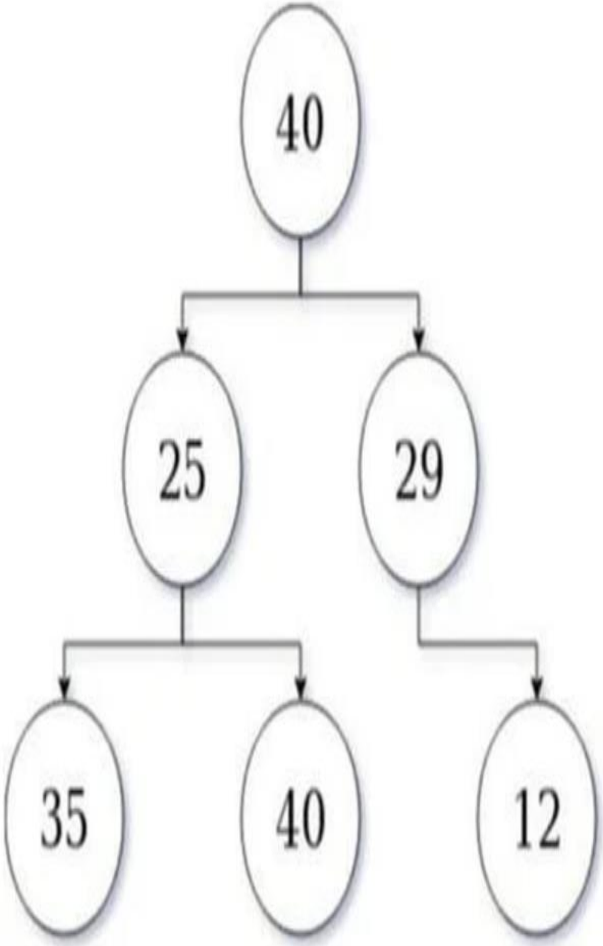
Greedy heuristics

- **Definition**: A basic algorithm design technique that **solves an optimization problem by finding locally optimal solutions**

- **How to be greedy?**
  - At every step, make your best move based on the current situation (**locally optimal choice – offers obvious and immediate benefit**).
  - Keep going until you're done (**yielding a locally optimal solution**).

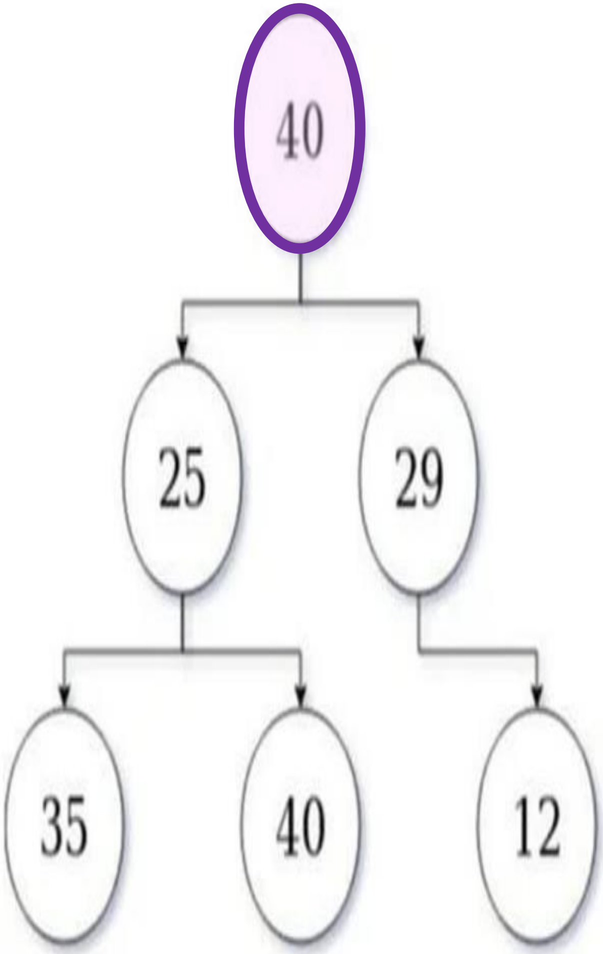
Greedy heuristics

- **Advantages**
  - Don’t need to pay much **effort** at each step.
  - Often find an **acceptable** solution in **a reasonable amount of time**.
- **Disadvantage**
  - Couldn’t take a broad view – **myotic**, sometimes far from the globally optimal solution



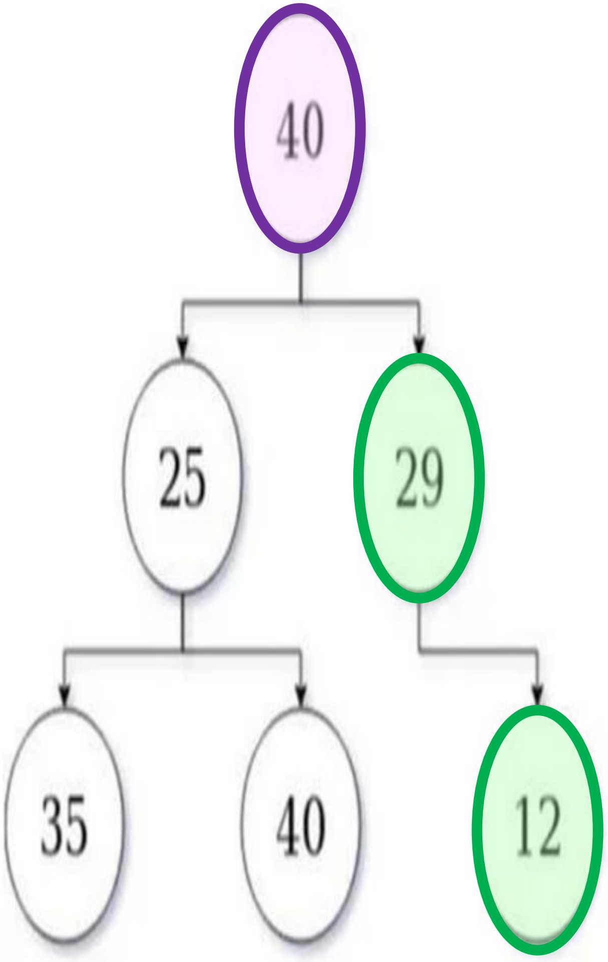
Greedy heuristics

- **Advantages**
  - Don’t need to pay much **effort** at each step.
  - Often find an **acceptable** solution in **a reasonable amount of time**.
- **Disadvantage**
  - Couldn’t take a broad view – **myotic**, sometimes far from the globally optimal solution



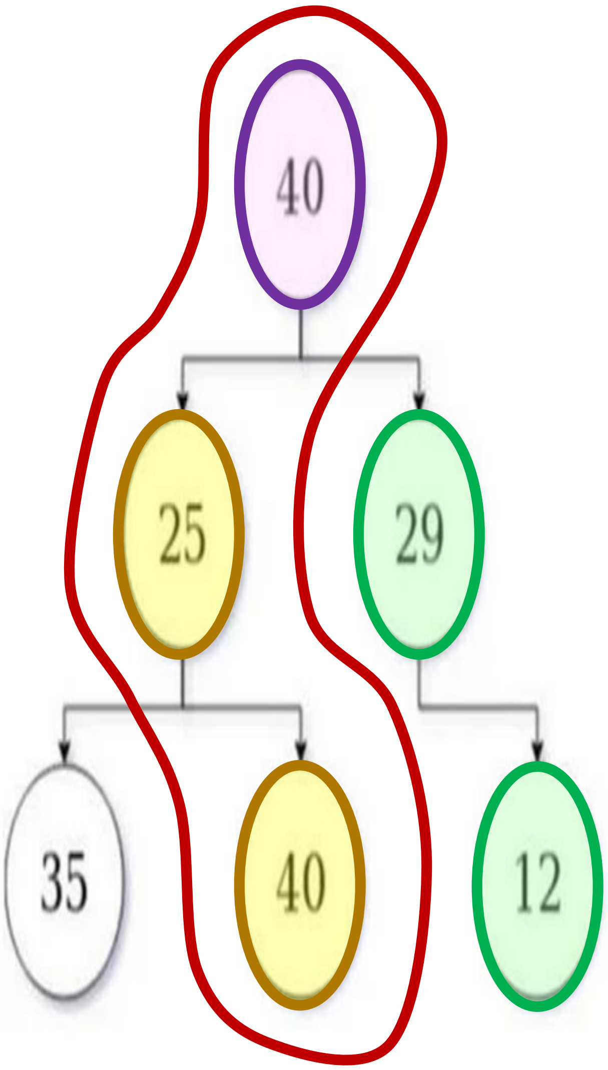
Greedy heuristics

- **Advantages**
  - Don’t need to pay much **effort** at each step.
  - Often find an **acceptable** solution in **a reasonable amount of time**.
- **Disadvantage**
  - Couldn’t take a broad view – **myotic**, sometimes far from the globally optimal solution

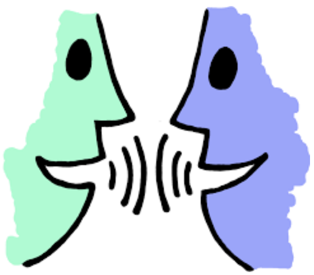


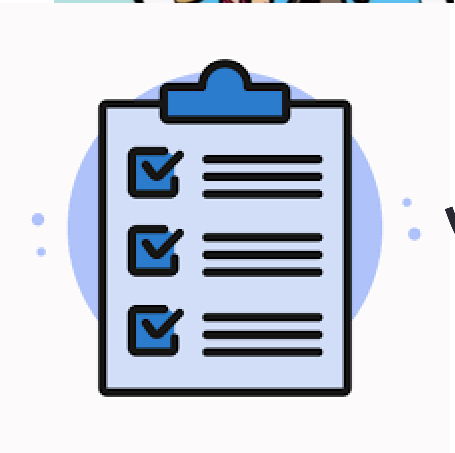
Greedy heuristics

- **Advantages**
  - ➢Don't need to pay much **effort** at each step.
  - ➢Often find an **acceptable** solution in **a reasonable amount of time**.
- **Disadvantage**
  - ➢Couldn't take a broad view – **myotic**, sometimes far from the globally optimal solution



Encoding problem: Background





Information Compression

Storage

Transmission

Encoding: to represent information using binary code?

Encoding problem: Definitions

- A binary code **C** for a character set **E** assigns **a unique binary string** to each character in **E**.

Example:

E = {a, b, c, d}.

C: a $\to$ 1, b $\to$ 01,

c $\to$ 001, d $\to$ 0001

How to decide an encoding of *S* using C is efficient?

- Encoding a sequence **S** of characters using **C**:
  - **S**: aabaacda $\to$ 11011100100011

Encoding problem: Definitions

- Average character length (of ***C*** on ***S***):

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

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

|**S**|

➢ **S** is a sequence, with each character **x** belonging to a character set **E**

➢ f(**x**) is the occurrence of **x** in **S**, and len(**x**) is the number of bits needed to encode **x** according to C.

➢ **In other words, L_C(S) is the average number of bits used for encoding a character in S.**

Encoding problem: Definitions

Recall that

E = {a, b, c, d}.

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

S: aabaacda $\rightarrow$ 110111001000011

Encoding problem: Definitions

Recall that

E = {a, b, c, d}.

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

S: aabaacda $\rightarrow$ 11011100100011

Σ_i **yi=y$_1$+y$_2$+ ⋯ + y$_n$**

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

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

|**S**|

Encoding problem: Definitions

Recall that

E = {a, b, c, d}.

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

S: aabaacda $\rightarrow$ 11011100100011

$\boxed{\Sigma_i \, \mathbf{yi = y_1 + y_2 + \cdots + y_n}}$

$$L_C(S) = \frac{\Sigma_{\mathbf{x} \in \mathbf{E}} f(\mathbf{x}) \times \mathrm{len}(\mathbf{x})}{\Sigma_{\mathbf{x} \in \mathbf{E}} f(\mathbf{x})}$$

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

|**S**|

L_C(S) = (5×1 + 1×2 + 1×3 + 1×4) / 8 = 14/8 = 1.75

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

Quick exercise

1. Suppose E = {'c', 'e', 'i', 'm', 'o', 't'},

C: 'c' → 10, 'e' → 01, 'i' → 001, 'm' → 000, 'o' → 1110, 't' → 1111,

S: 'committee',

Please compute L_C(S).

Encoding problem

- Efficient encoding of **S** is to find a code **C** that leads to the minimum **L_C(S)!!!**

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

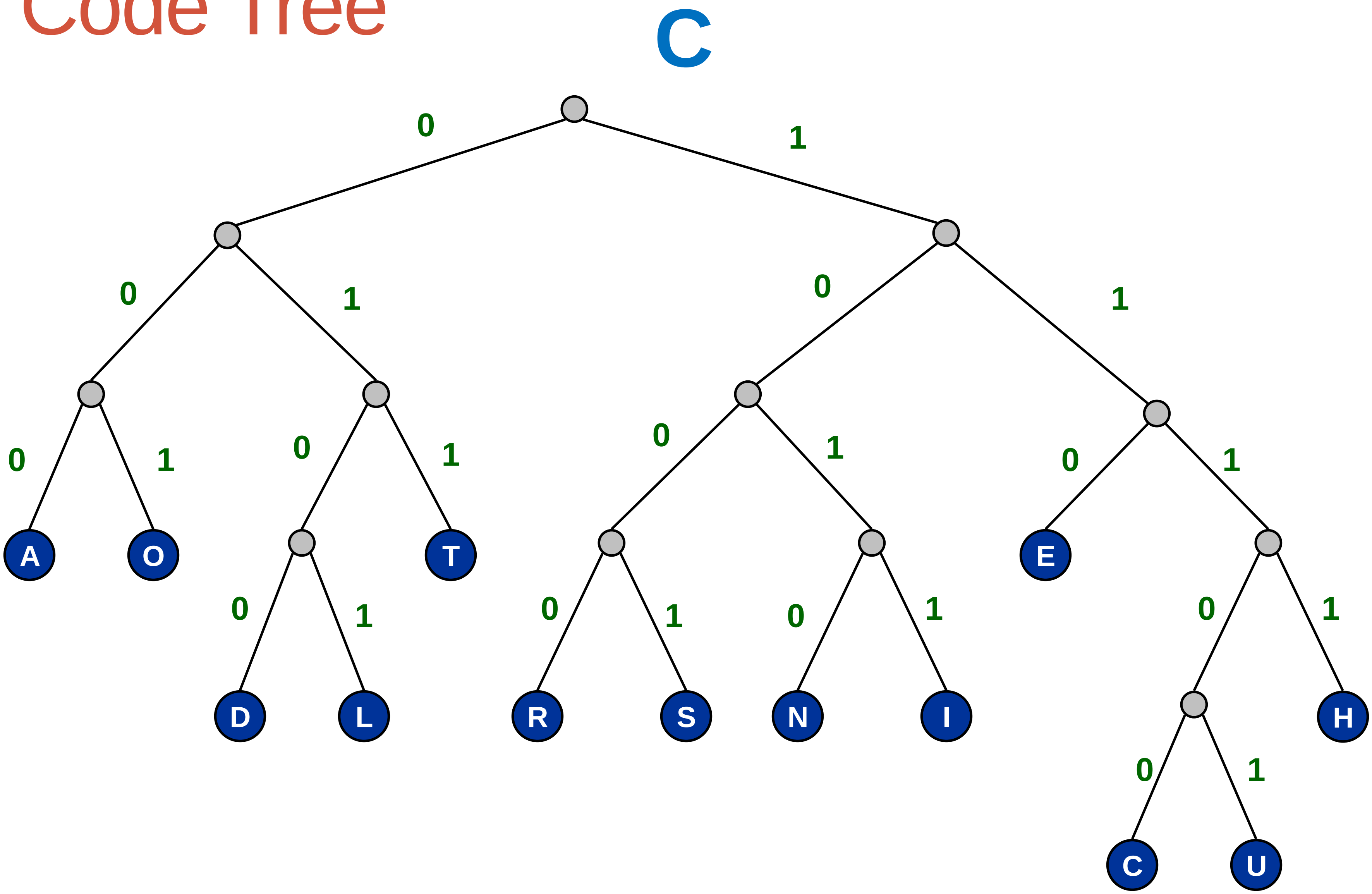
Code Tree

- Usually a binary tree.
- A graphical representation of a code **C**.
- **Example:**

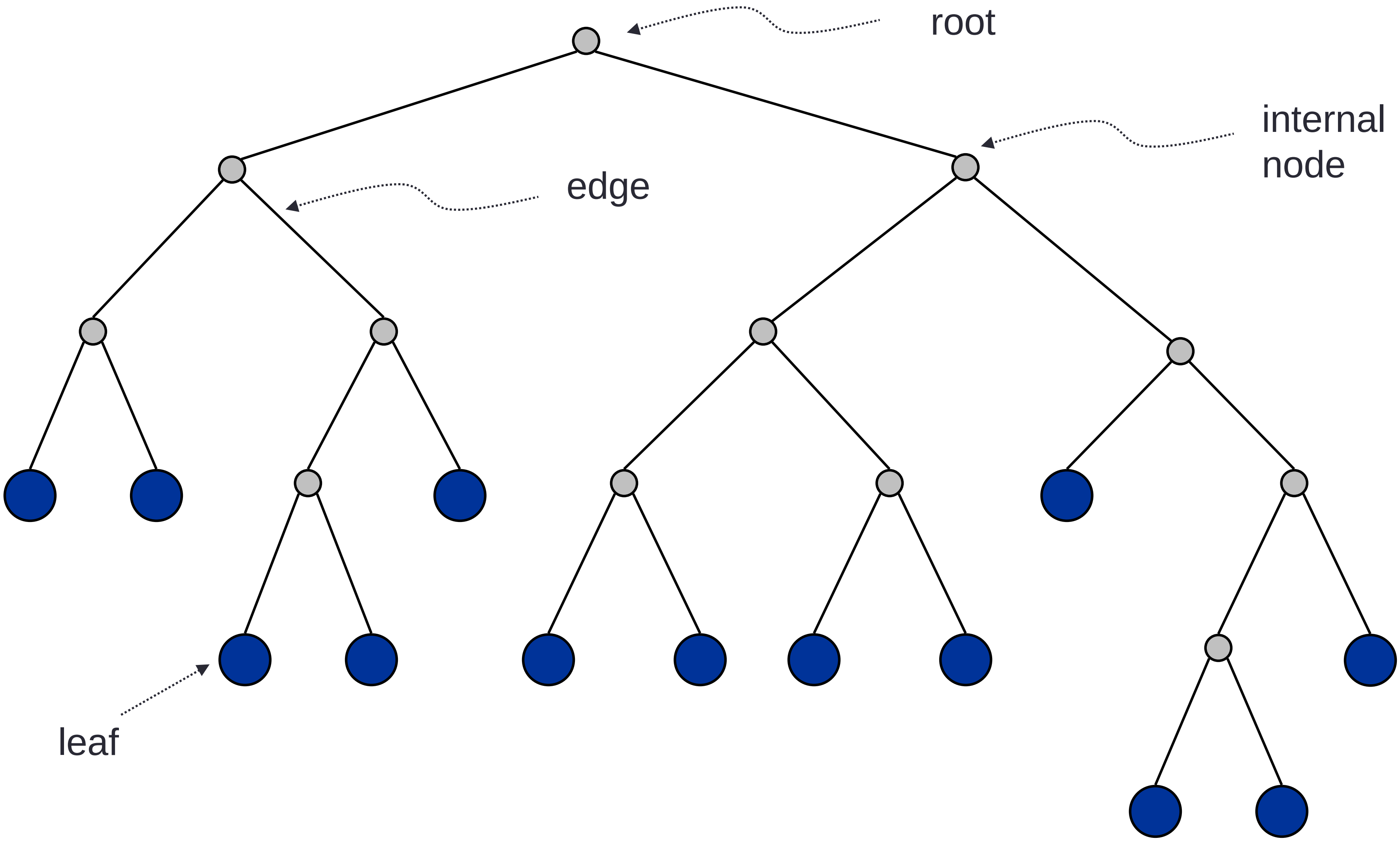
E = {A, C, D, E, H, I, L, N, O, R, S, T, U}.

C: A $\rightarrow$ 000, C $\rightarrow$ 11100, D $\rightarrow$ 0100, E $\rightarrow$ 110, H $\rightarrow$ 1111, I $\rightarrow$ 1011, L $\rightarrow$ 0101, N $\rightarrow$ 1010, O $\rightarrow$ 001, R $\rightarrow$ 1000, S $\rightarrow$ 1001, T $\rightarrow$ 011, U $\rightarrow$ 11101

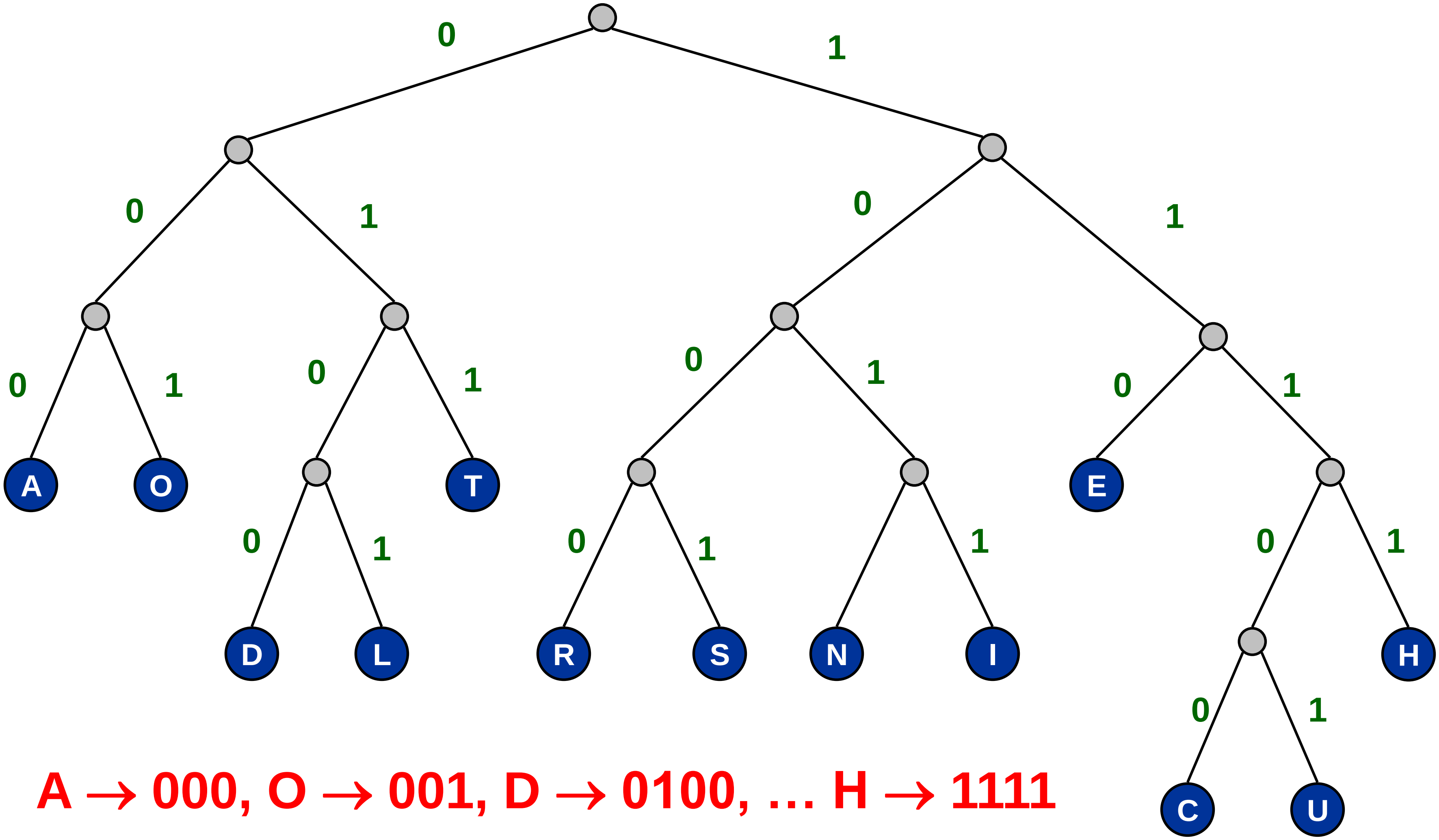
Code Tree



Binary tree

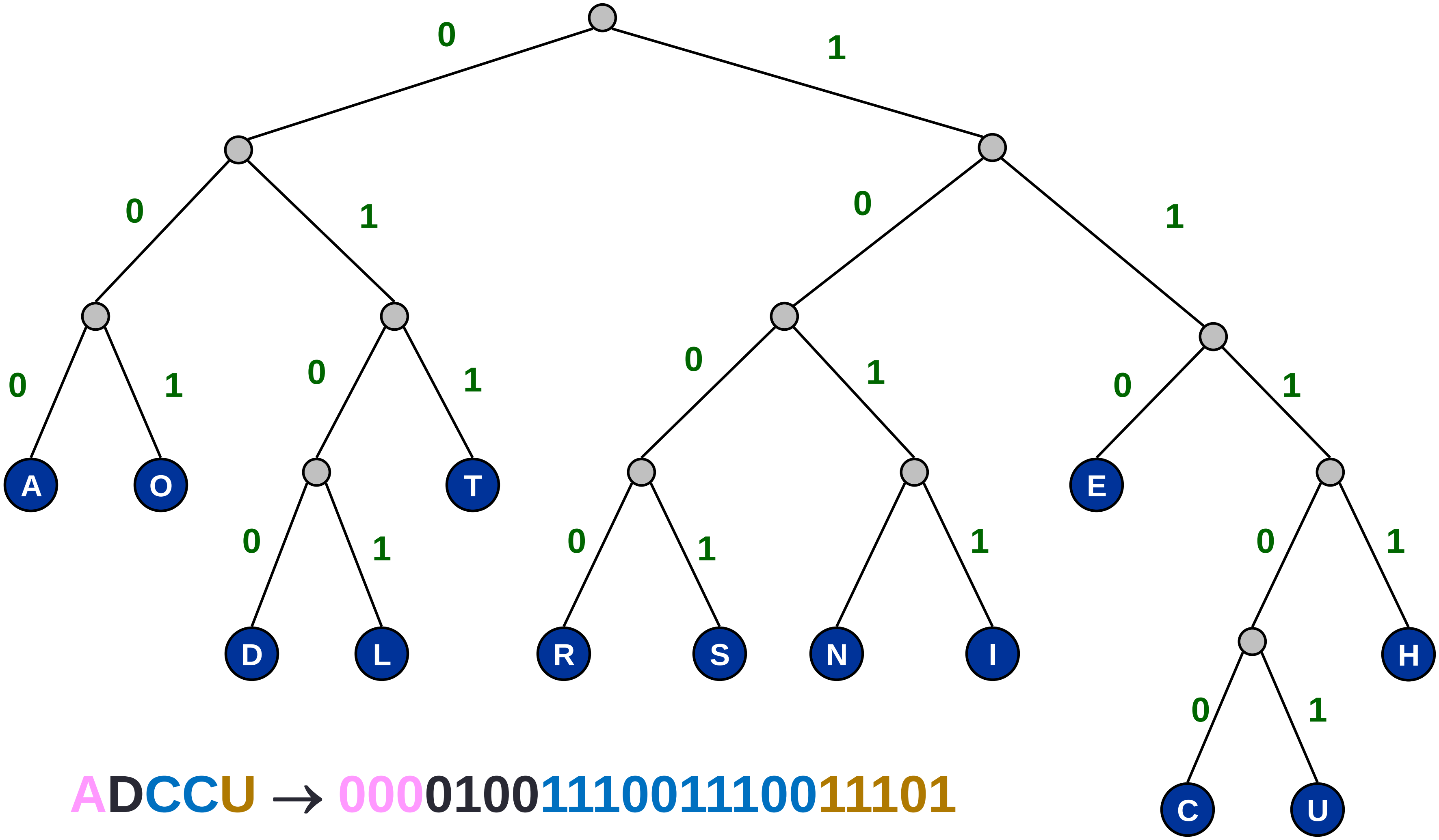


Code tree



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

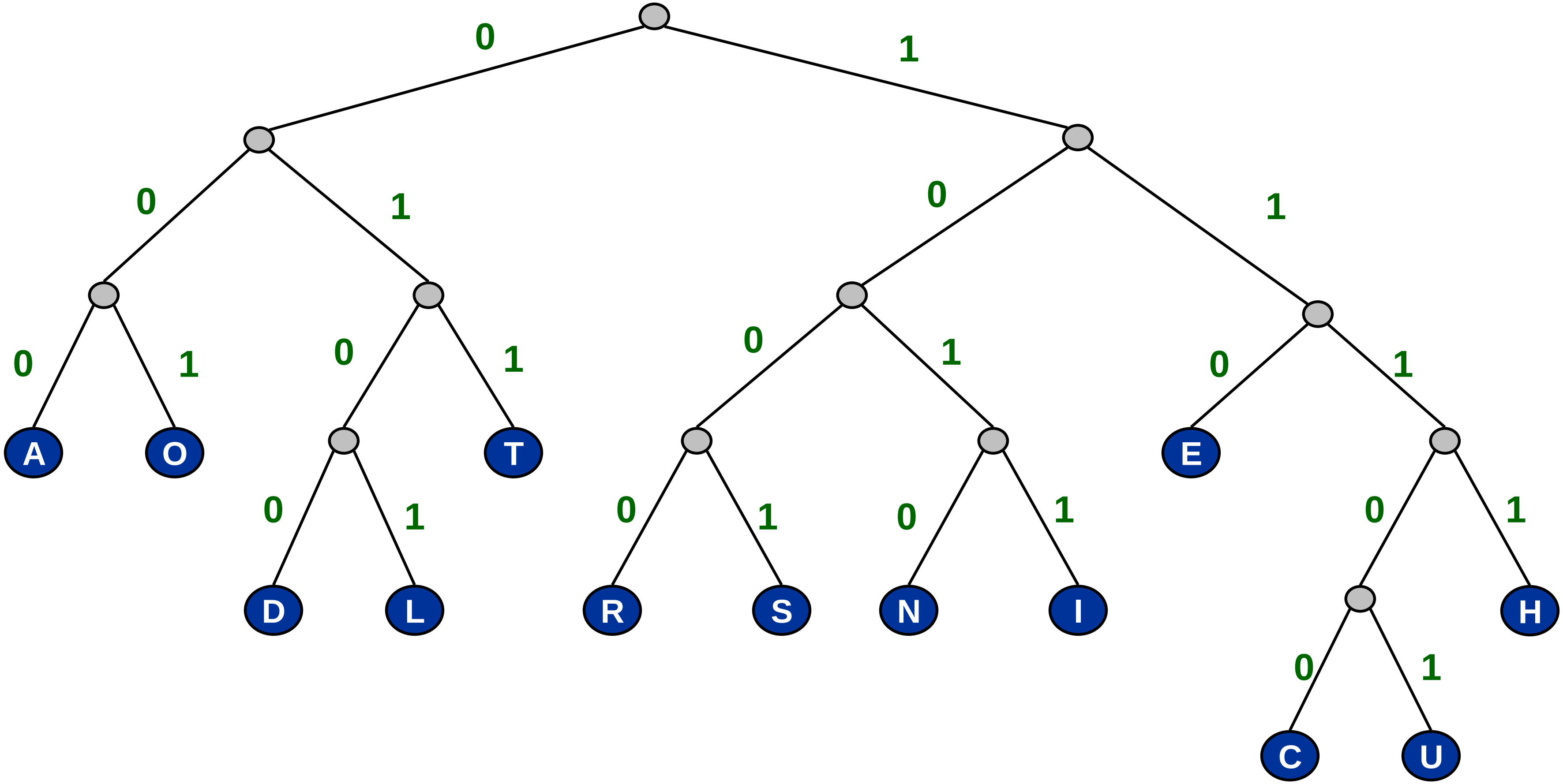
Code tree



ADCCU → 000010011100111001101

Quick exercise

2. Use the code tree C to encode **S:** ‘DELTA’, **and compute $L_C(S)$.**

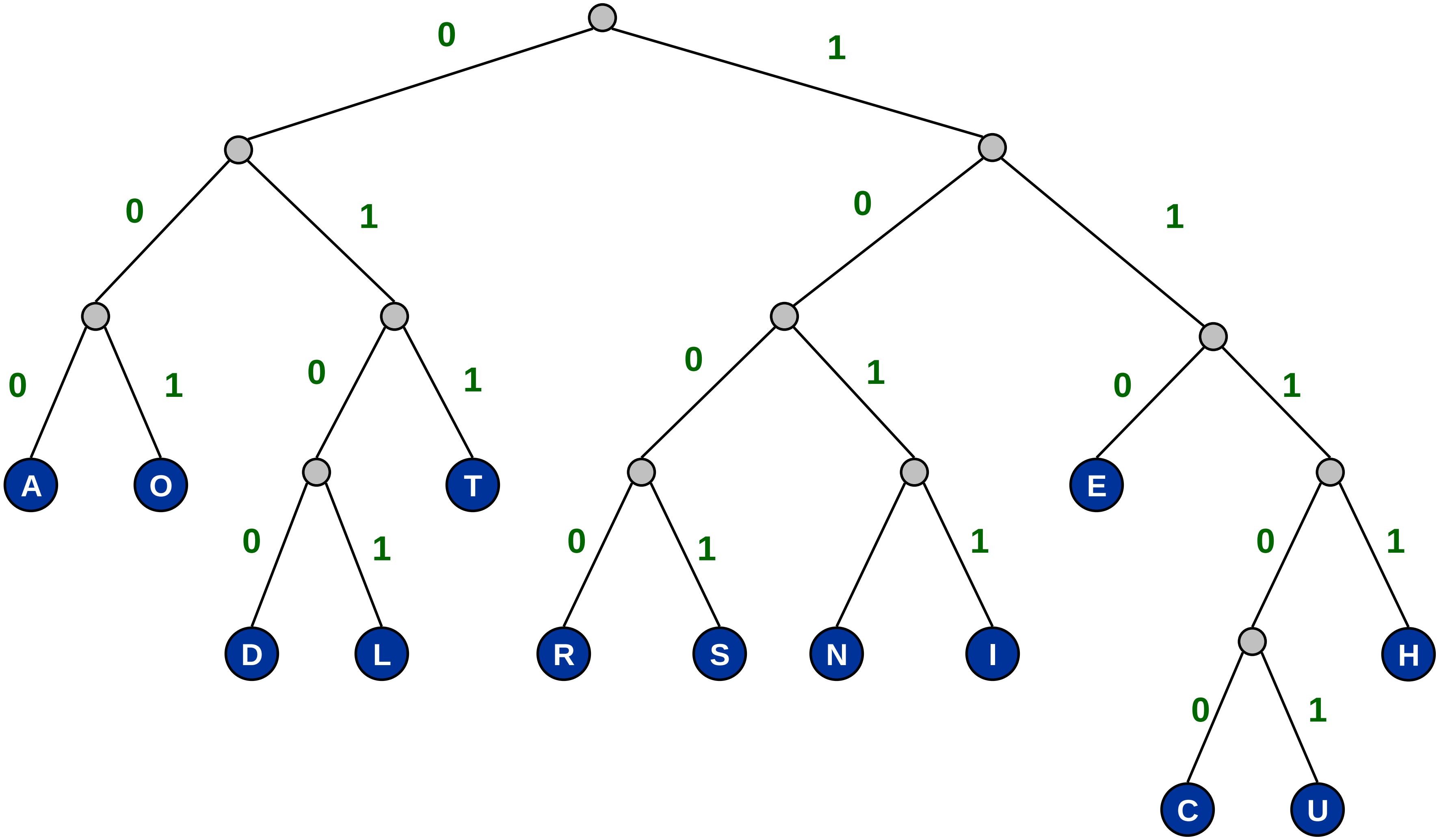


Computing $L_C(S)$ using a code tree

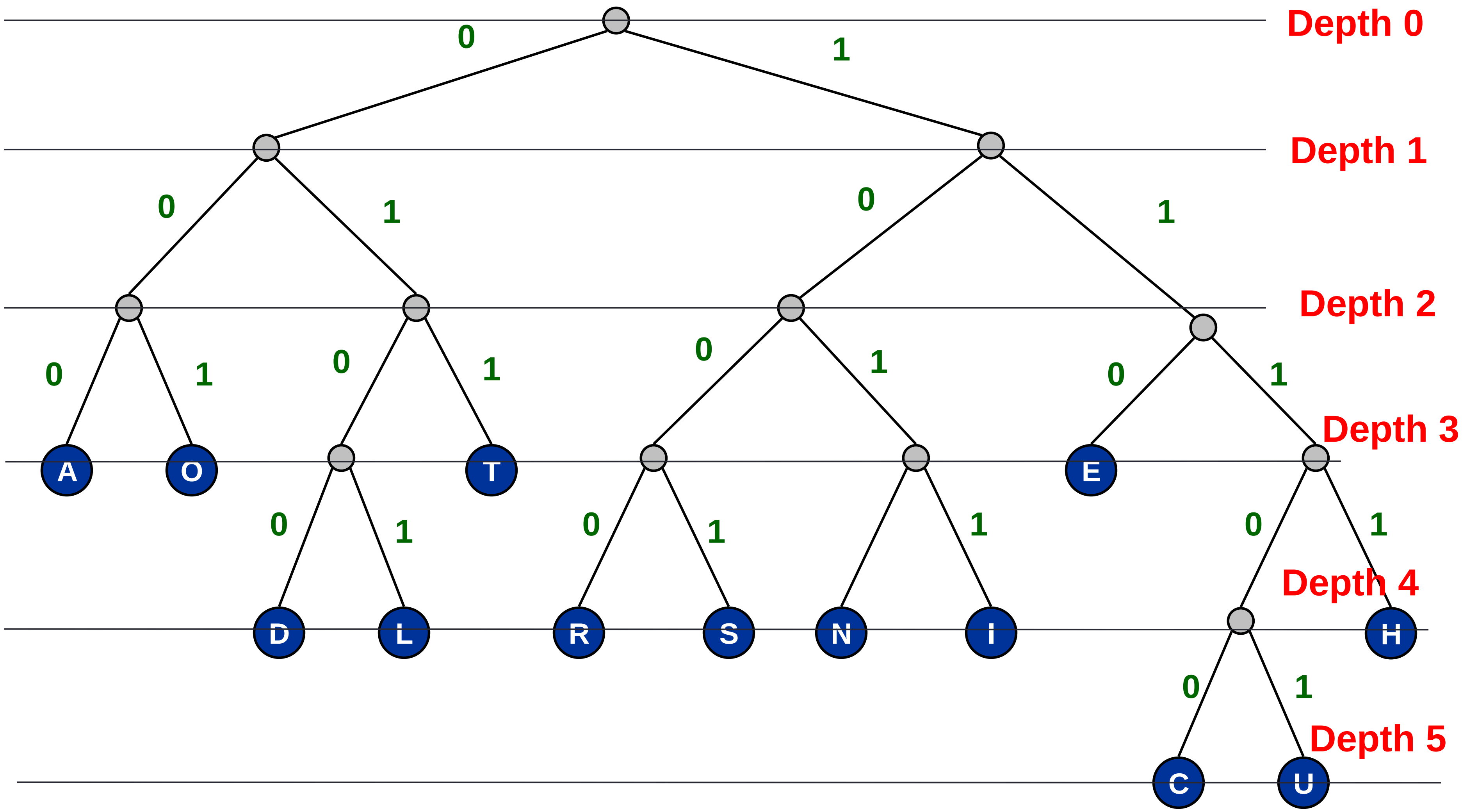
- Given a code tree **T** for code **C**, it is a simple matter to compute $\mathbf{L_C(S)}$.

- First, we need the notion of the depth of nodes in **T**.

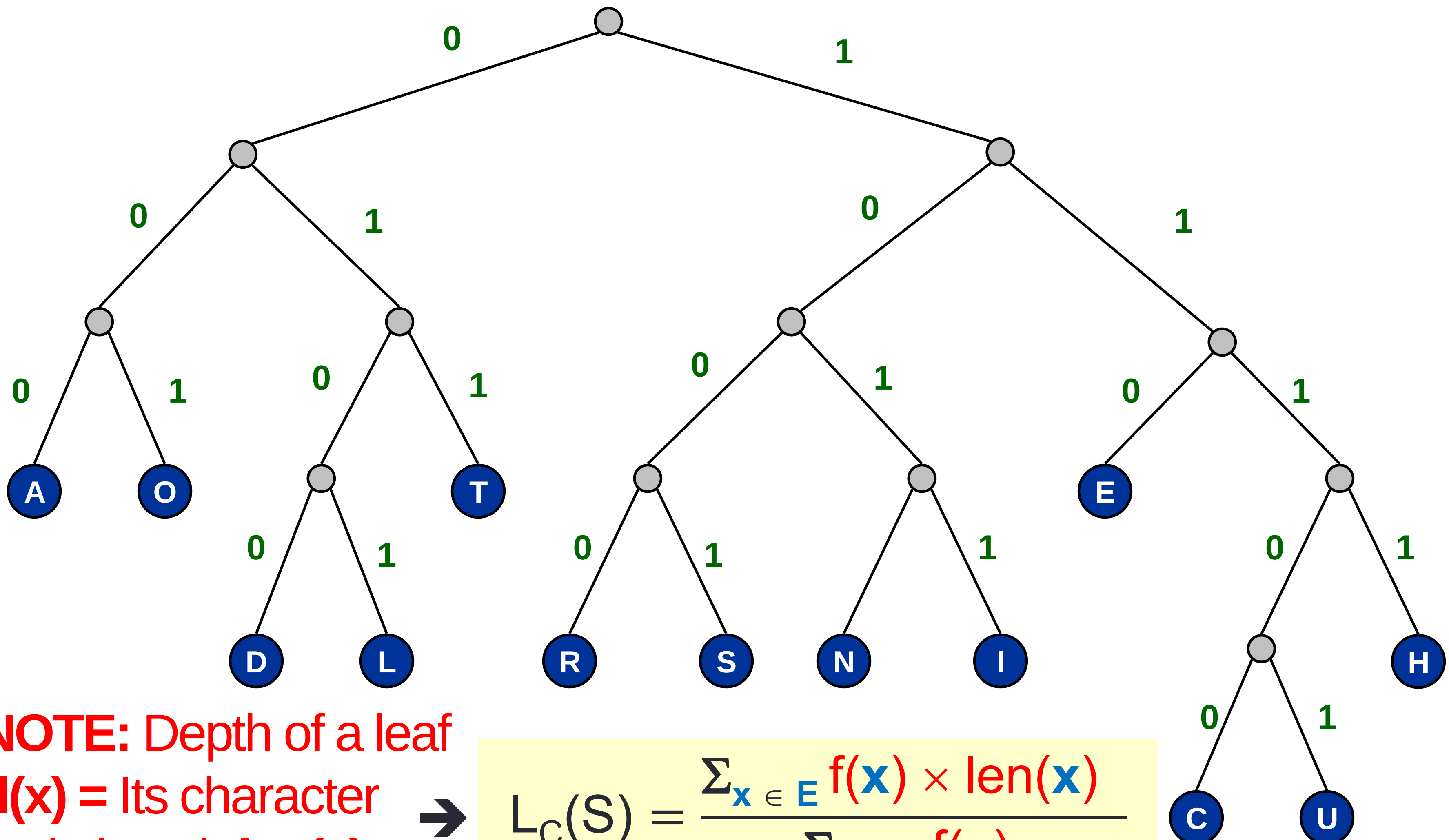
Computing $L_C(\mathbf{S})$ using a code tree



Computing $L_C(\mathbf{S})$ using a code tree



Computing $L_C(\mathbf{S})$ using a code tree



NOTE: Depth of a leaf **d(x) =** Its character code length **len(x)** ➔

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

Looking for an optimal code tree

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

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

- Or equivalently, minimizes

$$\Sigma_{\mathbf{x} \in \mathbf{E}} f(\mathbf{x}) \times len(\mathbf{x})$$

This is given by the input.

This is determined by the tree we construct.

Looking for an optimal code tree

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

$$L_C(S) = \frac{\Sigma_{\mathbf{x} \in \mathbf{E}} f(\mathbf{x}) \times \mathrm{len}(\mathbf{x})}{\Sigma_{\mathbf{x} \in \mathbf{E}} f(\mathbf{x})}$$

fixed; equal to **|S|**.

- Or equivalently, minimizes

$$\Sigma_{\mathbf{x} \in \mathbf{E}} f(\mathbf{x}) \times \mathrm{len}(\mathbf{x})$$

This is given by the input.

This is determined by the tree we construct.

Looking for an optimal code tree

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

$$L_C(S) = \frac{\Sigma_{\mathbf{x} \in \mathbf{E}} f(\mathbf{x}) \times \mathrm{len}(\mathbf{x})}{\Sigma_{\mathbf{x} \in \mathbf{E}} f(\mathbf{x})}$$

fixed; equal to **|S|**.

- Or equivalently, minimizes

$$\Sigma_{\mathbf{x} \in \mathbf{E}} f(\mathbf{x}) \times \mathrm{len}(\mathbf{x})$$

This is given
by the input.

This is determined
by the tree we construct.

We will use simplified
$L_C(S) = \Sigma_{\mathbf{x} \in \mathbf{E}} f(\mathbf{x}) \times \mathrm{len}(\mathbf{x})$
Hereafter!!!

Observations for Optimal Tree

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

1. Suppose that **T** has **n** leaves $s_1, s_2, \ldots, s_n$ with *decreasing frequency*, namely

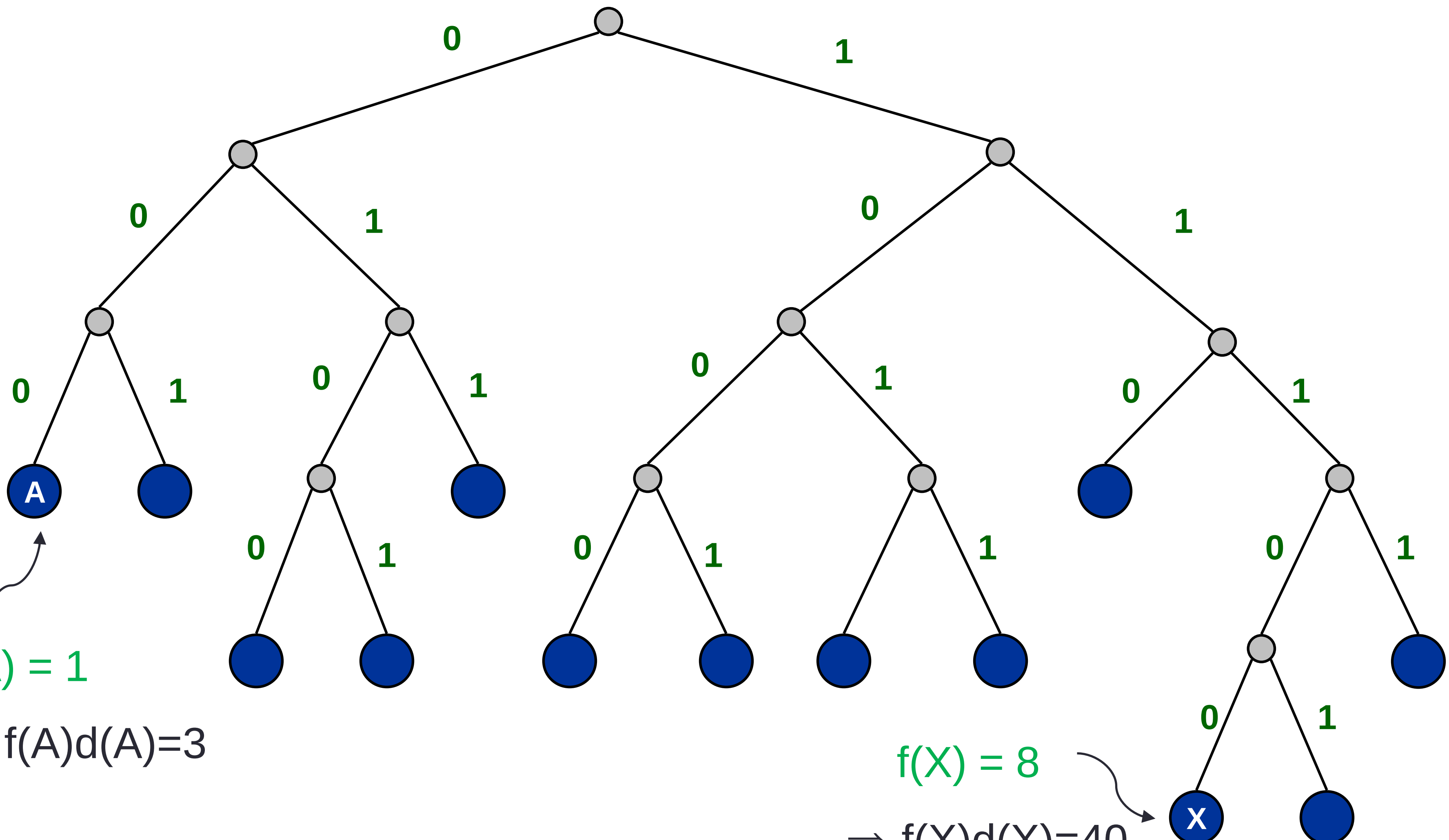
   $$\mathbf{f(s_1) \geq f(s_2) \geq \ldots \geq f(s_n).}$$

   Then they should have *increasing depths* in **T**,

   $$\mathbf{d(s_1) \leq d(s_2) \leq \ldots \leq d(s_n).}$$

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

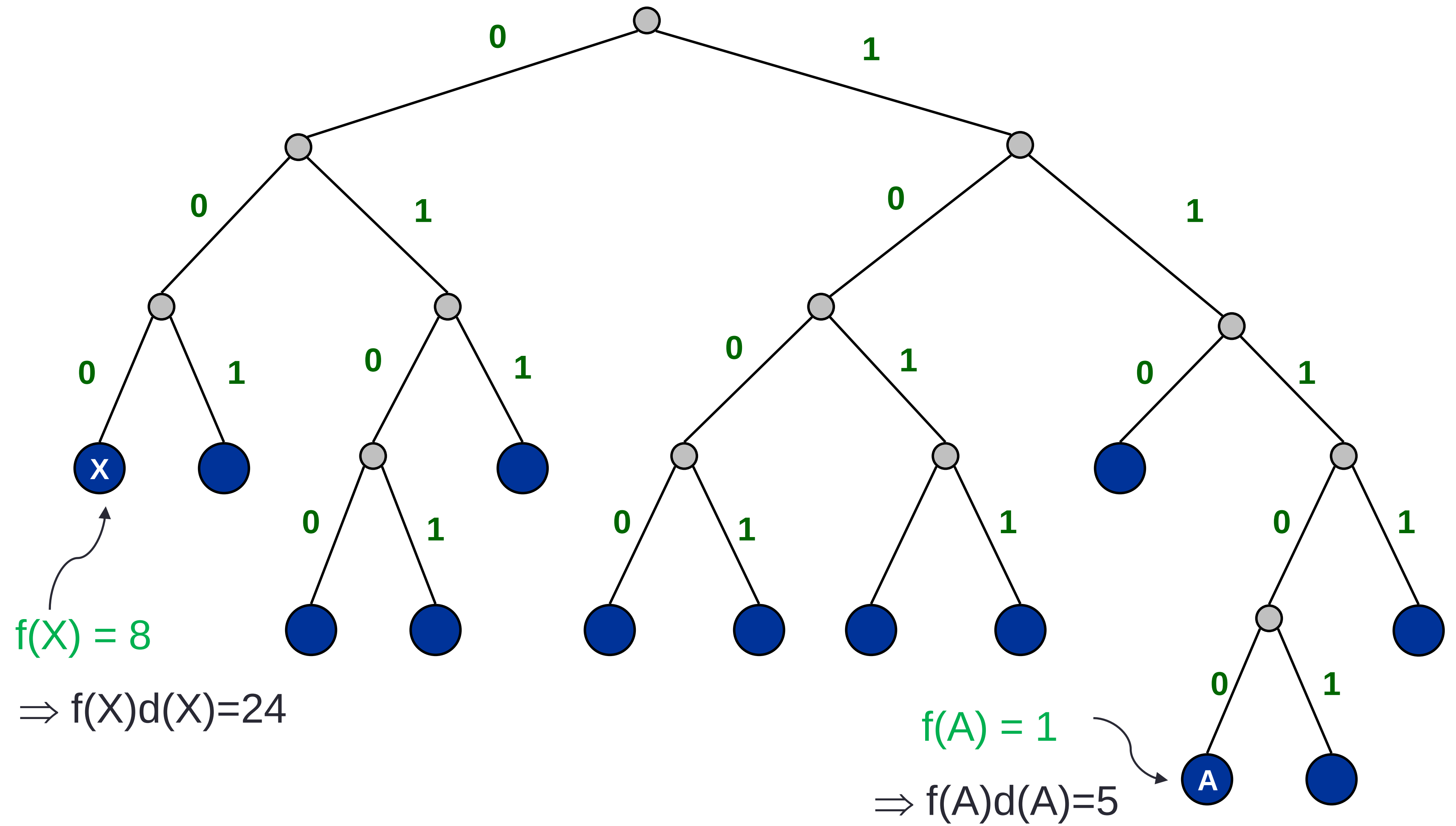
⇒ f(A)d(A)=3

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: $\mathbf{L_C(S)} = (\sum_{x \notin \{A,X\}} f(x)d(x)) + \boldsymbol{f(A)d(A)} + \boldsymbol{f(X)d(X)}$
- After the swap: $\mathbf{L_C(S)} = (\sum_{x \notin \{A,X\}} f(x)d(x)) + \boldsymbol{f(A)d(X)} + \boldsymbol{f(X)d(A)}$
- The difference between the new tree and the old tree is

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

- Thus, the new tree has **smaller** $\mathbf{L_C(S)}$, which **contradicts** that the old tree is an optimal tree.

Observations for Optimal Tree

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

1. Suppose that **T** has **n** leaves s_1, s_2,…, s_n with *decreasing frequency*, namely

   $$\mathbf{f(s_1) \geq f(s_2) \geq \ldots \geq f(s_n).}$$

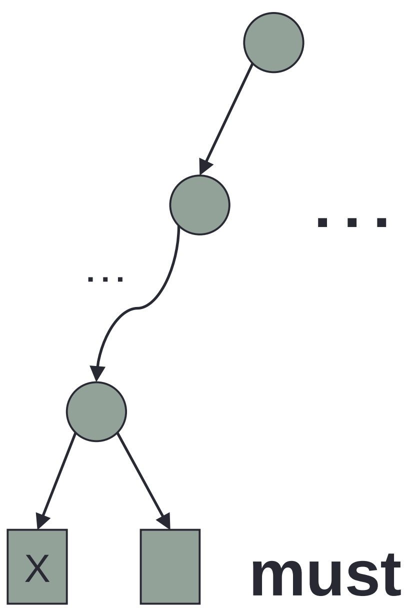
   Then they should have *increasing depths* in **T**,

   $$\mathbf{d(s_1) \leq d(s_2) \leq \ldots \leq d(s_n).}$$

2. The **leaf node** with the largest depth in **T** must have a sibling.

Observation 2

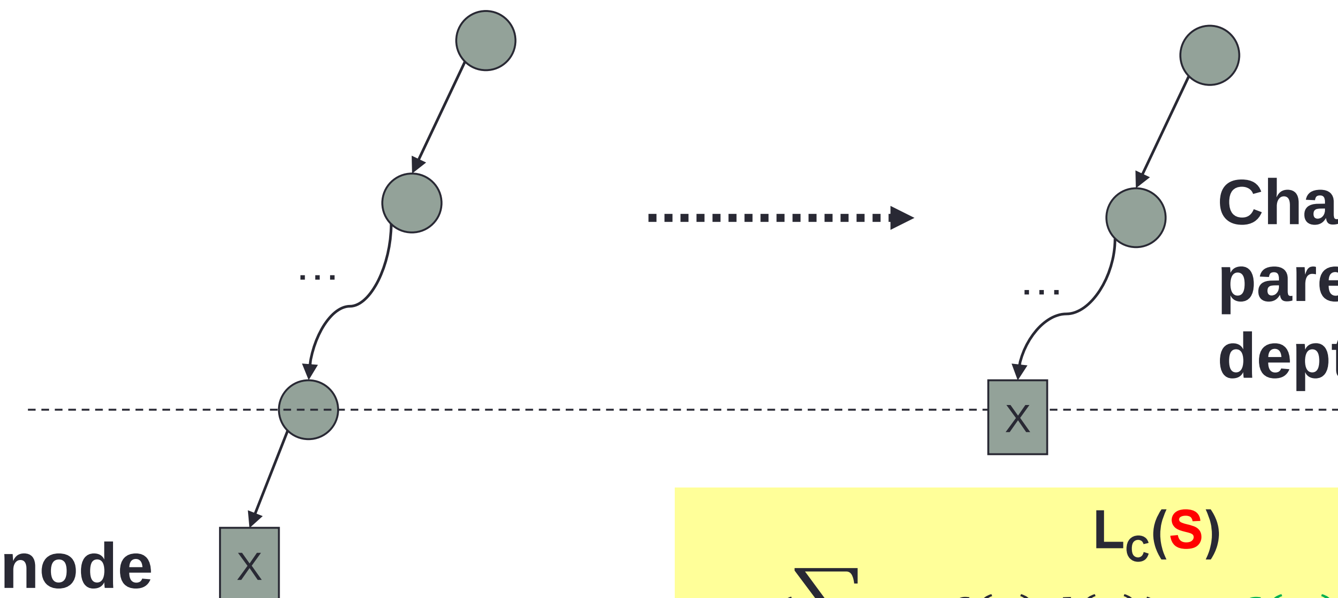
- We also observe that in **the optimal code tree** T, the leaf node with the largest depth must have a sibling.



Leaf node **must have a sibling here**

Observation 2

- If not, then



Change to the parent node, depth - 1

Leaf node

L_C(S) $=$

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

L_C(S)

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

$$= \left(\sum_{c \neq X} f(c)d(c)\right) + \boldsymbol{f(X)d(X) - f(X)}$$

This is smaller!

Idea for Building Optimal Tree

In an **optimal code tree** T, for every character x,

1. the larger the depth d(x), the smaller the frequency f(x); and
2. the leaf **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.

L_C(S) =

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

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

Idea for Building Optimal Tree

In an **optimal code tree T**, for every character x,

1. the larger the depth d(x), the smaller the frequency f(x); and
2. the leaf **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.

$$\mathbf{L_C(S)} =$$

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

$$= \underbrace{\sum_{x\notin\{u,v\}} f(x)d(x) + (\boldsymbol{f(u)} + \boldsymbol{f(v)})(\boldsymbol{d(u)} - \mathbf{1})}_{\text{As if it has n-1 leaves, with the old leaves u, v replaced by a new one with frequency f(u)+f(v). This sum is minimized for these n-1 leaves.}} + \underbrace{(\boldsymbol{f(u)} + \boldsymbol{f(v)})}_{\text{fixed}}$$

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.

fixed

Idea for Building Optimal Tree

In an **optimal code tree T**, for every character x,

1. the larger the depth d(x), the smaller the frequency f(x); and
2. the leaf **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 T** :

> 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 of characters **E**:

1. Build the code tree in a bottom-up manner.
2. It begins with a set of | **E** | leaves and performs a sequence of | **E** |-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(| **E** |^2) time.
- Using a min-Heap: O(| **E** | log | **E** |) time.

Huffman code (simple implementation)

```
Huffman(h):
      sort the nodes in h by their frequencies
      while len(h) > 1:
           get the least-frequent node left in h
           delete left from h

           get the 2nd least-frequent node right in h
           delete right from h

           merge left and right into a new leaf node last
           with last.freq = left.freq + right.freq,
           h.append(last)
      return last
```

Min-Heap

- A min-Heap is a **Tree-based data structure** that maintains a set **B** of ***n*** numbers and allows access to the minimum element in constant time.

- Performance analysis of a min-Heap:
  - Find-Min(S): return the minimum number in S (**O(1)**);
  - Delete-Min(S): delete the minimum number in S (**O(logn)**) - needs to fix the violated heap property;
  - Insert(x, S): insert a new number to S (**O(logn)**) - needs to fix the violated heap property.

- The time complexity of the Huffman algorithm is **O(nlogn)**. Using Min-heap to store **B**, each iteration requires **O(logn)** time to determine the minimum and insert the new number. There are **O(n)** iterations, one for each item.

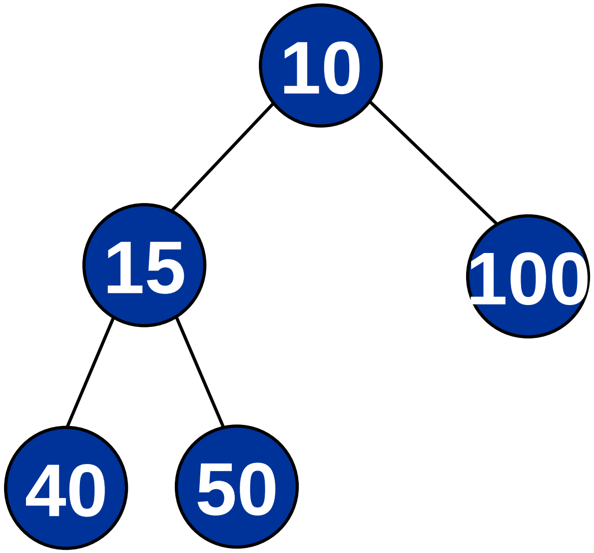
Min-Heap

- In a min-Heap, **the key at root** must be **minimum** among all keys present in the Heap. The same property must be recursively true for all nodes in the Tree.

- A binary heap is typically represented as **an array Arr**:
  - The root element will be at Arr[0].
  - Arr[(i-1)/2] returns the parent node
  - Arr[2*i+1] and Arr[2*i+2] return the left-child and right-child nodes

- **Heap sort**: using a min-Heap to sort an array costs O(nlogn) time.

An example of min-Heap tree



Huffman code (using a min-Heap)

```
Huffman2(h):
  construct a min-Heap for the nodes in h by their frequencies
  while len(h) > 1 :
     pop out a node left from min-Heap
     pop out a node right from min-Heap

     merge left and right into a new leaf node last
     with last.freq = left.freq + right.freq,
     push last into the min-Heap
  return last
```

Huffman code

Example:

Step 1

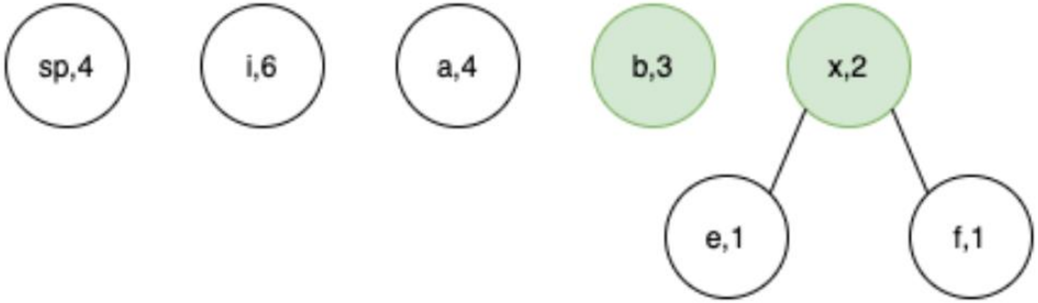
1. Suppose **S** is **'spspspsp iiiiii e f aaaa bbb'**. First we calculate the frequencies: **{'sp':4, 'i':6, 'e':1, 'f':1, 'a':4, 'b':3}** :
2. Start at the leaf nodes, and merge nodes in a bottom-up manner.

Huffman code

Example:

1. Suppose **S** is **'spspspsp iiiiii e f aaaa bbb'**. First we calculate the frequencies: **{'sp':4, 'i':6, 'e':1, 'f':1, 'a':4, 'b':3}** :
2. Start at the leaf nodes, and merge nodes in a bottom-up manner.

Step 2

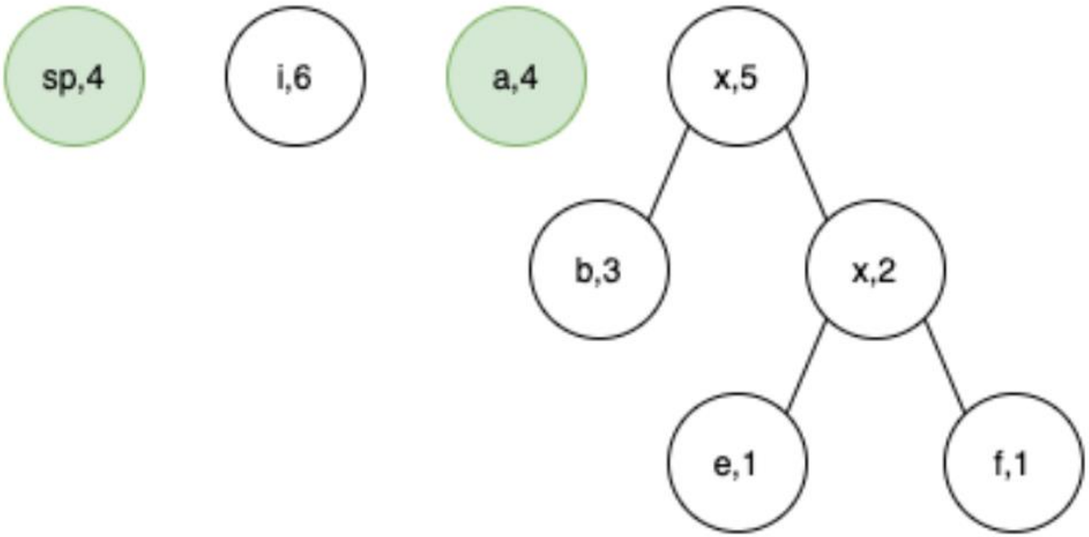


Huffman code

Example:

1. Suppose **S** is **'spspspsp iiiiii e f aaaa bbb'**. First we calculate the frequencies: **{'sp':4, 'i':6, 'e':1, 'f':1, 'a':4, 'b':3}** :
2. Start at the leaf nodes, and merge nodes in a bottom-up manner.

Step 3

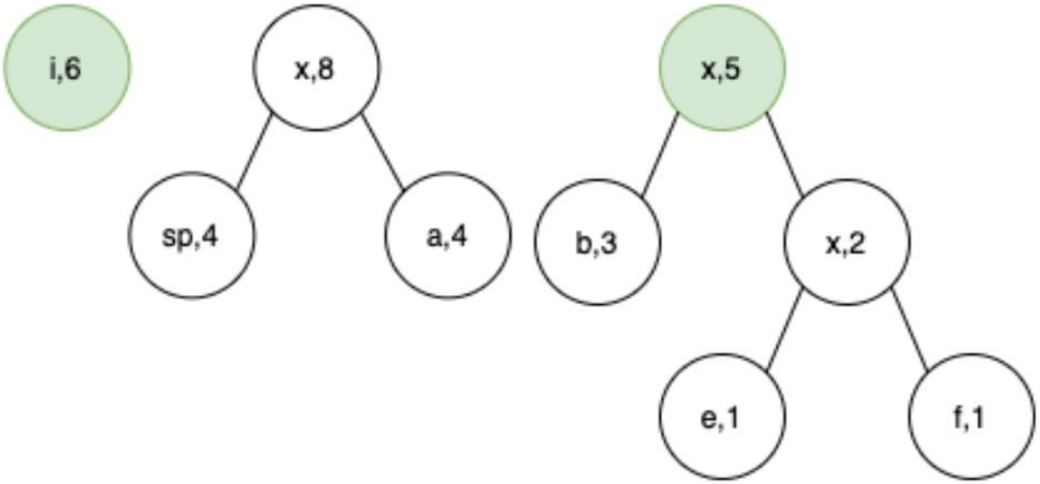


Huffman code

Example:

1. Suppose **S** is **'spspspsp iiiiii e f aaaa bbb'**. First we calculate the frequencies: **{'sp’:4, ‘i’:6, ‘e’:1, 'f':1, ‘a’:4, 'b':3}** :
2. Start at the leaf nodes, and merge nodes in a bottom-up manner.

Step 4

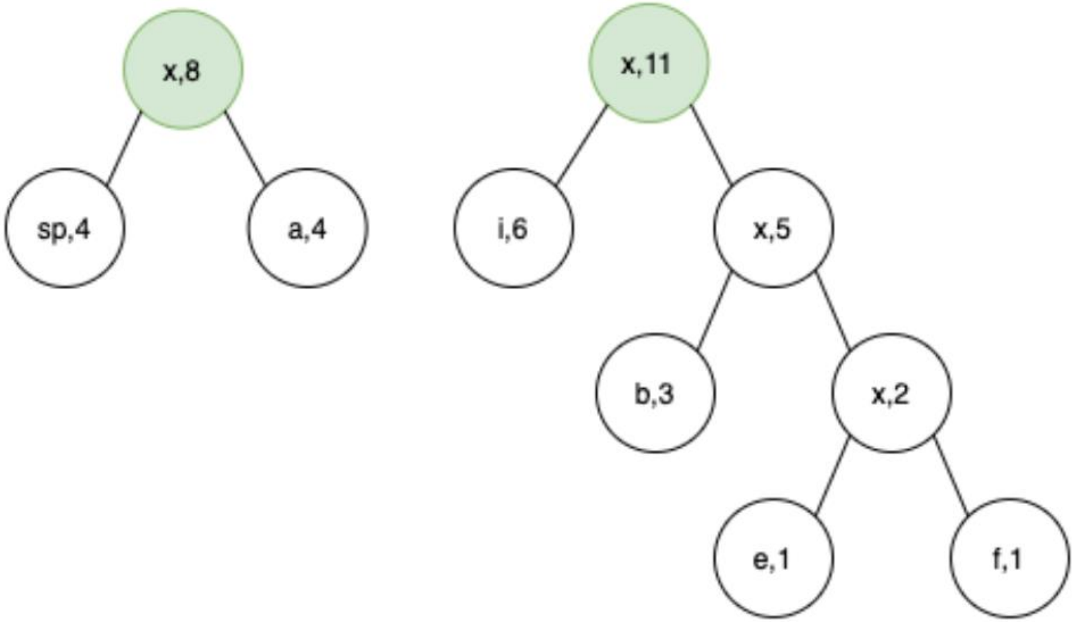


Huffman code

Example:

1. Suppose **S** is **'spspspsp iiiiii e f aaaa bbb'**. First we calculate the frequencies: **{'sp':4, 'i':6, 'e':1, 'f':1, 'a':4, 'b':3}** :
2. Start at the leaf nodes, and merge nodes in a bottom-up manner.

Step 5

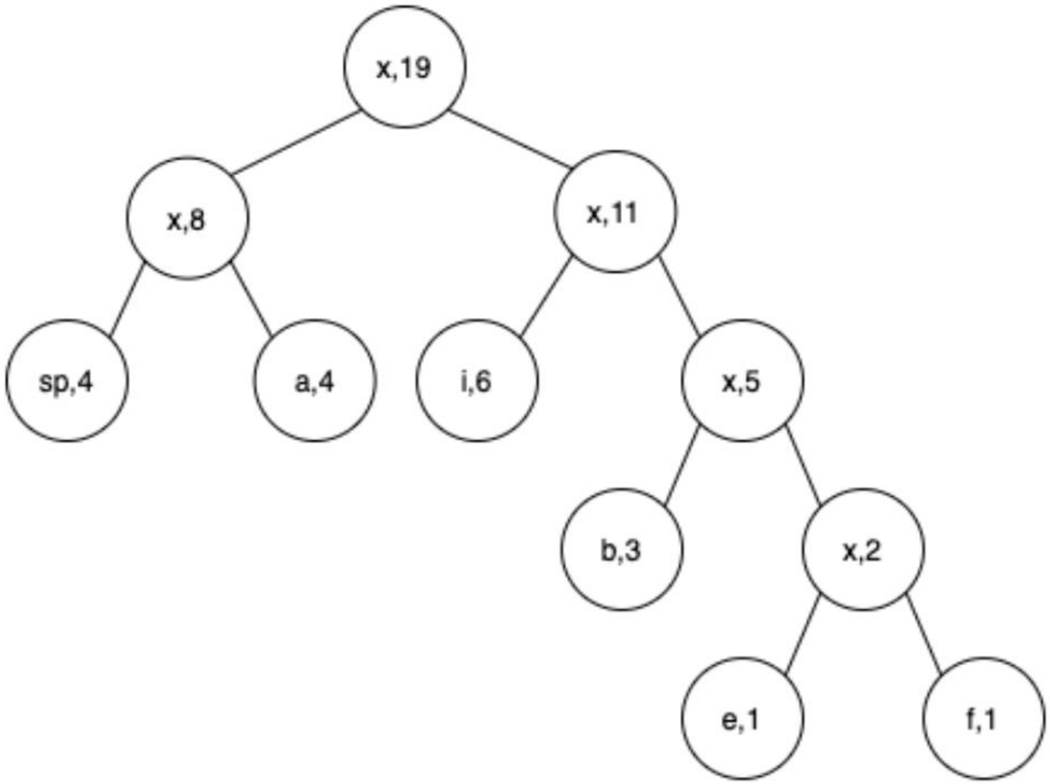


Huffman code

Example:

1. Suppose **S** is **'spspspsp iiiiii e f aaaa bbb'**. First we calculate the frequencies: **{'sp’:4, ‘i’:6, ‘e’:1, 'f':1, ‘a’:4, 'b':3}** :
2. Start at the leaf nodes, and merge nodes in a bottom-up manner.

Step 6

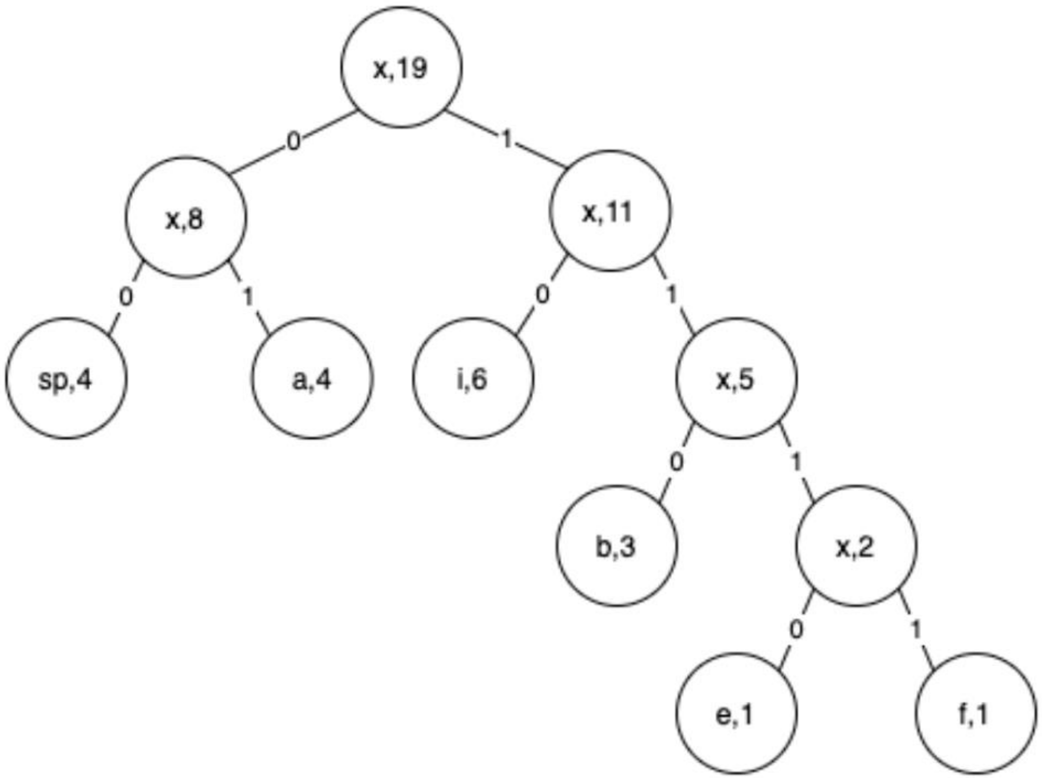


Huffman code

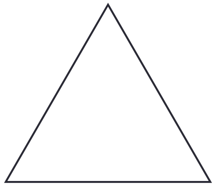
Example:

1. Suppose **S** is **'spspspsp iiiiii e f aaaa bbb'**. First we calculate the frequencies: **{'sp':4, 'i':6, 'e':1, 'f':1, 'a':4, 'b':3}** :
2. Start at the leaf nodes, and merge nodes in a bottom-up manner.

Step 7

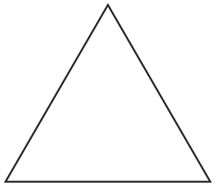


Proof of correctness (rough ideas)



Any optimal tree

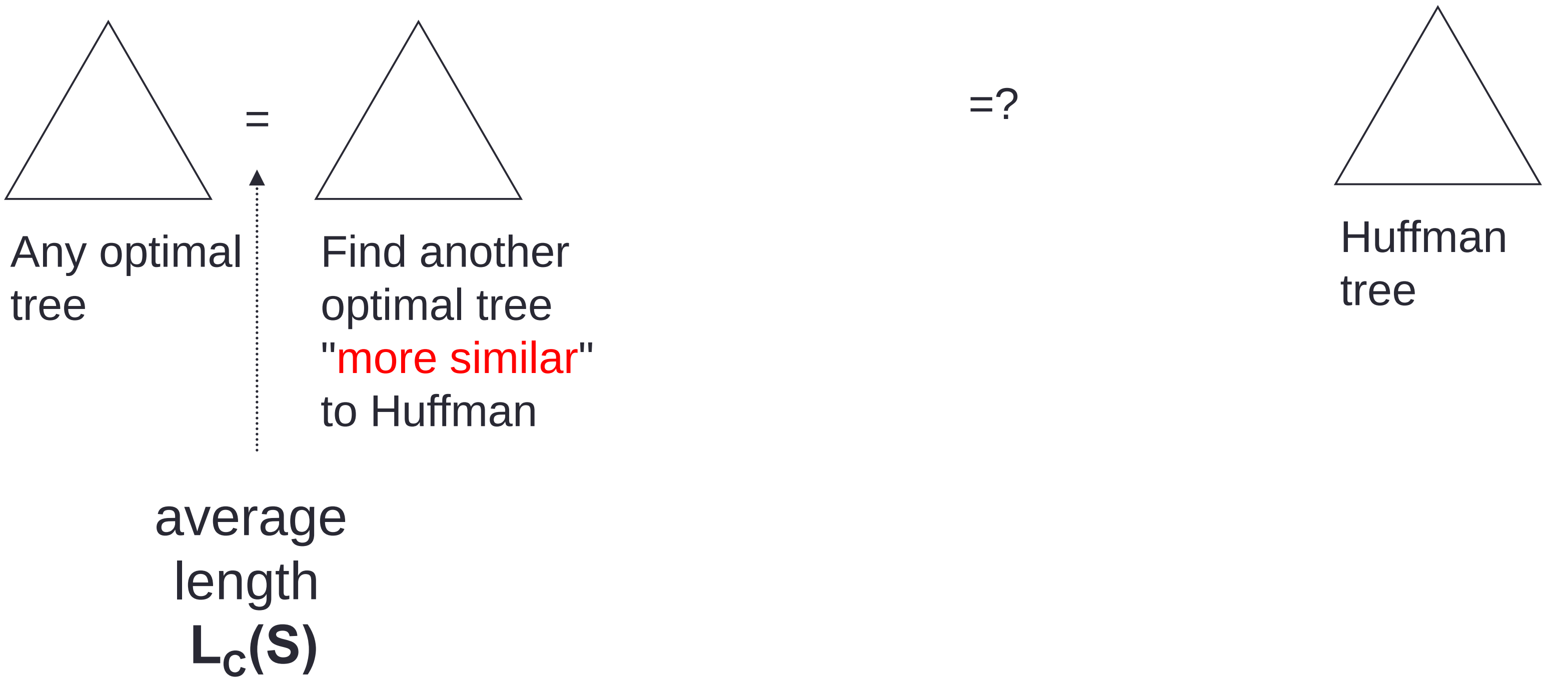
=?



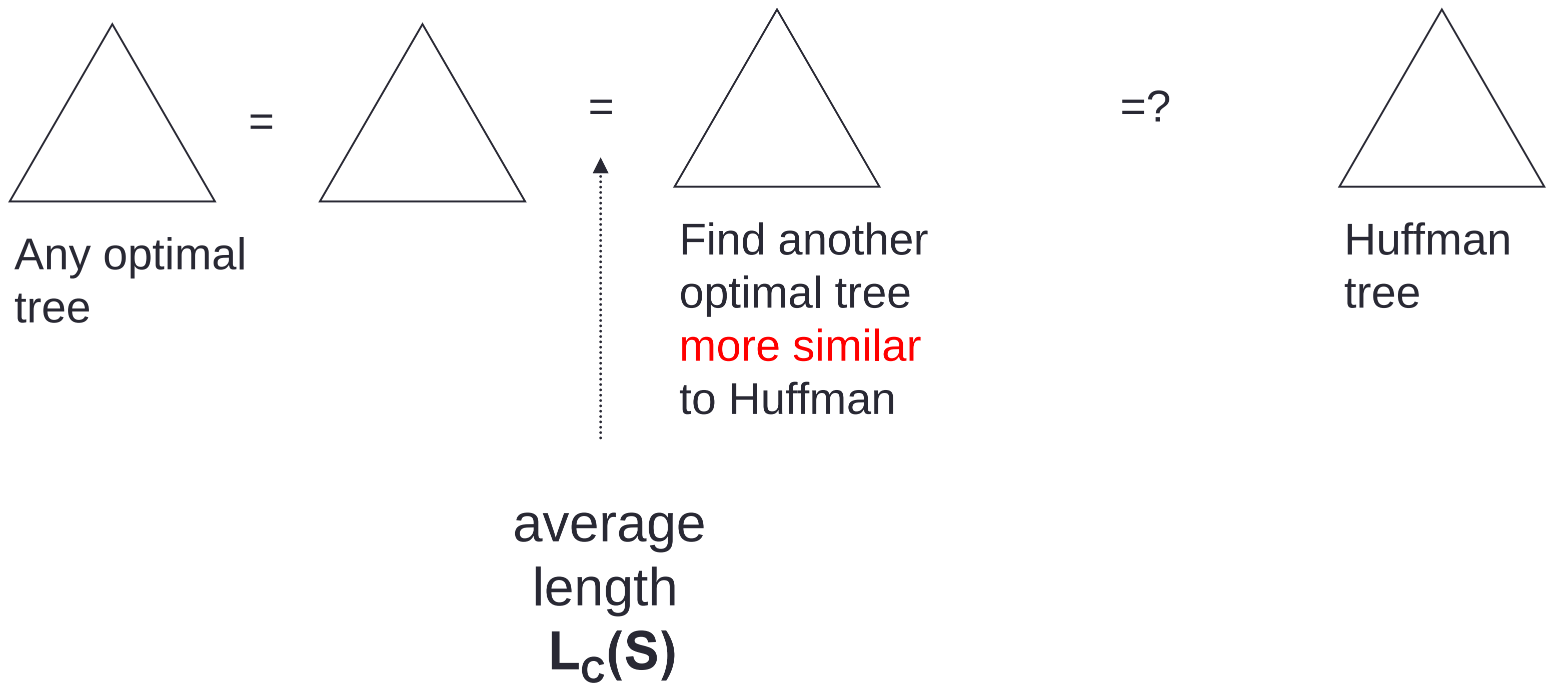
Huffman tree

average length **$L_C(S)$**

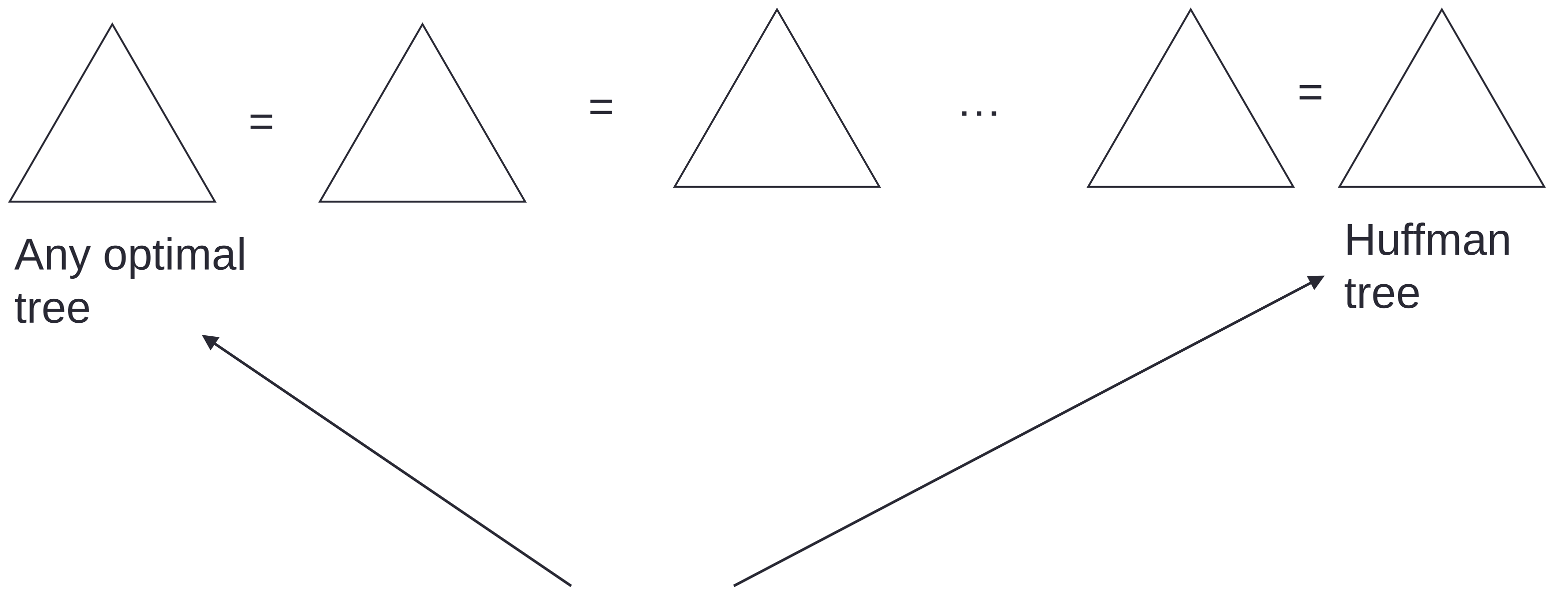
Proof of correctness (rough ideas)



Proof of correctness (rough ideas)

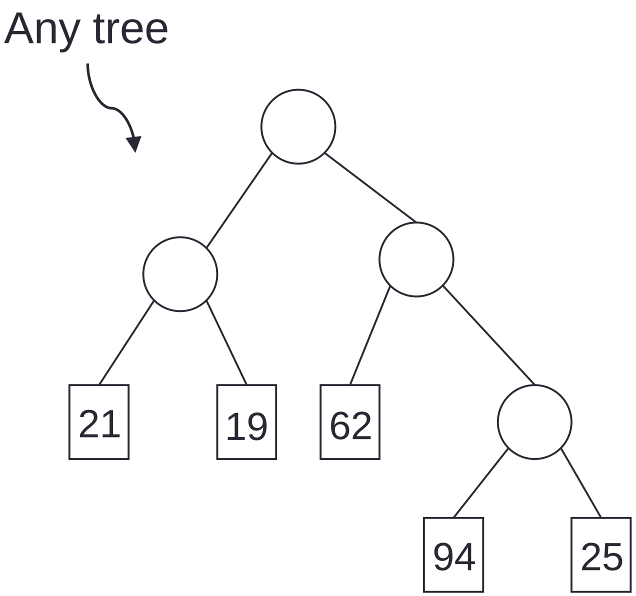


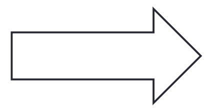
Proof of correctness (rough ideas)

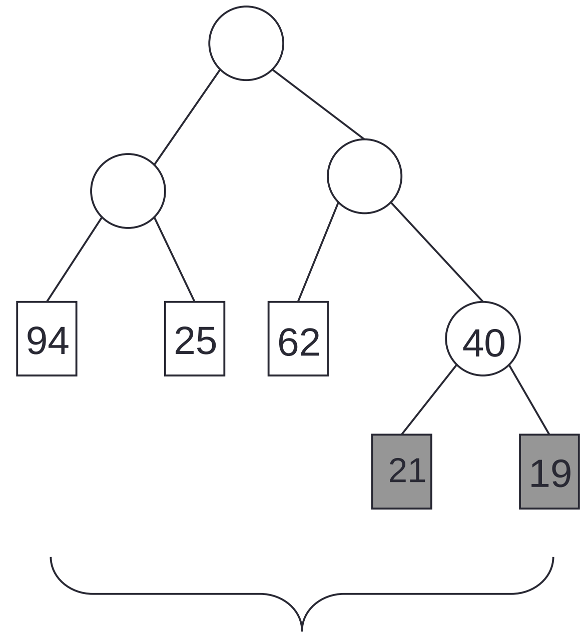


equal average length, hence, Huffman is optimal.

Basic step for the transformation





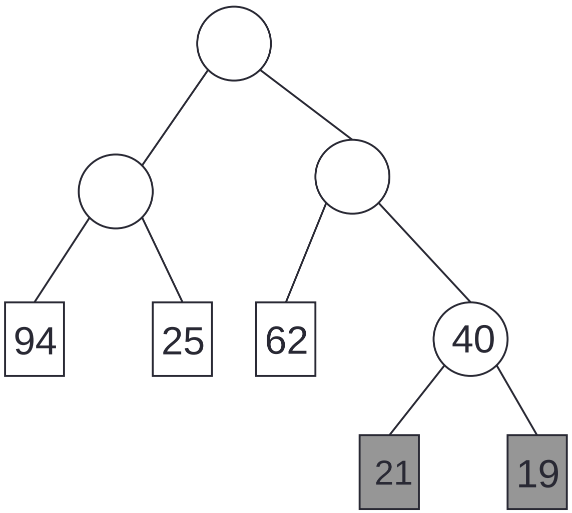


The change will not increase average length **$L_C(S)$**

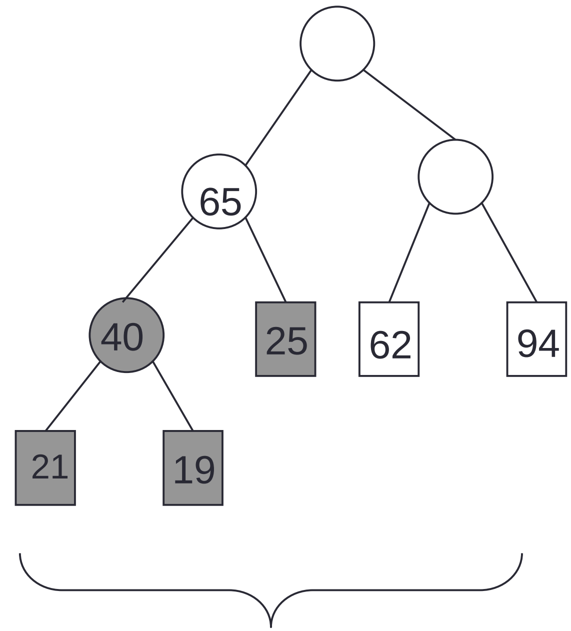
In the Huffman tree, the two leaves with the **lowest frequency** appear as **sibling leaves with maximum depth**

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

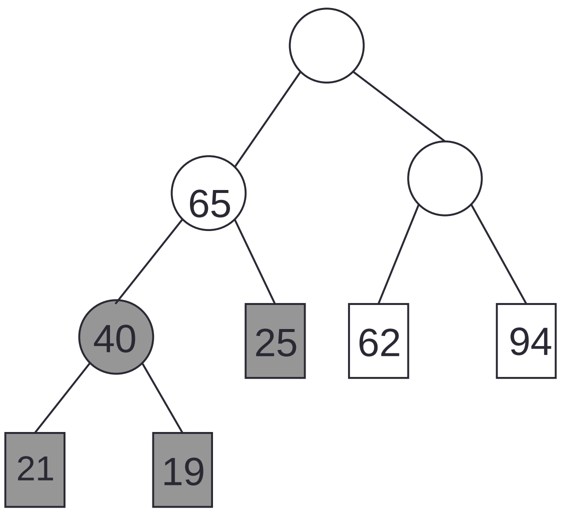


The change will not increase average length $L_C(S)$

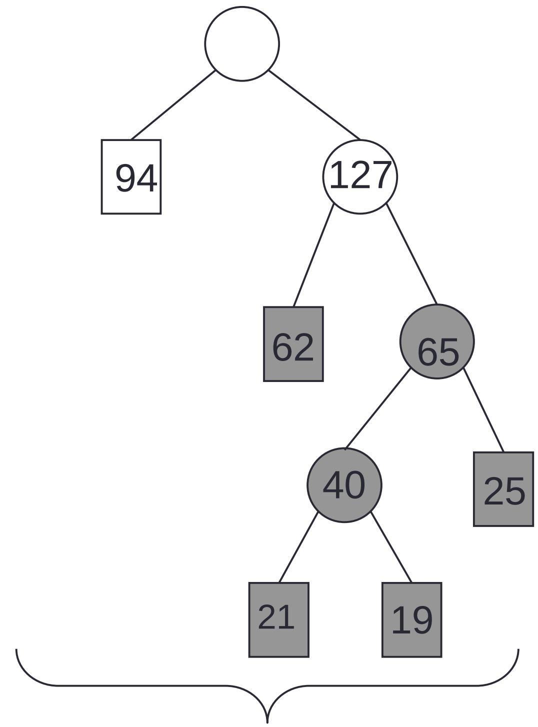


This tree has the **same set of “leaves”** as the one we get after the **second** step of the algorithm.

Basic step for the transformation



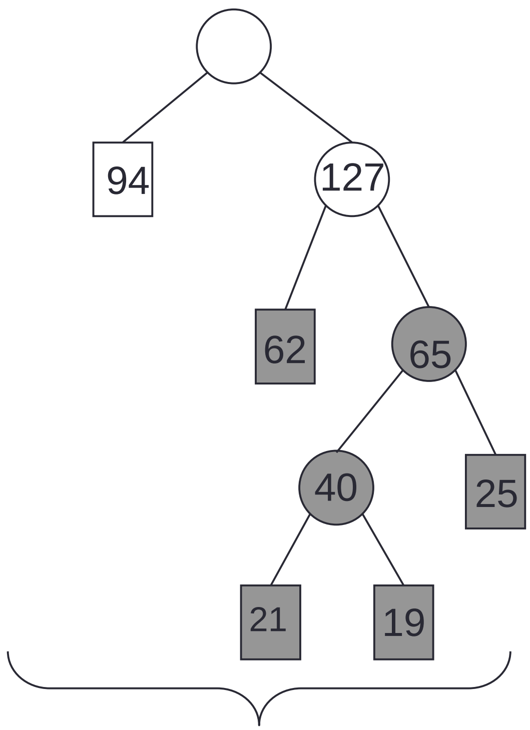
The change will not increase average length $L_C(S)$



This tree has the **same set of “leaves”** as the one we get after the **third** 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 $\mathbf{L_C(S)}$.
- The $\mathbf{L_C(S)}$ of Huffman tree is **no greater than** that of any code tree.
- Huffman code has **the minimum average character length** $\mathbf{L_C(S)}$.



This is a Huffman Tree.

Quick exercise

3. Given a set of characters A, B, C, D and their corresponding frequencies:

Character	A	B	C	D
Frequency	5	10	7	3

Compute the average character length of the Huffman code.