

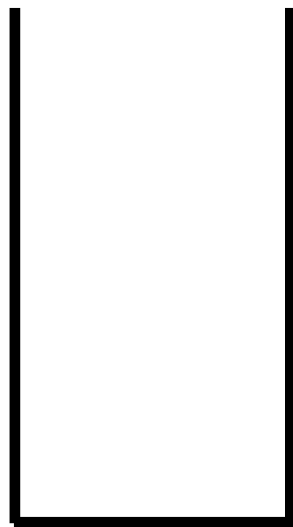
Amortized Analysis

Stack operations

Given an empty stack S and a sequence of $\text{Push}(S, x)$, $\text{Pop}(S)$, $\text{MultiPop}(S, k)$ operations.

What is the worst-case running time for a sequence of n Push , Pop , and MultiPop operations?

Push and Pop



S

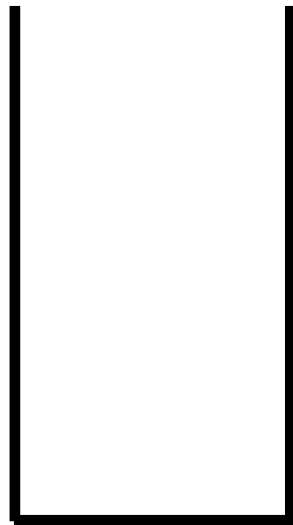
Push(S , 1)

Push(S , 3)

Pop(S)

...

Push and Pop



S

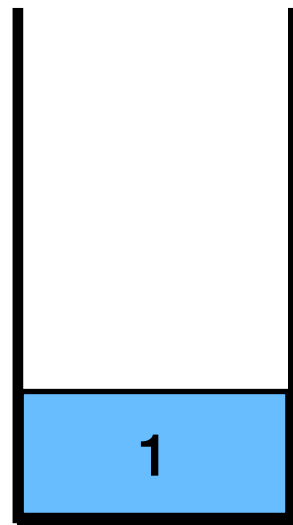
Push(S , 1)

Push(S , 3)

Pop(S)

...

Push and Pop



S

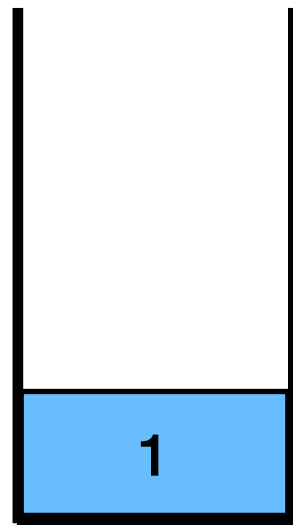
Push(S, 1)

Push(S, 3)

Pop(S)

...

Push and Pop



S

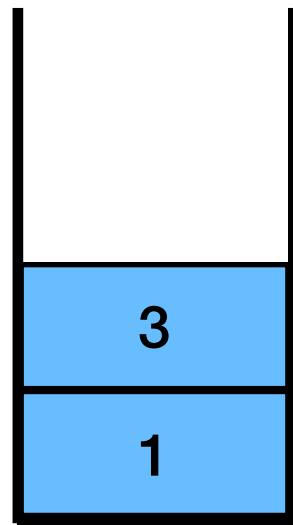
Push(S, 1)

Push(S, 3)

Pop(S)

...

Push and Pop



S

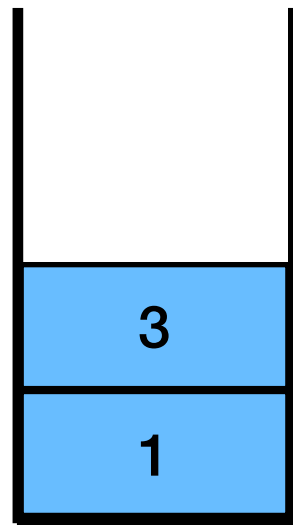
Push(S, 1)

Push(S, 3)

Pop(S)

...

Push and Pop



S

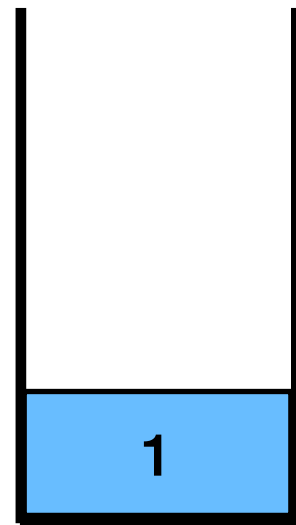
Push(S, 1)

Push(S, 3)

Pop(S)

...

Push and Pop



Push(S, 1)

Push(S, 3)

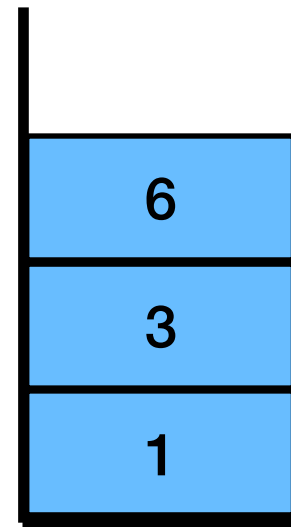
Pop(S)

...

Every Push and Pop operation needs $O(1)$ time.

MultiPop

```
MultiPop(S, k){  
  while(!S.empty() and k > 0){  
    S.Pop();  
    k ← k - 1;  
  }  
}
```



S

MultiPop(S, 2)

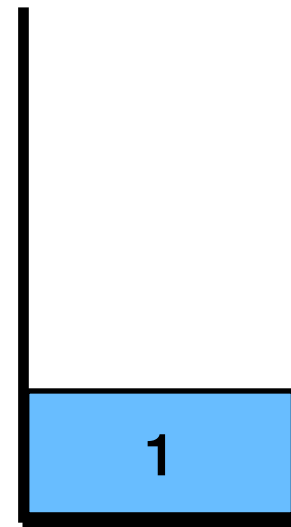
Push(S, 3)

Pop(S)

...

MultiPop

```
MultiPop(S, k){  
  while(!S.empty() and k > 0){  
    S.Pop();  
    k ← k - 1;  
  }  
}
```



S

MultiPop(S, 2)

Push(S, 3)

Pop(S)

...

Every MultiPop operation needs $O(k)$ time.

Naive analysis

Note that there are at most n $\text{Push}(S, x)$ in a sequence of n operations.



$\text{MultiPop}(S, k)$ needs $O(k) = O(n)$ time because there are at most n elements in the stack at any time.



The running time of a sequence of n $\text{Pop}()$, $\text{Push}()$, and $\text{MultiPop}()$ operations is $O(n^2)$ in the worst case.

Aggregate analysis

Note that there are at most n $\text{Push}(S, x)$ in a sequence of n operations.



All $\text{MultiPop}(S, k)$ operations need $O(n)$ time because there are at most n elements in the stack at any time.



The running time of a sequence of n $\text{Pop}()$, $\text{Push}()$, and $\text{MultiPop}()$ operations is **$O(n)$** in the worst case.

The accounting method

i-th operation	actual cost c_i	amortized cost \hat{c}_i
Push(S, x)	1	2
Pop()	1	0
MultiPop(S, k)	$\min(S , k)$	0

At Push(S, x), it **prepays** the cost for Pop x.

We have $\sum_{1 \leq i \leq n} c_i \leq \sum_{1 \leq i \leq n} \hat{c}_i = \mathbf{O(n)}$.

Huffman Codes

Data compression

Input: a sequence S of n characters, where each character c_i for each $i \in [1, k]$ has f_i occurrences in the sequence.

Output: a binary string B that represents S . In other words, one can recover S by decompressing B .

Example.

Let S be “I can can a can.”

The binary representation
of S is \rightarrow

```
1001001000000100
1100011010000110
0111011000000100
1100011010000110
0111011000000100
1000011000000100
1100011010000110
0111011001110100
```


Data compression

Input: a sequence S of n characters, where each character c_i for each $i \in [1, k]$ has f_i occurrences in the sequence.

Output: a binary string B that represents S . In other words, one can recover S by decompressing B .

Example.

Let S be “I can can a can.”

The binary representation of S is \rightarrow

There are a lot of repeated substrings.

1001001000000100
1100011010000110
0111011000000100
1100011010000110
0111011000000100
1000011000000100
1100011010000110
0111011001110100

Character code

Let S be “I can can a can.”

1001001000000100
1100011010000110
0111011000000100
1100011010000110
0111011000000100
1000011000000100
1100011010000110
0111011001110100

Raw data: 128 bits.



0001010011011010
0110110110010011
01100000

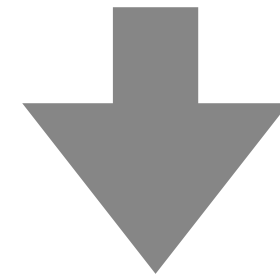
Compressed data:
39 bits. (Rate = 30.5%)

Character code - encoding

Each character is represented by a unique binary string.

character	codeword
'n'	11
'a'	10
','	01
'c'	001
'I'	0001
'.'	0000

I can can a can.



0001010011011010
0110110110010011
01100000

Prefix code

It is a **character code** in which no codeword is a prefix of some other codeword.

character	codeword
'n'	11
'a'	10
' '	01
'c'	001
'I'	0001
'.'	0000

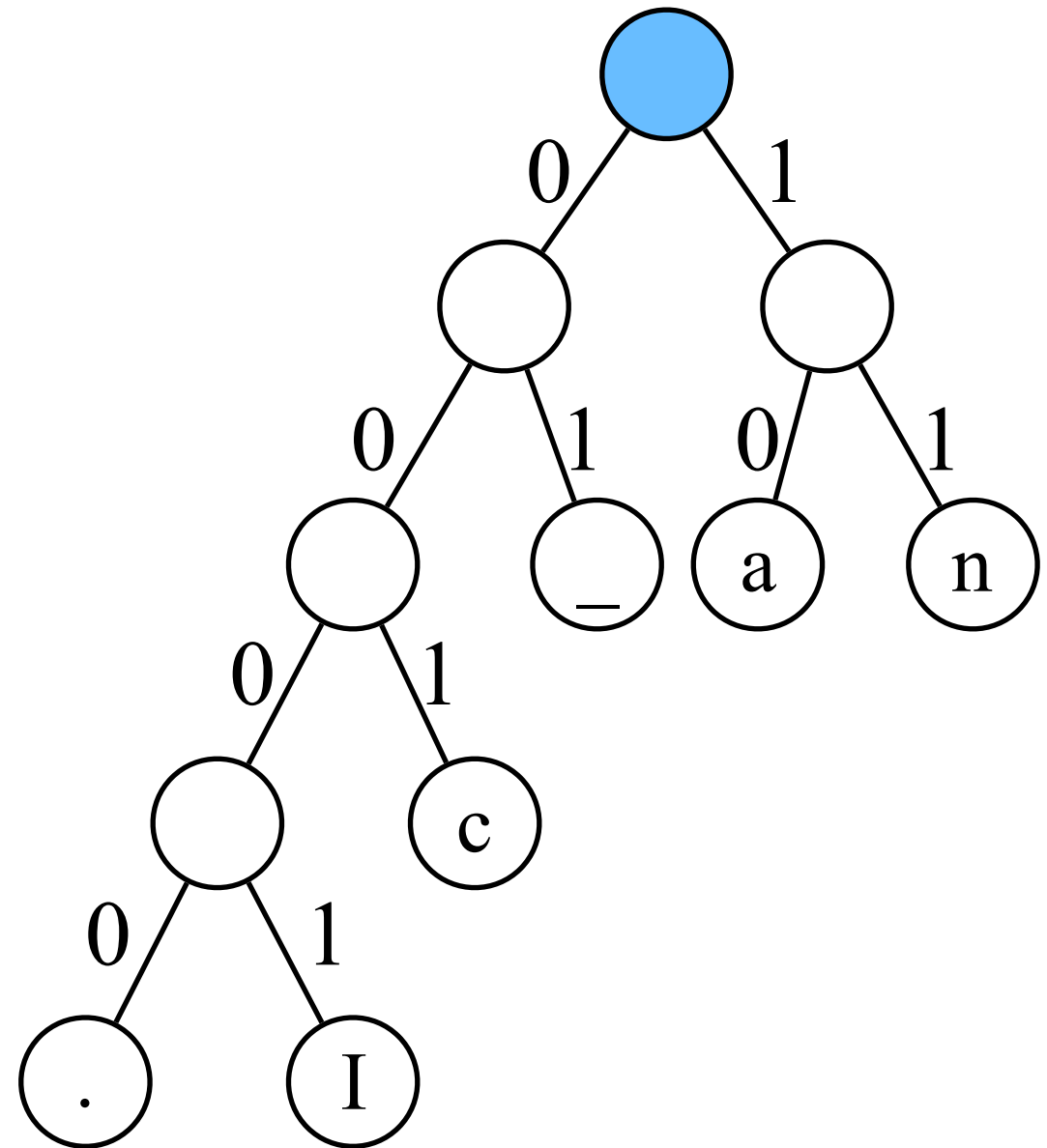
character	codeword
'n'	11
'a'	10
' '	01
'c'	001
'I'	0001
'.'	1

a prefix

Not a prefix code.

Prefix code - decoding

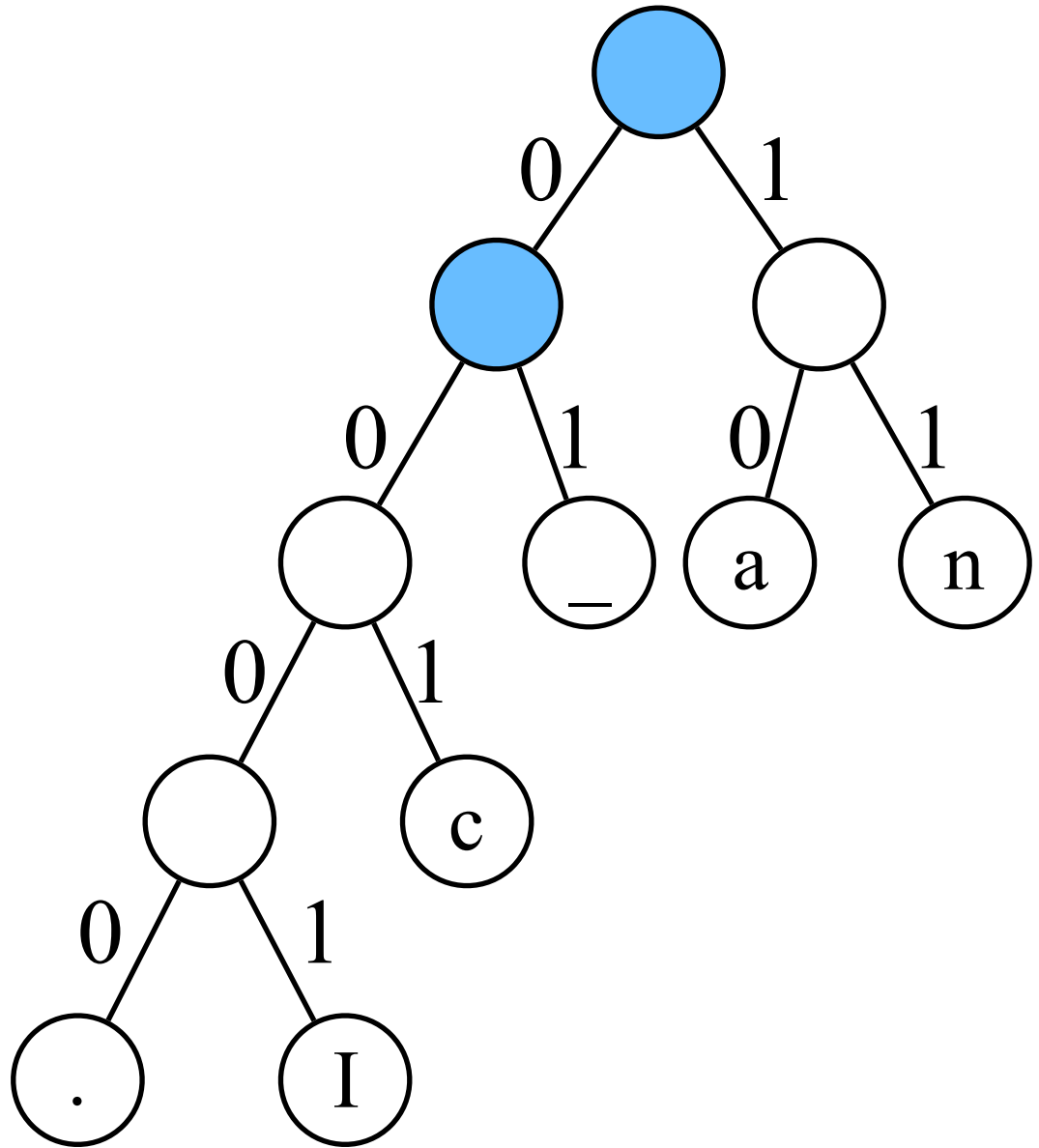
character	codeword
'n'	11
'a'	10
‘ ’	01
'c'	001
'I'	0001
‘ ’ .	0000



000101001101101001101101100100110110000

Prefix code - decoding

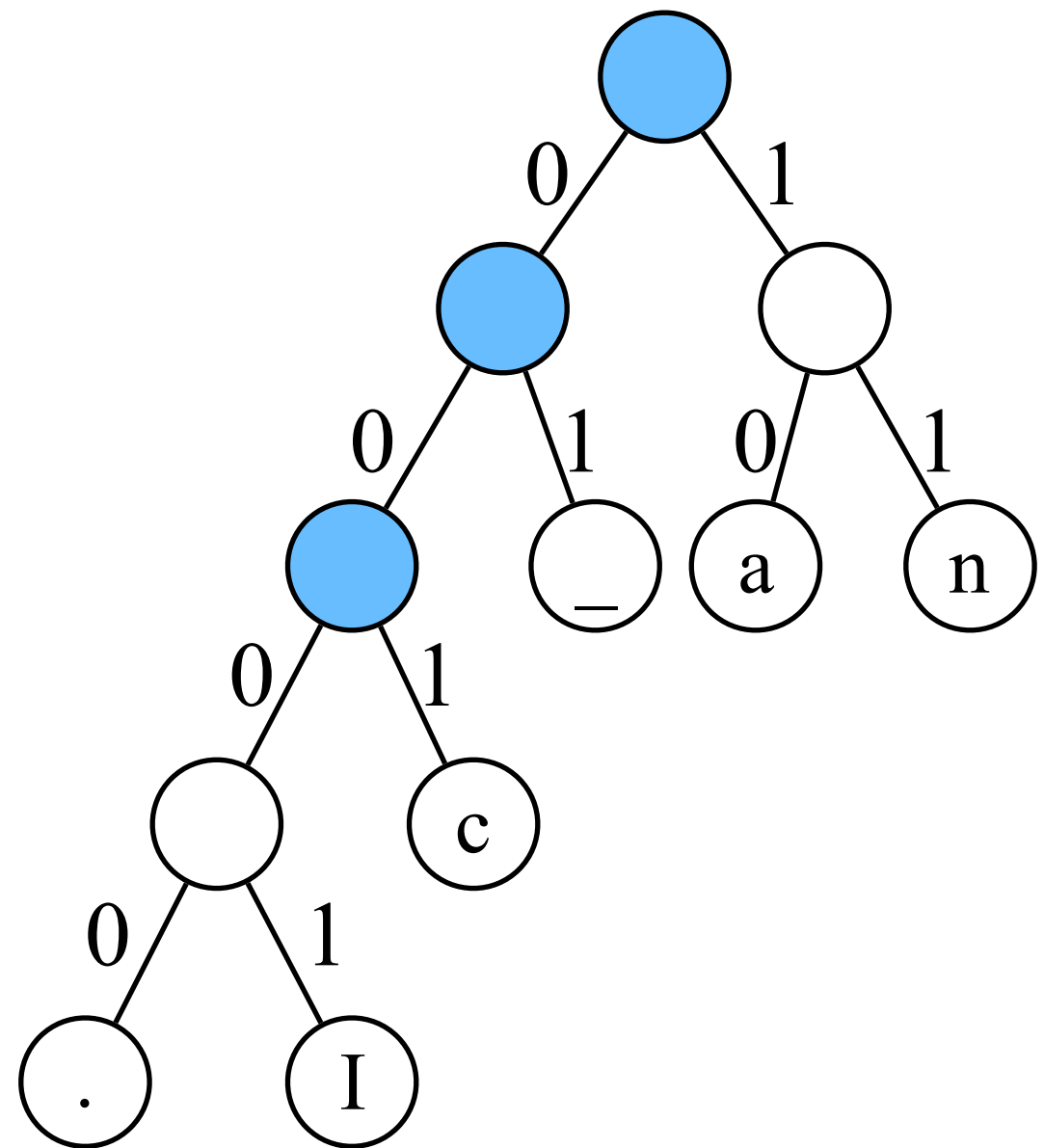
character	codeword
'n'	11
'a'	10
' '	01
'c'	001
'I'	0001
'.'	0000



000101001101101001101101100100110110000

Prefix code - decoding

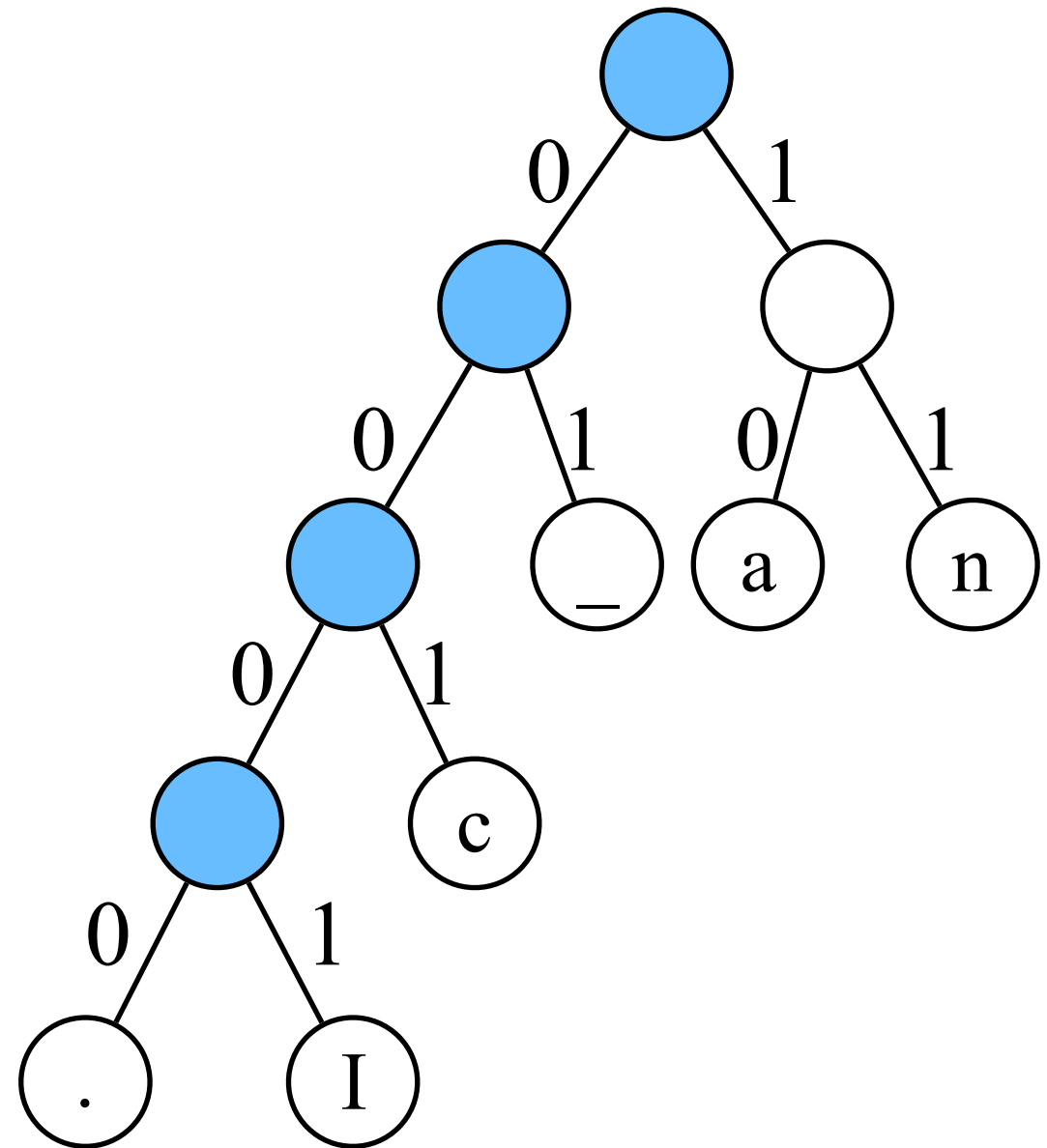
character	codeword
'n'	11
'a'	10
' '	01
'c'	001
'I'	0001
'.'	0000



000101001101101001101101100100110110000

Prefix code - decoding

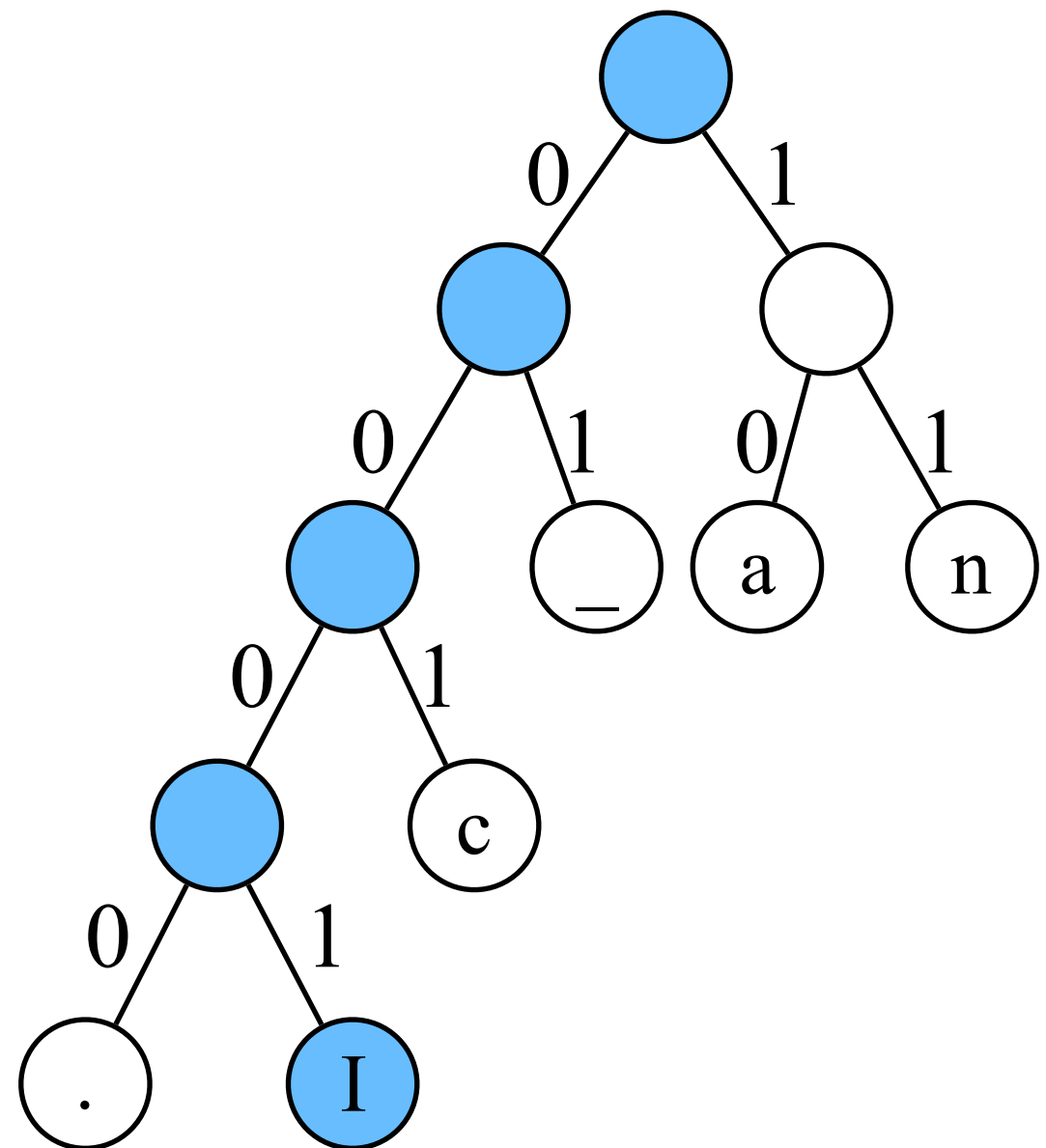
character	codeword
'n'	11
'a'	10
‘ ’	01
'c'	001
'I'	0001
‘ ’ .	0000



000101001101101001101101100100110110000

Prefix code - decoding

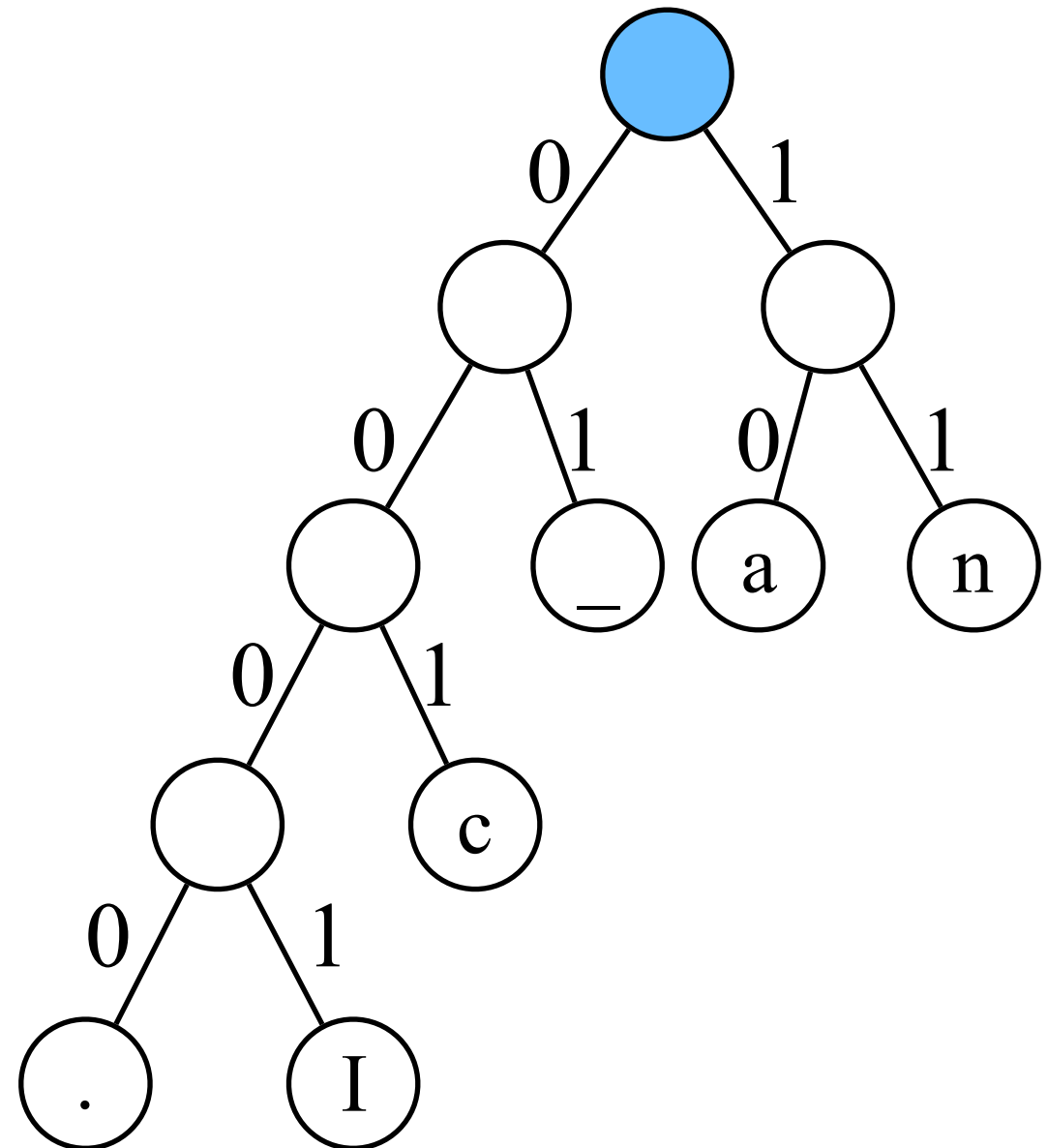
character	codeword
'n'	11
'a'	10
' '	01
'c'	001
'I'	0001
'.'	0000



000101001101101001101101100100110110000

Prefix code - decoding

character	codeword
'n'	11
'a'	10
‘ ’	01
'c'	001
'I'	0001
‘ ’ .	0000

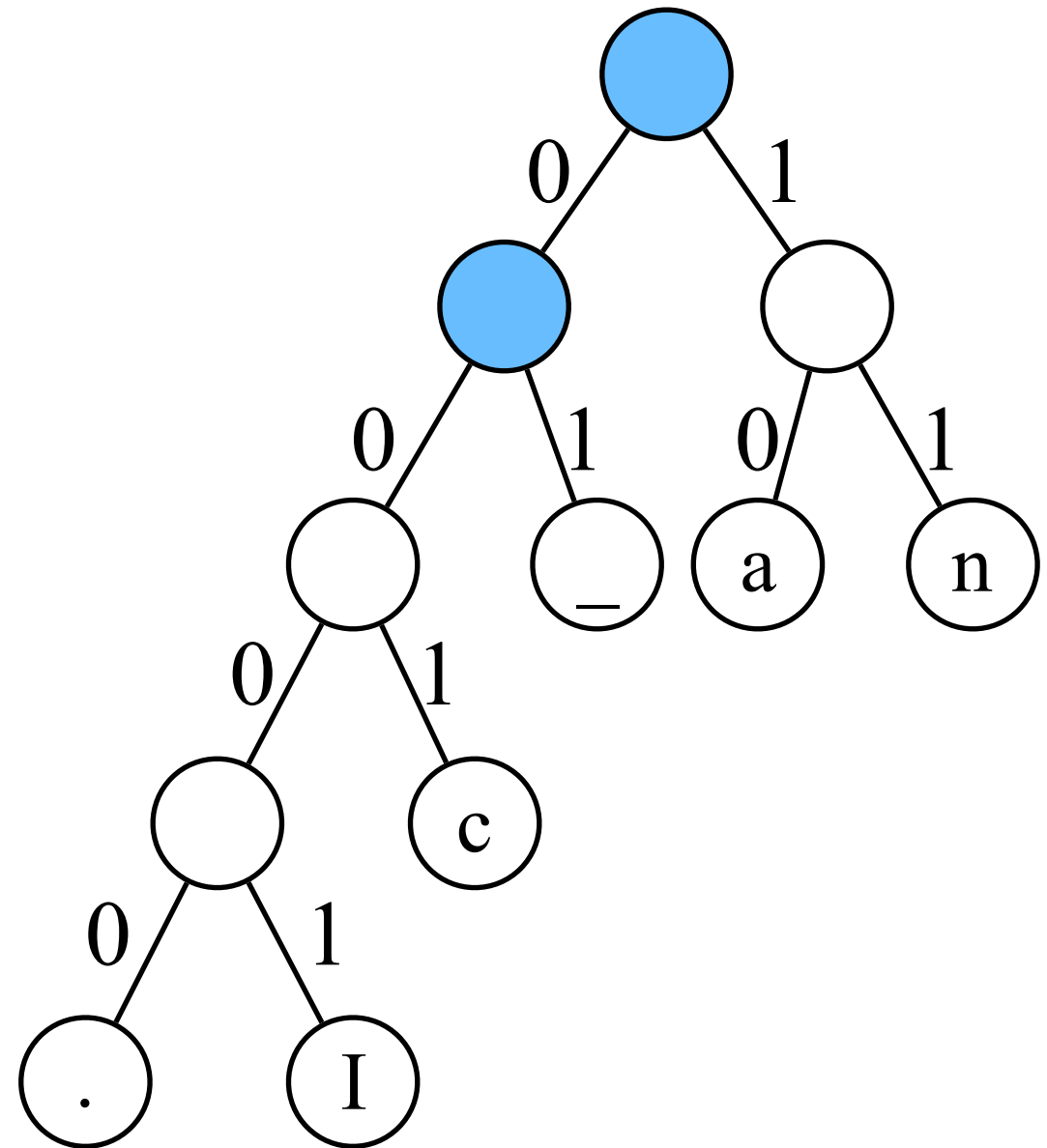


01001101101001101101100100110110000

I

Prefix code - decoding

character	codeword
'n'	11
'a'	10
‘ ’	01
'c'	001
'I'	0001
‘ ’ .	0000

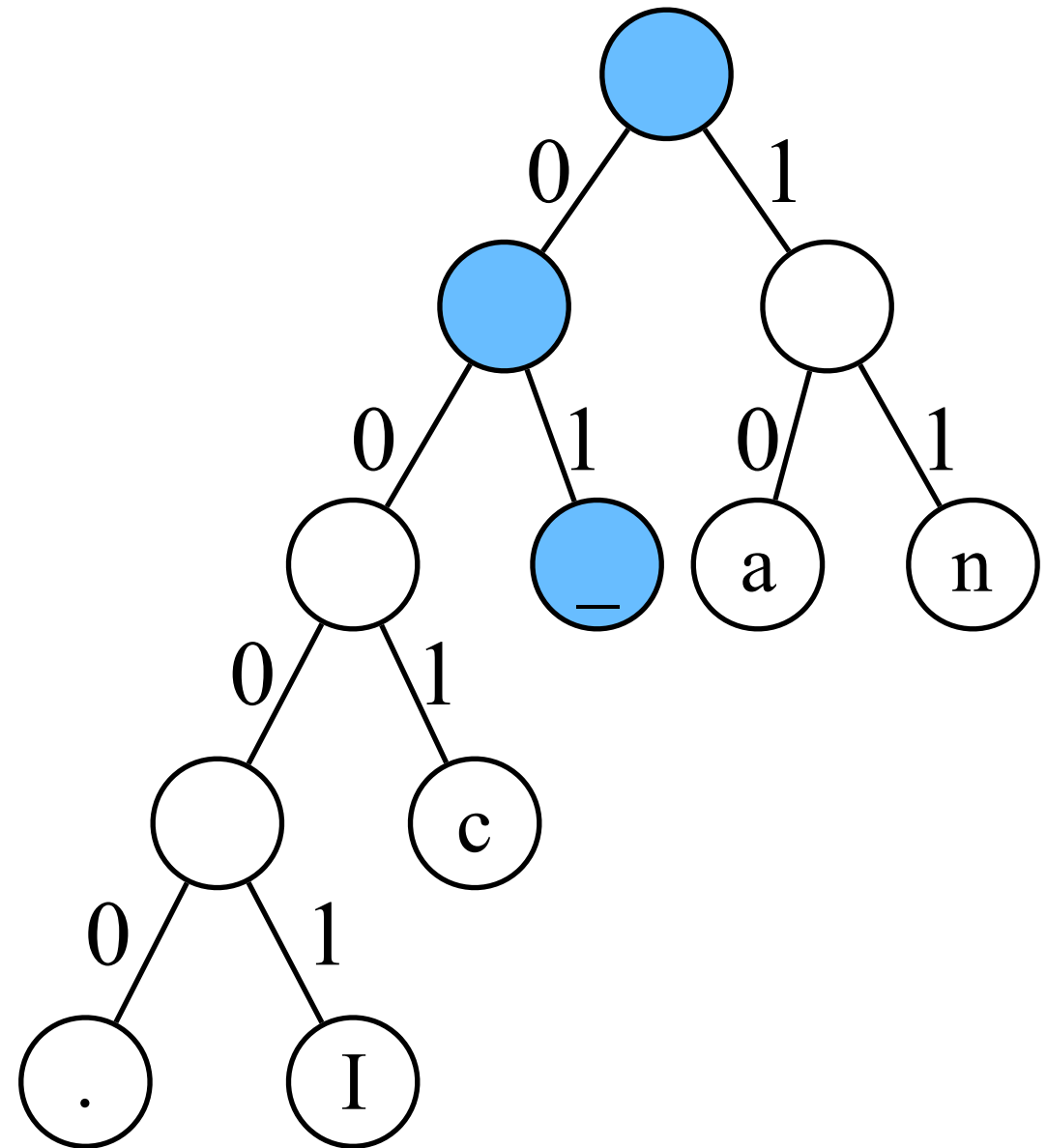


01001101101001101101100100110110000

I

Prefix code - decoding

character	codeword
'n'	11
'a'	10
' '	01
'c'	001
'I'	0001
'.'	0000

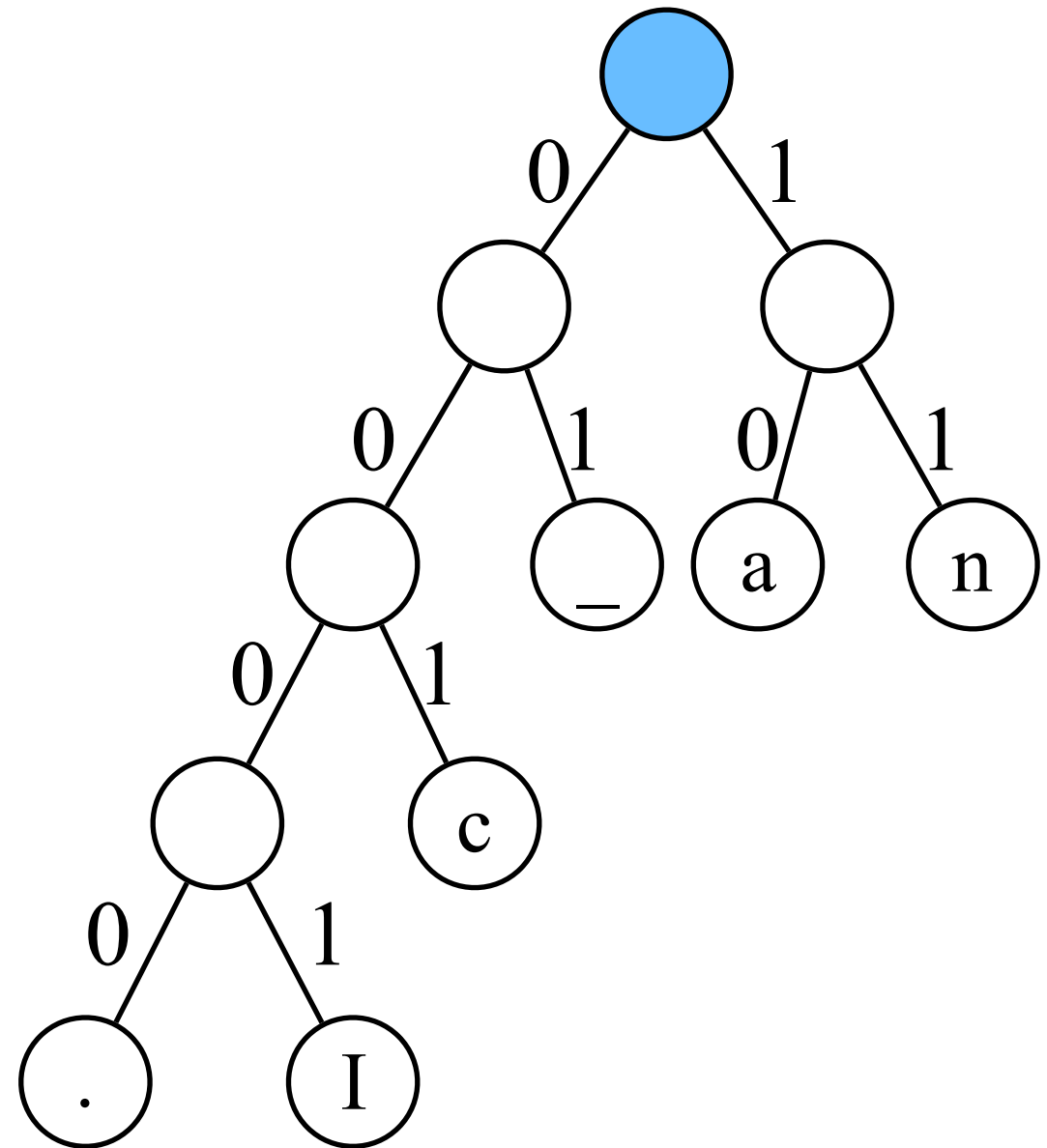


01001101101001101101100100110110000

I

Prefix code - decoding

character	codeword
'n'	11
'a'	10
‘ ’	01
'c'	001
'I'	0001
‘ ’ .	0000

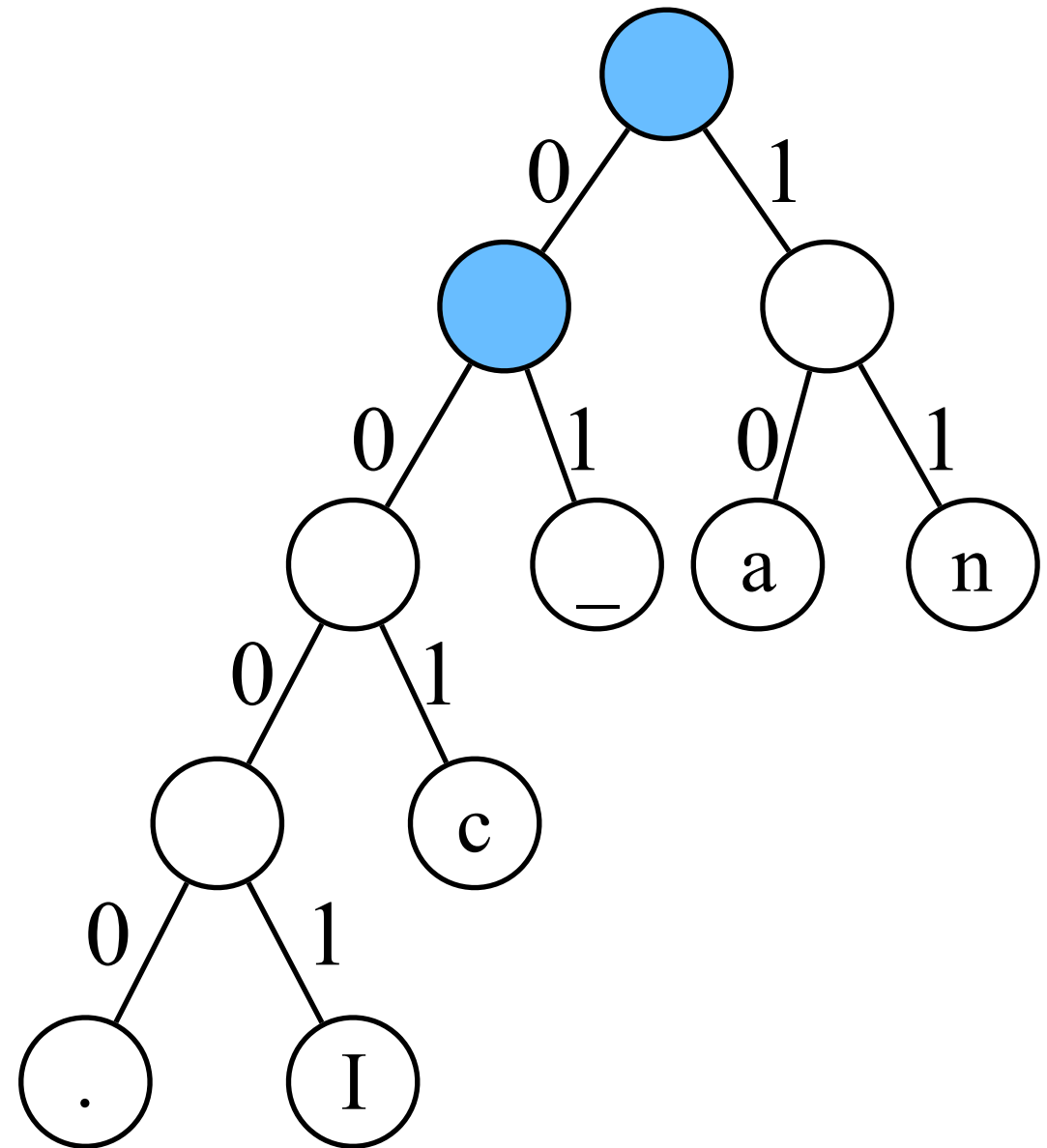


001101101001101101100100110110000

I

Prefix code - decoding

character	codeword
'n'	11
'a'	10
‘ ’	01
'c'	001
'I'	0001
‘ ’ .	0000

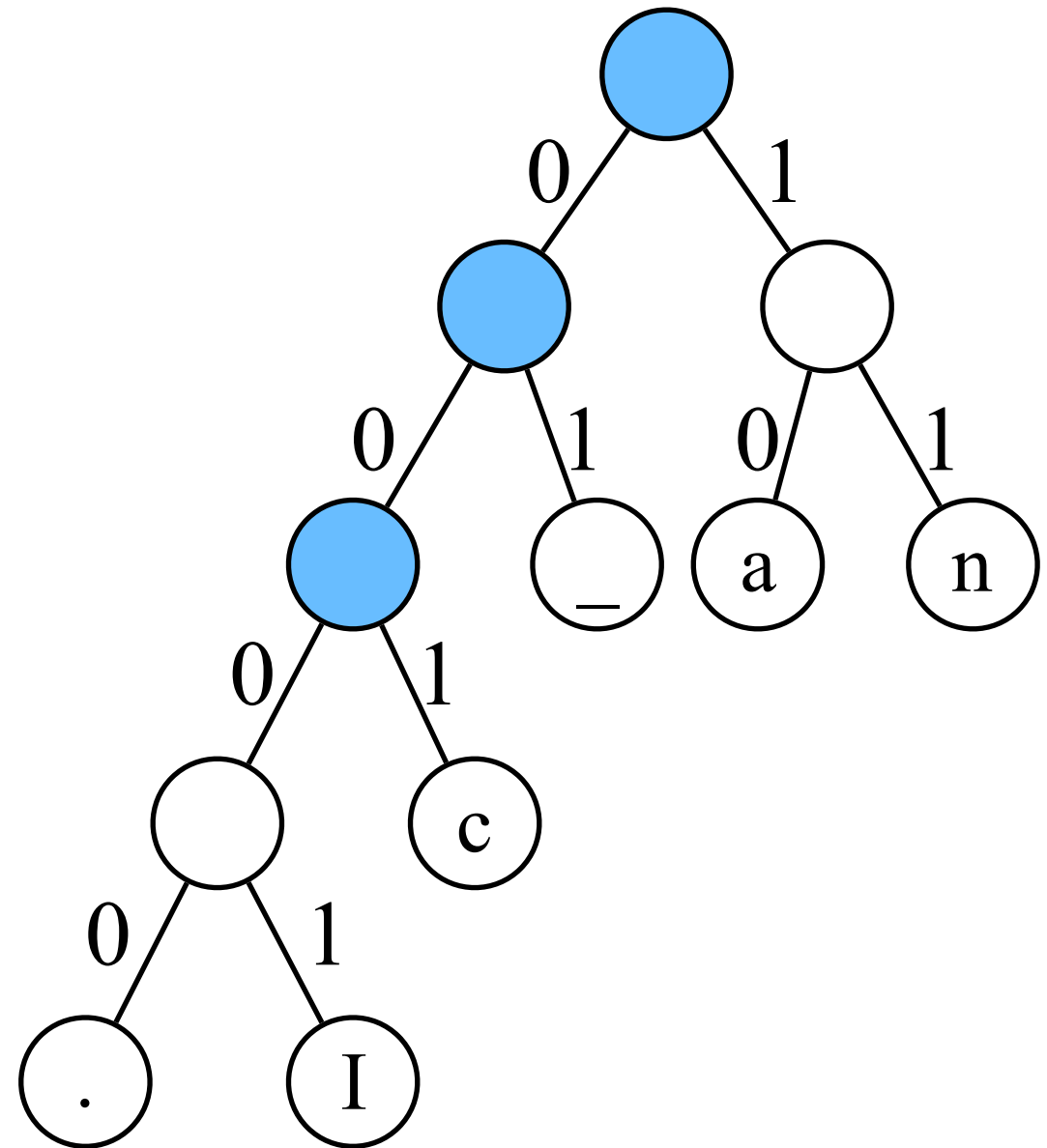


001101101001101101100100110110000

I.

Prefix code - decoding

character	codeword
'n'	11
'a'	10
','	01
'c'	001
'I'	0001
'.'	0000

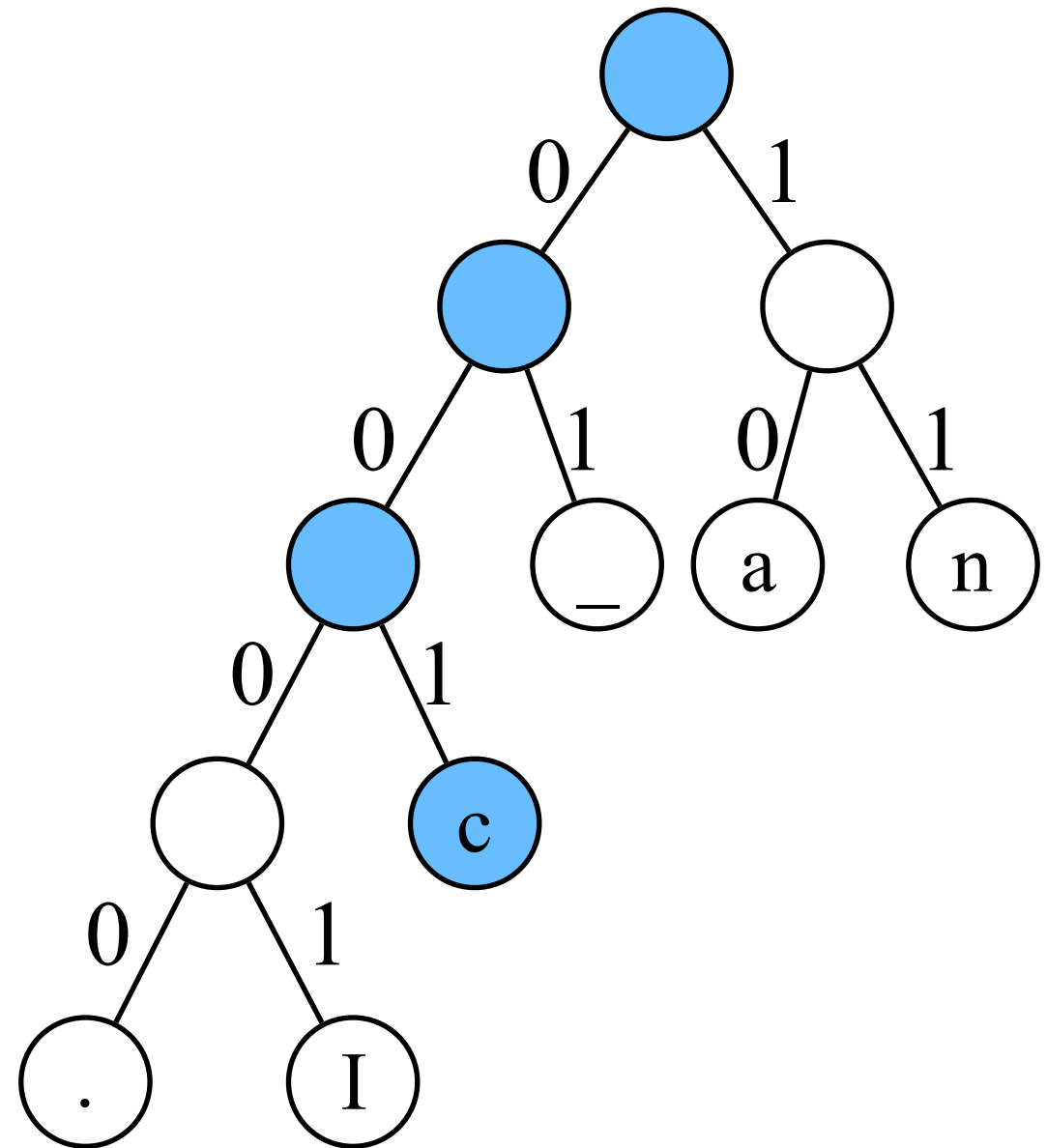


001101101001101101100100110110000

I_

Prefix code - decoding

character	codeword
'n'	11
'a'	10
‘ ’	01
'c'	001
'I'	0001
‘ ’ .	0000

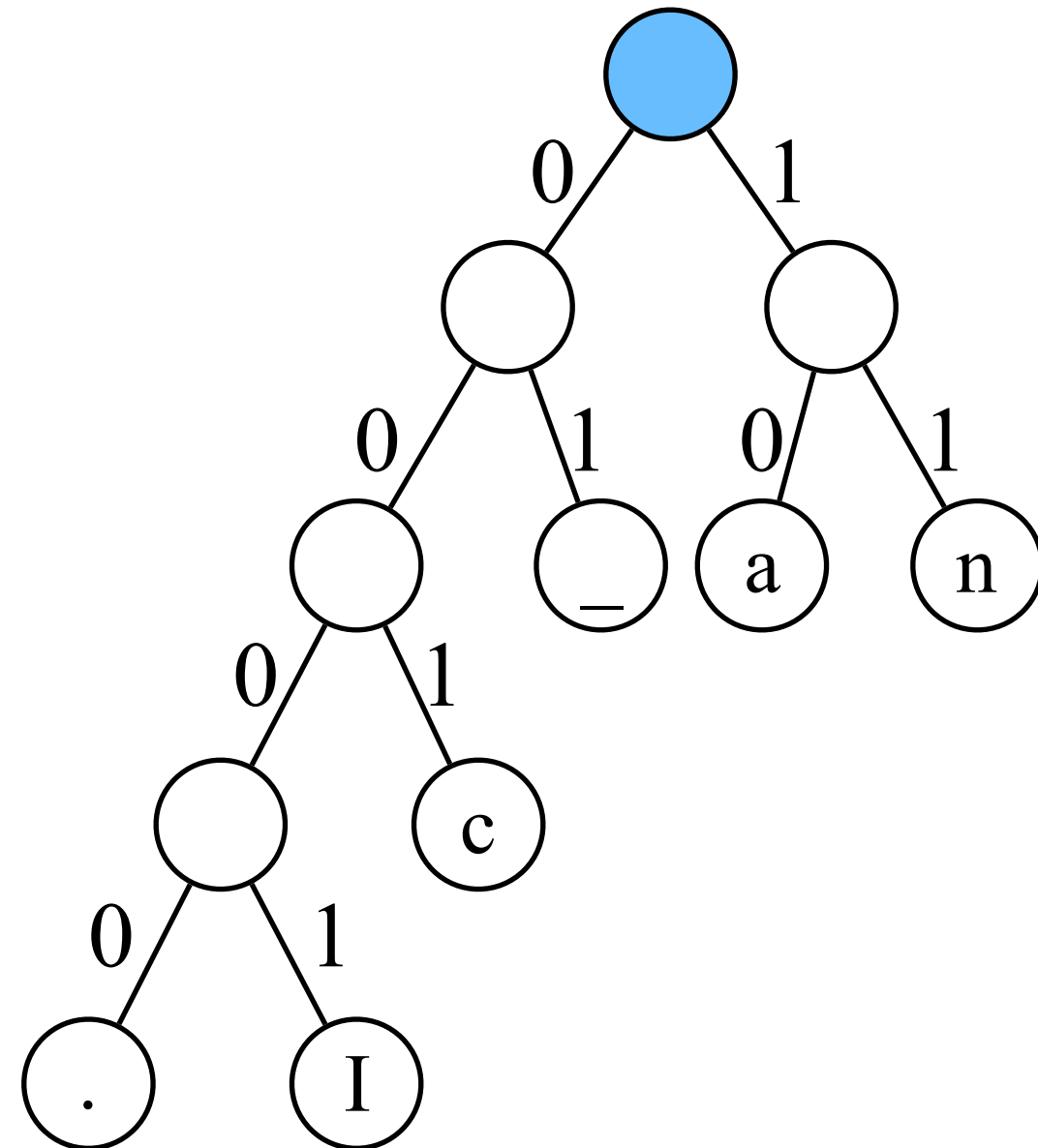


001101101001101101100100110110000

I.

Prefix code - decoding

character	codeword
'n'	11
'a'	10
','	01
'c'	001
'I'	0001
',' .	0000



101101001101101100100110110000

I c

Repeat this procedure until all codewords are decoded.

Huffman code

Input: a sequence S of n characters, where each character c_i for each $i \in [1, k]$ has f_i occurrences in the sequence.

Output: a prefix code mapping each c_i to a codeword e_i so that $\sum_{1 \leq i \leq k} f_i \cdot \text{length}(e_i)$ is minimized. In other words, the desired output is a prefix code that minimizes the length of compressed data.

Huffman code - constructing

Construct_Huffman_Code( input) {

Represent each character c_i by a tree composed of a single root node with frequency f_i .

character	frequency
'n'	3
'a'	4
','	4
'c'	3
'I'	1
'.'	1

While (there are multiple connected components) {

(1) Pick two trees T_1 , T_2 whose root has lowest frequencies.

(2) Connect T_1 , T_2 by a new node z . Let z .left be the root of T_1 and z .right be the root of T_2 . Let z .freq be the sum of the frequency at the roots of T_1 and T_2 .

}

}

Huffman code - constructing

1

.

1

I

3

c

3

n

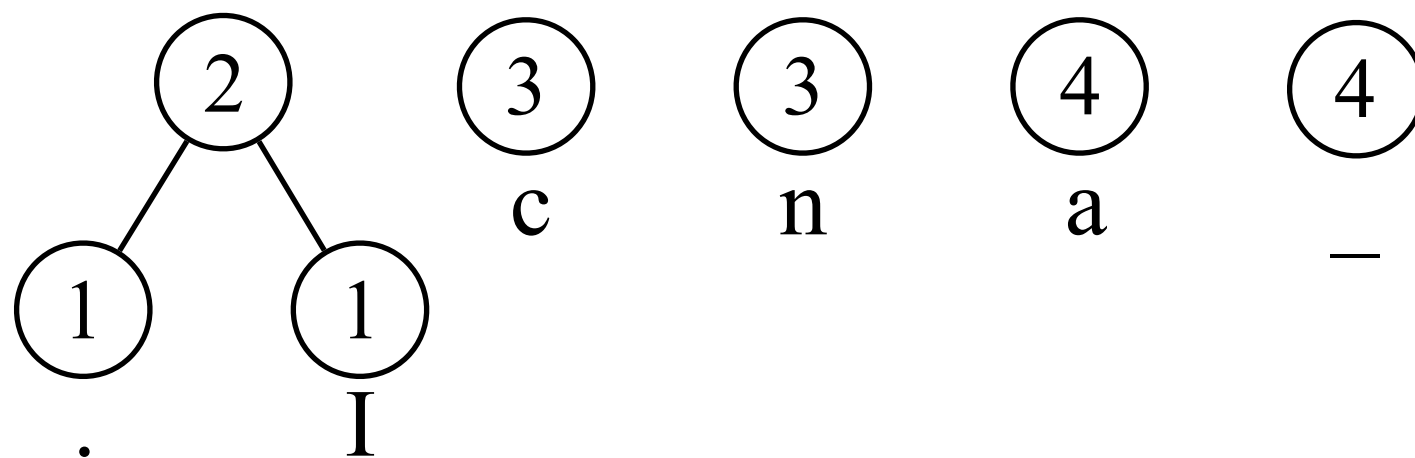
4

a

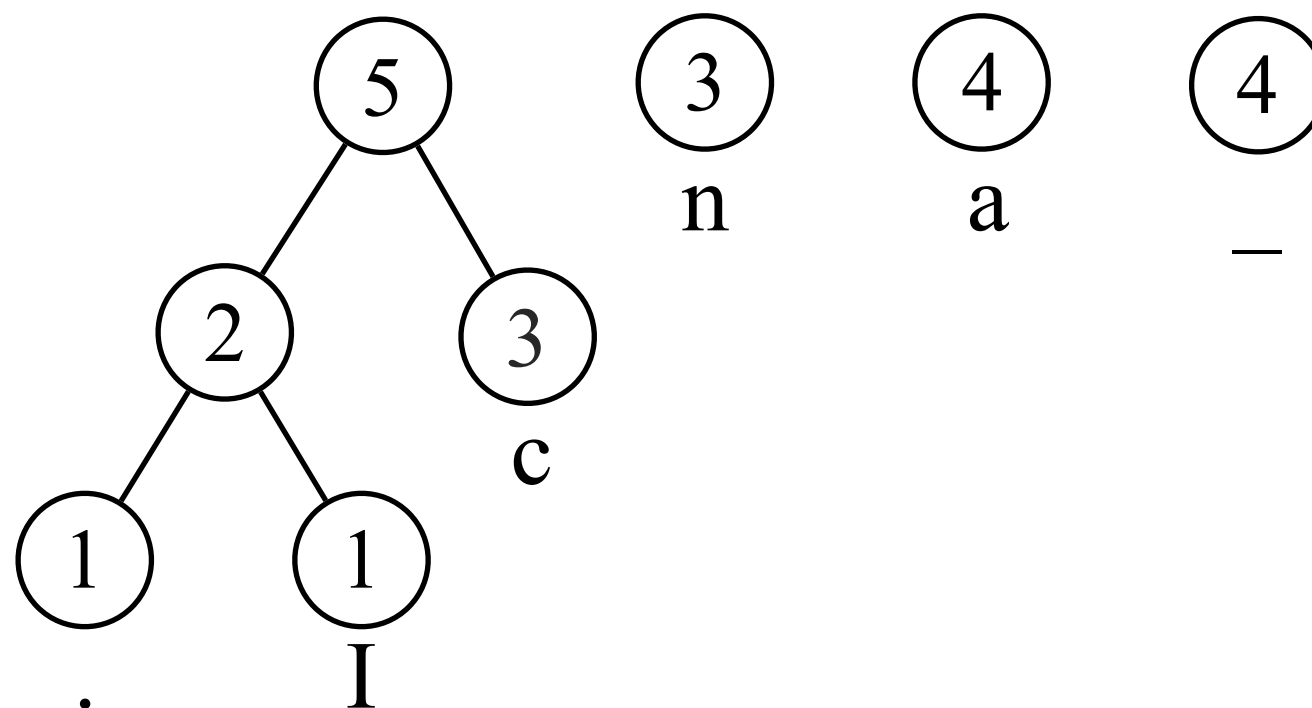
4

—

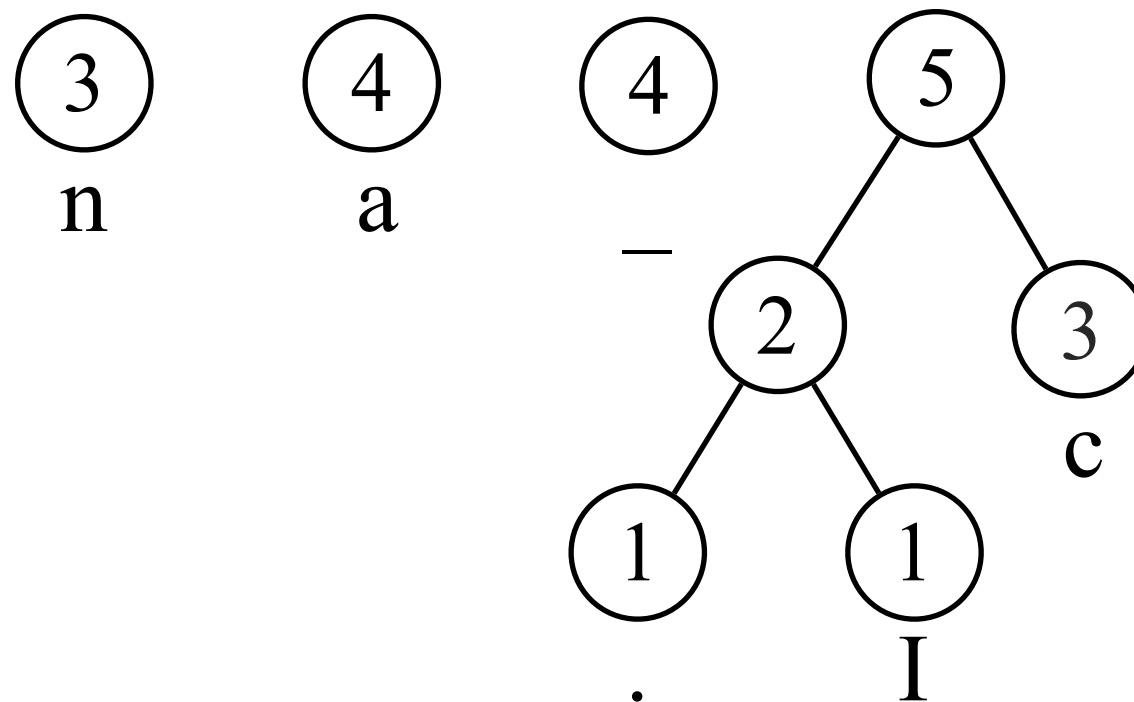
Huffman code - constructing



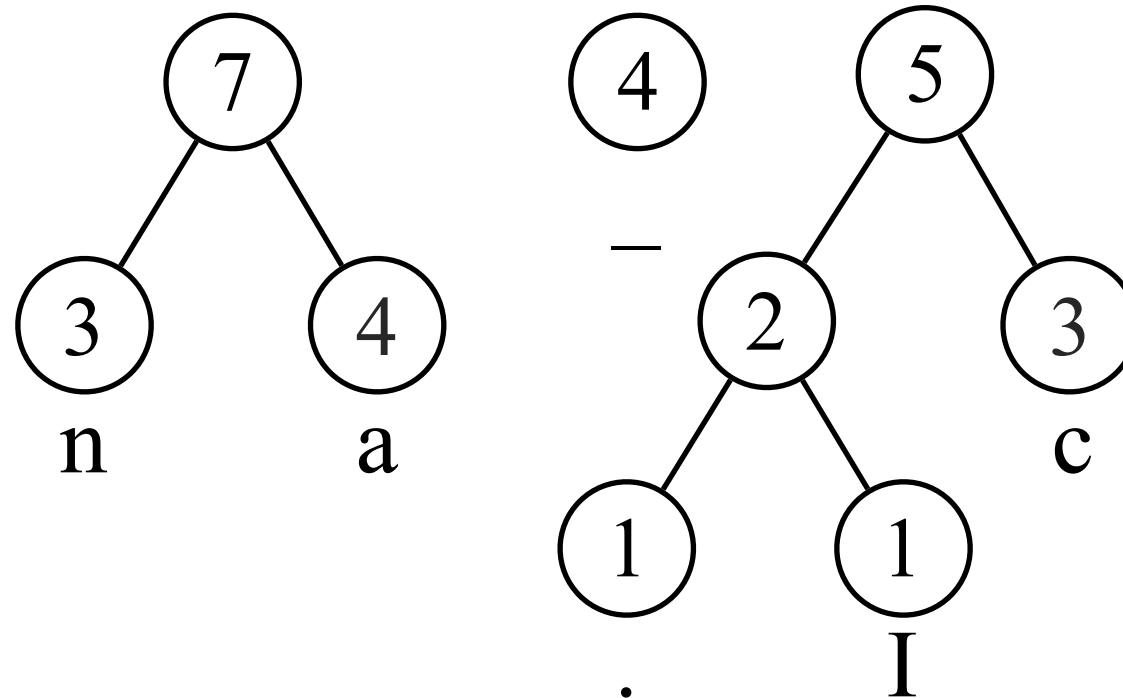
Huffman code - constructing



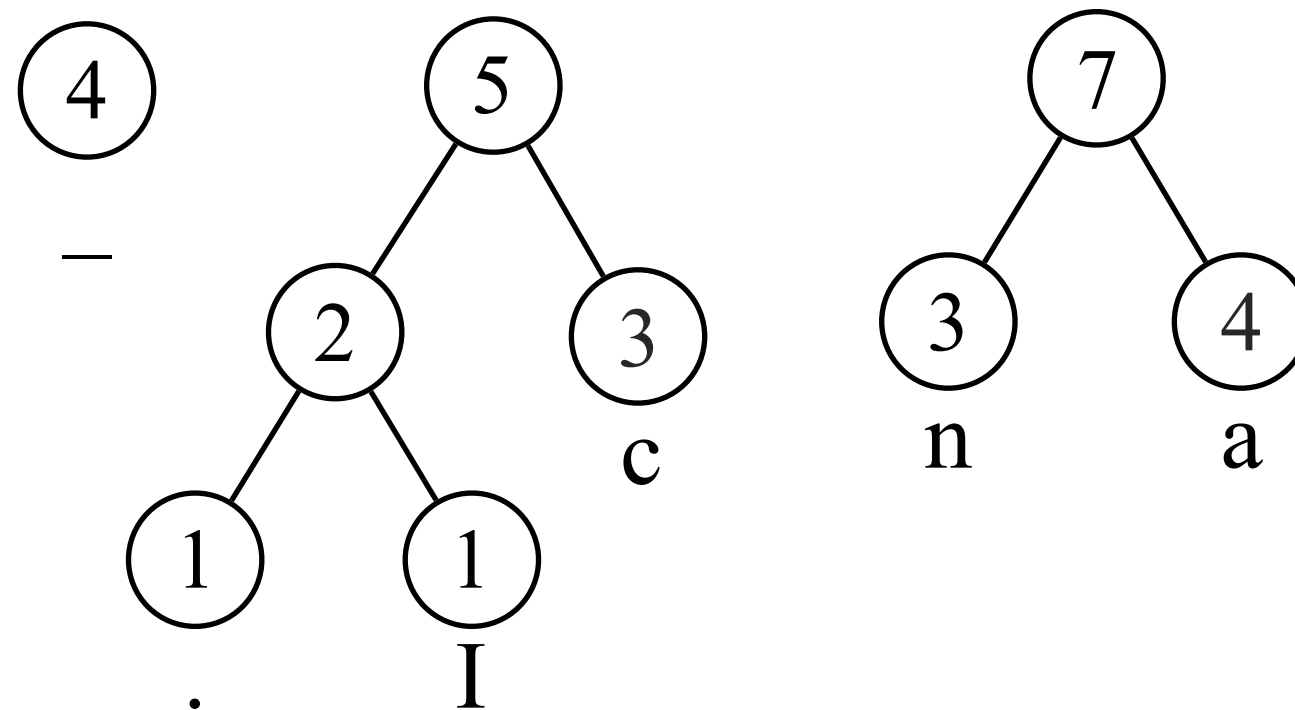
Huffman code - constructing



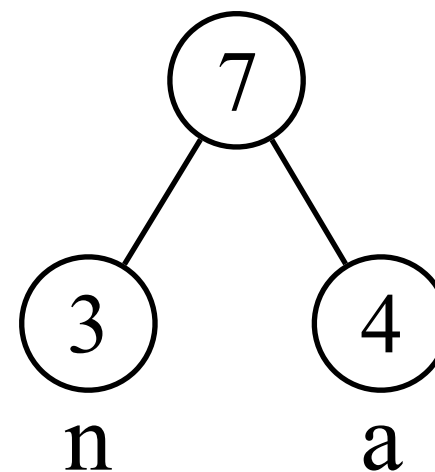
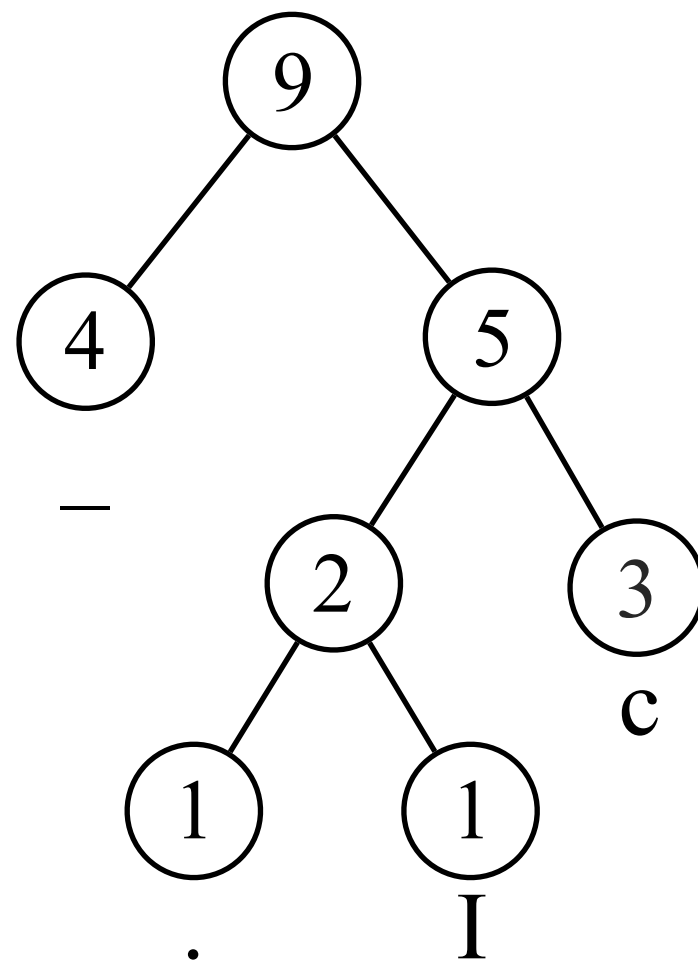
Huffman code - constructing



Huffman code - constructing

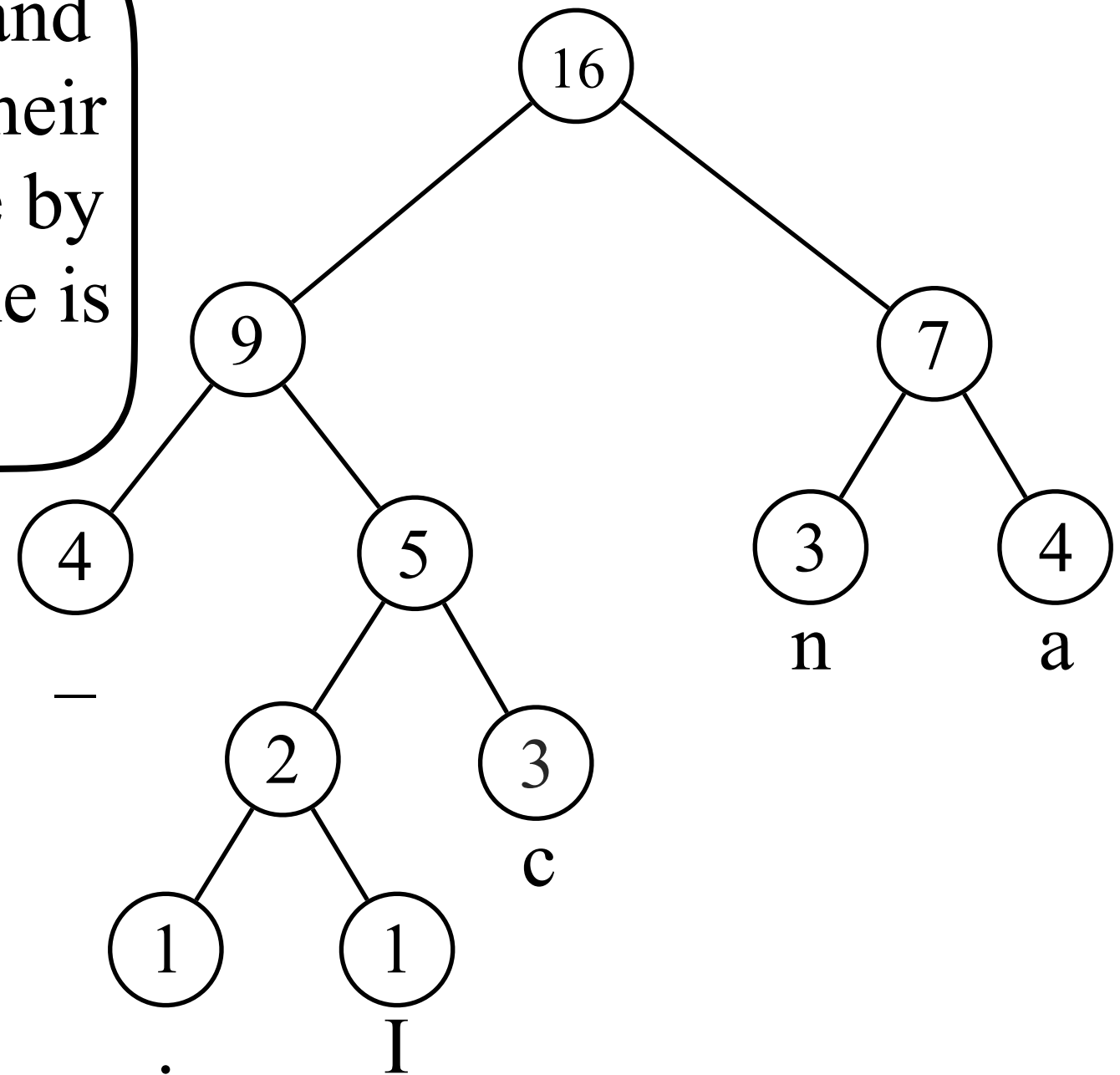


Huffman code - constructing



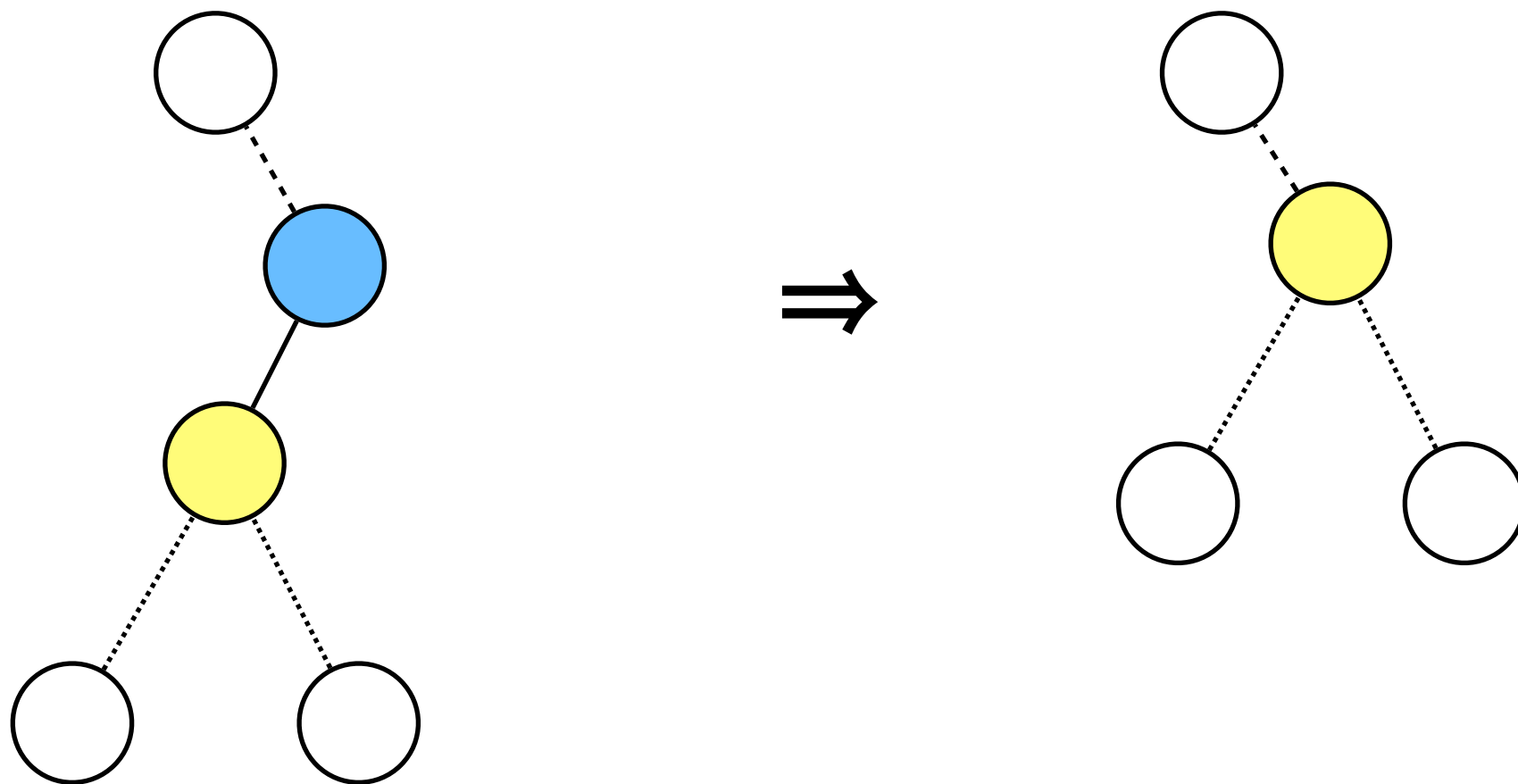
Huffman code - constructing

Replacing the lowest frequency and the second lowest frequency by their sum can be done in $O(\log n)$ time by a min-heap. The total running time is $O(n \log n)$.



Why is Huffman code optimal?

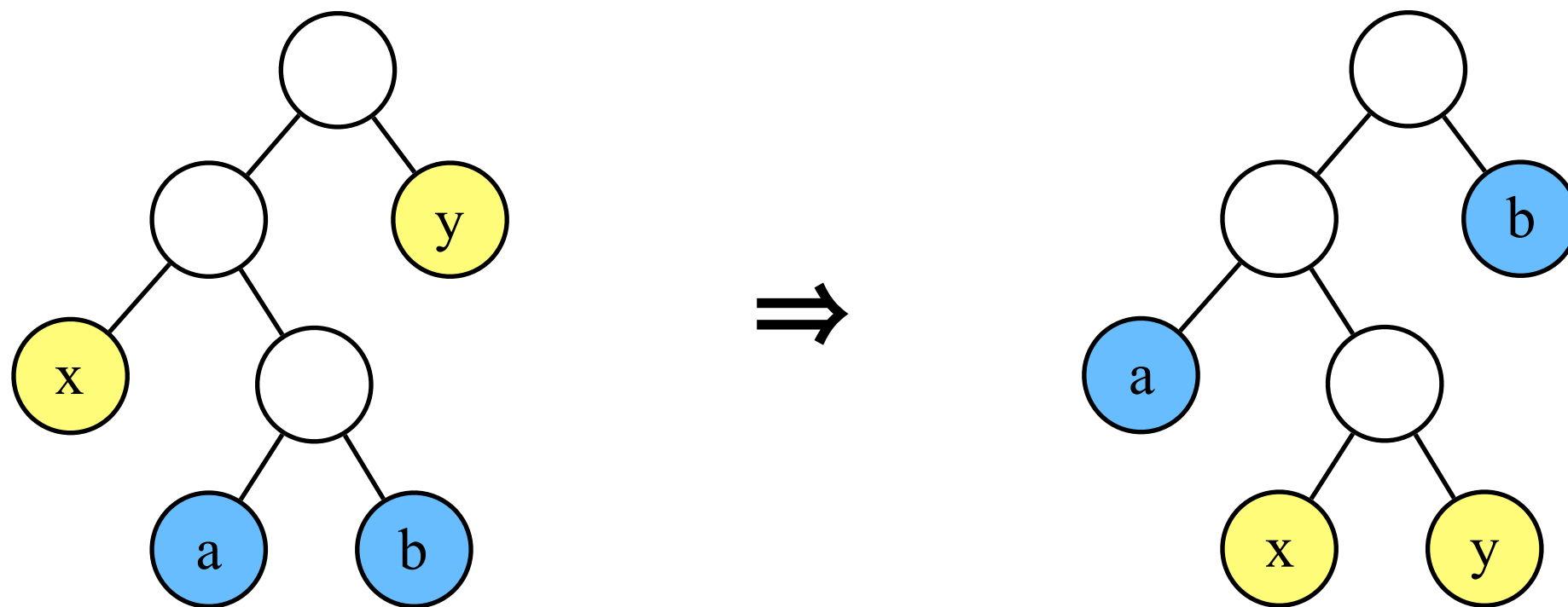
Claim 1. If a prefix code is optimal ($\sum_{1 \leq i \leq k} f_i \cdot \text{length}(e_i)$ is minimized), then in the tree T representing the code every internal node has exactly two child nodes.



We can get a more succinct code by removing the internal nodes that have one child node.

Why is Huffman code optimal?

Claim 2. Let x, y be the characters that have the lowest frequencies. There exists an optimal code so that the codewords for x and y have the same length and differ only in the last bit.



Let leaves a, b be siblings of maximum depth. Replacing a, b with x, y cannot increase $\sum_{1 \leq i \leq k} f_i \cdot \text{length}(e_i)$.

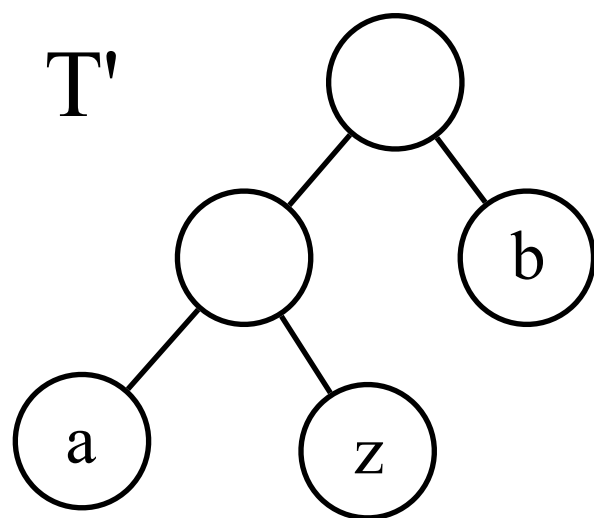
Why is Huffman code optimal?

Claim 3. Let C be the character set. Let x, y be the characters that have the lowest frequencies in C .

Let $C' = C - \{x, y\} \cup \{z\}$ and $z.\text{freq} = x.\text{freq} + y.\text{freq}$.

Let T' be any tree representing an optimal code for C' .

Then, the tree T obtained from T' by replacing z with an internal node having x and y as children, represents an optimal code for C .

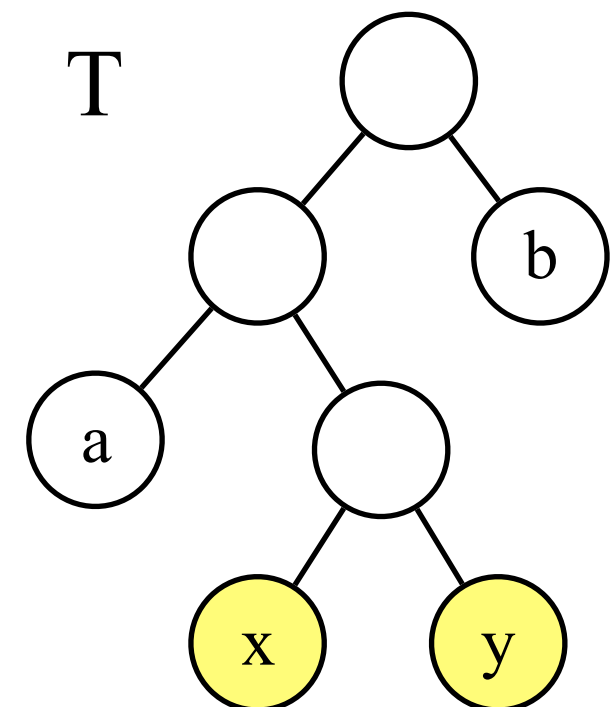


To show:

If T' is optimal for C' ,
then T is optimal for C .

Note that

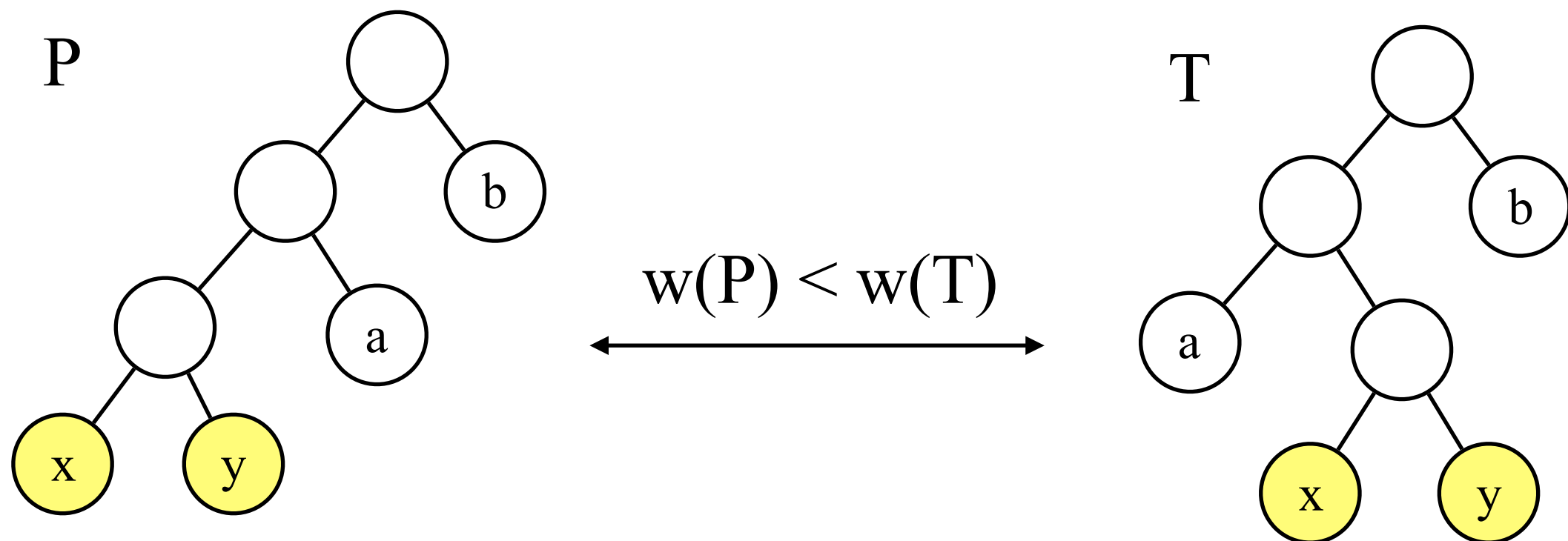
$$w(T') = w(T) - z.\text{freq}.$$



Proof of Claim 3

Suppose tree T does not represent the optimal code, then there exists a tree P representing the optimal code so that $w(P) < w(T)$ and x, y are siblings in P (due to Claim 1).

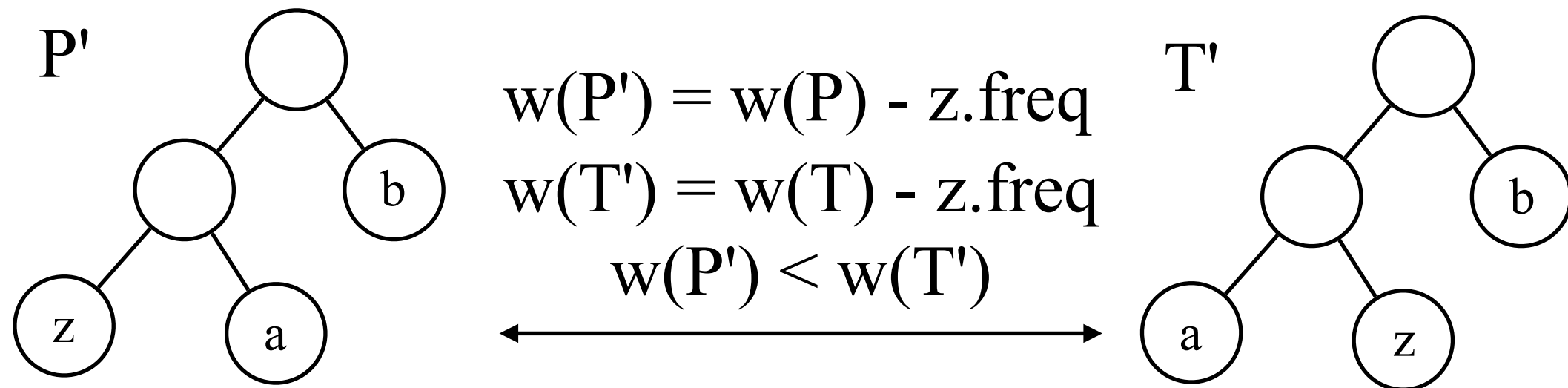
Let P' be the tree obtained from P by replacing the subtree rooted at the parent of x and y with a node z and letting $z.\text{freq} = x.\text{freq} + y.\text{freq}$.



Proof of Claim 3

Suppose tree T does not represent the optimal code, then there exists a tree P representing the optimal code so that $w(P) < w(T)$ and x, y are siblings in P .

Let P' be the tree obtained from P by replacing the subtree rooted at the parent of x and y with a node z and letting $z.\text{freq} = x.\text{freq} + y.\text{freq}$.



$w(P') < w(T')$ contradicts the optimality of T' , completing the proof of Claim 3.