Introduction to Algorithms

Meng-Tsung Tsai

11/12/2019

Announcements

Programming Quiz 1 will be held in EC 315/316/324 this Saturday (Nov 16) 13:30 - 17:30.

There are 5 problem sets and you may bring codes/slides/ebooks (electronic copies) with you using a USB flash drive and/or physical books/cheeting sheets.

The total size of e-files cannot exceed 200 MB, the number of physical books is at most 2, and the number of cheeting sheets is at most 4.

1. (60%) Monotonic Paths -- A Yes/No problem.
2. (20%) Young Tableau -- A variation. You need to get familar with how to search in a Young tableau.
3. (15%) Enclosing Squares -- A variation.
4. (15%) A challening problem. (DP/Greedy/D&C)
5. (15%) A more challenging problem. (DP/Greedy/D&C)

It is hard to get fewer than 30 points. Please attend this quiz.

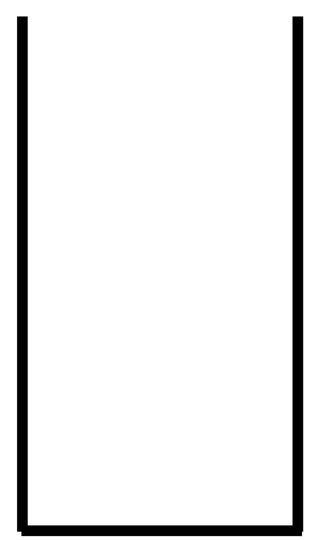
Amortized Analysis

Stack operations

Given an empty stack S and a sequence of Push(S, x), Pop(S), 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

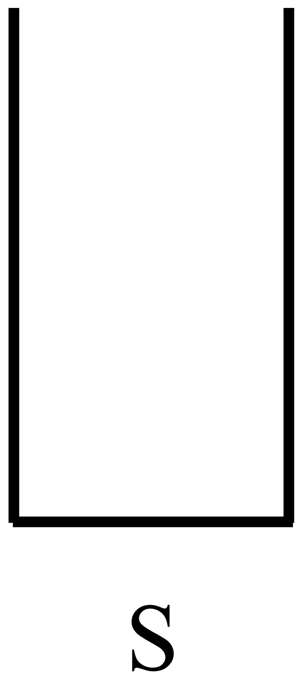
Push(S, 1)

Push(S, 3)

Pop(S)

...

Push and Pop



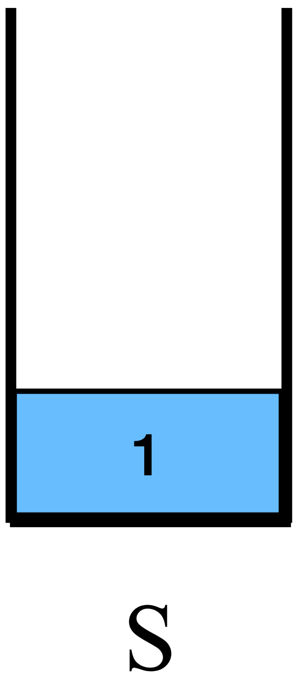
Push(S, 1)

Push(S, 3)

Pop(S)

...

Push and Pop



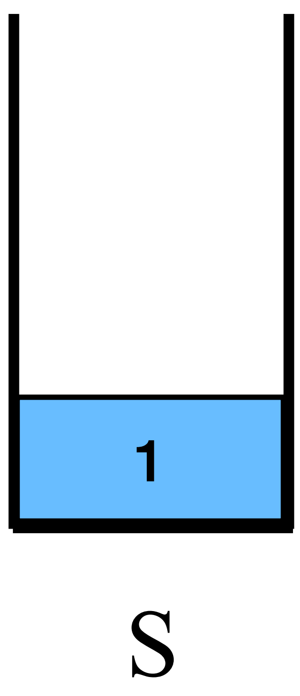
Push(S, 1)

Push(S, 3)

Pop(S)

...

Push and Pop



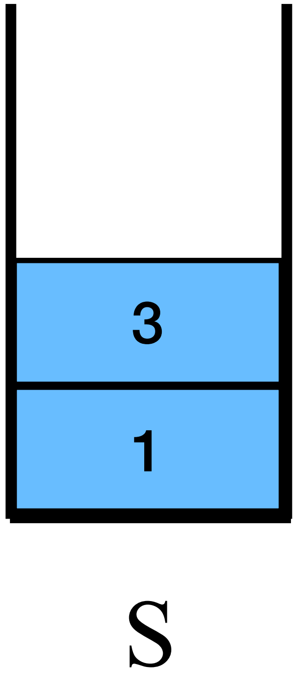
Push(S, 1)

Push(S, 3)

Pop(S)

…

Push and Pop



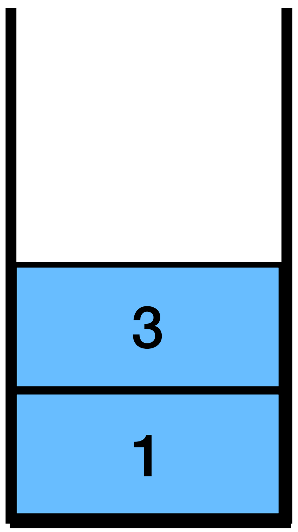
Push(S, 1)

Push(S, 3)

Pop(S)

...

Push and Pop



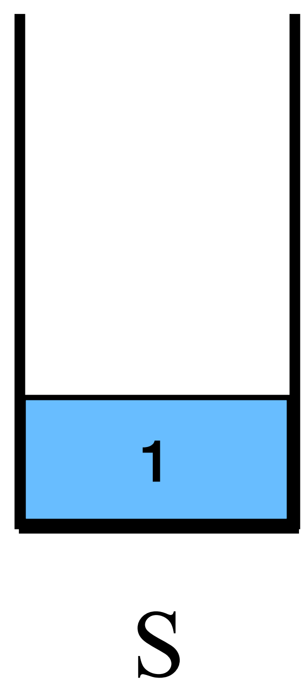
Push(S, 1)

Push(S, 3)

Pop(S)

...

Push and Pop

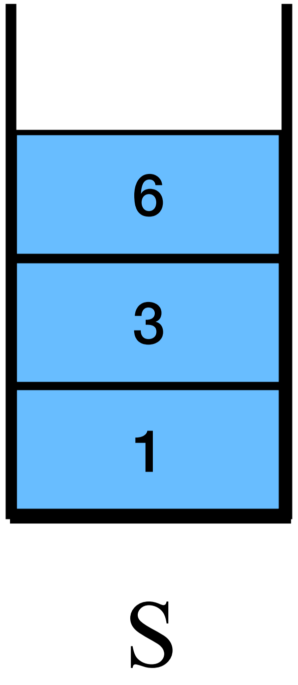


Push(S, 1)
Push(S, 3)
Pop(S)
...

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

MultiPop

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



MultiPop(S, 2)

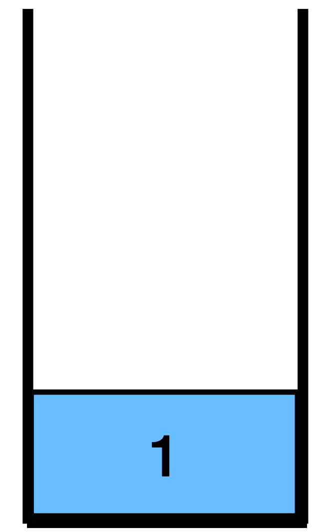
Push(S, 3)

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)

...

Eevery MultiPop operation needs O(k) time.

Naive analysis

Note that there are at most n Push(S, x) in a sequence of n operations.

$$\Downarrow$$

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

$$\Downarrow$$

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

Aggregate analysis

Note that there are at most n Push(S, x) in a sequence of n operations.

$\Downarrow$

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

$\Downarrow$

The running time of a sequence of n Pop(), Push(), and 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 $\sum_{1 \leq i \leq n} c_i \leq \sum_{1 \leq i \leq n} \hat{c}_i = O(n)$.

The potential method

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

At Push(S, x), instead of prepaying for x, we prepay for the potential change to the system. We define the system potential to be # elements in stack S. Let $\mathbf{\Phi}_i$ be the system potential after i operations are performed.

The potential method

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

$$
\begin{aligned}
O(n) = \sum_{1\le i\le n} \hat{c}_i &= \sum_{1\le i\le n} c_i + \mathbf{\Phi}_i - \mathbf{\Phi}_{i-1} \\
&= \mathbf{\Phi}_n - \mathbf{\Phi}_0 + \sum_{1\le i\le n} c_i \\
&\ge \sum_{1\le i\le n} c_i
\end{aligned}
$$

Incrementing a binary counter

Given a k-bit binary counter and a sequence of n Inc() operations.

What is the worst-case running time for a sequence of n Inc() operations?

Inc

```
Inc(A){
    for(i=0; i<k; ++i){
        A[i] = 1 - A[i];
        if(A[i] == 1) break;
    }
}
```

counter	A[3] (2^3)	A[2] (2^2)	A[1] (2^1)	A[0] (2^0)
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0

Inc

```
Inc(A){
    for(i=0; i<k; ++i){
        A[i] = 1 - A[i];
        if(A[i] == 1) break;
    }
}
```

The values of the gray cells are changed by Inc().

counter	A[3] (2^3)	A[2] (2^2)	A[1] (2^1)	A[0] (2^0)
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0

Naive analysis

Each Inc() changes at most k bits.

$\Downarrow$

The running time of a sequence of n Inc() operations is O(nk) in the worst case.

Aggregate analysis

A[0] changes every Inc(), A[1] changes every other Inc(), A[2] changes once every four Inc(), ...

Total running time is thus O(n)+O(n/2)+... = O(n).

counter	A[3] (2^3)	A[2] (2^2)	A[1] (2^1)	A[0] (2^0)
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0

The accounting method

i-th Inc()	actual cost c_i	amortized cost $\hat{c}_i$
set α 1-bits to 0 and set β 0-bits to 1	$\alpha + \beta$	$2\beta = 2$ (β must be 1)

While setting a bit to 1, it prepays the cost for resetting the bit back to 0. We have $\sum_{1 \leq i \leq n} c_i \leq \sum_{1 \leq i \leq n} \hat{c}_i = O(n)$.

The potential method

i-th Inc()	actual cost c_i	amortized cost $\hat{c}_i$
set α 1-bits to 0 and set β 0-bits to 1	$\alpha + \beta$	$\hat{c}_i = c_i + \mathbf{\Phi}_i - \mathbf{\Phi}_{i-1}$ $= \alpha + \beta - (\alpha - \beta)$ $= 2\beta = 2$

We define the system potential to be # 1-bits in array S. Let $\mathbf{\Phi}_i$ be the system potential after i operations are performed.

The potential method

i-th Inc()	actual cost c_i	amortized cost $\hat{c}_i$
set α 1-bits to 0 and set β 0-bits to 1	$\alpha + \beta$	$\hat{c}_i = c_i + \boldsymbol{\Phi}_i - \boldsymbol{\Phi}_{i-1}$ $= \alpha + \beta - (\alpha - \beta)$ $= 2\beta = 2$

$$\color{red}{O(n)} = \sum_{1 \le i \le n} \hat{c}_i = \sum_{1 \le i \le n} c_i + \boldsymbol{\Phi}_i - \boldsymbol{\Phi}_{i-1}$$

$$= \boldsymbol{\Phi}_n - \boldsymbol{\Phi}_0 + \sum_{1 \le i \le n} c_i$$

$$\ge \sum_{1 \le i \le n} c_i$$

Exercise

Input: an n by n boolean matrix M.

Output: Mv for all v $\in \{0, 1\}^n$.

Show that this problem can be solved in $O(2^n n)$ time.

<u>Example</u>.

$$\text{Let } M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

$$M\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},\ M\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix},\ M\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix},\ M\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

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.

<u>Example</u>.

Let S be "I can can a can."

The binary representation of S is →

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.

<u>Example</u>.

Let S be "I can can a can."

The binary representation of S is →

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

0001010011011010
0110110110010011
0110000

Raw data: 128 bits.

⇒

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
0110000

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

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

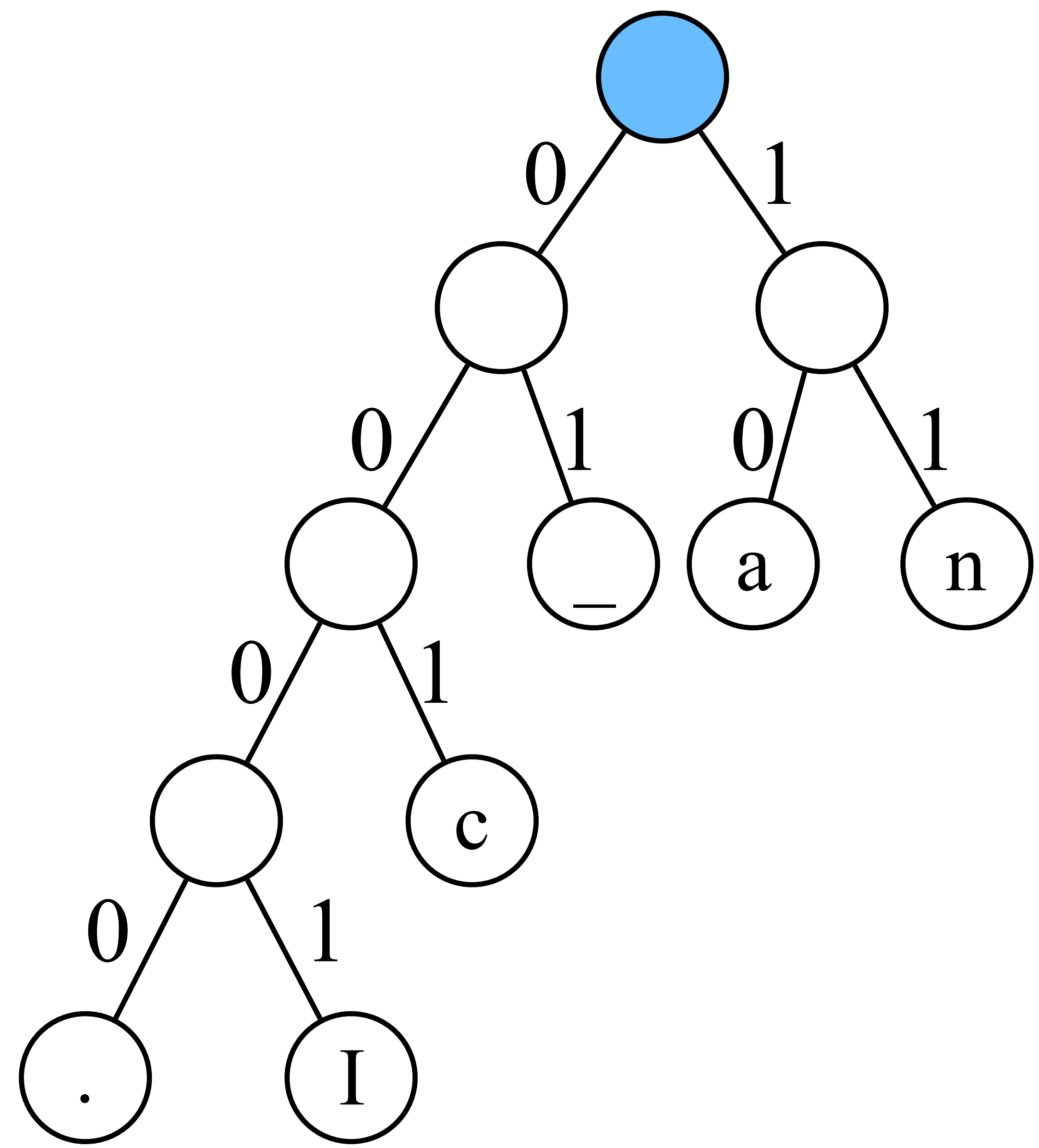
character	codeword
'n'	11
'a'	10
' '	01
'c'	001
'I'	0001
'.'	1

a prefix

Not a prefix code.

Prefix code - decoding

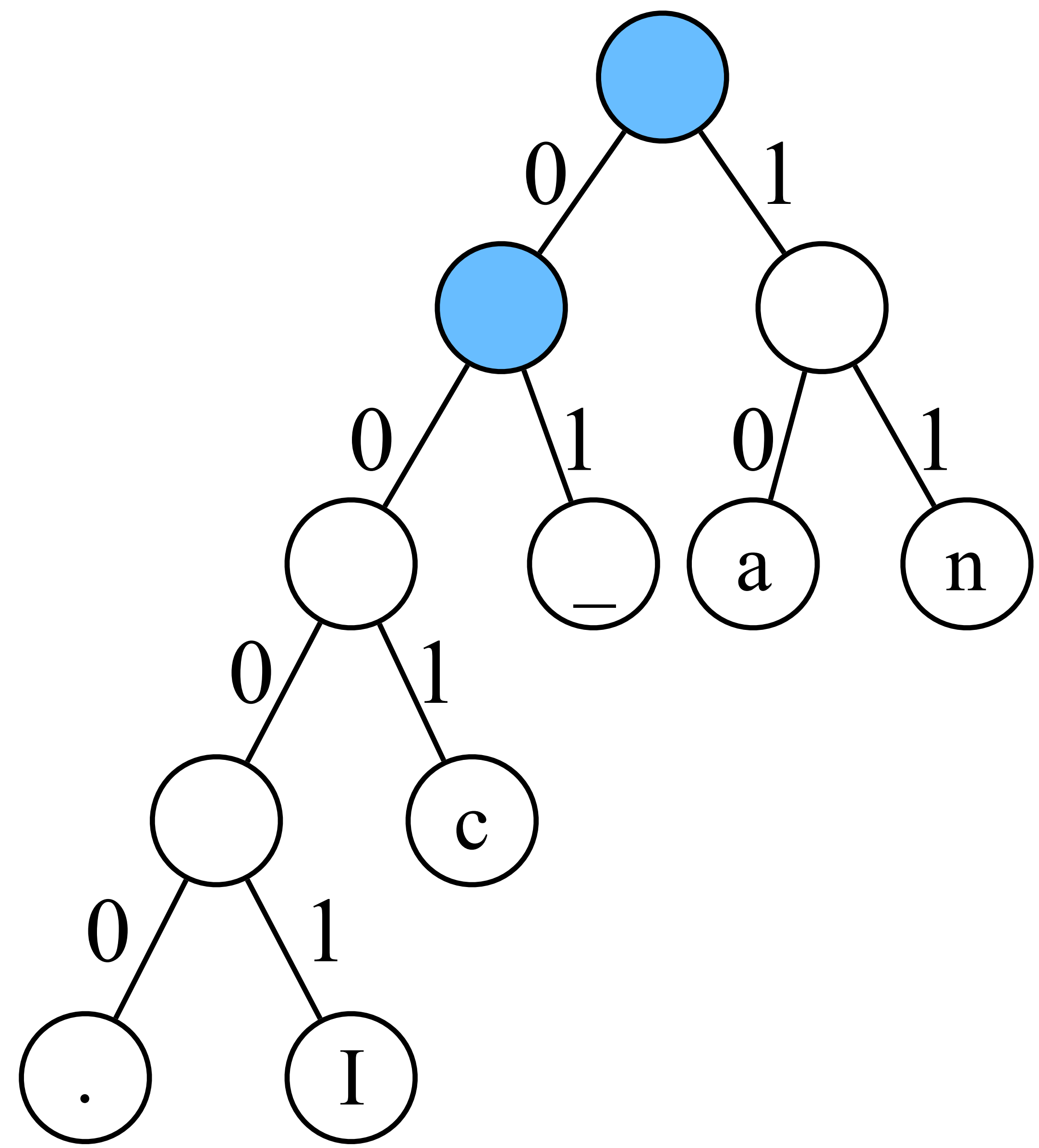
character	codeword
’n’	11
‘a’	10
‘ ’	01
‘c’	001
‘I’	0001
‘.’	0000



000101001101101001101101100100110110000

Prefix code - decoding

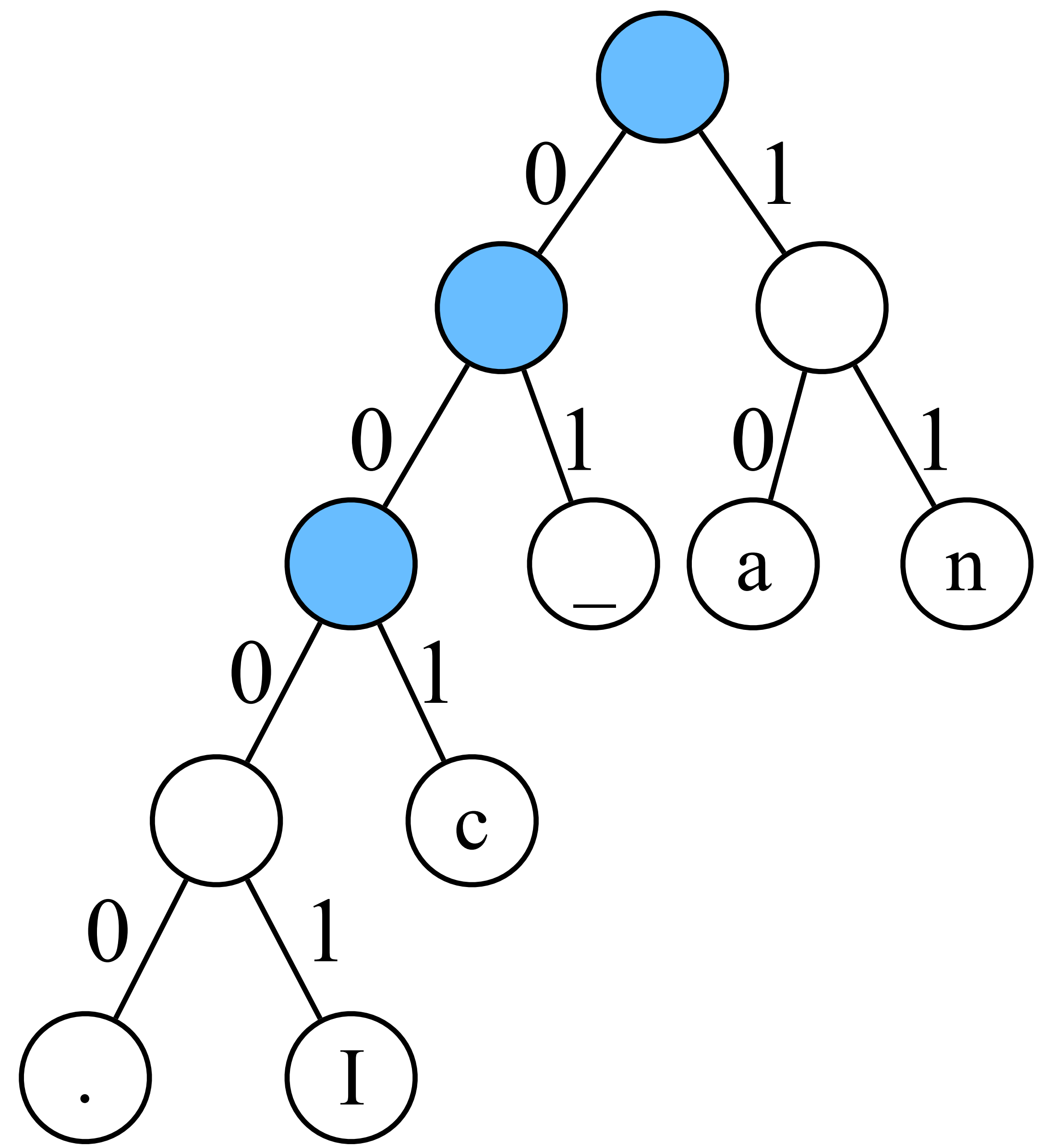
character	codeword
'n'	11
'a'	10
' '	01
'c'	001
'I'	0001
'.'	0000



0001010011011010011011011001001101100000

Prefix code - decoding

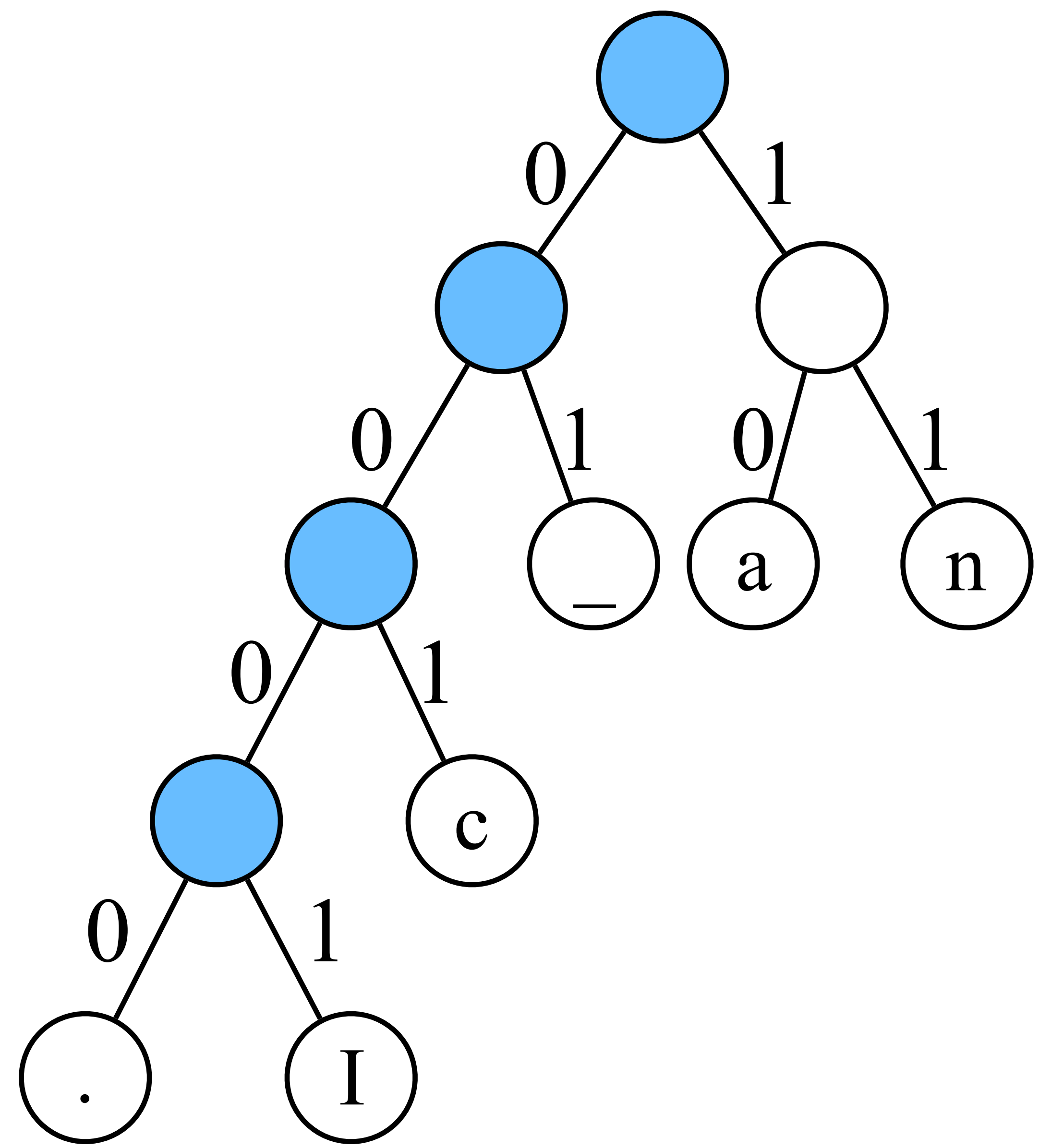
character	codeword
'n'	11
'a'	10
' '	01
'c'	001
'I'	0001
'.'	0000



000101001101101001101101100100110110000

Prefix code - decoding

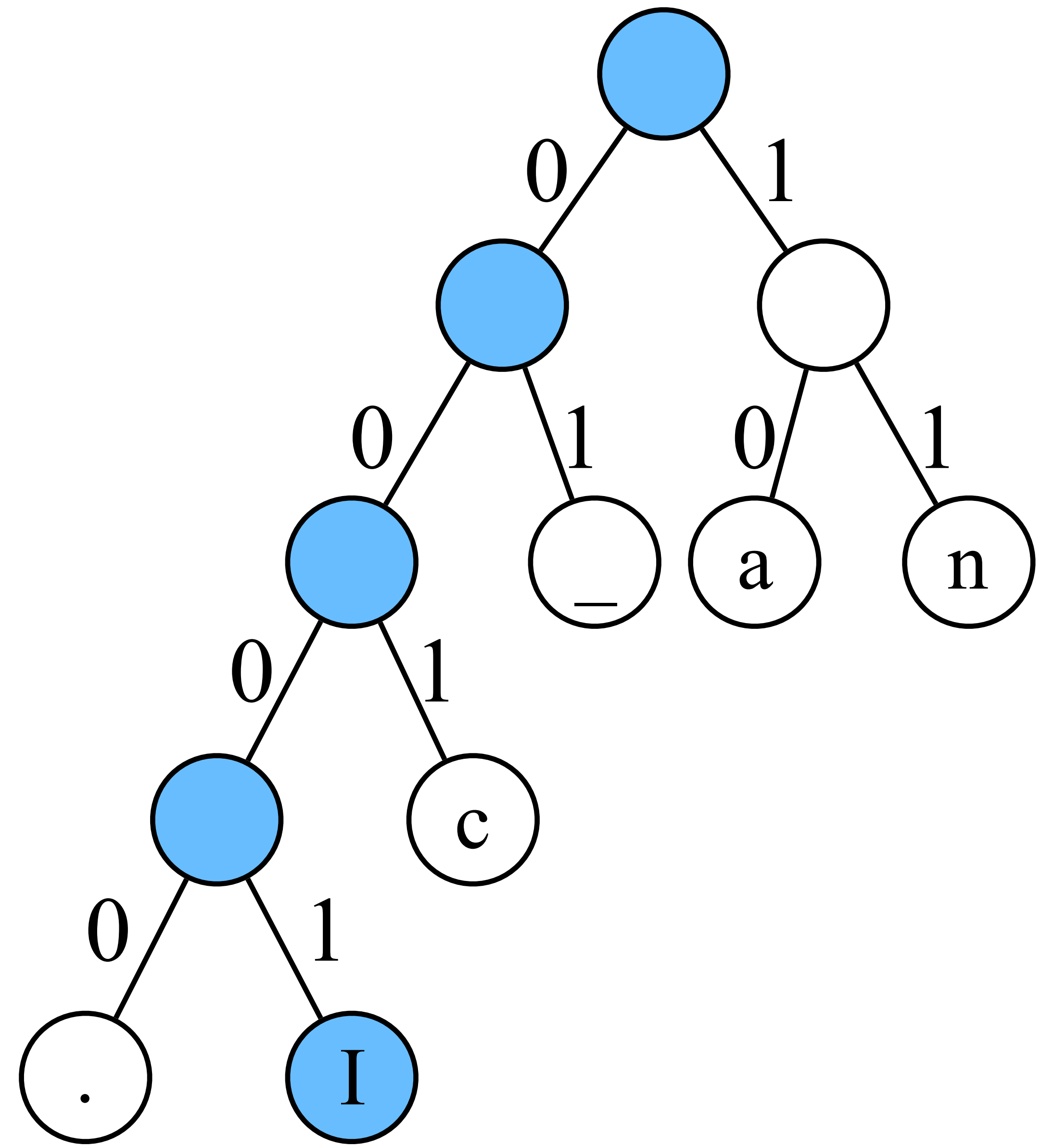
character	codeword
'n'	11
'a'	10
' '	01
'c'	001
'I'	0001
'.'	0000



0001010011011010011011011001001101100000

Prefix code - decoding

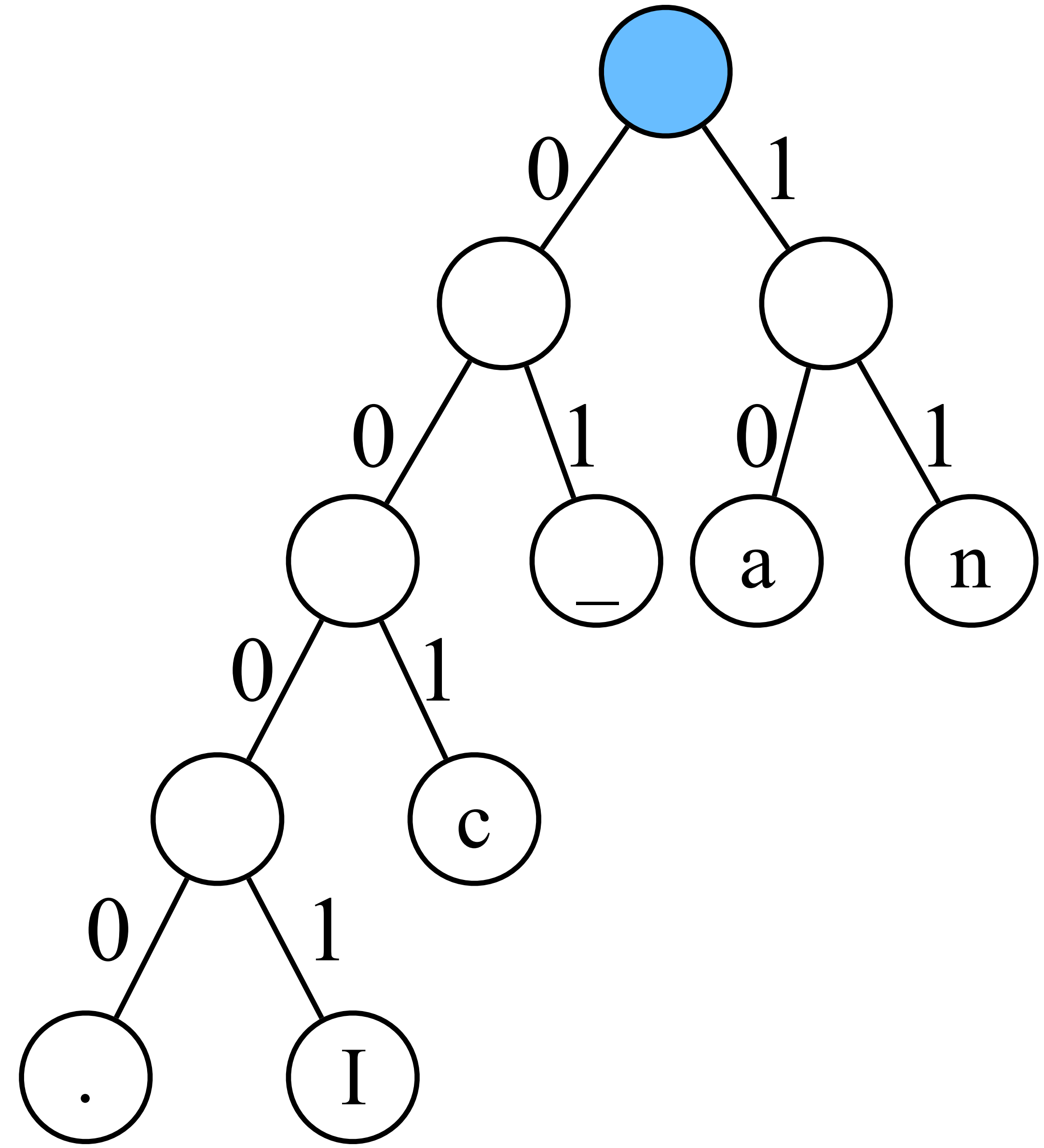
character	codeword
’n’	11
‘a’	10
‘ ’	01
‘c’	001
‘I’	0001
‘.’	0000



00010100110110100110110110010011010000

Prefix code - decoding

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

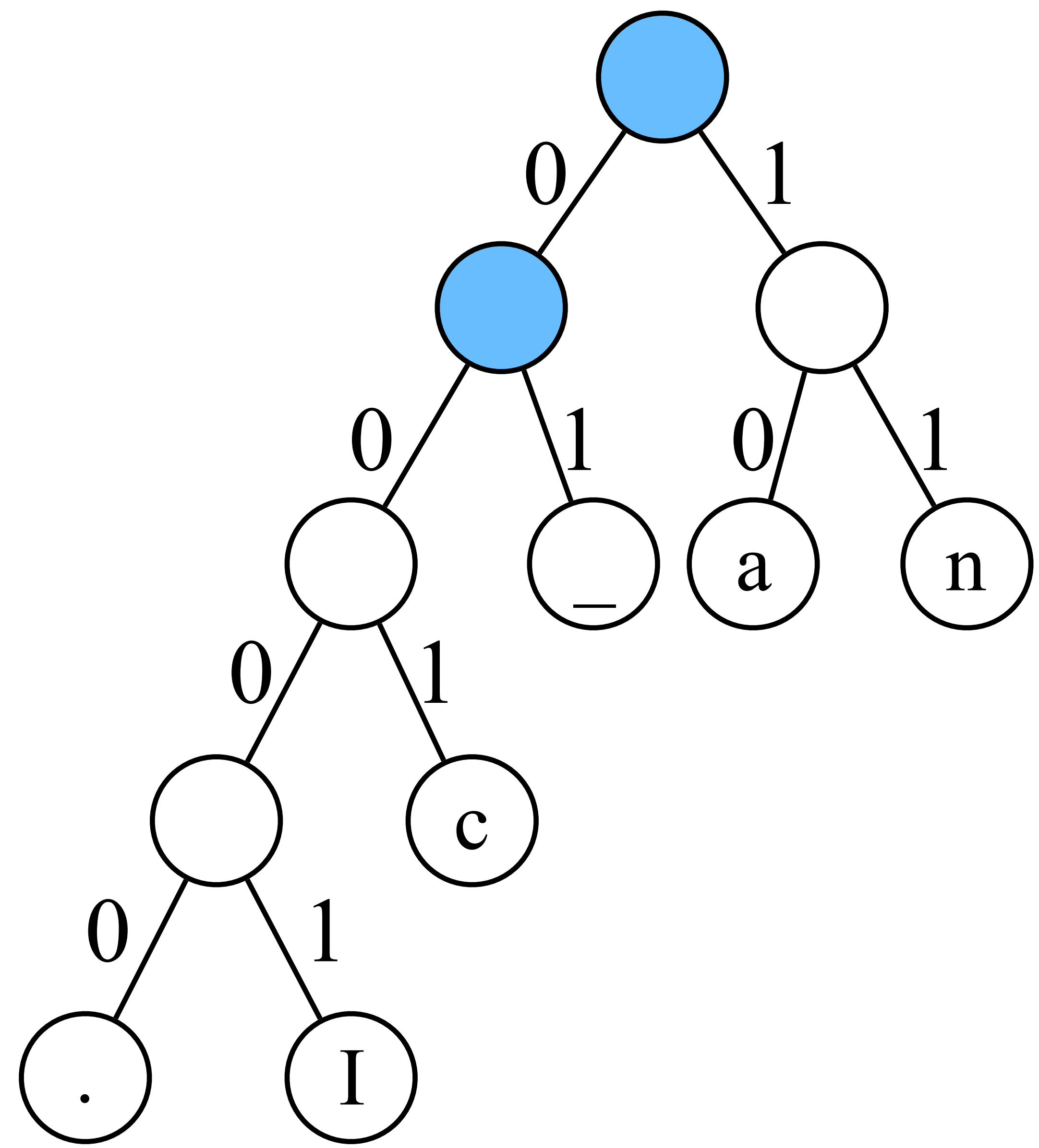


01001101101001101101100100110110000

I

Prefix code - decoding

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

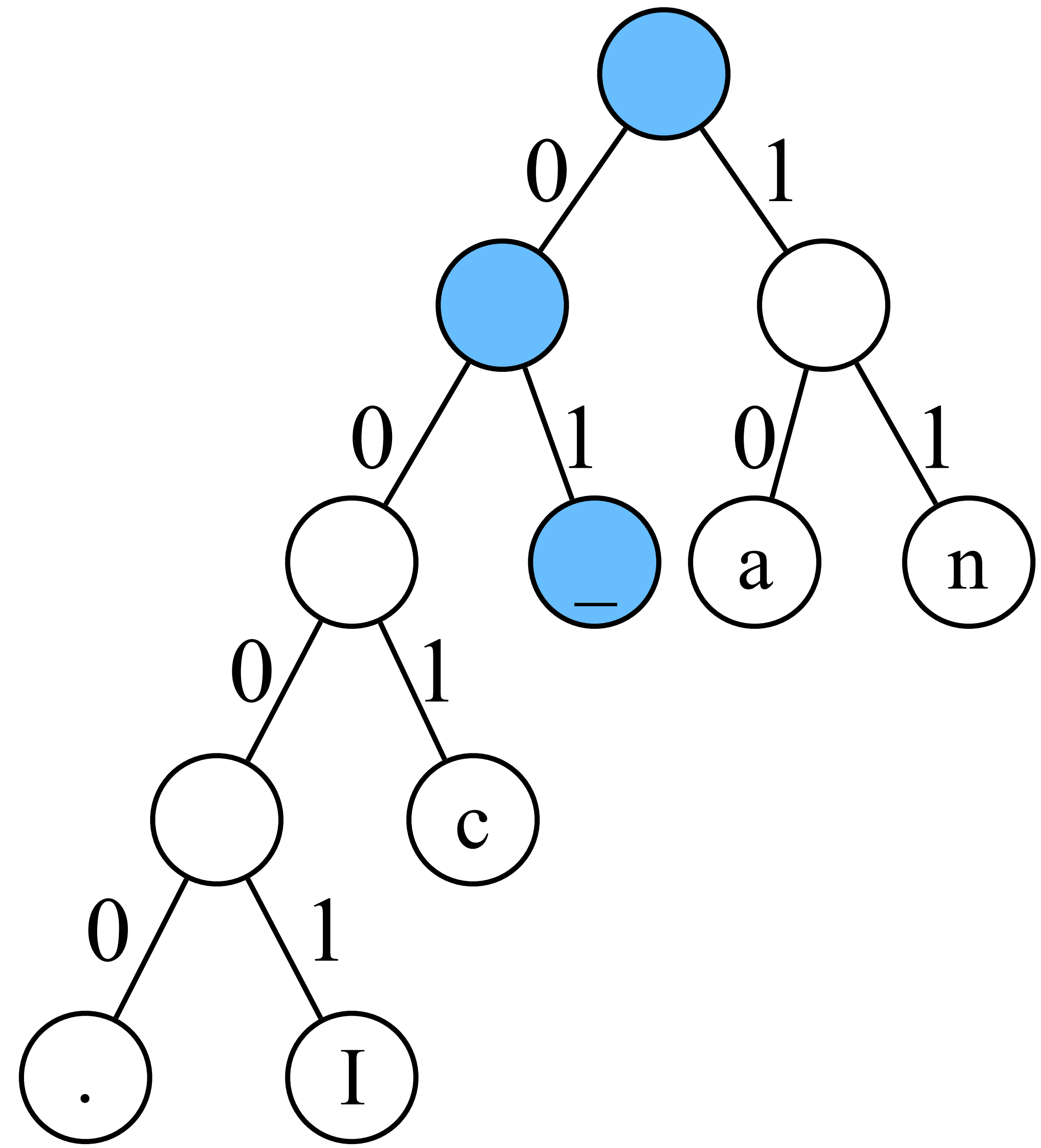


010011011010011011011001001101100000

I

Prefix code - decoding

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

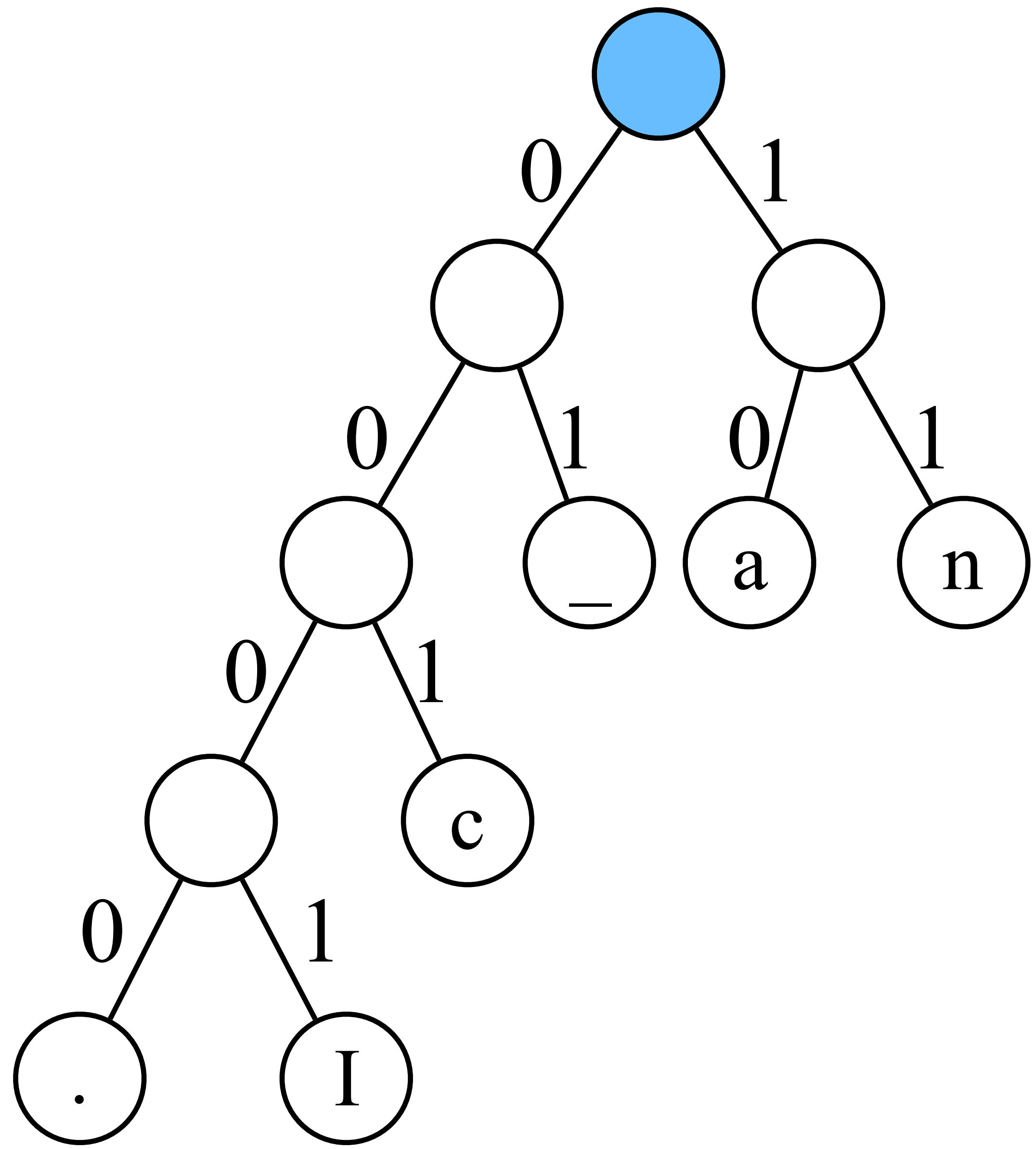


0100110110100110110110010011011000

I

Prefix code - decoding

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

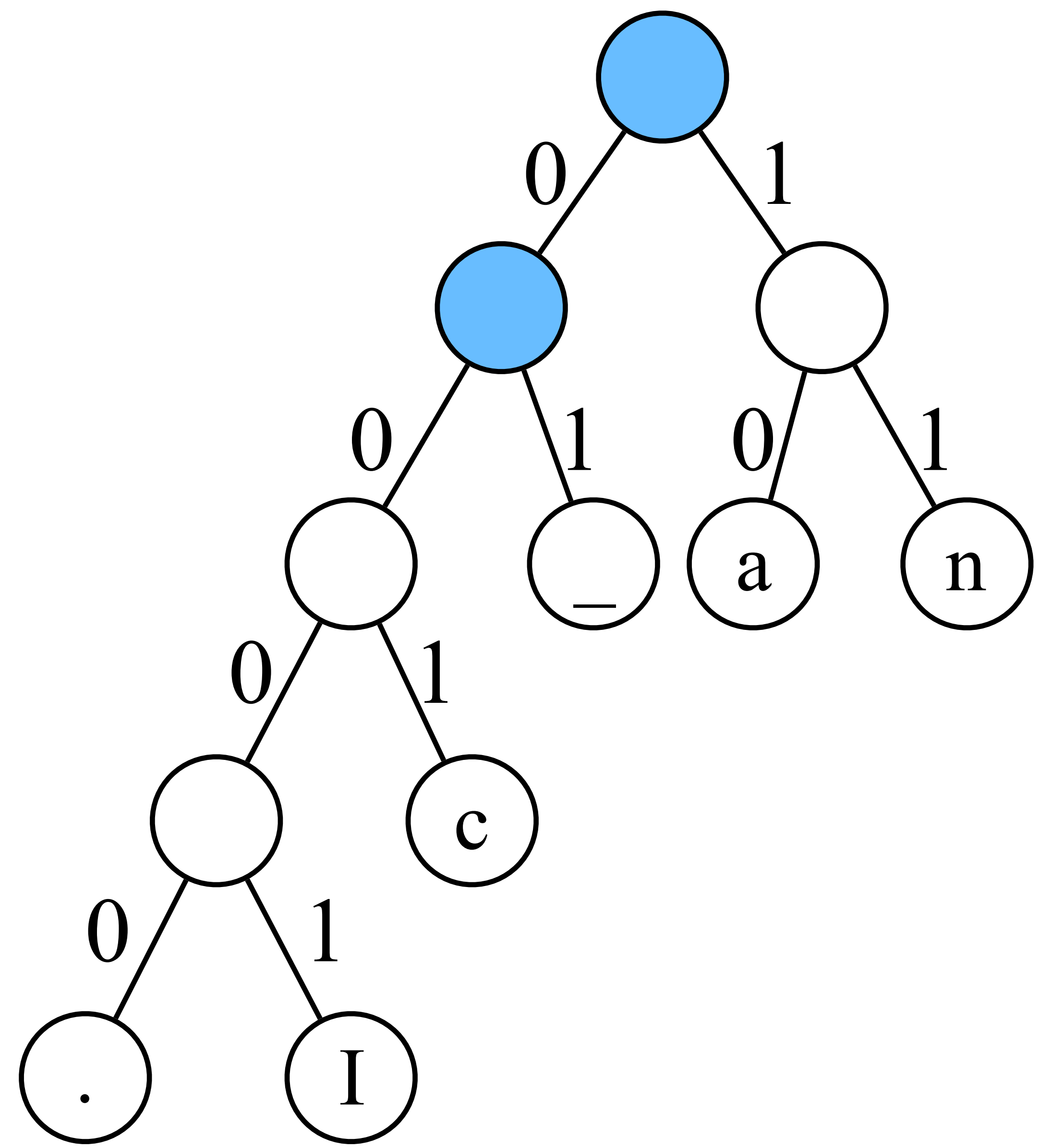


0011011010011011011001001101100000

I_

Prefix code - decoding

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

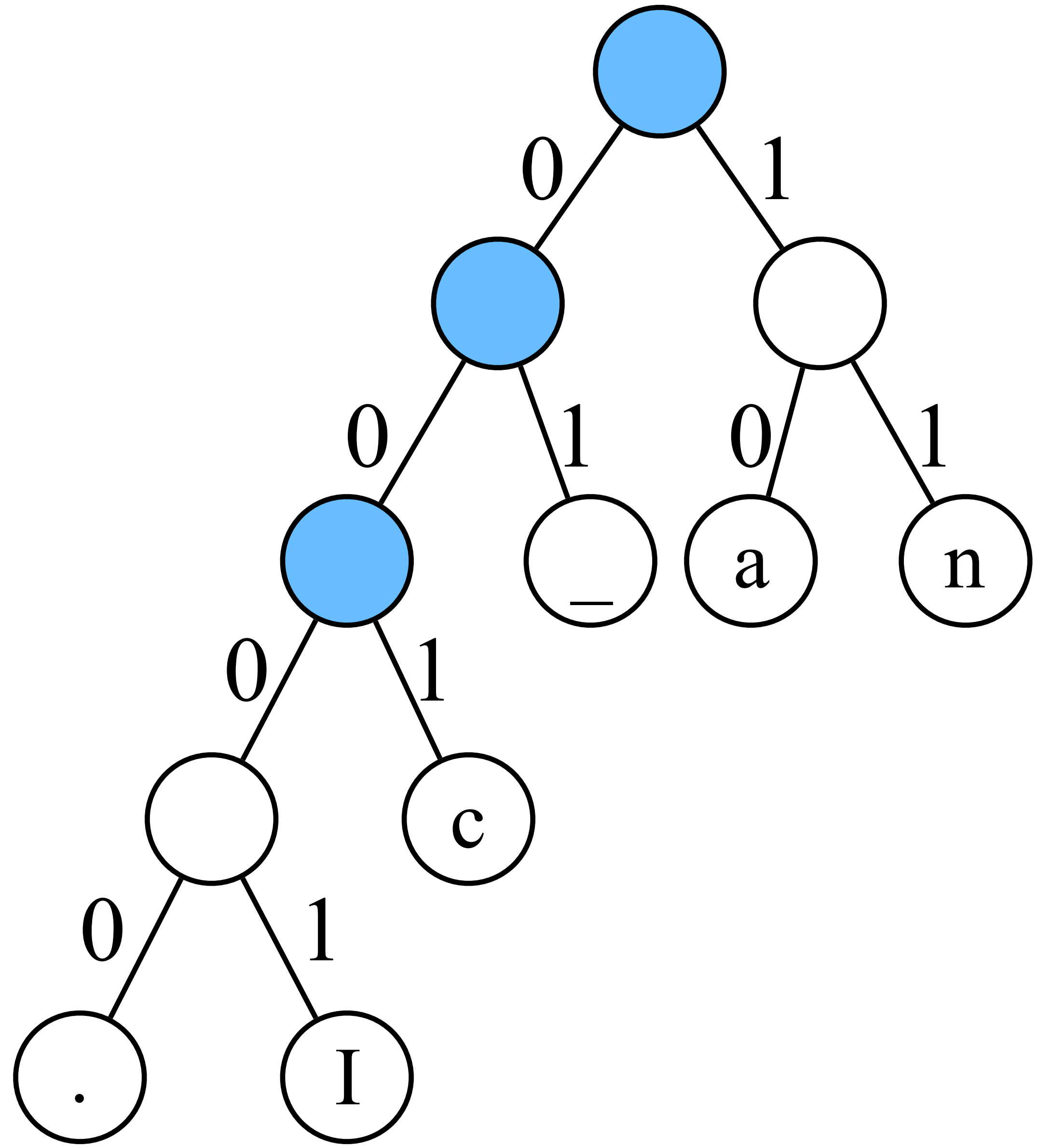


001101101001101101100100110110000

I_

Prefix code - decoding

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

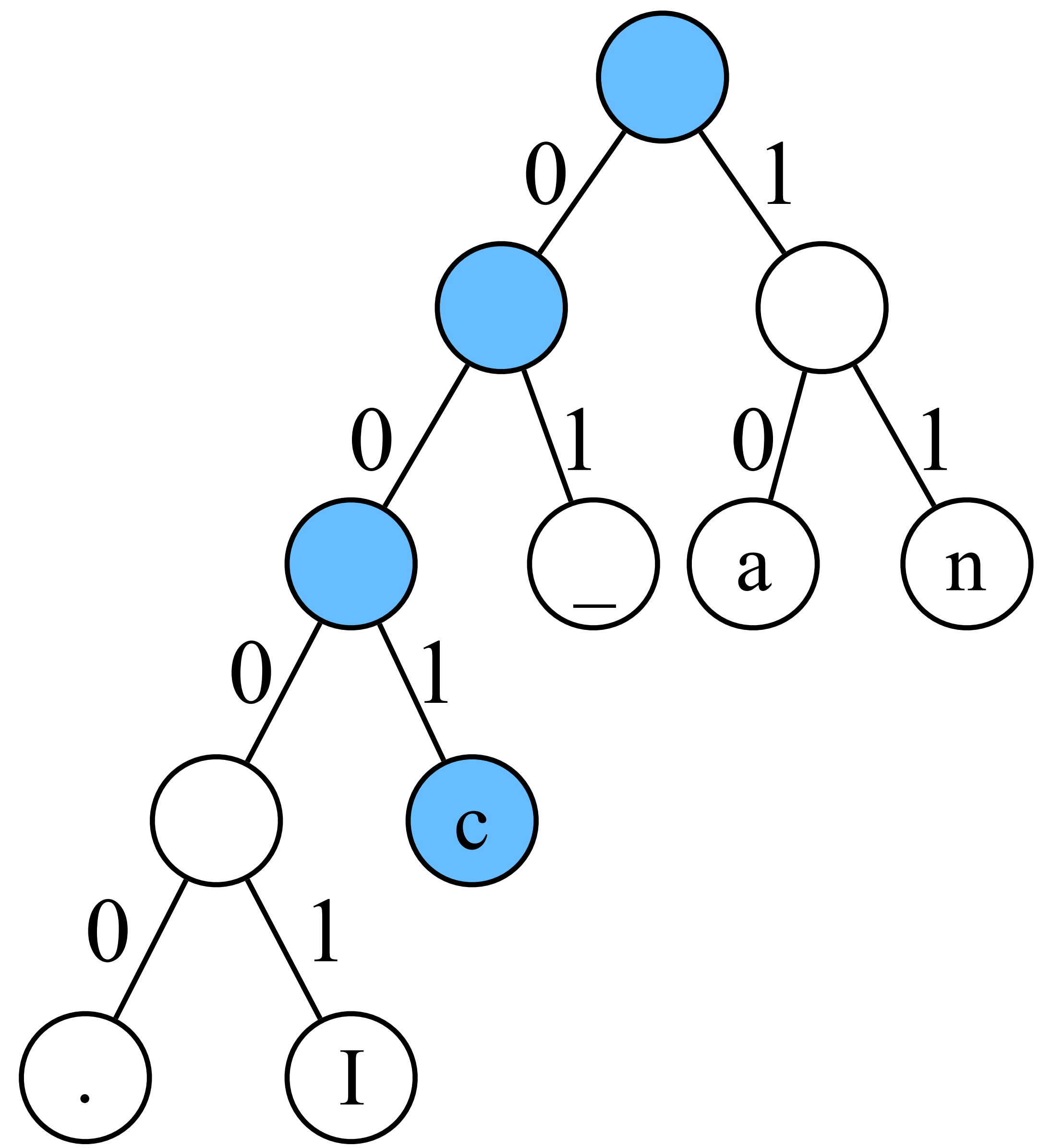


0011011010011011011001001101100000

I_

Prefix code - decoding

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

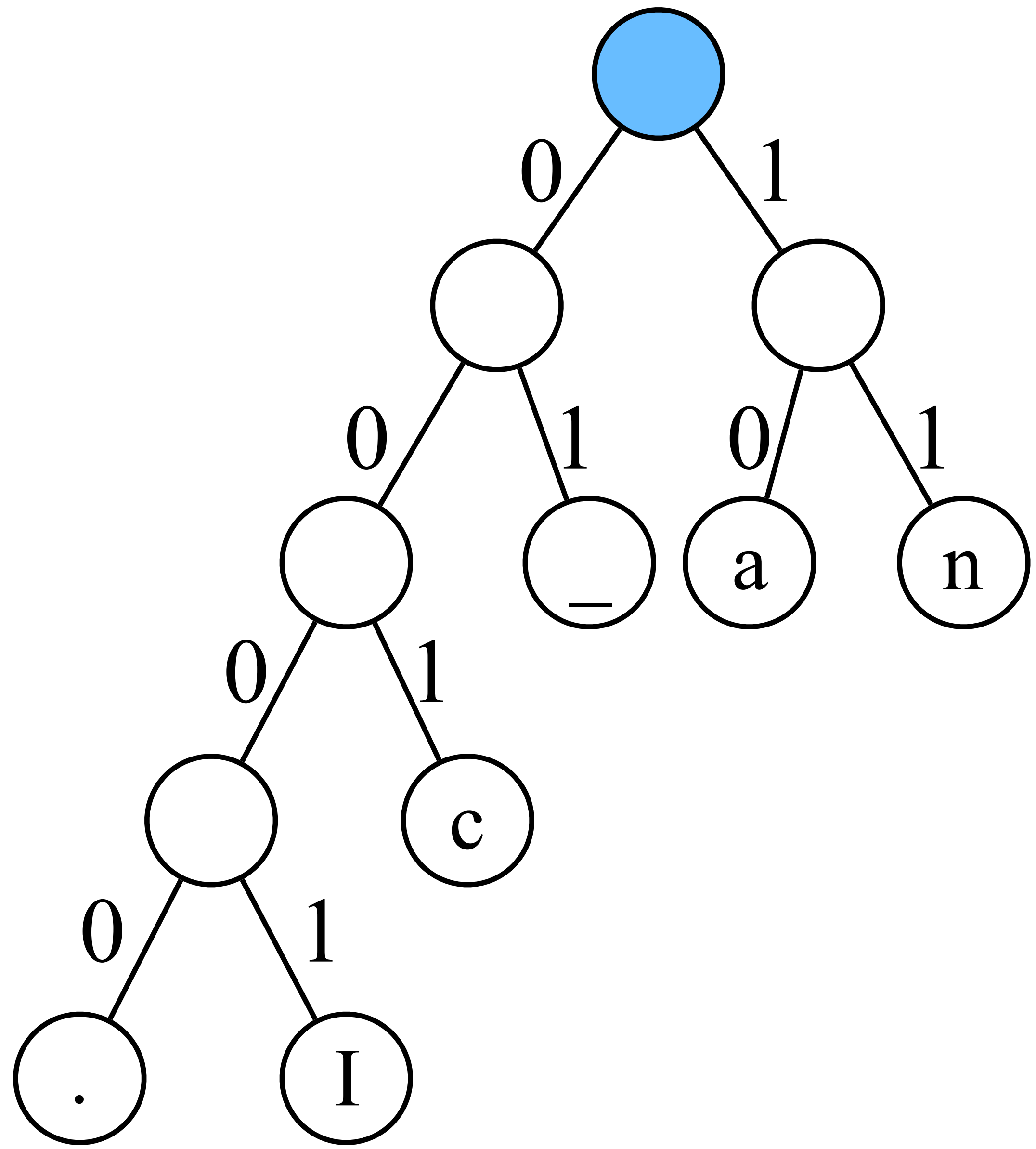


0011011010011011011001001101100001

I_

Prefix code - decoding

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



10110100110110110010011010110000

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

character	frequency
’n’	3
‘a’	4
‘ ’	4
‘c’	3
‘I’	1
‘.’	1

Construct_Huffman_Code(input){

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

While (there are multiple connected components){

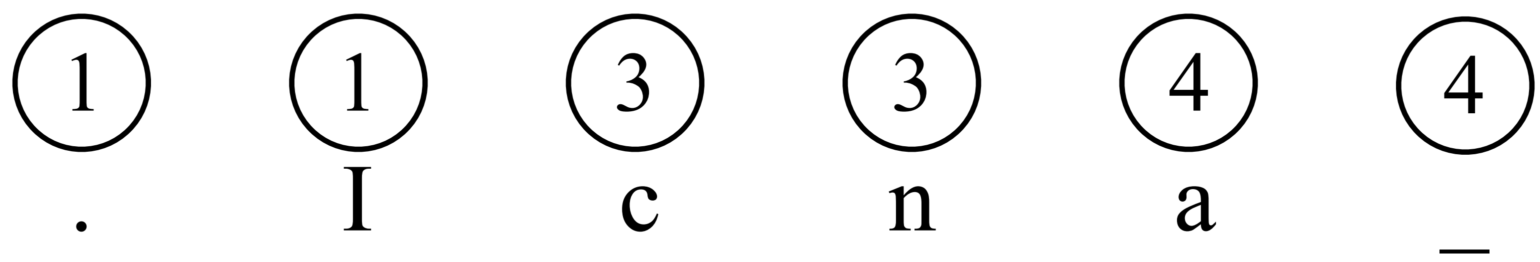
(1) Pick two trees T_1, T_2 whose root has lowest frequencies.

(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.

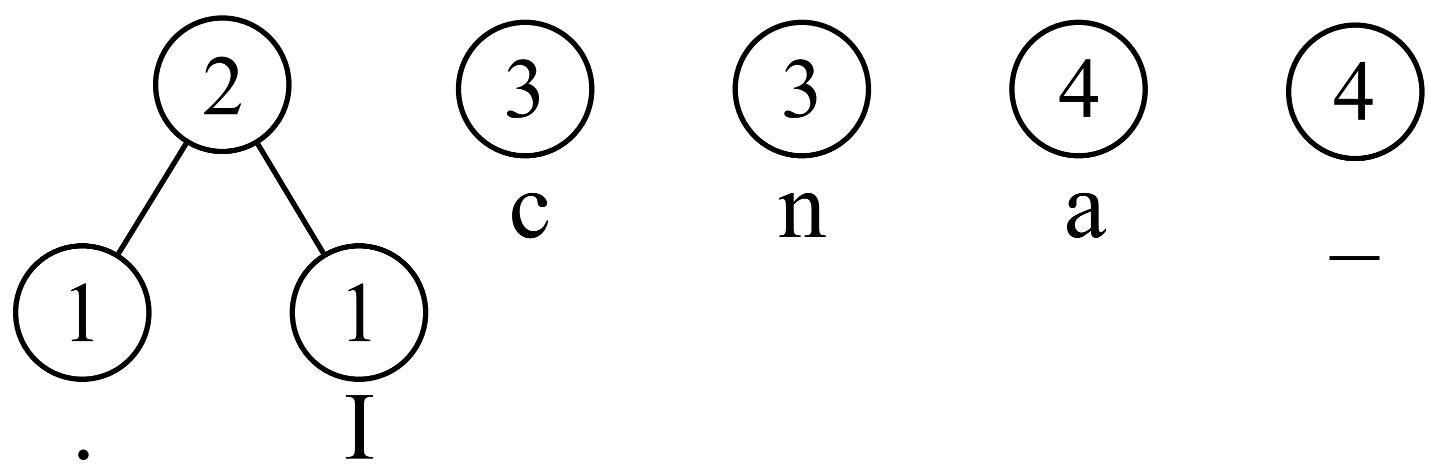
}

}

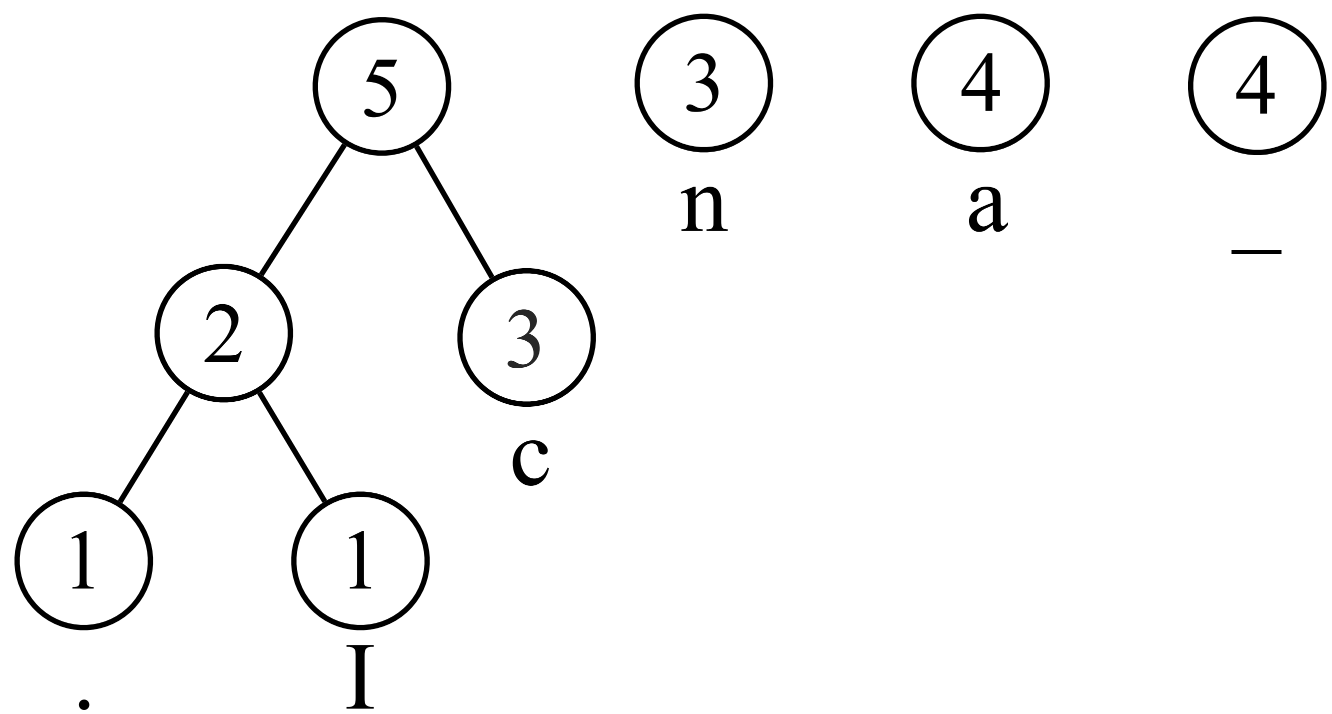
Huffman code - constructing



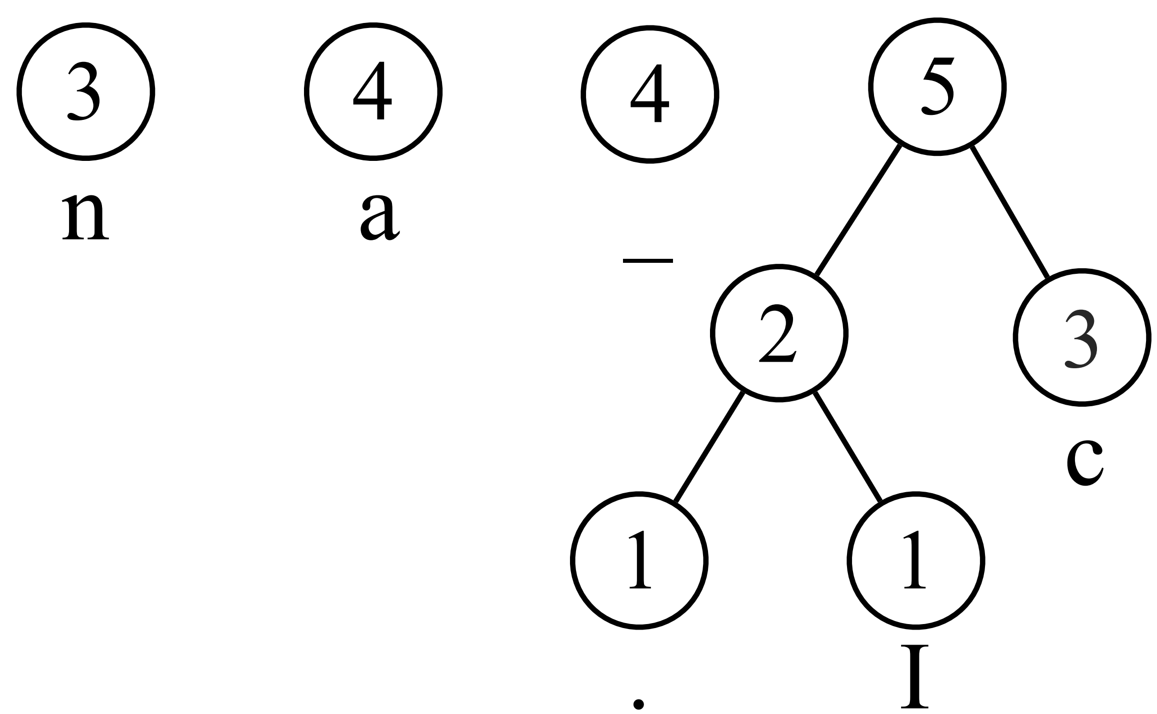
Huffman code - constructing



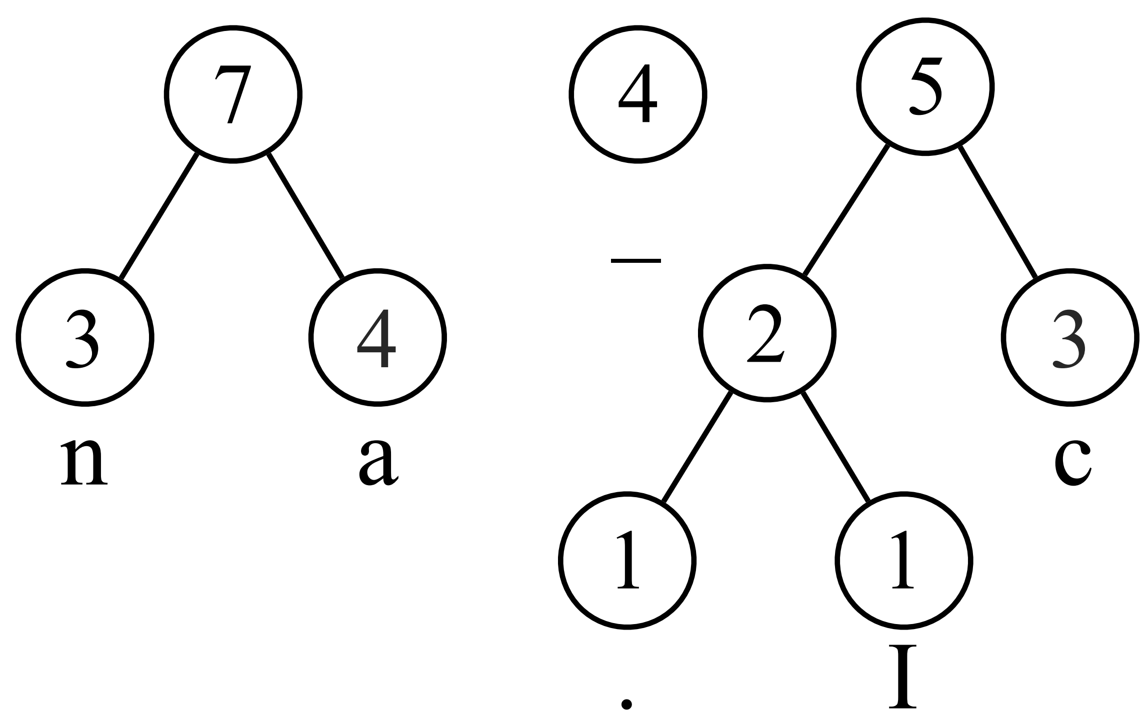
Huffman code - constructing



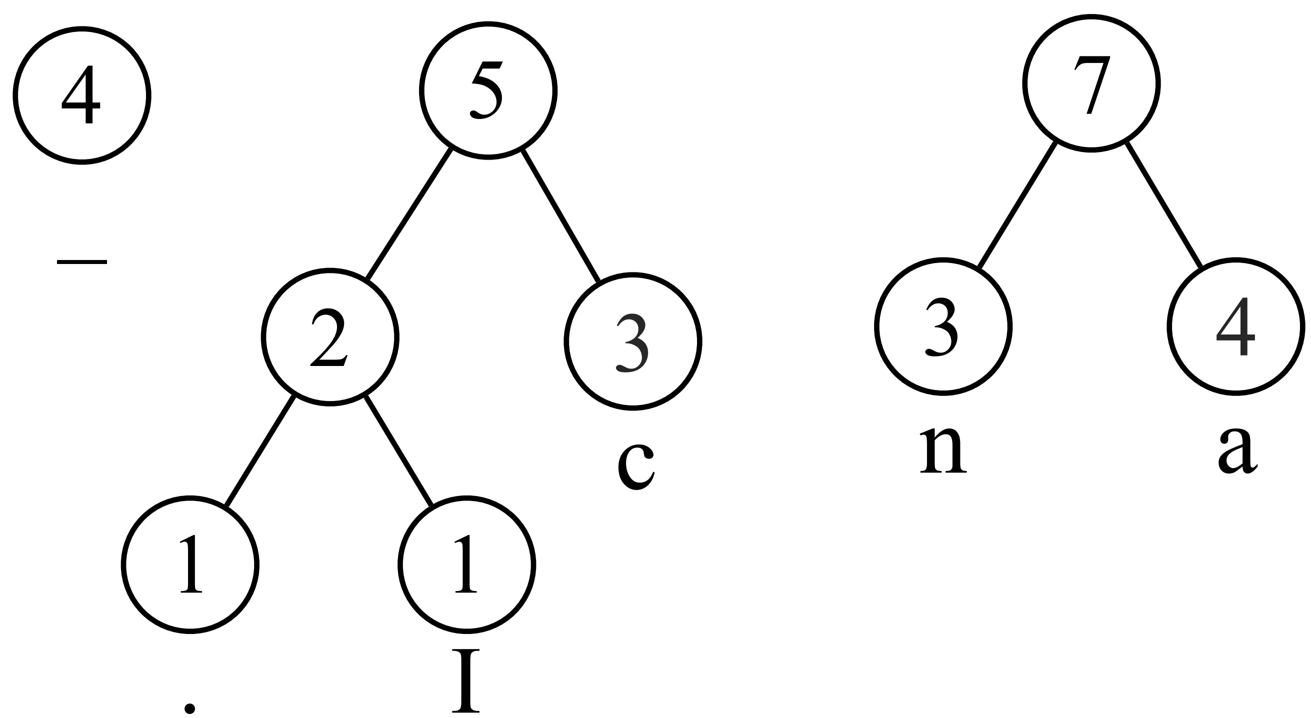
Huffman code - constructing



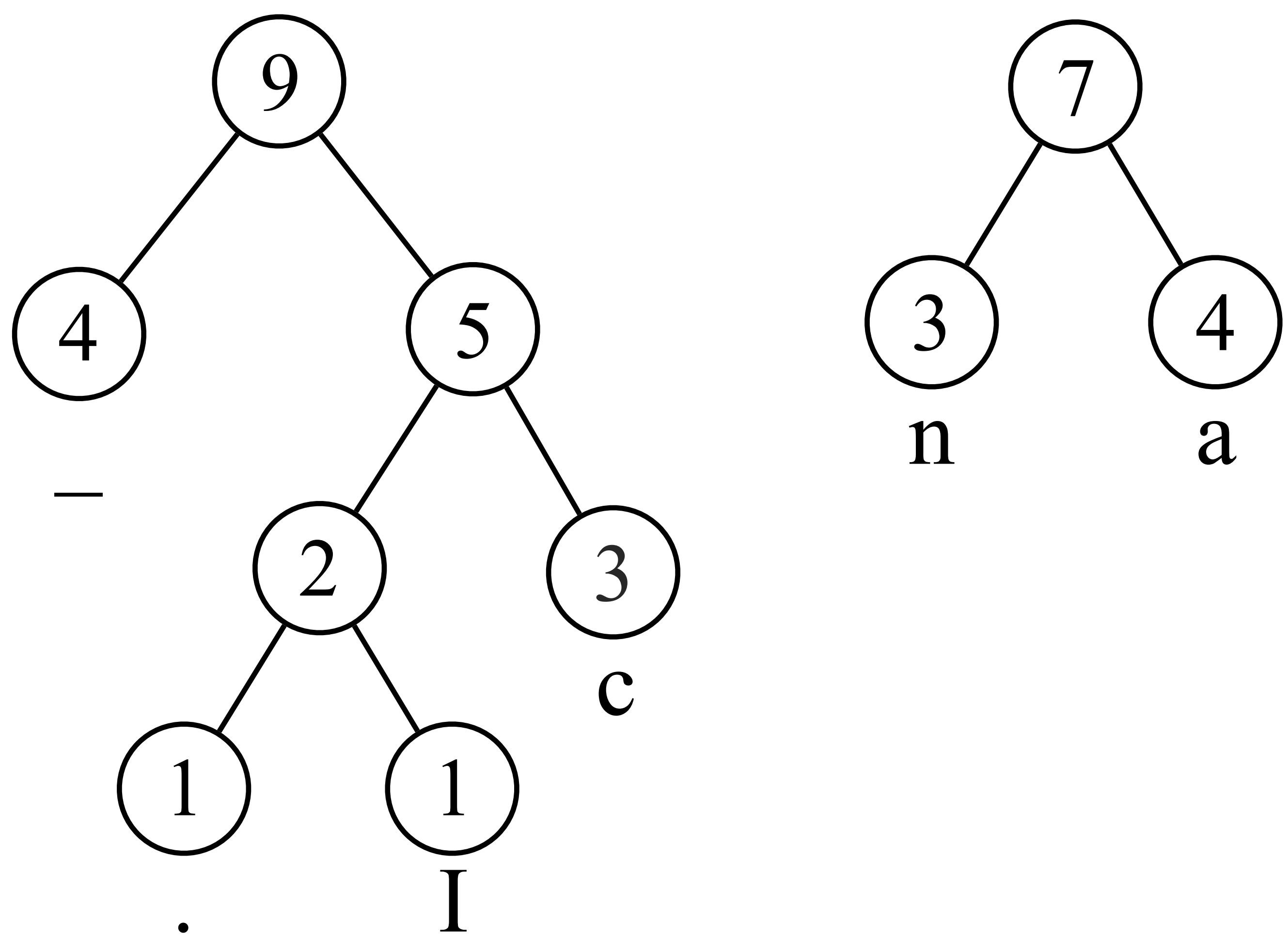
Huffman code - constructing



Huffman code - constructing

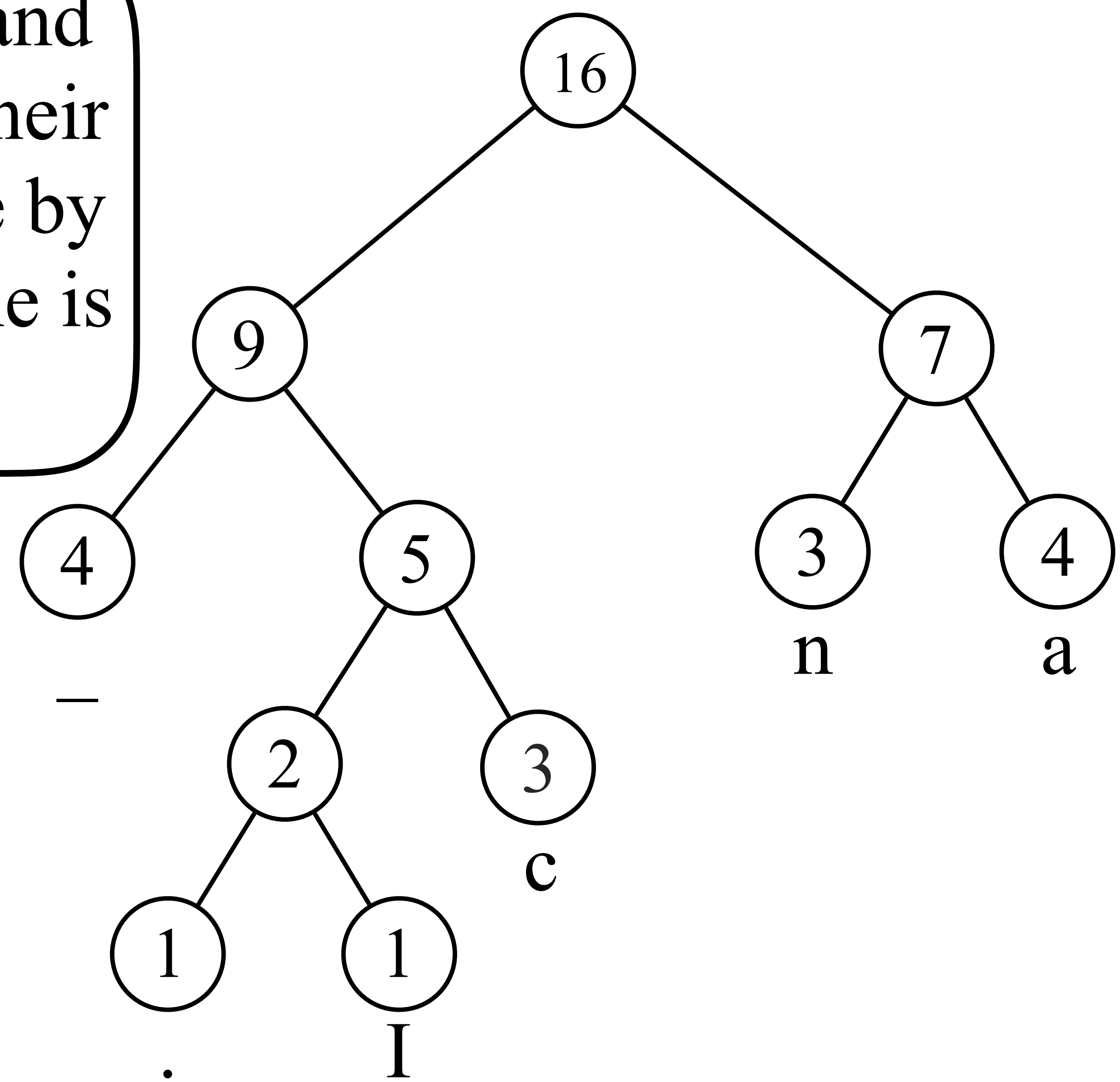


Huffman code - constructing



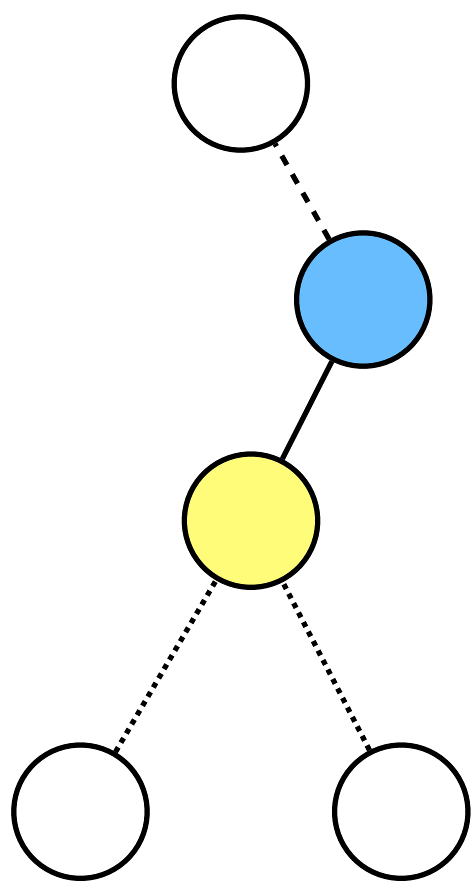
Huffman code - constructing

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).

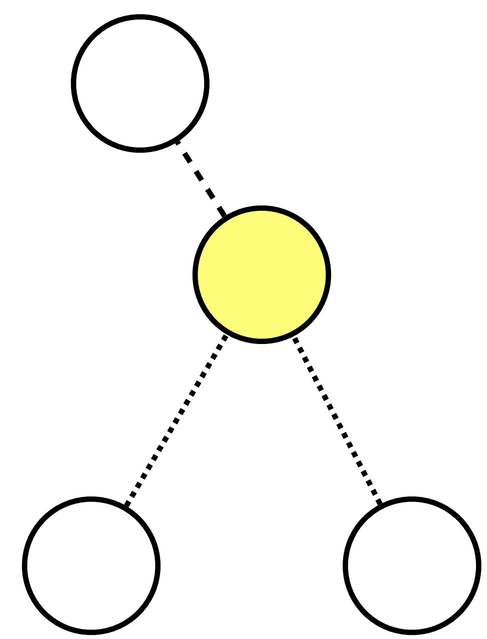


Why is Huffman code optimal?

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.



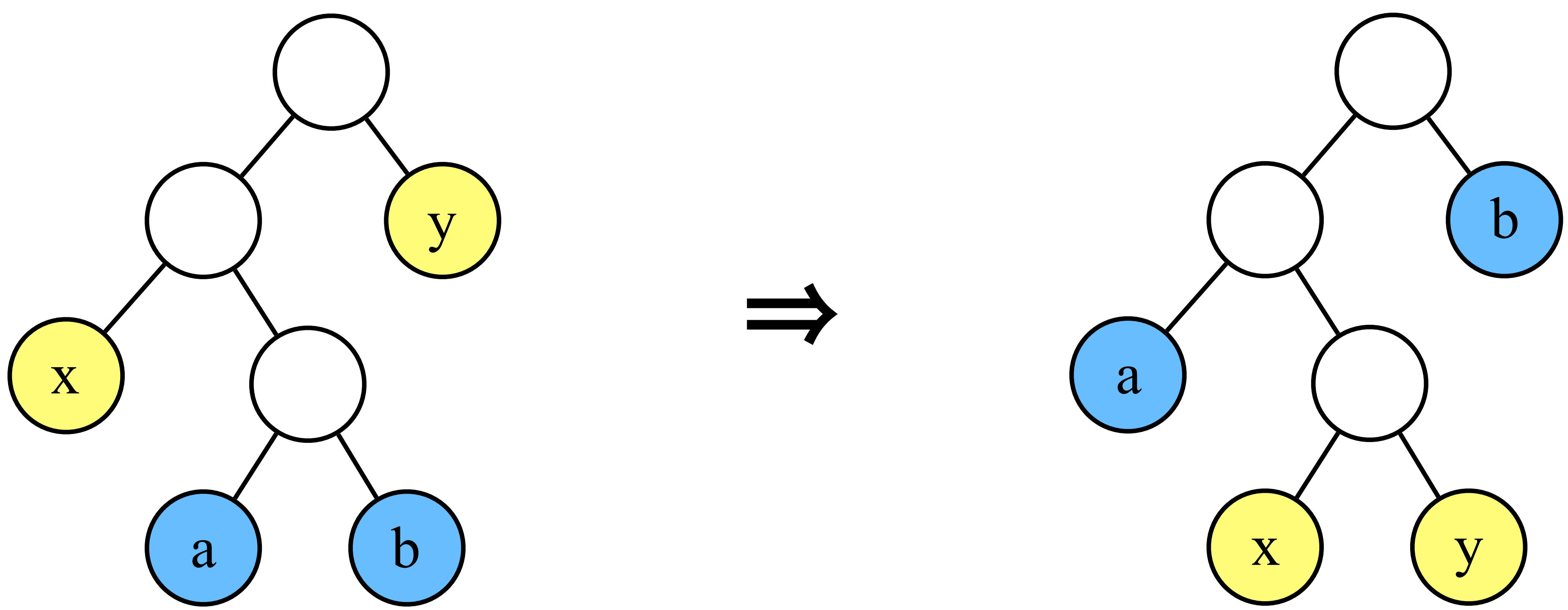
$\Rightarrow$



We can get a more succint 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.

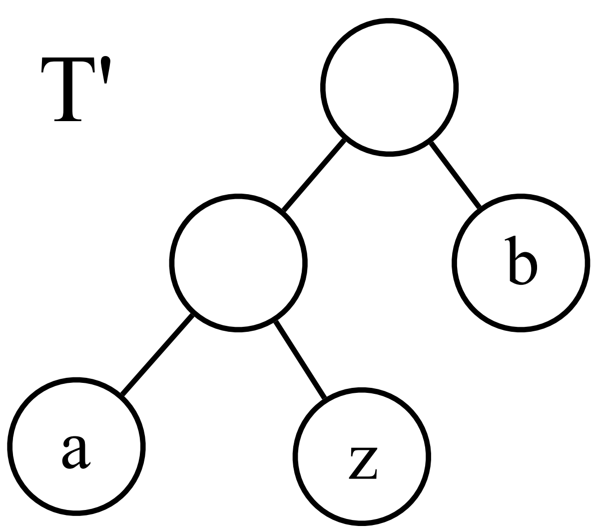


Let leaves a, b be siblings of maximum depth. Replacing a, b with x, y cannot increase $\sum_{1\leq i\leq k} f_i \cdot \text{length}(e_i)$.

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.

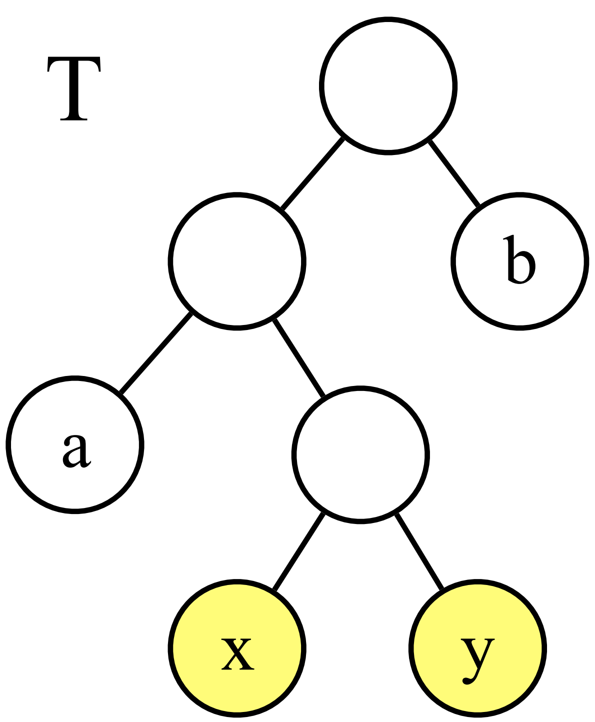
Let C' = C - {x, y} ∪ {z} and z.freq = x.freq + y.freq. Let T' be any tree representing an optimal code for C'. Then, the tree T obtained from T' by replacing z with an internal node having x and y as children, represents an optimal code for C.



To show:

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

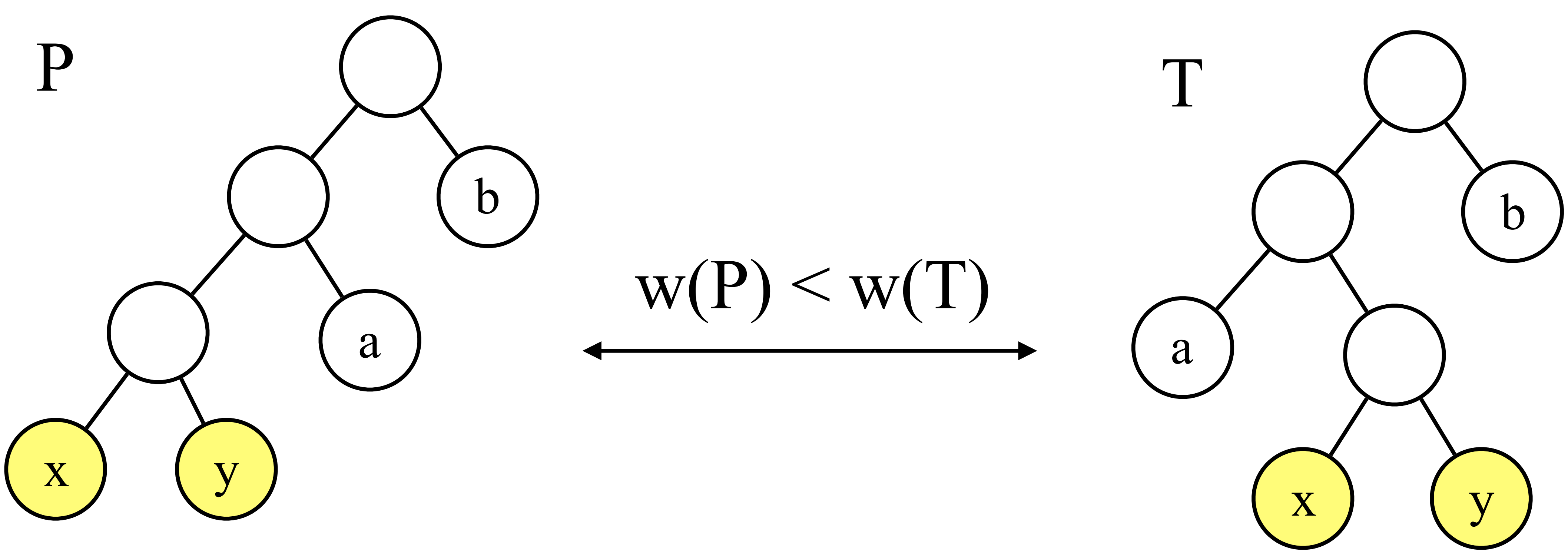
Note that
w(T') = w(T) - z.freq.



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.freq = x.freq + y.freq.

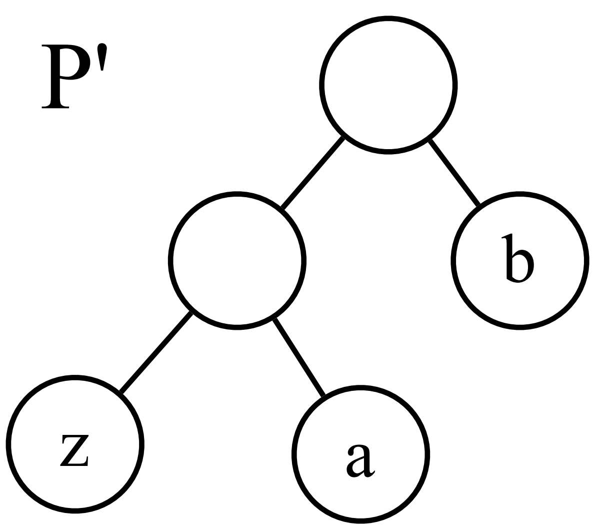


Proof of Claim 3

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.

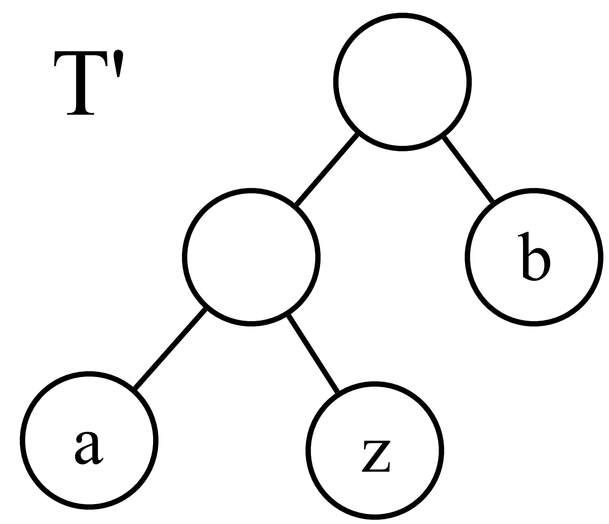
Let P' be the tree obtained from P by replacing the subtree rooted at the parent of x and y with a node z and letting z.freq = x.freq + y.freq.

P'



$$w(P') = w(P) - z.freq$$
$$w(T') = w(T) - z.freq$$
$$w(P') < w(T')$$

T'



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