Huffman Coding Procedure

INTRODUCTION

- Developed by D.A.Huffman in 1952
- Optimal prefix code
- Lossless Data Compression
- Variable Length Coding
- Widely used in mainstream compression formats such as JPEG,PNG,MP3,ZIP

Data Compression

Given a collection of letters, numbers, or other symbols, find the most efficient method to represent them using a binary code.

Data Compression

Given a collection of letters, numbers, or other symbols, find the most efficient method to represent them using a binary code.

Text!

T e x t !

R

G

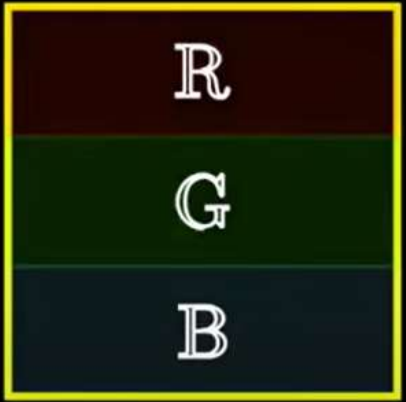
B

Data Compression

Given a collection of letters, numbers, or other symbols, find the most efficient method to represent them using a binary code.

Text!

T e x t !



R

G

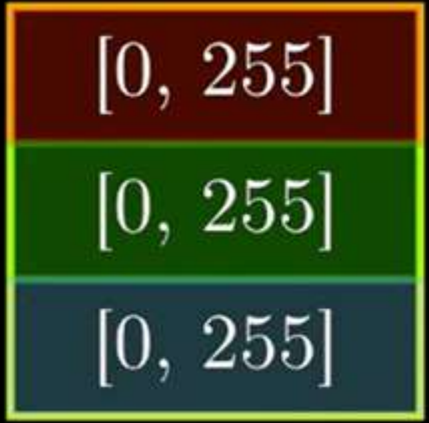
B

Data Compression

Given a collection of letters, numbers, or other symbols, find the most efficient method to represent them using a binary code.

Text!

T e x t !



[0, 255]

[0, 255]

[0, 255]

Data Compression

Given a collection of letters, numbers, or other symbols, find the most efficient method to represent them using a binary code.

Text!

T e x t !

0 1 0 1 1 1 1 1

0 1 0 1 1 1 1 0

1 0 0 1 1 0 1 0

Huffman Code

variable length coding

fixed length coding

- Consider the following short text:

Eerie eyes seen near lake.

- Count up the occurrences of **all characters** in the text

a	10
b	101
c	110
c	111

1 0 1 0 1 1 0 1 1 1

Ahmed Fawzy

1 0 1 0 1 1 0 1 1 1

Huffman Code

Eerie eyes seen near lake.

- What **characters** are present?

E e r i space y s n a l k .

Huffman Code

Eerie eyes seen near lake.

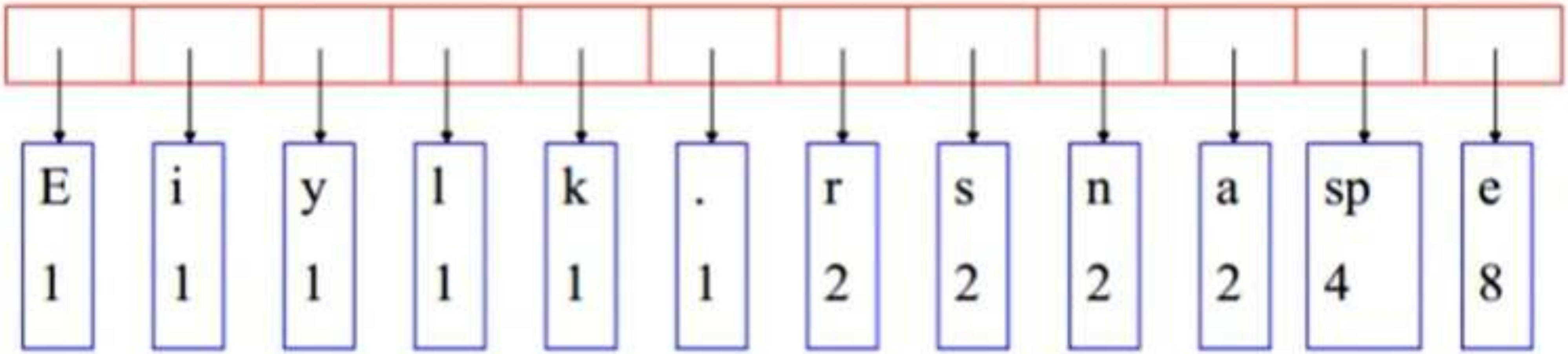
- What is the frequency of each character in the text?

Char	Freq.	Char	Freq.
E	1	*s*	2
e	8	*n*	2
r	2	*a*	2
i	1	*l*	1
space	4	*k*	1
y	1	.	1

Building Tree

- The queue after inserting all nodes

Eerie eyes seen near lake.

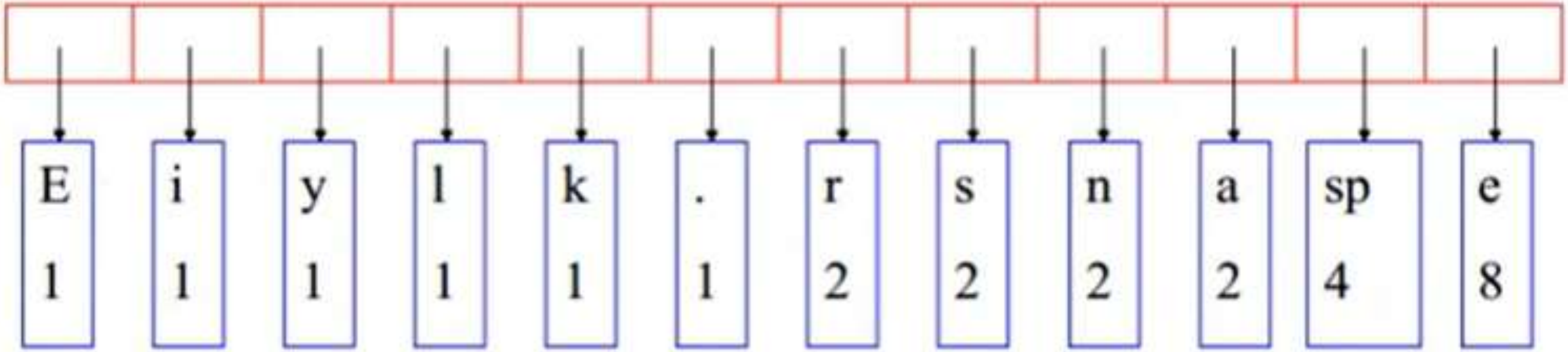


E	i	y	l	k	.	r	s	n	a	sp	e
1	1	1	1	1	1	2	2	2	2	4	8

Building Tree

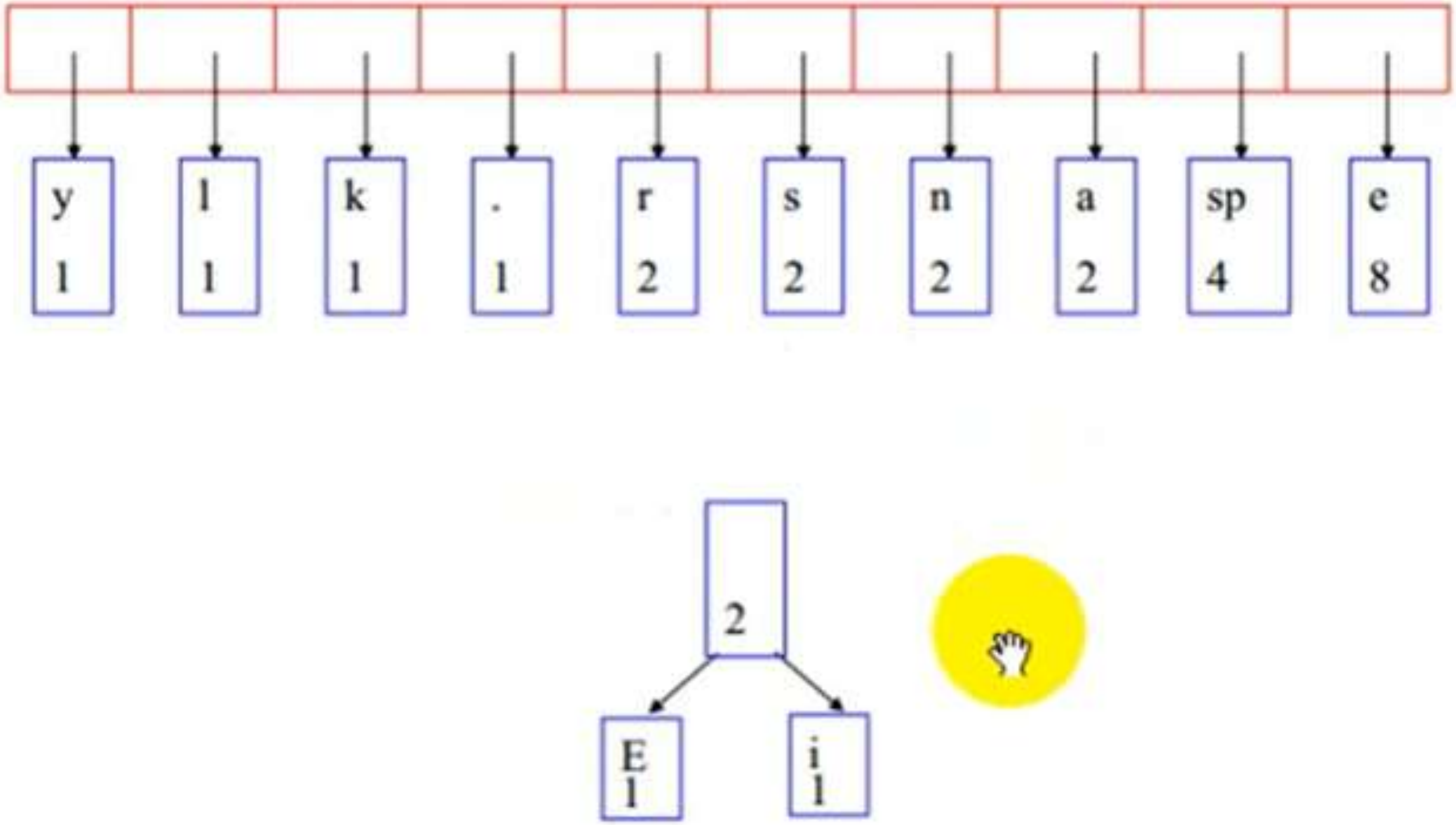
- The queue after inserting all nodes **Eerie eyes seen near lake.**

w:2

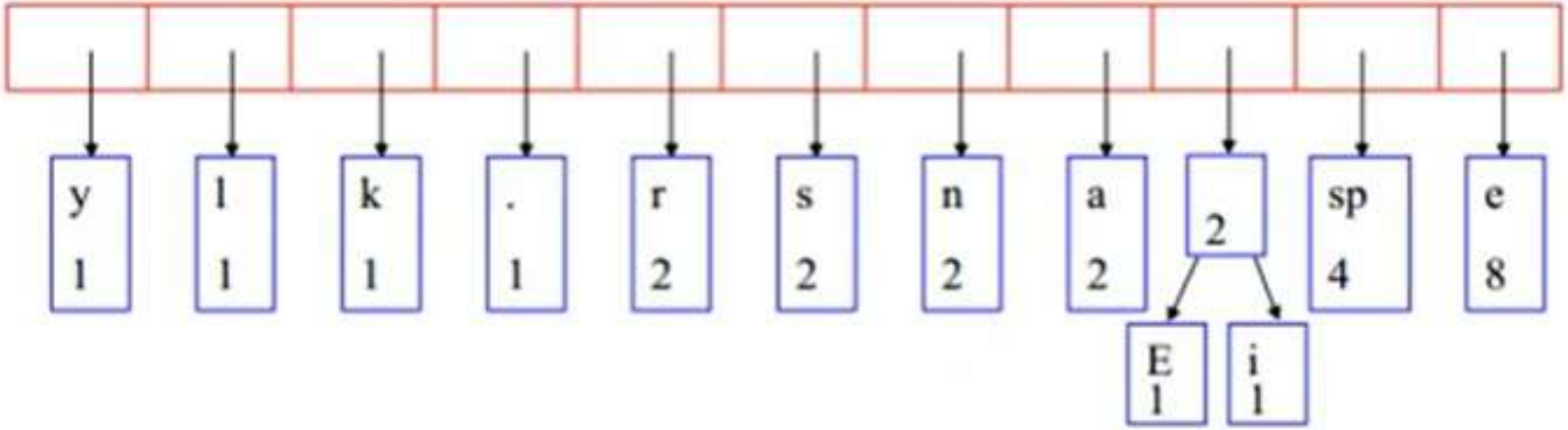


E	i	y	l	k	.	r	s	n	a	sp	e
1	1	1	1	1	1	2	2	2	2	4	8

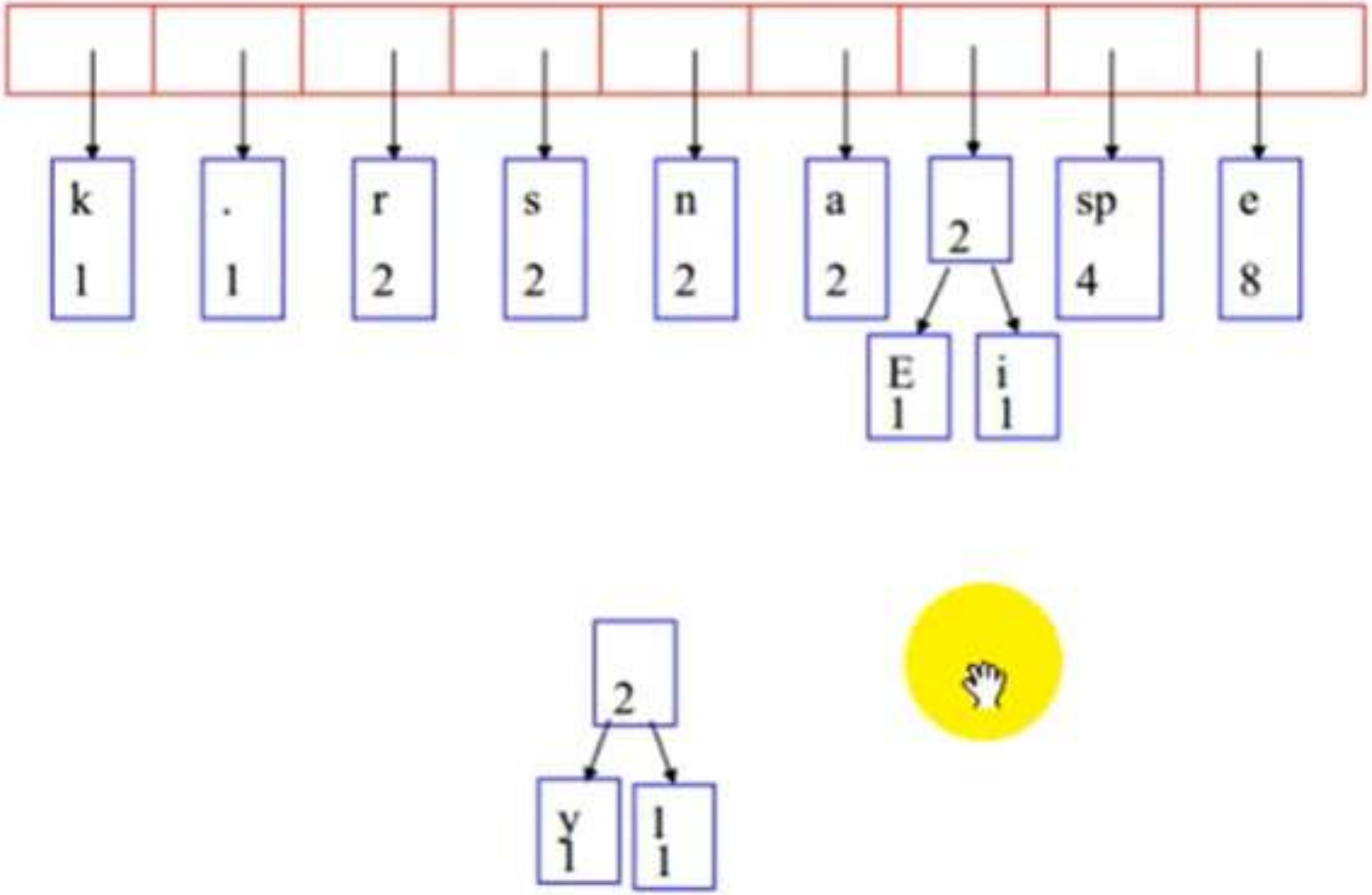
Building Tree



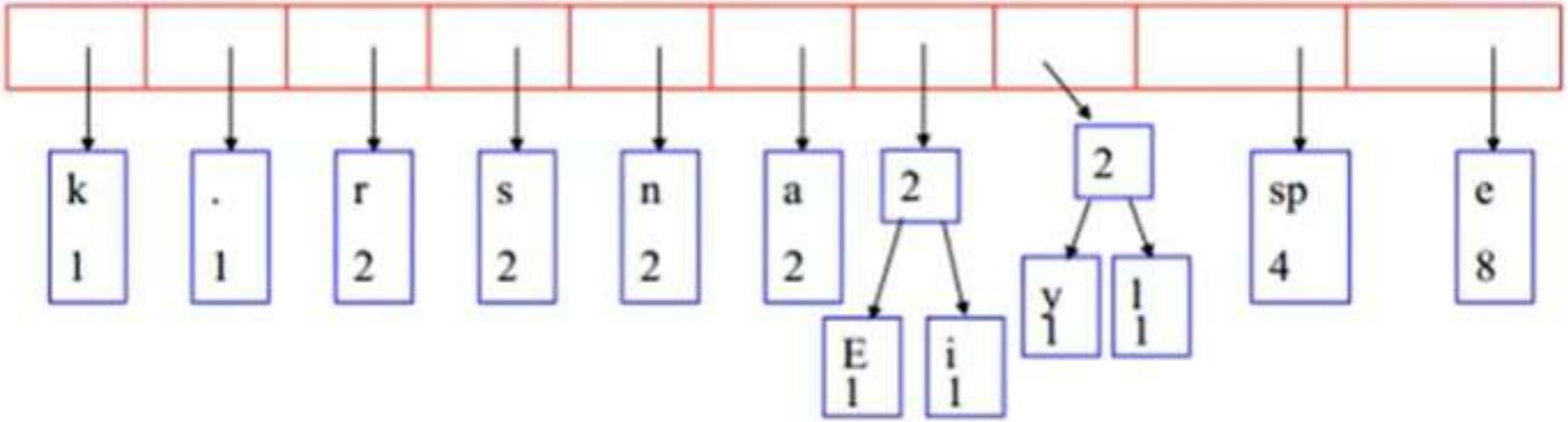
Building Tree



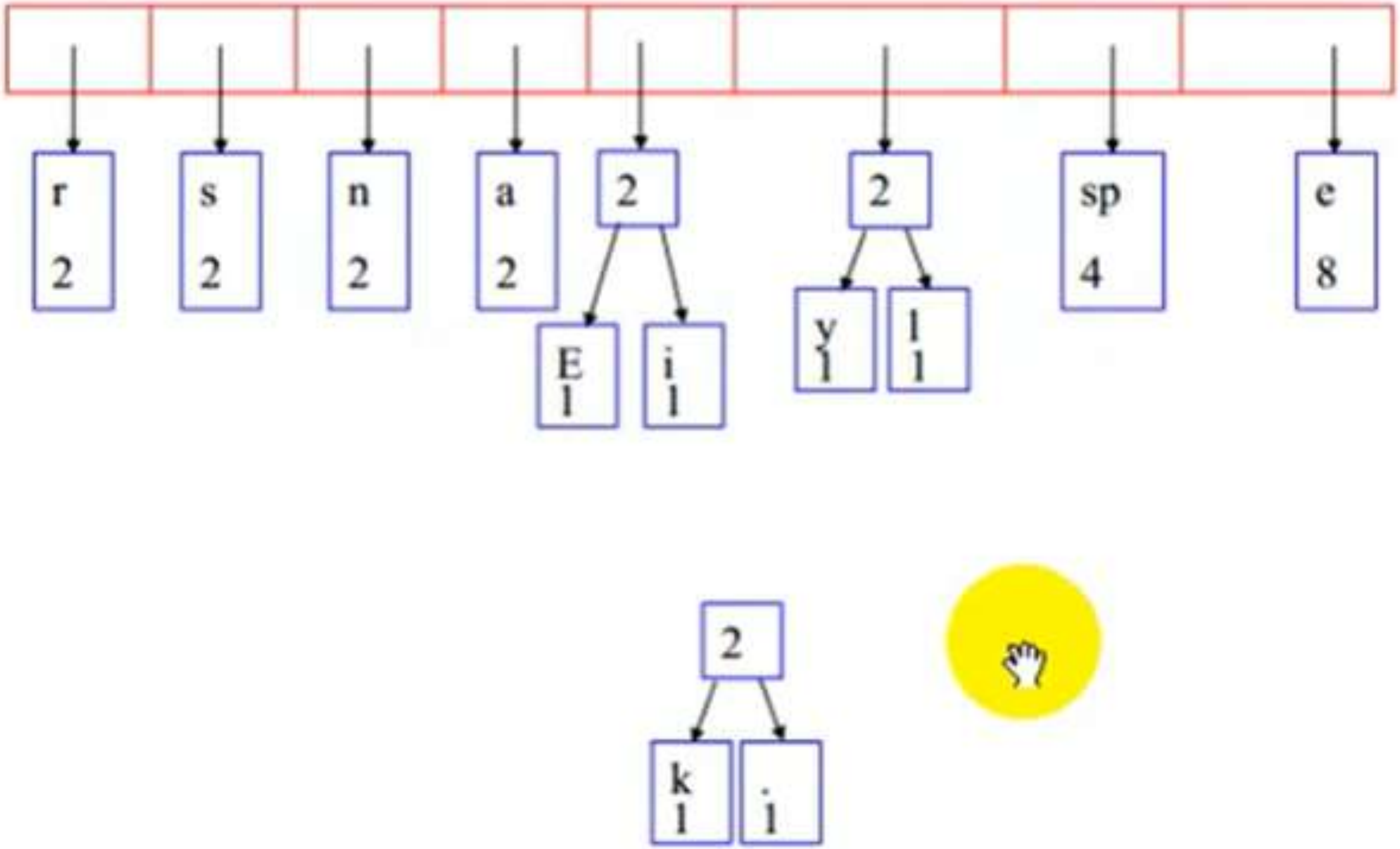
Building Tree



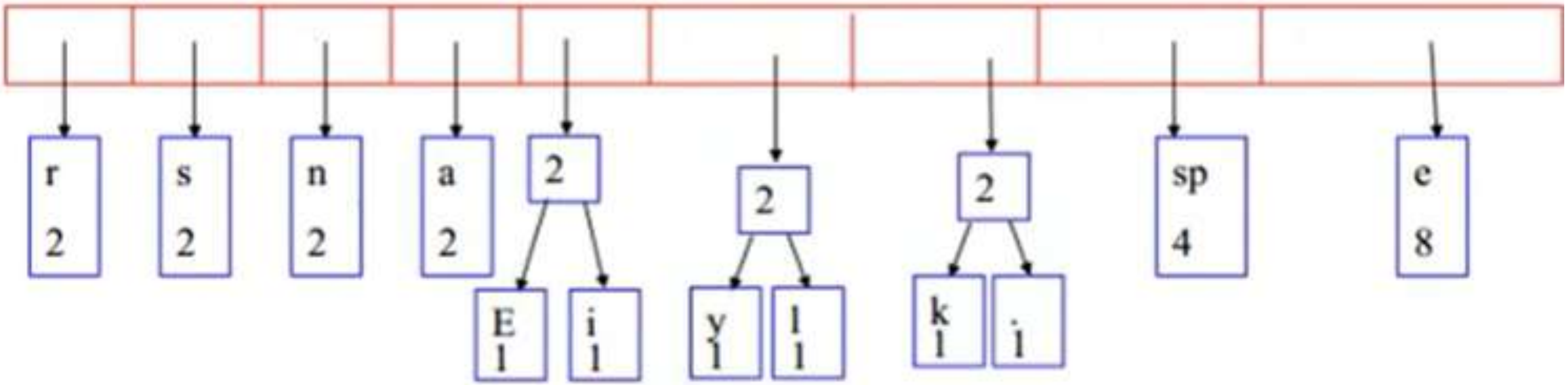
Building Tree



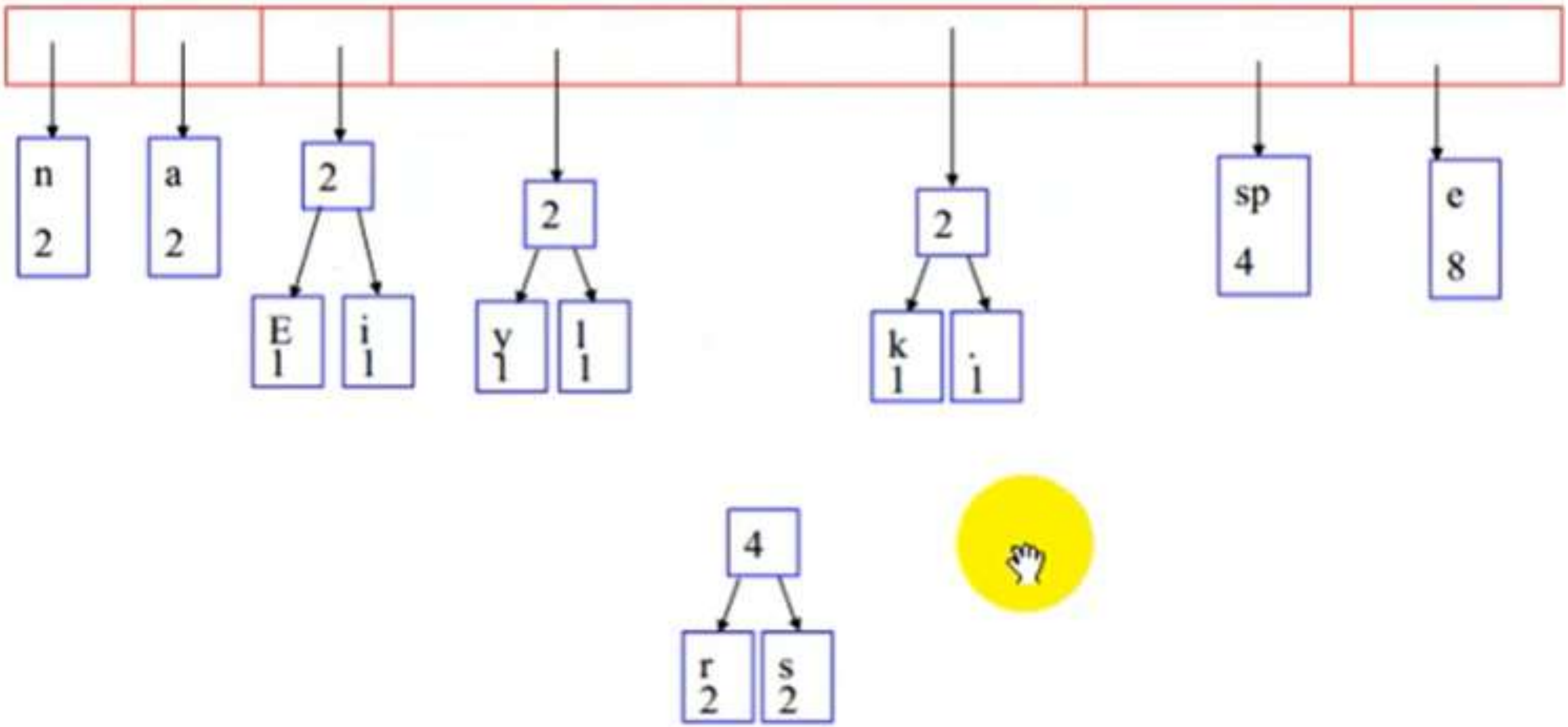
Building Tree



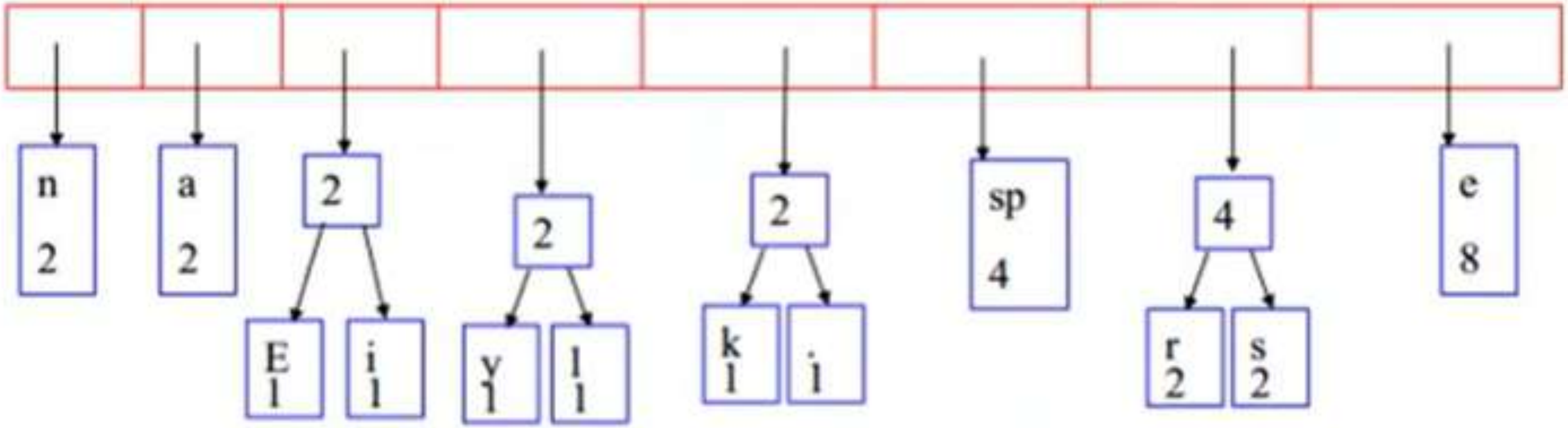
Building Tree



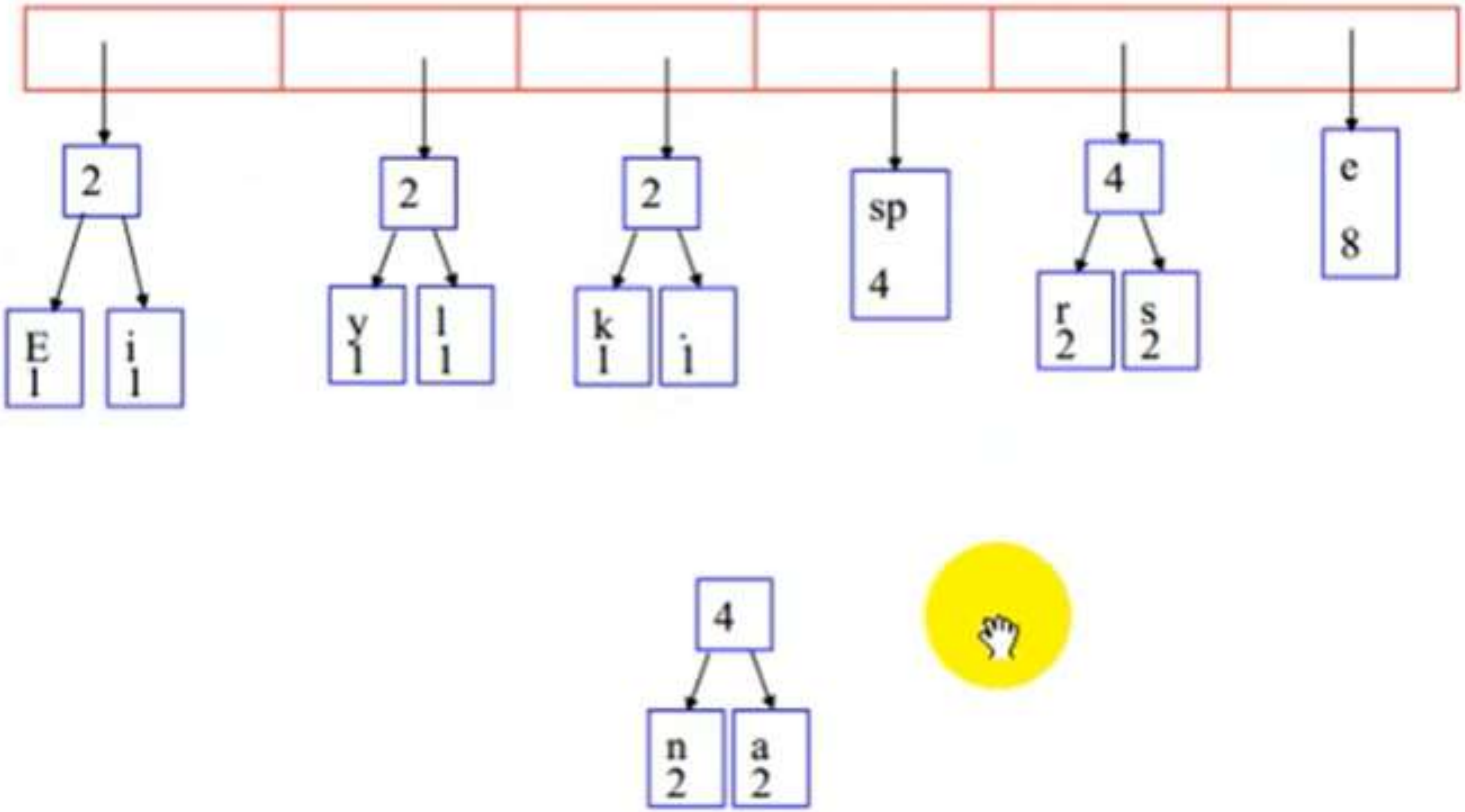
Building Tree



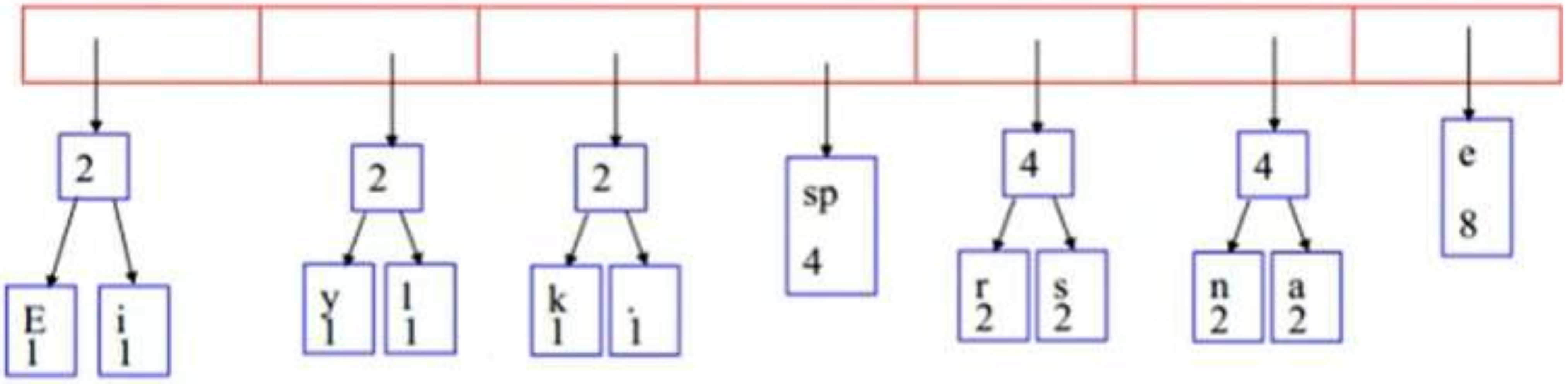
Building Tree



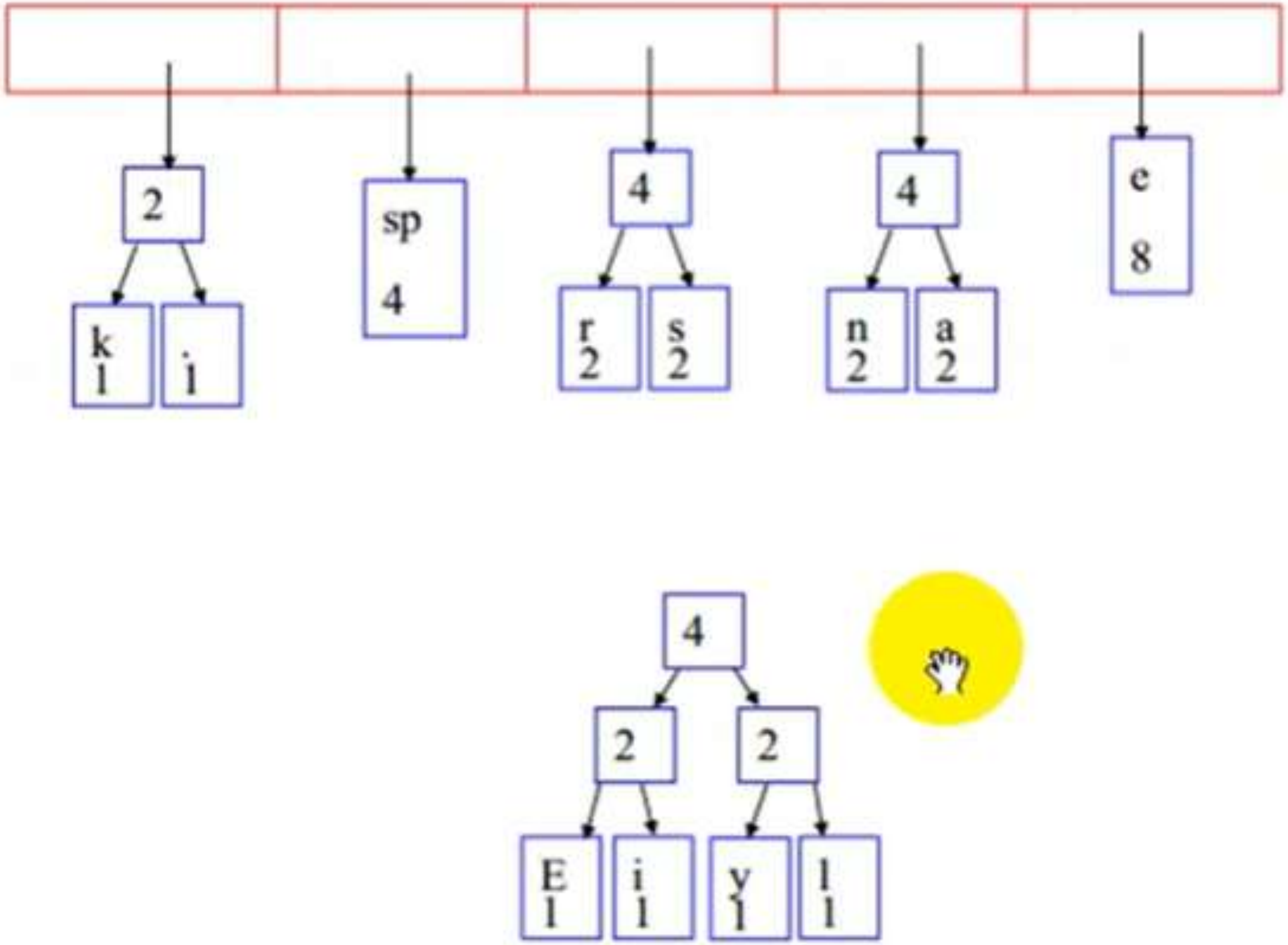
Building Tree



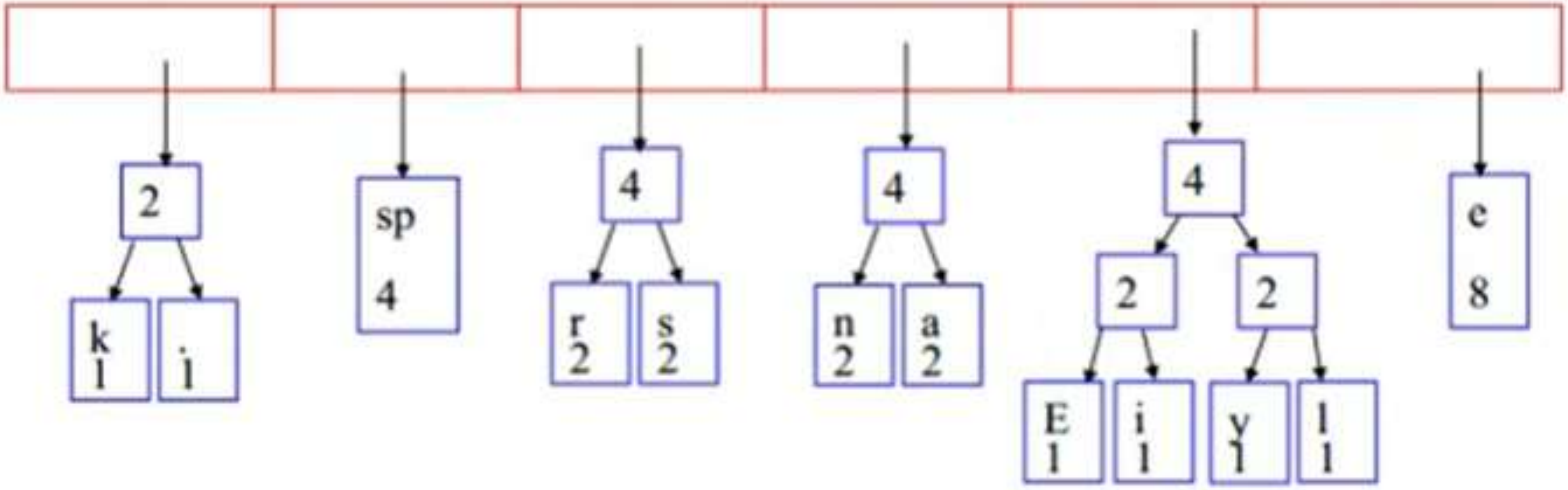
Building Tree



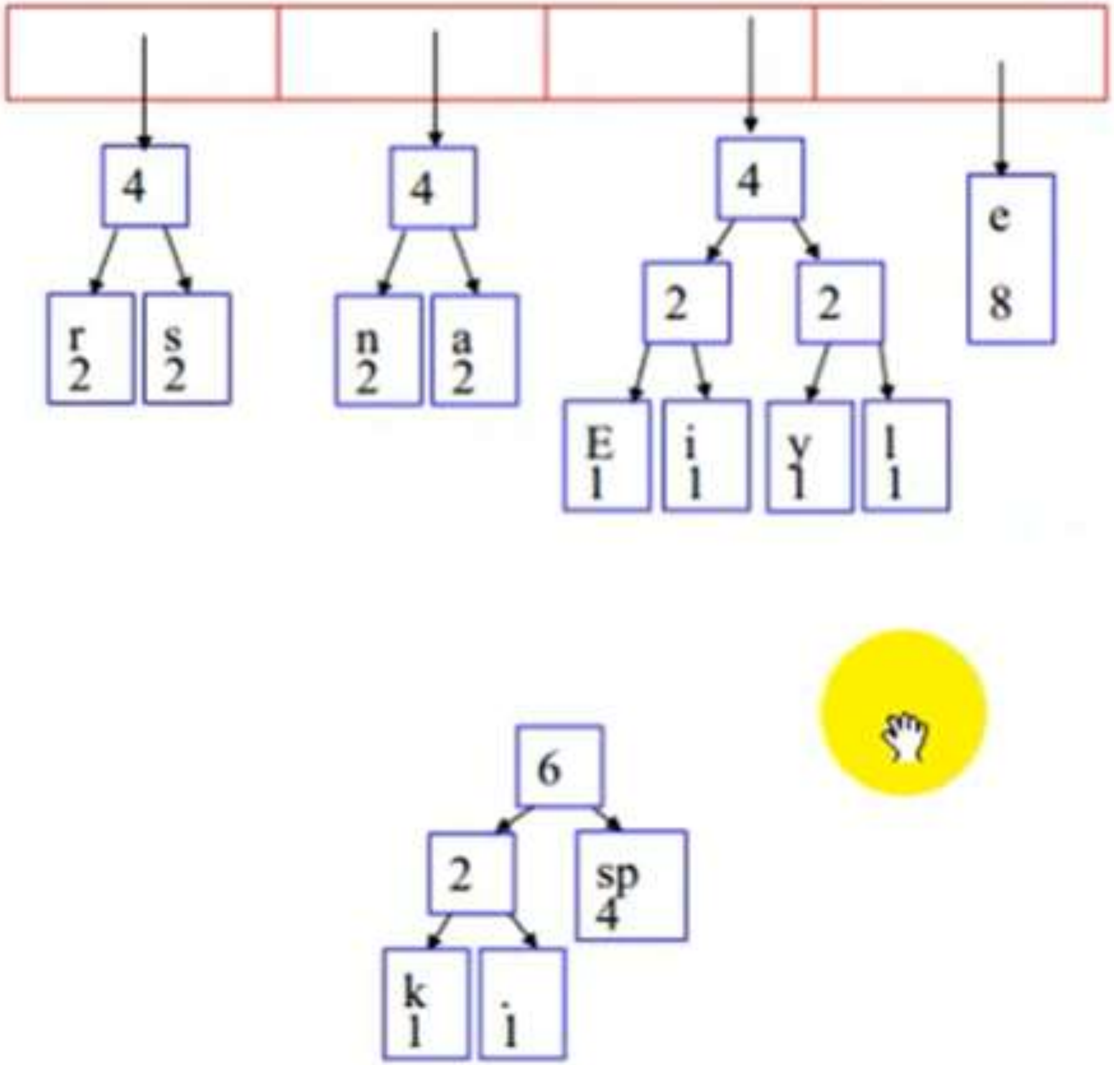
Building Tree



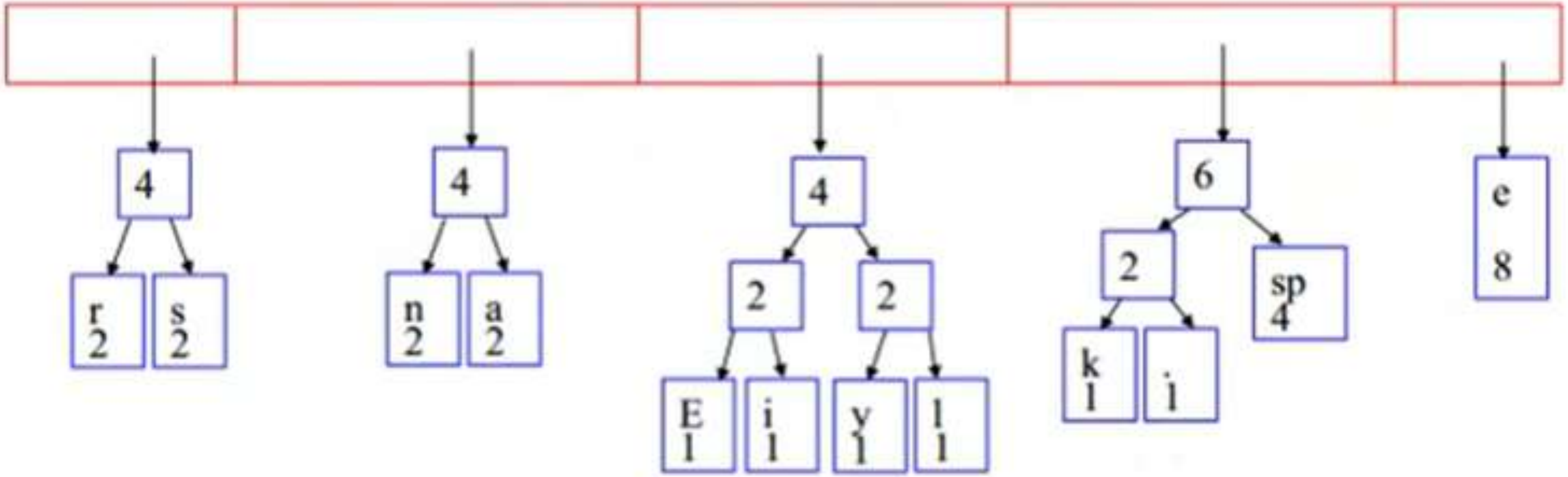
Building Tree



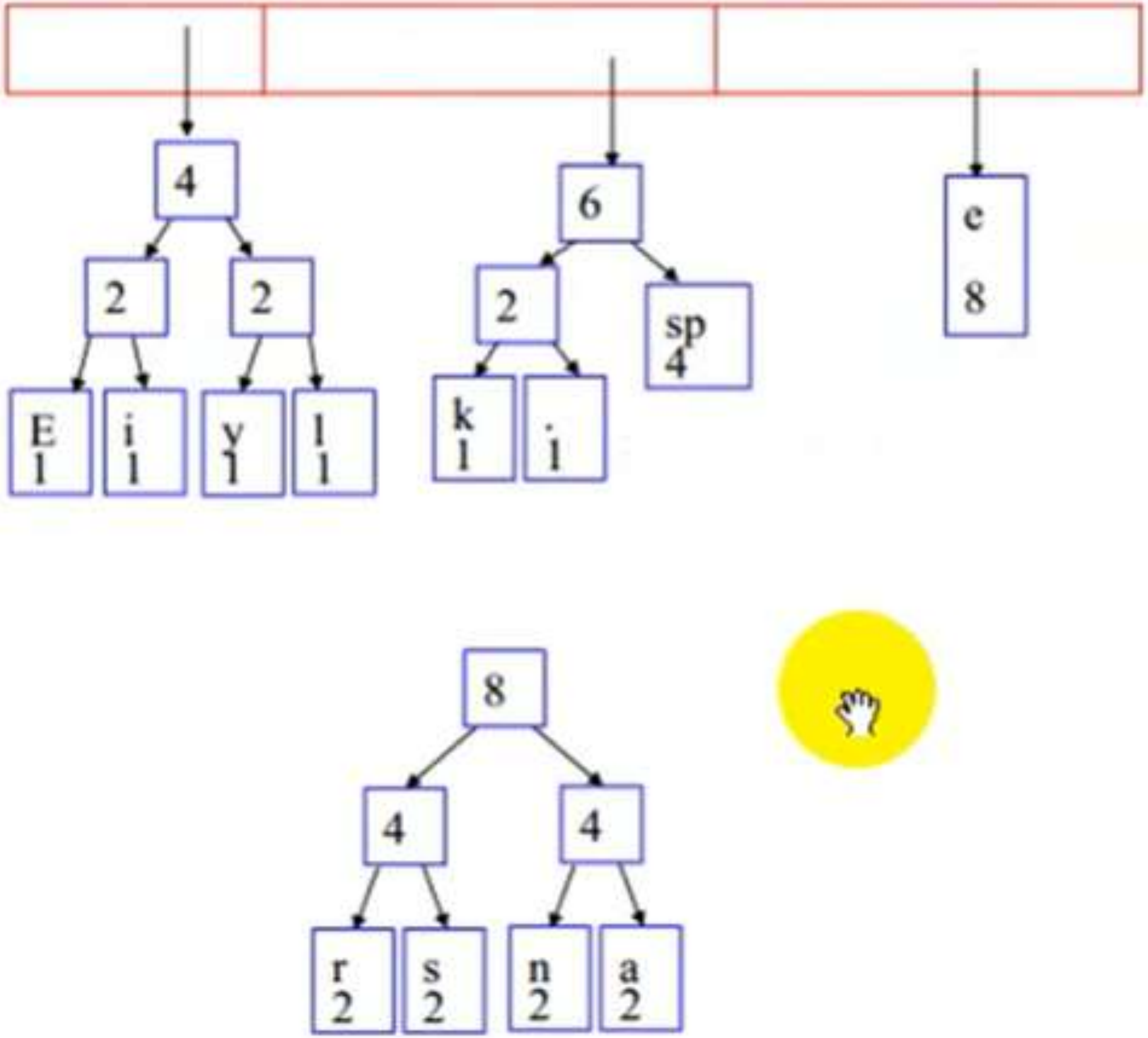
Building Tree



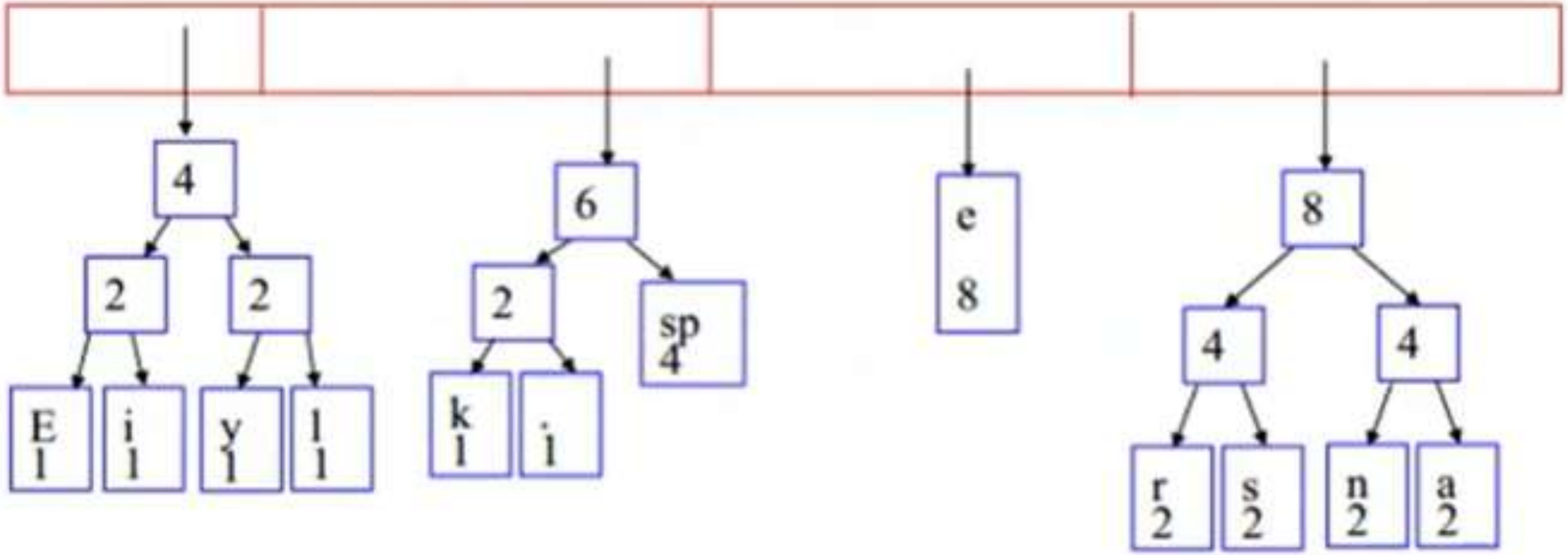
Building Tree



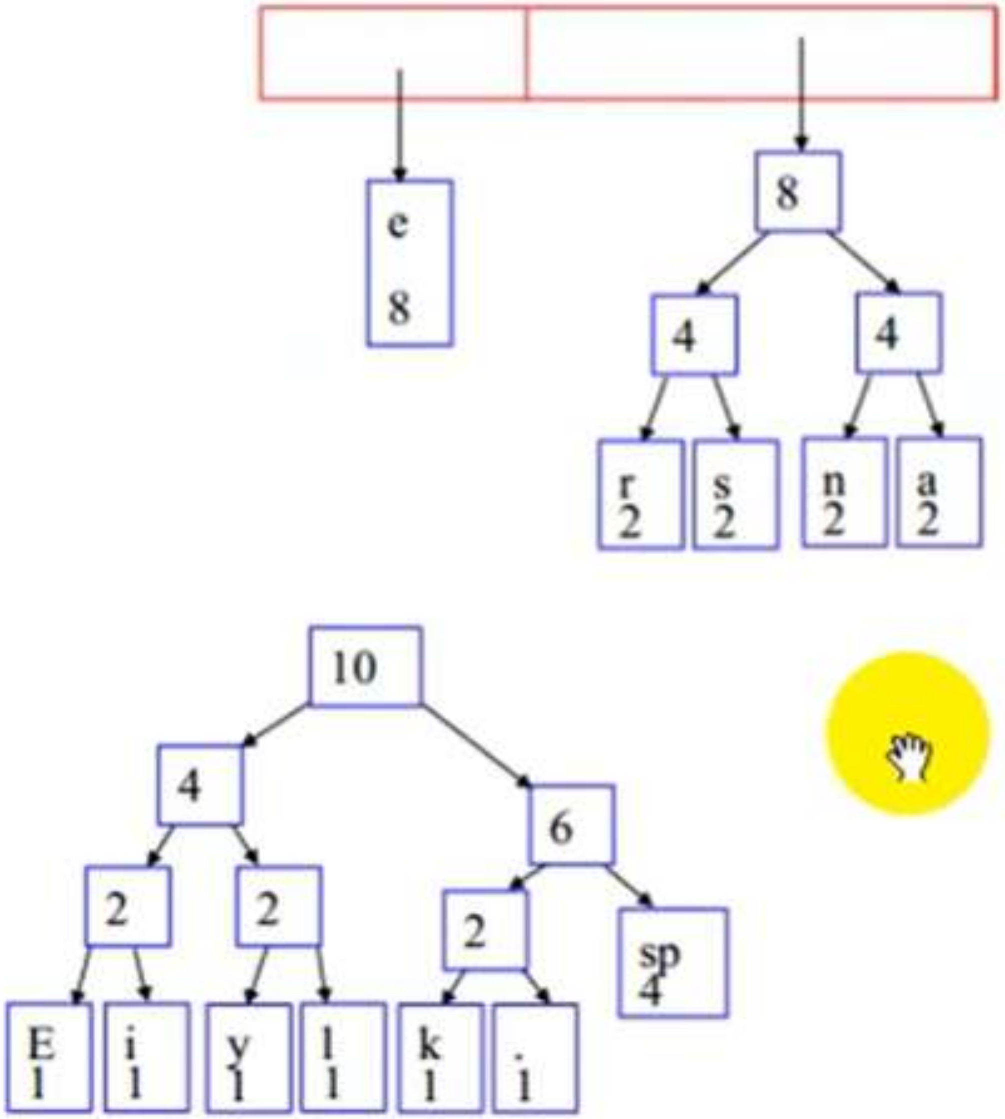
Building Tree



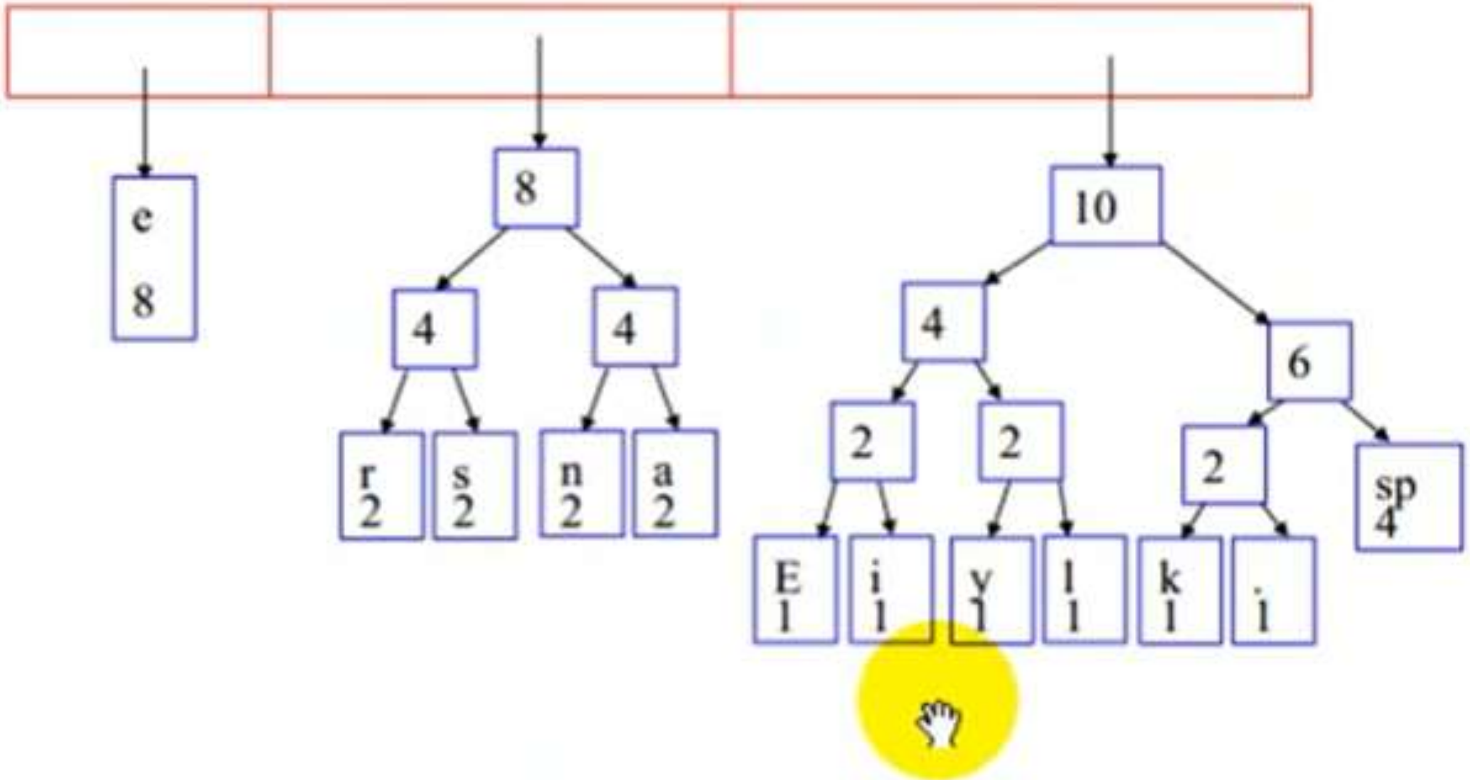
Building Tree



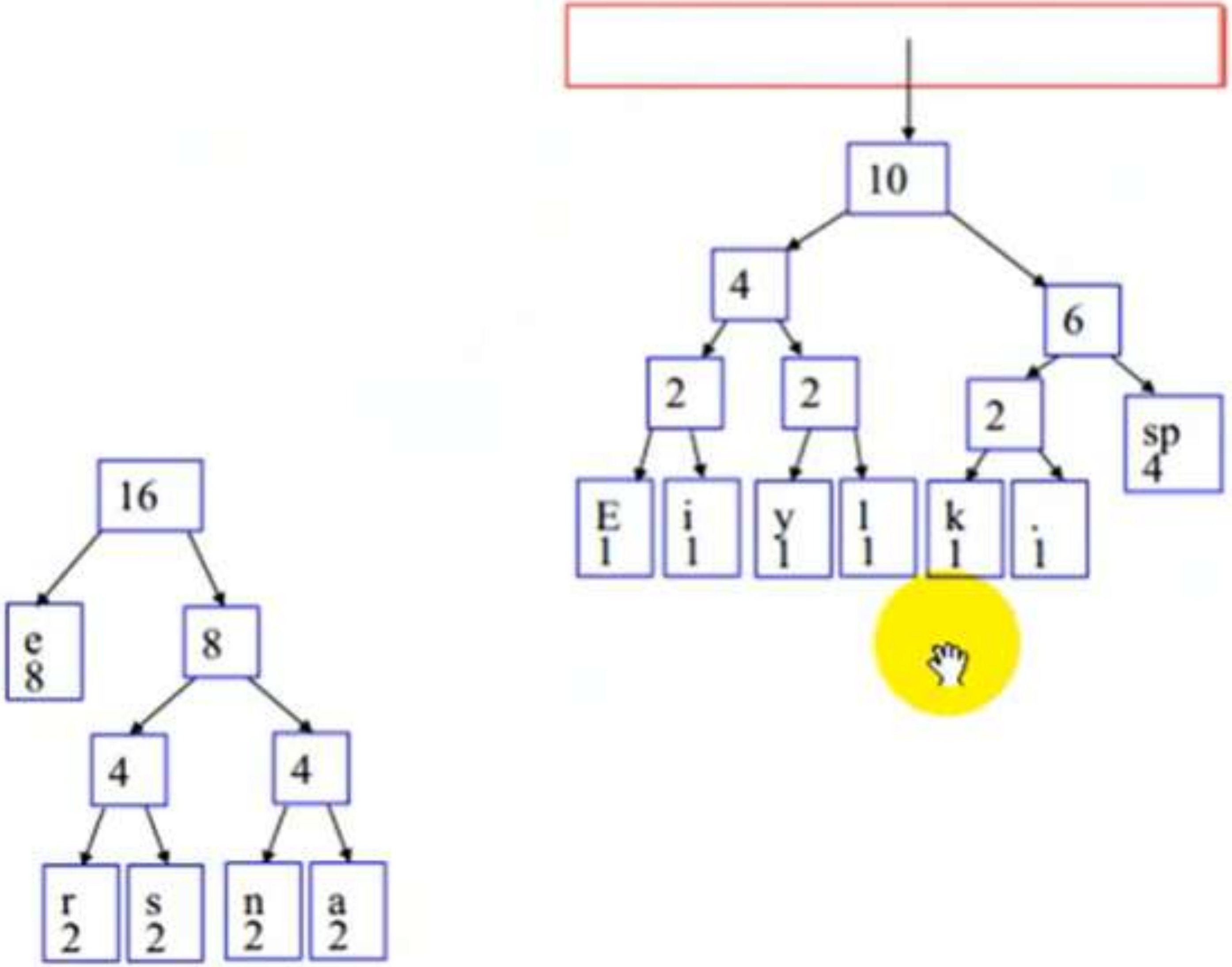
Building Tree



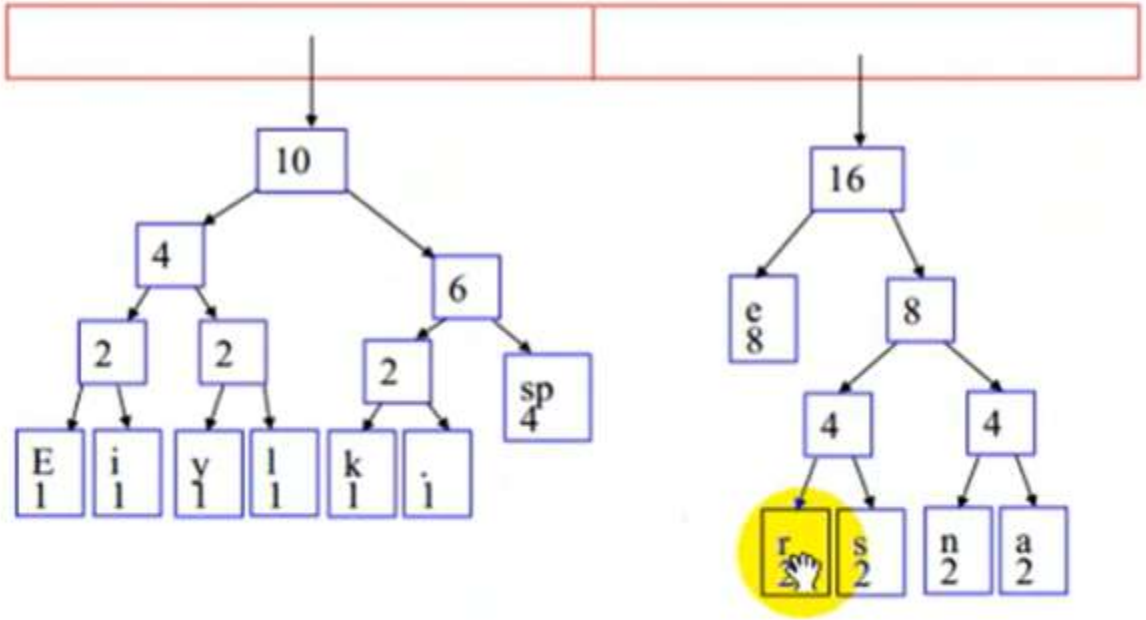
Building Tree



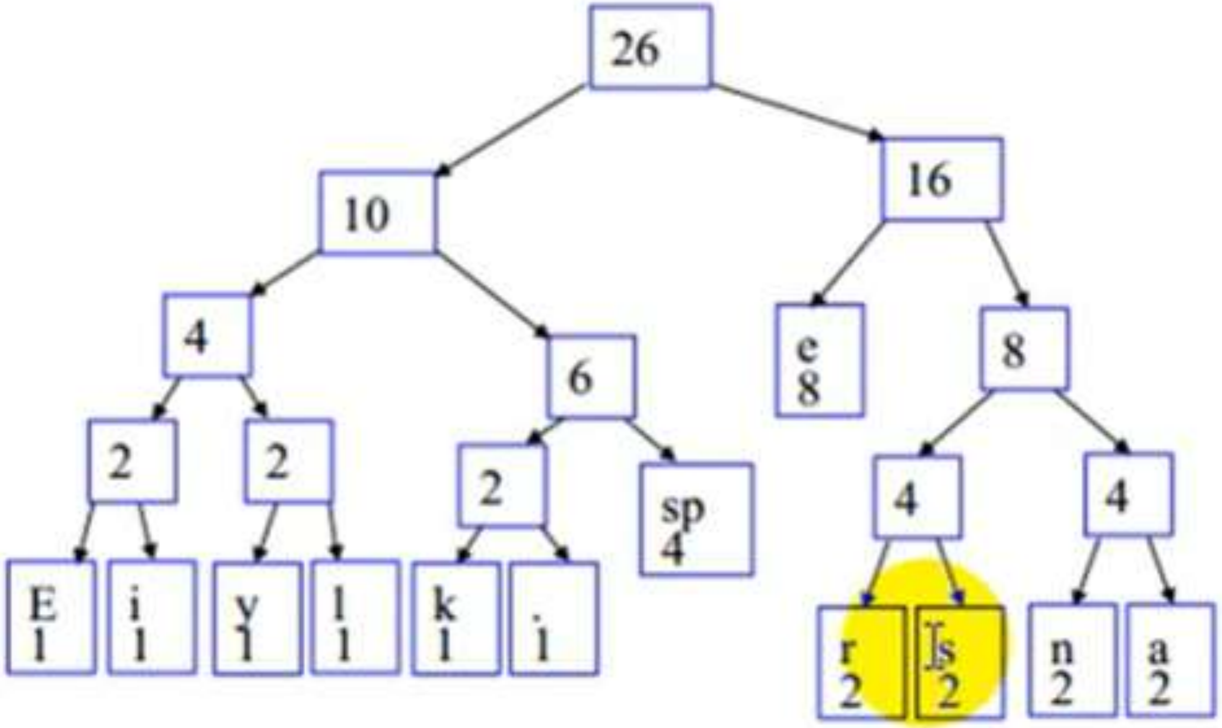
Building Tree



Building Tree

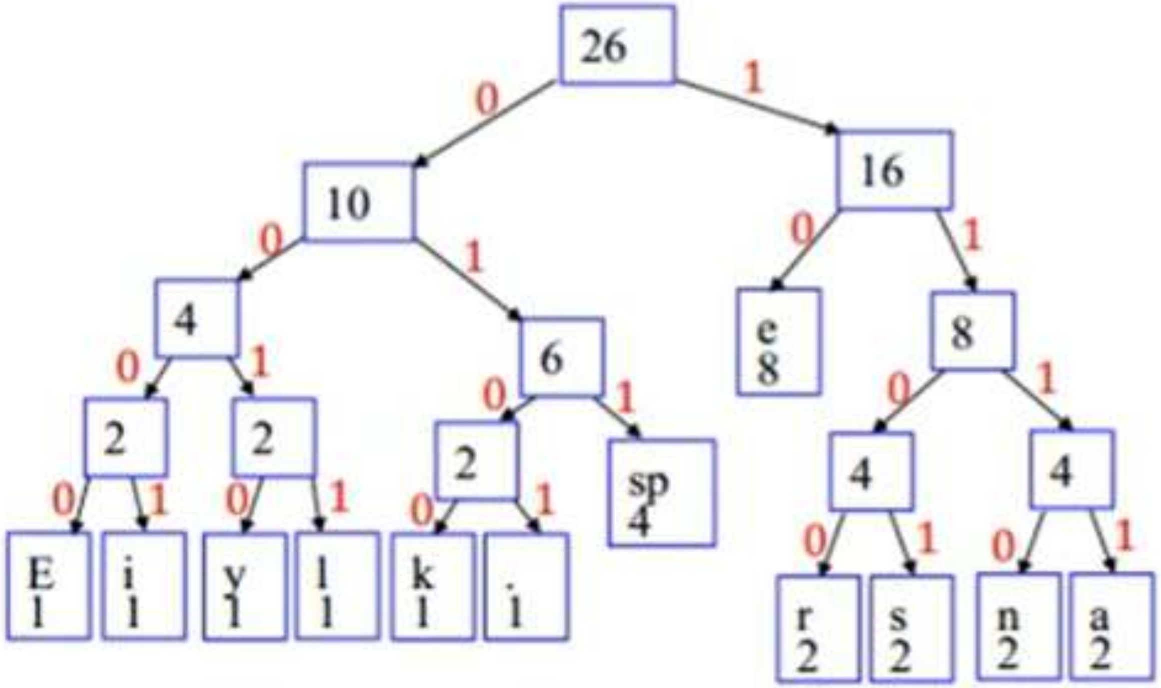


Building Tree



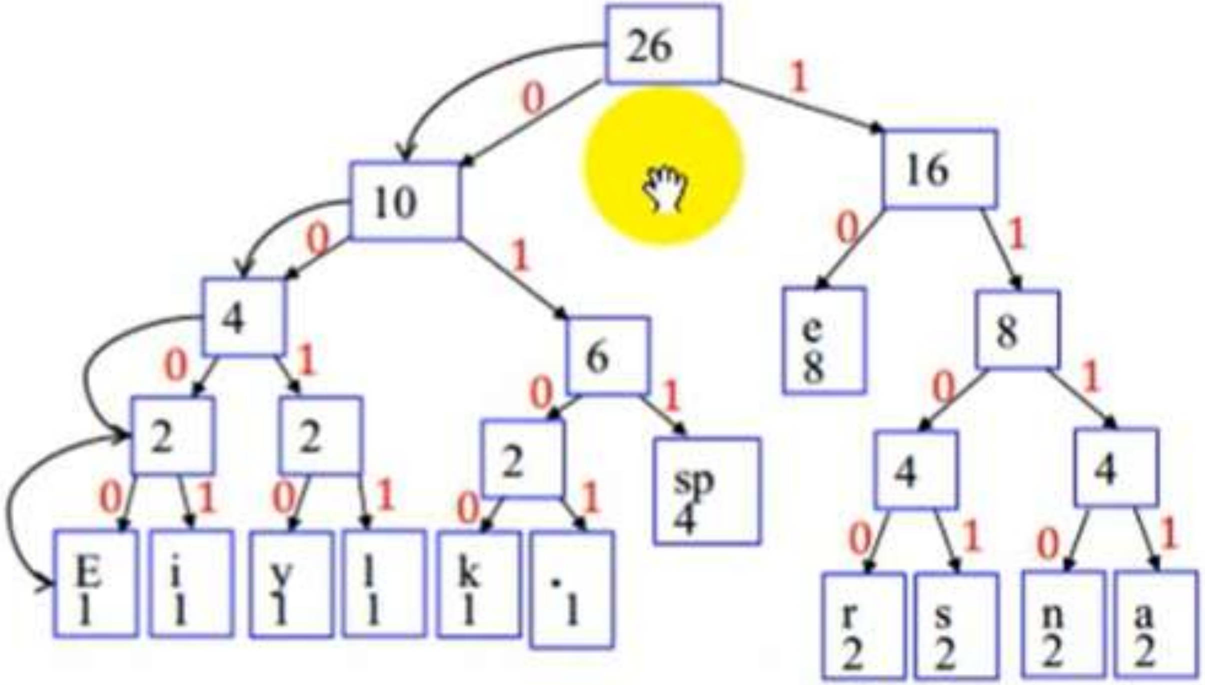
Building Tree

- $Left\ branch\ = 0$
- $Right\ branch\ = 1$



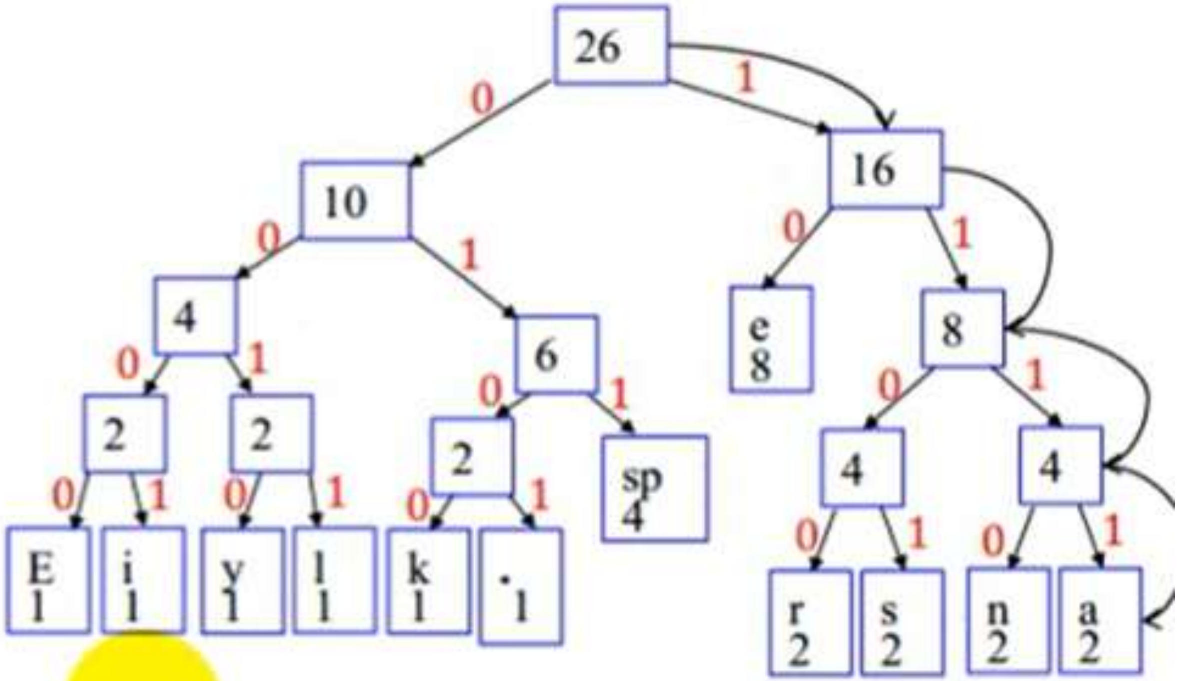
Traverse Tree for Codes

Char	Code
E	
i	
y	
l	
k	
.	
space	
e	
r	
s	
n	
a	



Traverse Tree for Codes

Char	Code
E	0000
i	0001
y	0010
l	0011
k	0100
.	0101
space	011
e	10
r	1100
s	1101
n	1110
a	1111



Encoding Text

Char	Code
E	0000
i	0001
y	0010
l	0011
k	0100
.	0101
space	011
e	10
r	1100
s	1101
n	1110
a	1111

Eerie eyes seen near lake.

Code:

0000

Encoding Text

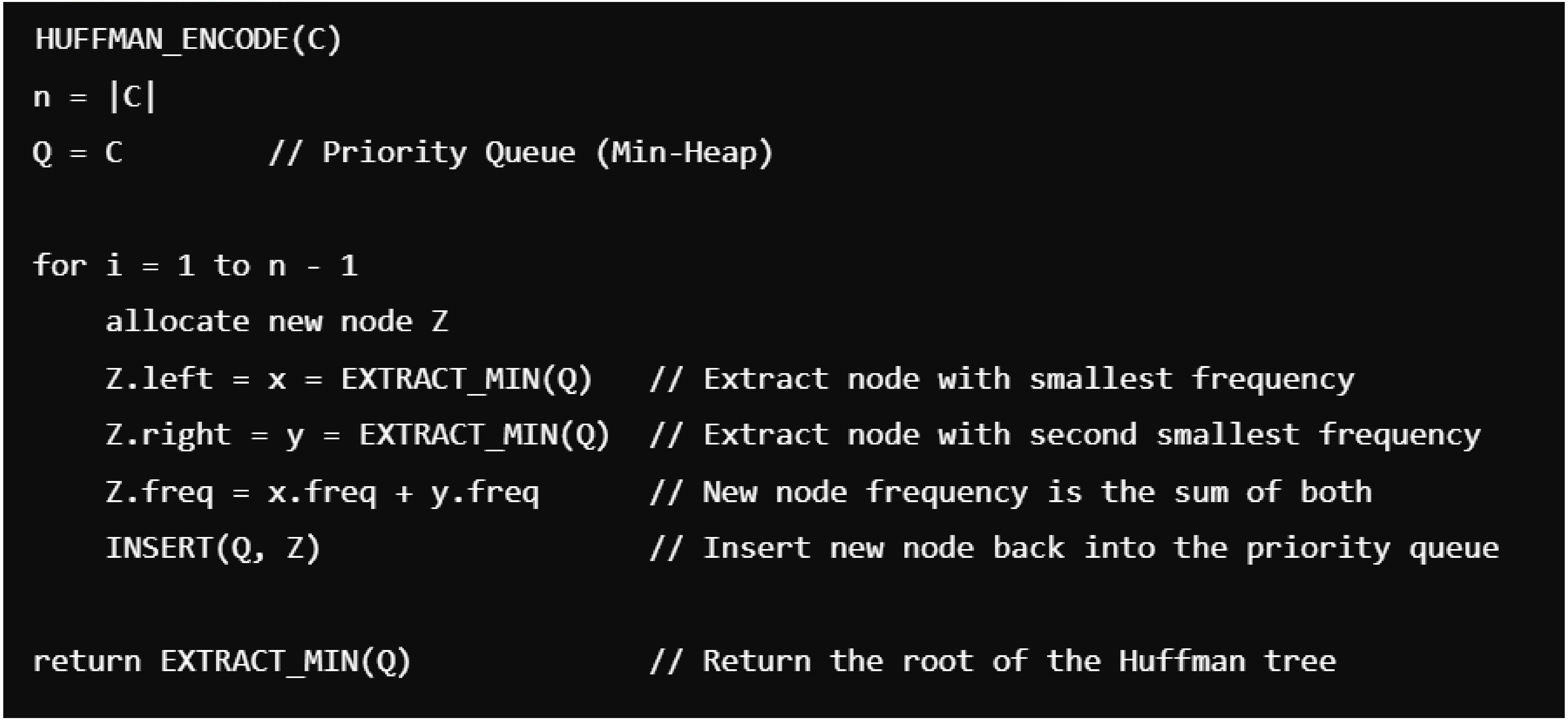
Char	Code
E	0000
i	0001
y	0010
l	0011
k	0100
.	0101
space	011
e	10
r	1100
s	1101
n	1110
a	1111

Eerie eyes seen near lake.

Code:

000010110000011001110001010
11010111101101011100111110
101111110001100111111010010
0101

Huffman Code



```
HUFFMAN_ENCODE(C)
n = |C|
Q = C        // Priority Queue (Min-Heap)

for i = 1 to n - 1
    allocate new node Z
    Z.left = x = EXTRACT_MIN(Q)   // Extract node with smallest frequency
    Z.right = y = EXTRACT_MIN(Q)  // Extract node with second smallest frequency
    Z.freq = x.freq + y.freq      // New node frequency is the sum of both
    INSERT(Q, Z)                  // Insert new node back into the priority queue

return EXTRACT_MIN(Q)             // Return the root of the Huffman tree
```

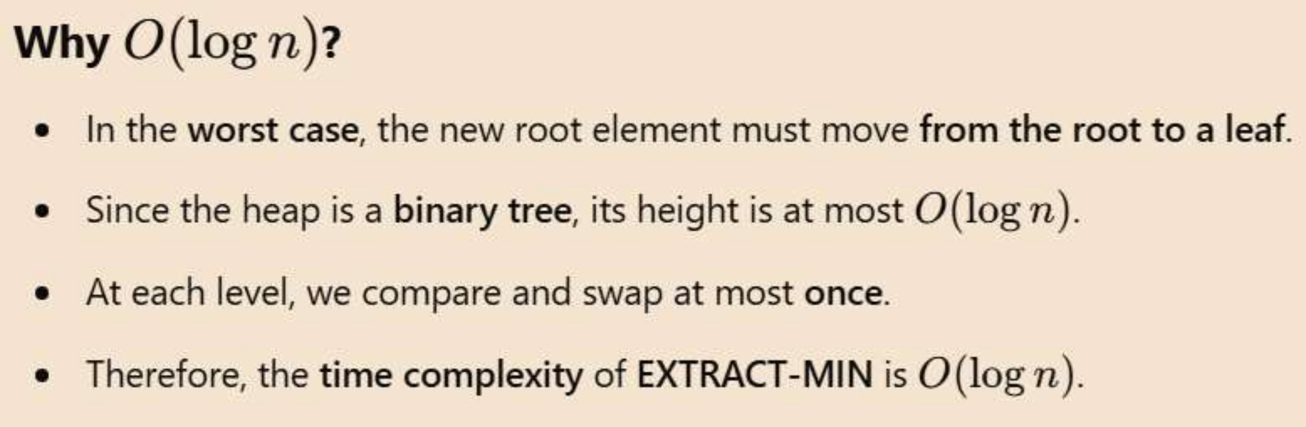
Huffman Code – Analysis

```
$HUFFAMN(C)$
    $n = |C| \longleftarrow 1$
    $Q = C \longleftarrow O(n) \quad BUILD-MIN-HEAP$
    $for \;\; i = 1 \;\; to \;\; n-1 \longleftarrow n-1$
        $allocate\ new\ node \;\; Z \longleftarrow n-1$
        $Z.left = x = EXTRACT-MIN(Q) \leftarrow (n-1)\lg n$
        $Z.right = y = EXTRACT-MIN(Q) \leftarrow (n-1)\lg n$
        $Z.freq = x.freq + y.freq \longleftarrow n-1$
        $INSERT(Q, Z) \longleftarrow (n-1)\lg n$
    $return \;\; EXTRACT-MIN(Q) \longleftarrow \lg n$
```

$\therefore T(n) = O(n \lg n)$

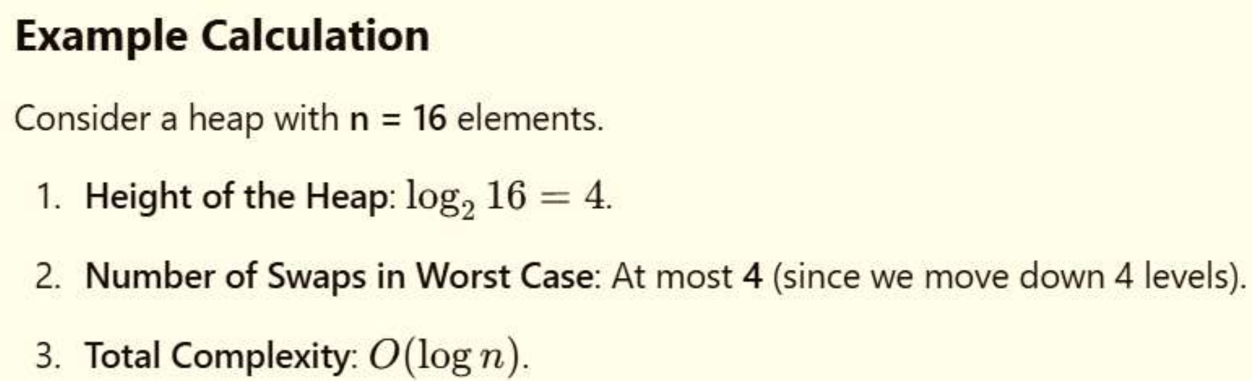
Steps in EXTRACT-MIN(Q)

1. **Removing the Root Element (Minimum)**
   - The minimum element is always at the root.
   - We remove it, which leaves a hole at the root.
2. **Replacing with the Last Element**
   - The last element in the heap is moved to the root position.
3. **Heapify (Heap Reordering Downward)**
   - Since the new root element might not maintain the **min-heap property**, we perform a **heapify-down** (or **percolate-down**) operation.
   - We swap the new root with the smaller of its two children, **recursively**, until the heap property is restored.



Why $O(\log n)$?

- In the **worst case**, the new root element must move **from the root to a leaf**.
- Since the heap is a **binary tree**, its height is at most $O(\log n)$.
- At each level, we compare and swap at most **once**.
- Therefore, the **time complexity** of EXTRACT-MIN is $O(\log n)$.

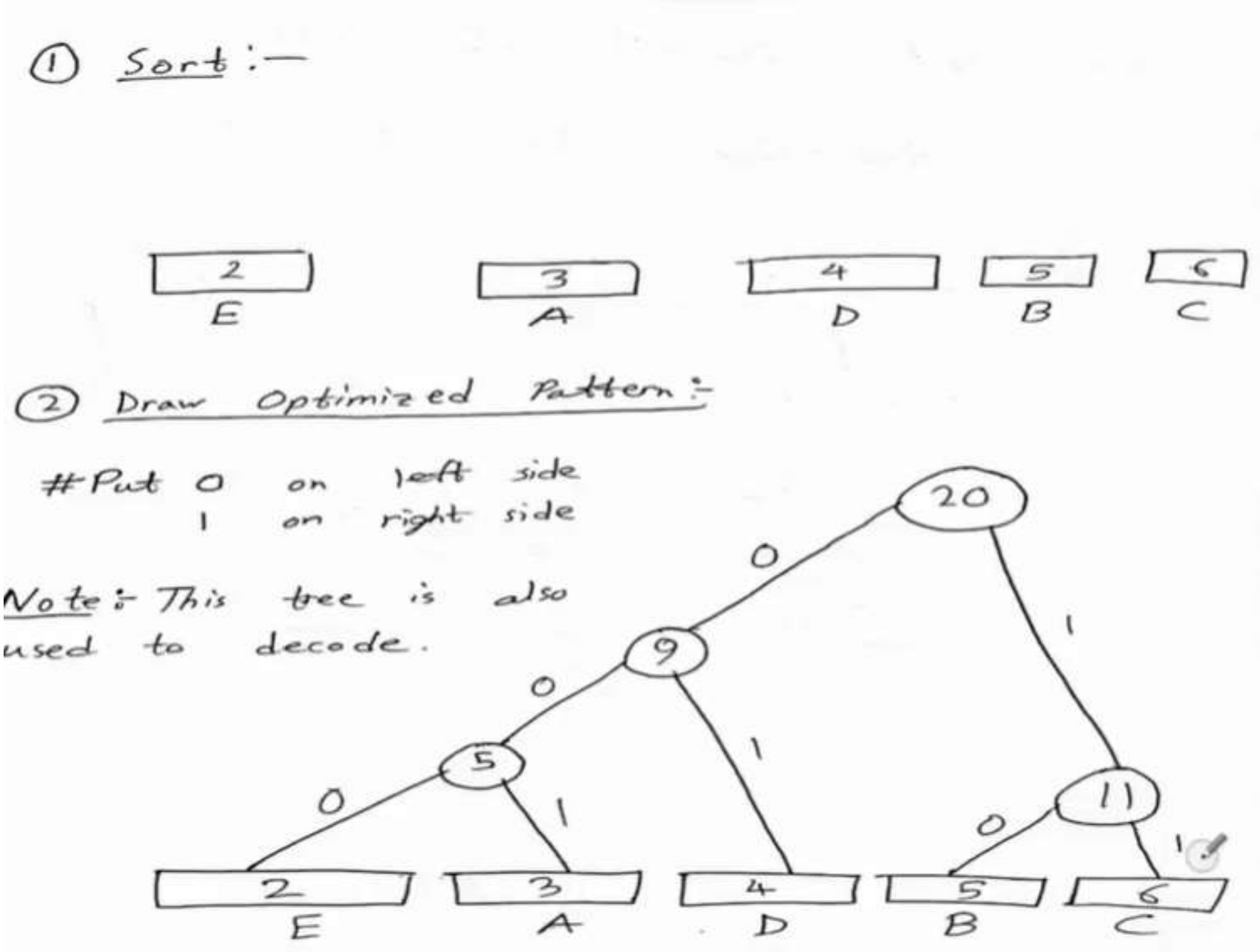


Example Calculation

Consider a heap with **n = 16** elements.

1. **Height of the Heap**: $\log_2 16 = 4$.
2. **Number of Swaps in Worst Case**: At most **4** (since we move down 4 levels).
3. **Total Complexity**: $O(\log n)$.

Hufman Decoding Procedure



① Sort:—

2	3	4	5	6
E	A	D	B	C

② Draw Optimized Pattern:-

#Put 0 on left side
1 on right side

Note:- This tree is also used to decode.

20
0
1
9
0
1
5
0
1
11
0
1
2
E
3
A
4
D
5
B
6
C

Hufman Decoding Procedure

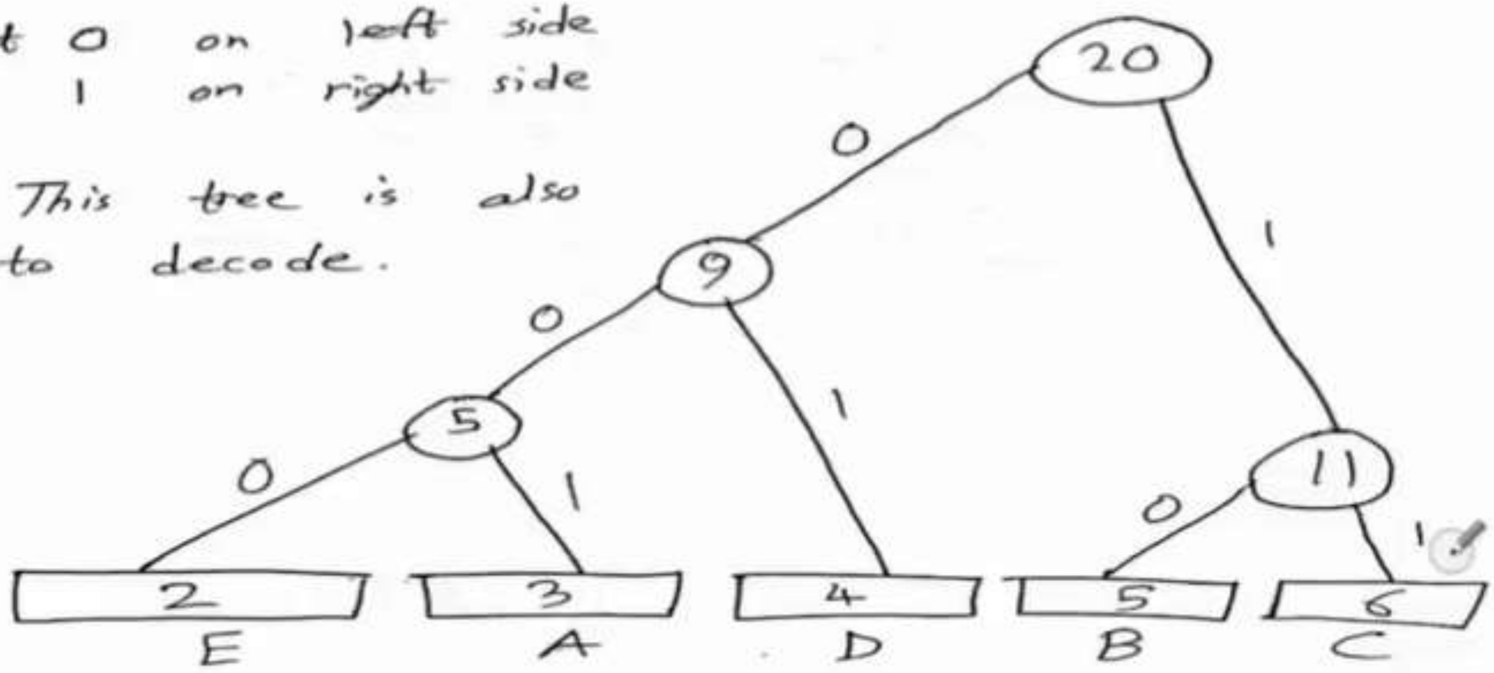
① Sort :—

2	3	4	5	6
E	A	D	B	C

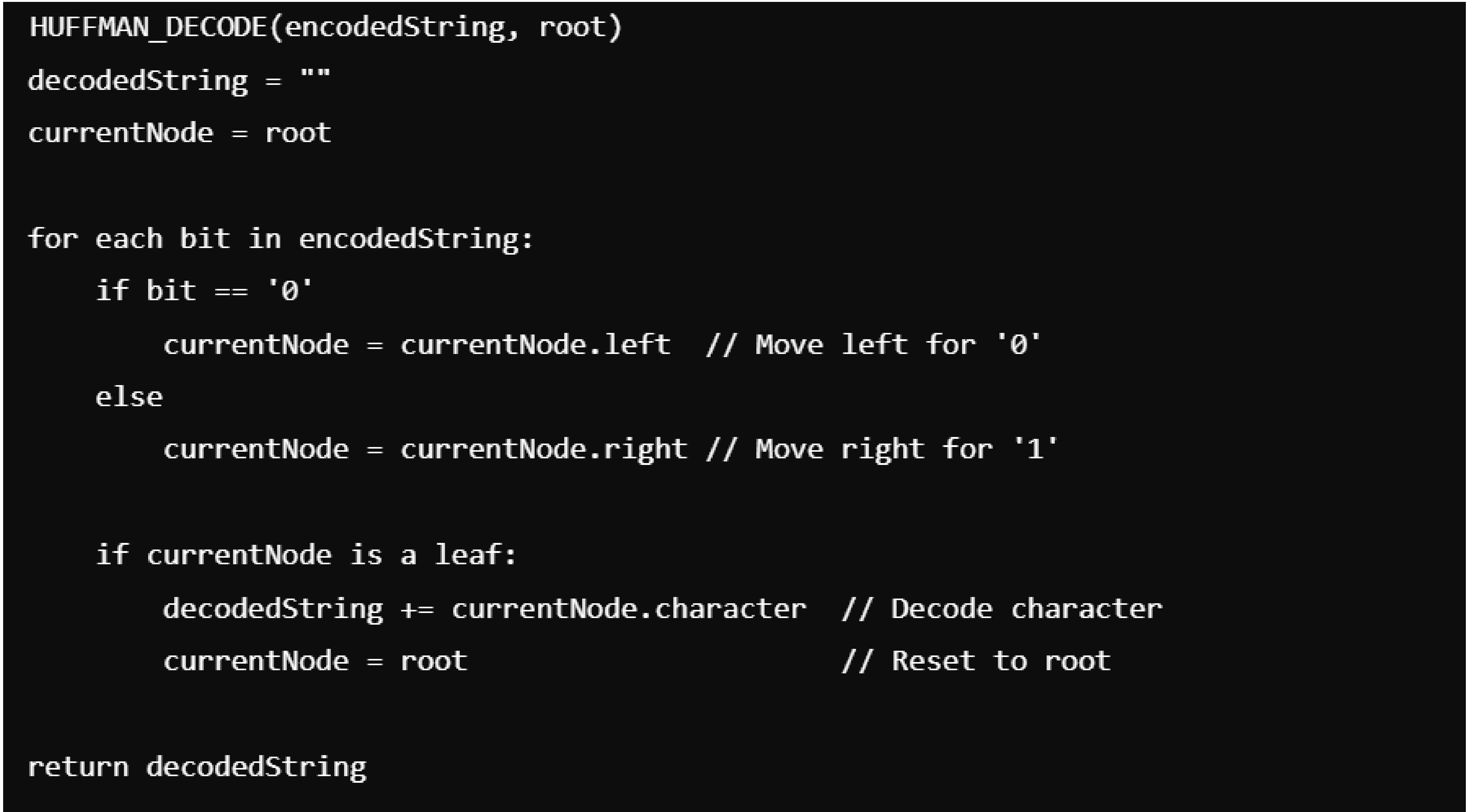
② Draw Optimized Pattern :-

Put 0 on left side
1 on right side

Note :- This tree is also used to decode.



Decoding Algorithm



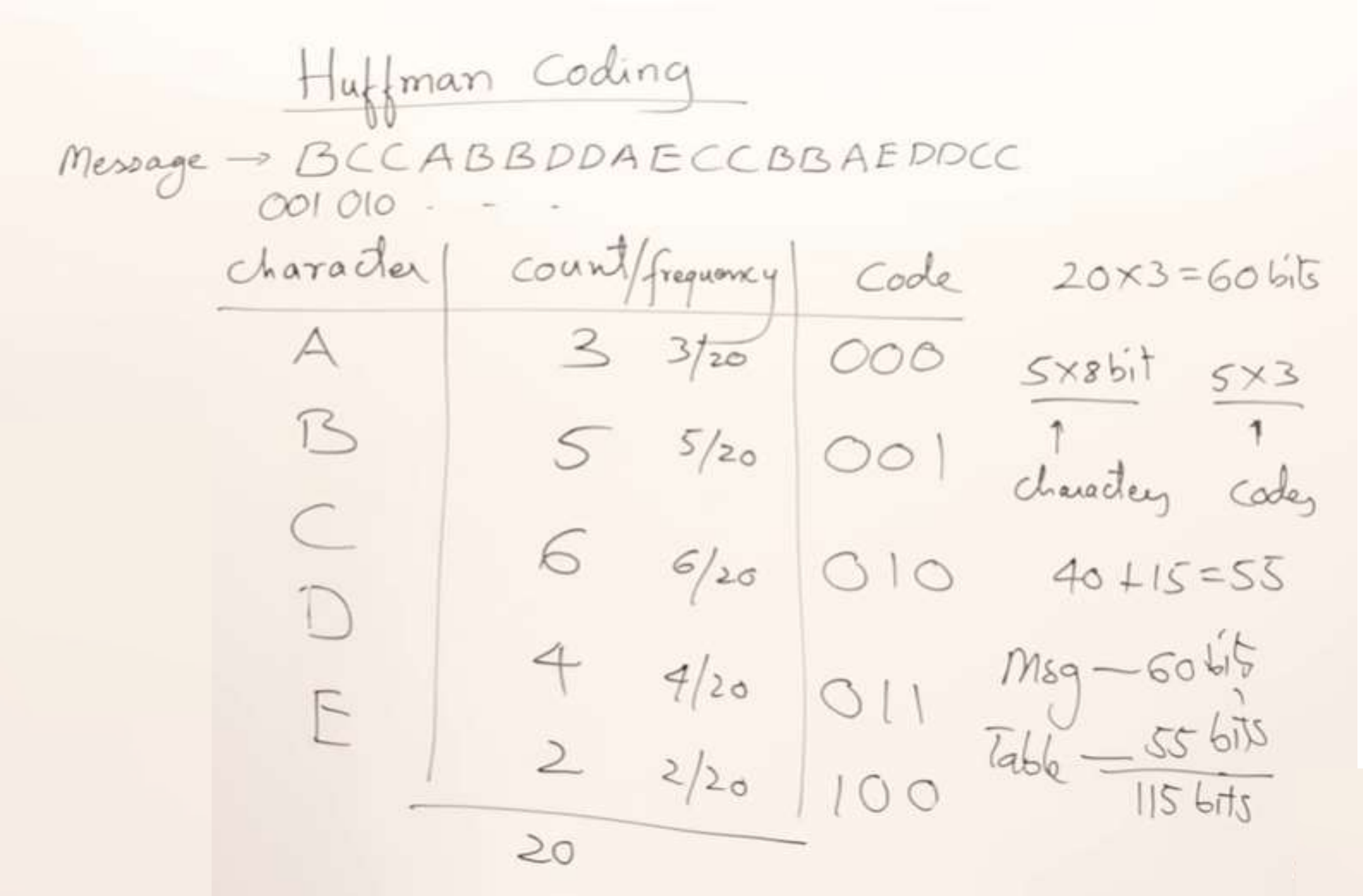
```
HUFFMAN_DECODE(encodedString, root)
decodedString = ""
currentNode = root

for each bit in encodedString:
    if bit == '0'
        currentNode = currentNode.left  // Move left for '0'
    else
        currentNode = currentNode.right // Move right for '1'

    if currentNode is a leaf:
        decodedString += currentNode.character  // Decode character
        currentNode = root                      // Reset to root

return decodedString
```

Overall Time Complexity:

- For each of the `m` bits in the encoded string, we may have to traverse up to the height of the tree `h`, which is **O(log n)**.
- Therefore, the total time complexity for decoding is:

$$\text{Time Complexity} = O(m \cdot h) = O(m \cdot \log n)$$

FIXED LENGTH CODE



Huffman Coding

Message → BCCABBDDAECCBBAEDDCC

001 010 . . .

character	count/frequency		Code
A	3	3/20	000
B	5	5/20	001
C	6	6/20	010
D	4	4/20	011
E	2	2/20	100
	20		

20×3=60 bits

5×8bit ↑ characters

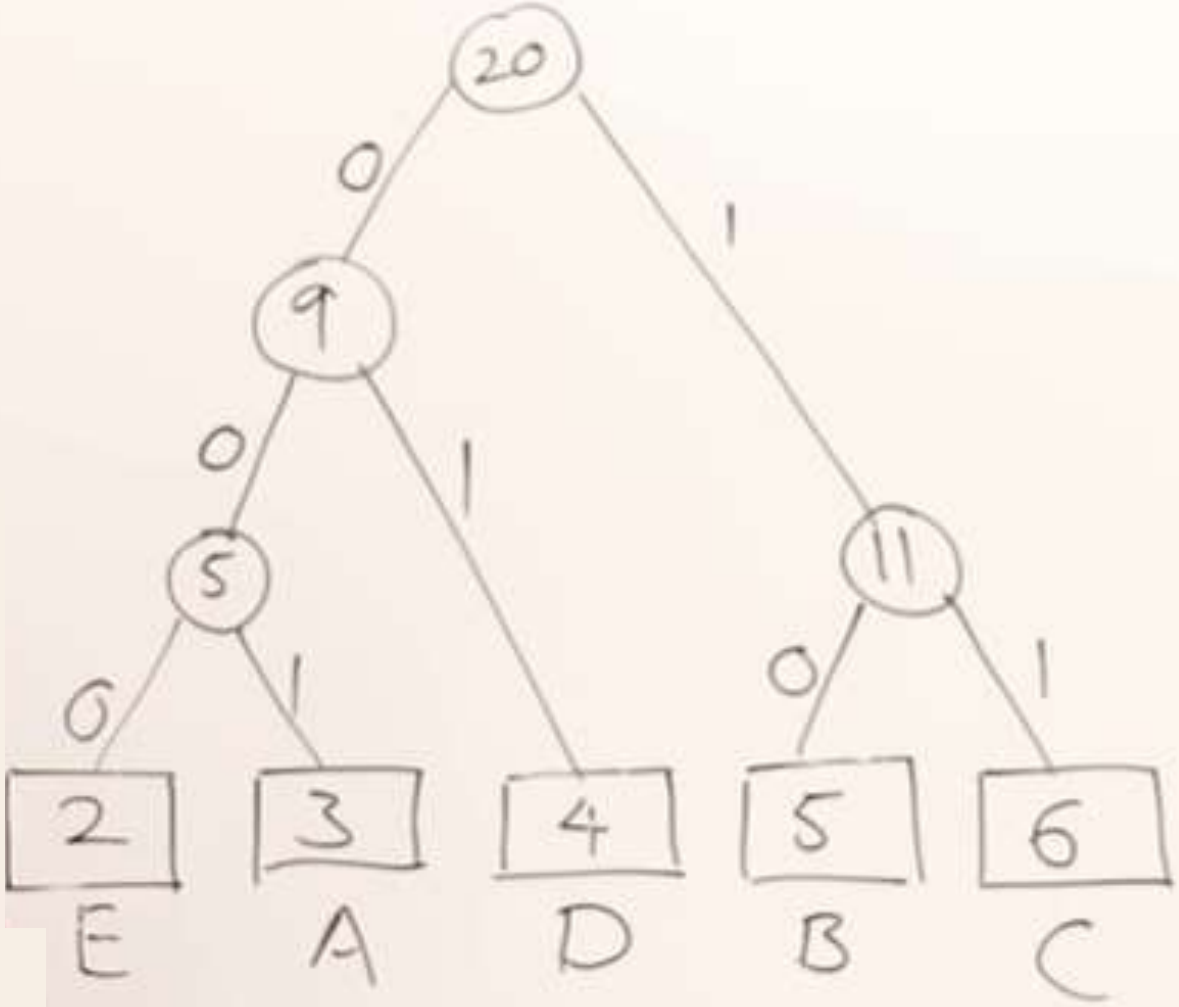
5×3 ↑ codes

40+15=55

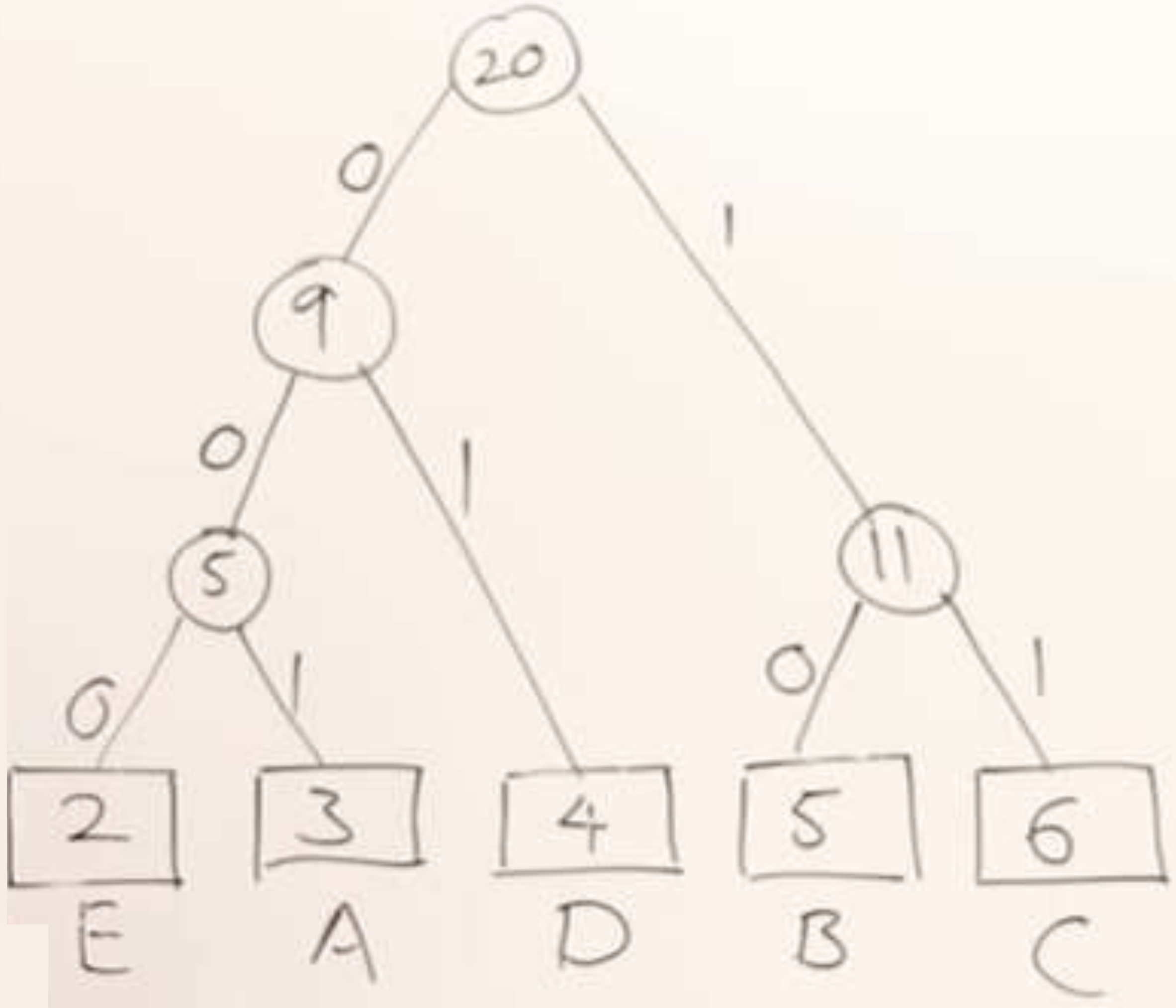
Msg — 60 bits

Table — 55 bits

115 bits



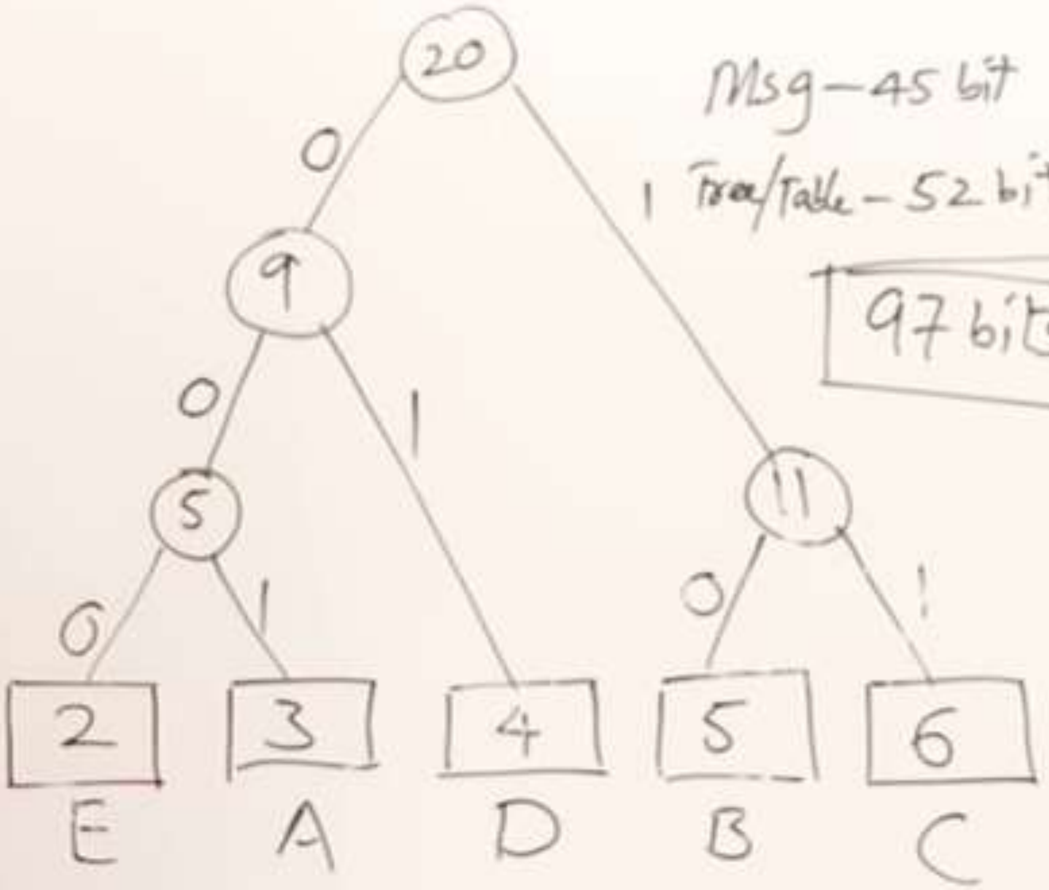
Char	count	Code
A	3	001
B	5	10
C	6	11
D	4	01
E	2	000
	20	



char	count	code	
A	3	001	3x3=9
B	5	10	5x2=10
C	6	11	6x2=12
D	4	01	4x2=8
E	2	000	2x3=6
	20		45 bits

Huffman Coding

Message → BCCABBDDAECCBBAEDDCC
10 11 11 001 10 10 01 01 - - -



Msg – 45 bit

Tree/Table – 52 bits

$\boxed{\text{97 bits}}$

Char	count	Code	
A	3	001	3×3=9
B	5	10	5×2=10
C	6	11	6×2=12
D	4	01	4×2=8
E	2	000	2×3=6
5x 8bit 40bits	20 +	12 bit	45 bits