Data Compression Basics

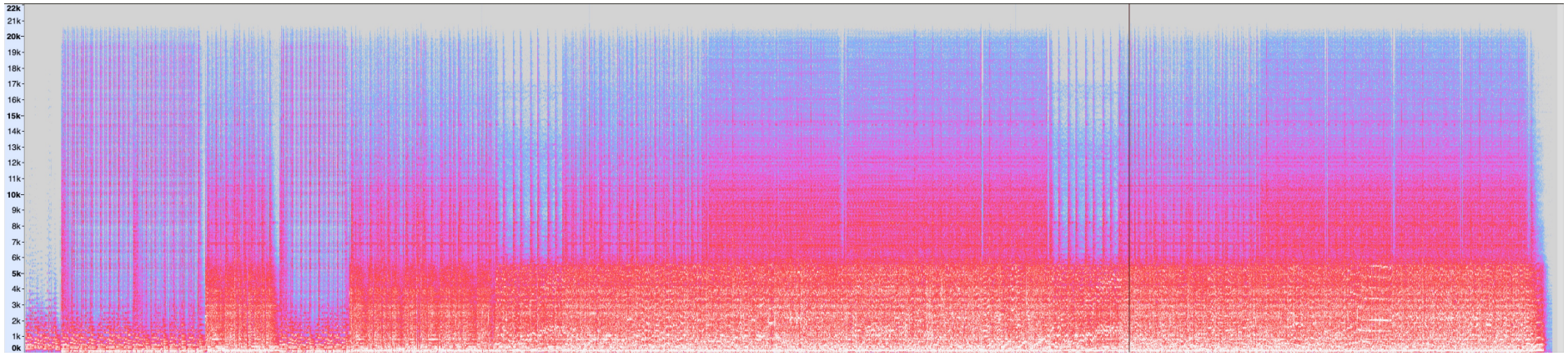
Kristian Guillaumier

We will not be covering lossy compression

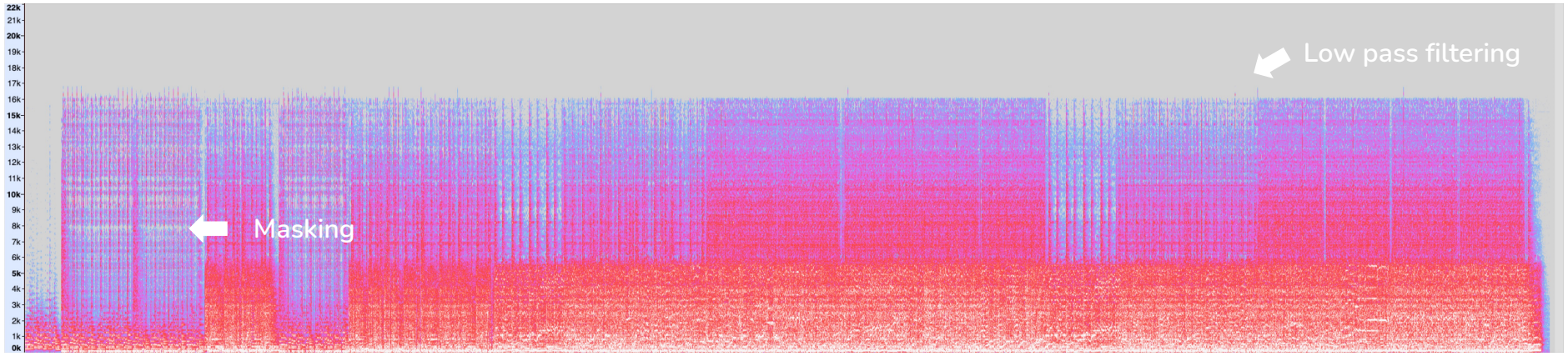
PNG lossless

JPG lossy

...continued



CDDA 44.1kHz, 16-bit



MP3 128kbps, 44.1kHz, 16-bit

Data compression

- The ASCII character set has about 100 printable characters.
- We can easily see that the number of bits required to encode the 100 characters in binary is given by $\lceil \log_2 100 \rceil = \lceil 6.64 \rceil = 7$ bits.
- The seven bits allow for 128 characters.
- ASCII adds an 8th bit for parity checking (or extensions).

In general, if the size of the character set is C then we need $\lceil log_2 C \rceil$ bits to encode each character.

...continued

- Let's say that we have a file that can only contain a, e, i, s, t, sp (space) and nl (new line) i.e., my character set C has 7 characters.
- Let's say that the file contains 10 'a' characters, 15 'e' characters, and so on...
- The table below shows some stats...

Character	Code	Frequency	Total bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
sp	101	13	39
nl	110	1	3
		Total:	174

← This is the file size

Example from Weiss

...continued

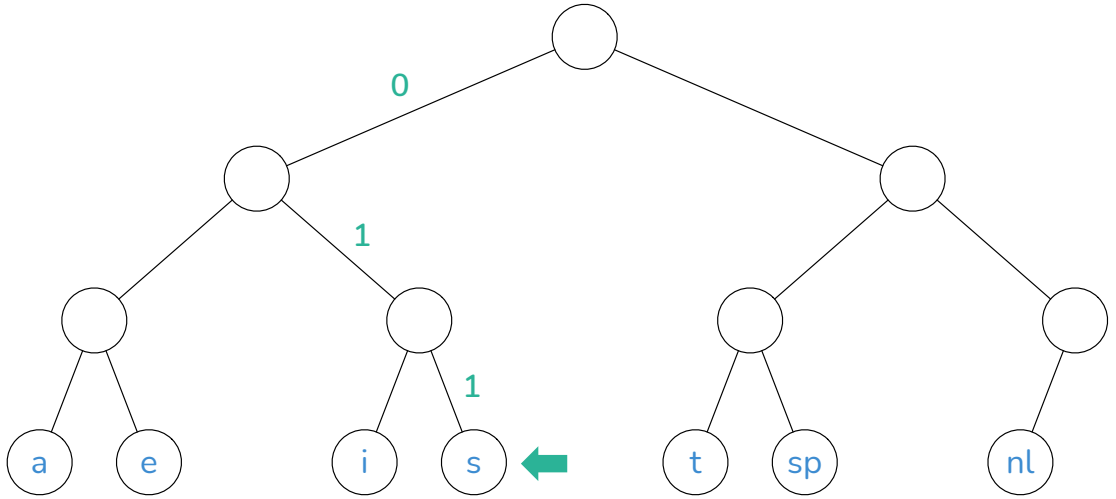
- In many data sequences (such text files) there is usually a large disparity between the most commonly occurring characters and the least commonly occurring ones.
- For example, the letter ‘e’ is more common than the letter ‘q’ in English text.
- We will use this property to our advantage and formulate a strategy where:
  - Character code length varies from one character to another.
  - Commonly used characters will have shorter codes.
  - Rarely used characters will have longer codes.

…continued

- Note that if all characters occur with the same frequency, good compression cannot be achieved.
  - Try compressing a file with random data – you'll barely get anything.
  - That is why plain text files compress much better than binaries.

Prefix codes

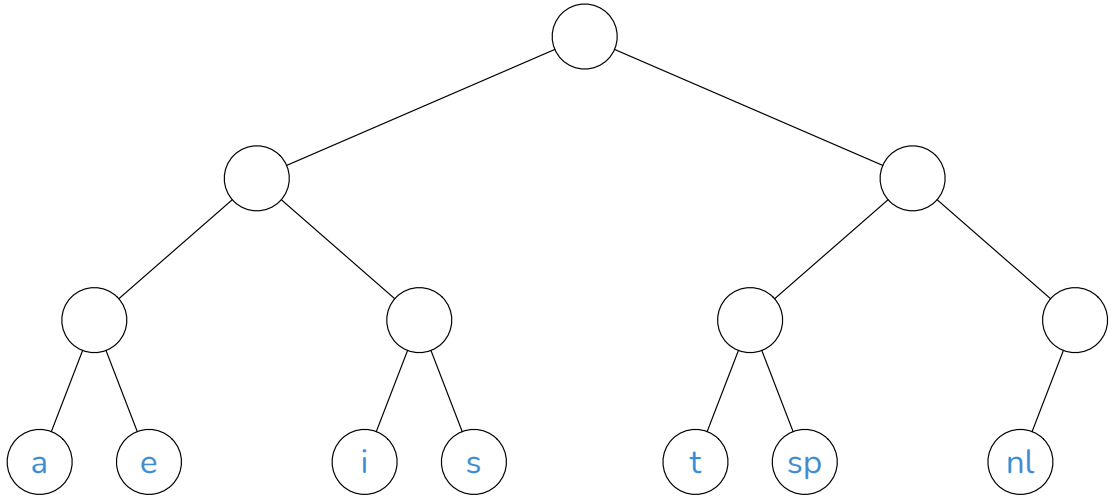
- The binary code shown in the table before can be represented by a prefix tree (values in the leaf nodes) below.
- Left branching indicates a 0, right branching indicates a 1. So, left → right → right is 011 and will encode the character ‘s’.



Example from Weiss

...continued

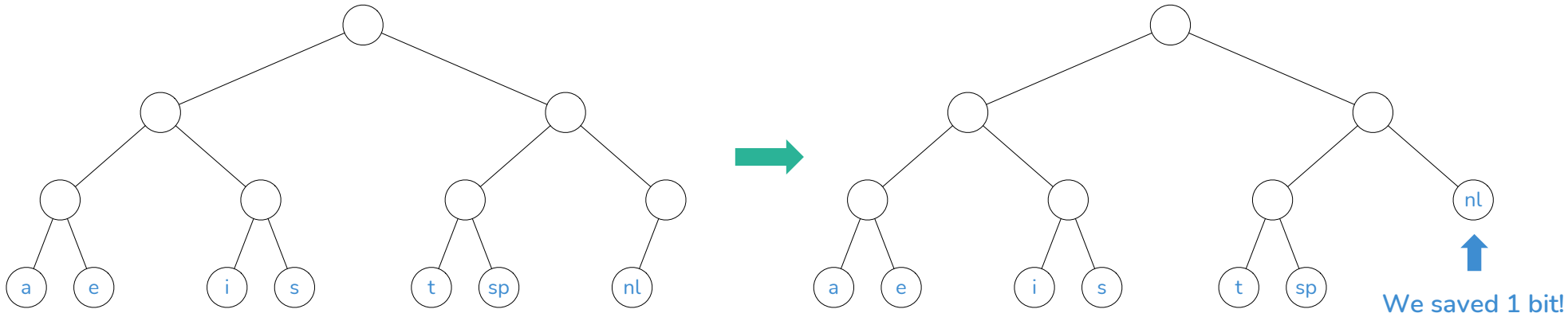
- If a character c is found at depth d and occurs f times, then $Cost(c) = d \times f$. In this example $Cost(\text{“}s\text{”}) = 3 \times 3 = 9$.
- The cost of the tree is the sum of the cost of every character in it. In this example, the cost will be 174.



Example from Weiss

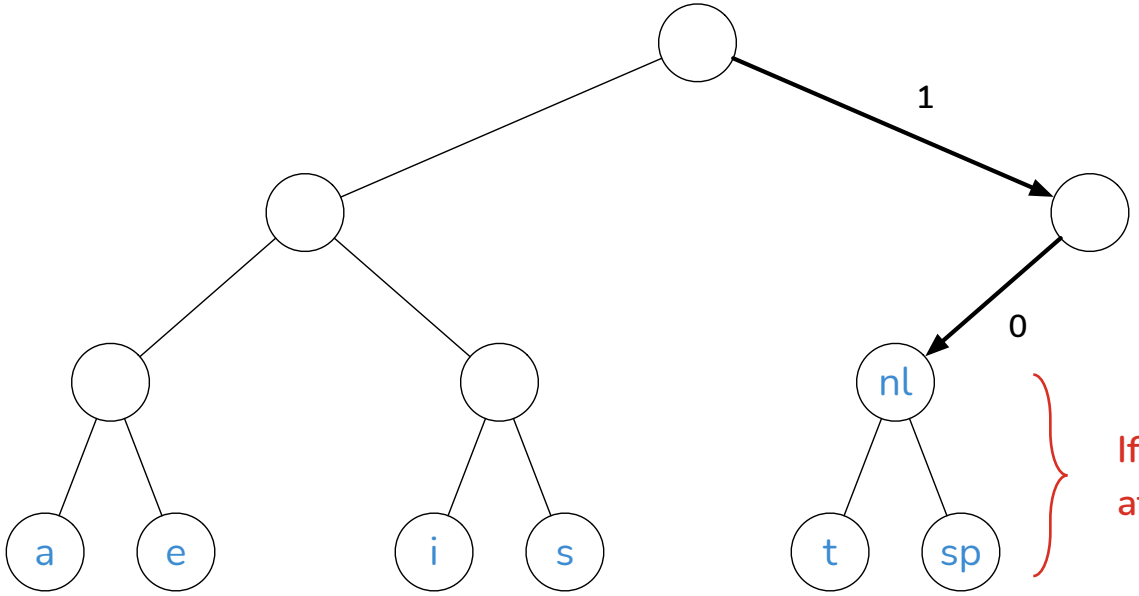
...continued

- If we reorganise the tree a bit and make leaves shallower, we we would also reduce the total cost of the tree.
- In this example, we shift the new line character one level up (without 'damaging' anything) and reduce the total cost of the tree from 174 to 173.



...continued

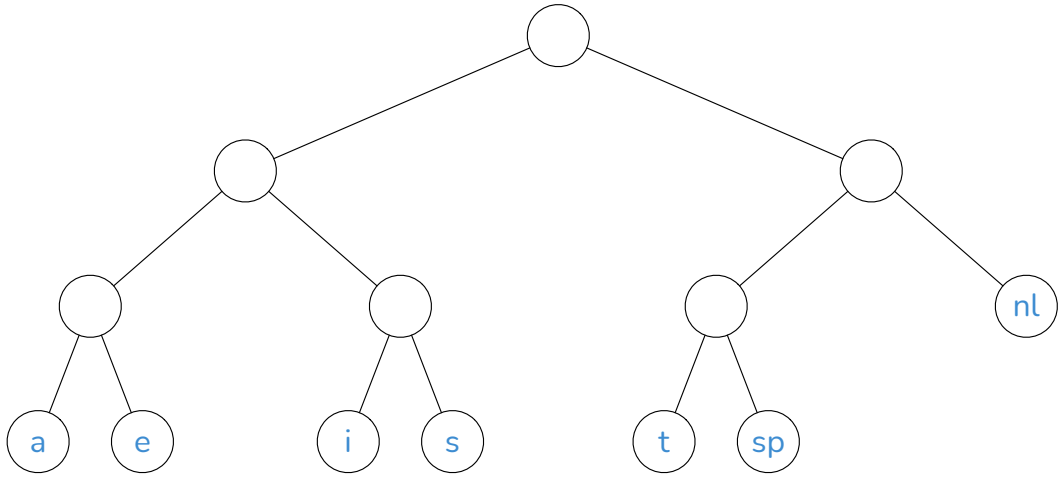
- Clearly, all characters must be in the leaves otherwise we will get ambiguous encoding (caused by common prefixes – this is very bad because we cannot decode).



If we read the sequence 10, do we stop at nl, or do we proceed to 't' or sp?

A simple example

- Consider the input string 010011111000101100010011 that we want to decode.
- We treat every 0 or 1 as a left or right move. When we reach a leave, we decoded a character. Then continue, restarting from the root.
- In this tree 010 (L → R → L) gives ‘i’, continuing with 011 (L → R → R) gives us ‘s’, continuing with 11 (R → R) gives us a nl, etc… Note that the character codes have different lengths.



Shannon-Fano coding

- Step 1: compute, or estimate, the frequency f of every character c in our character set.

Character	Frequency
e	10
a	100
d	50
c	80
b	90

Frequency table

...continued

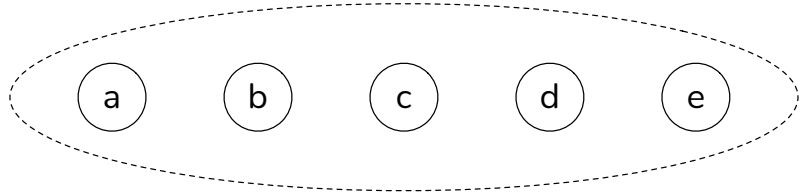
- Step 2: create a forest of trees containing a tree for each character (the tree is just a root node). The character table is sorted by frequency to help us visualise what's going on.

Character	Frequency
e	10
a	100
d	50
c	80
b	90

Frequency table

Character	Frequency
a	100
b	90
c	80
d	50
e	10

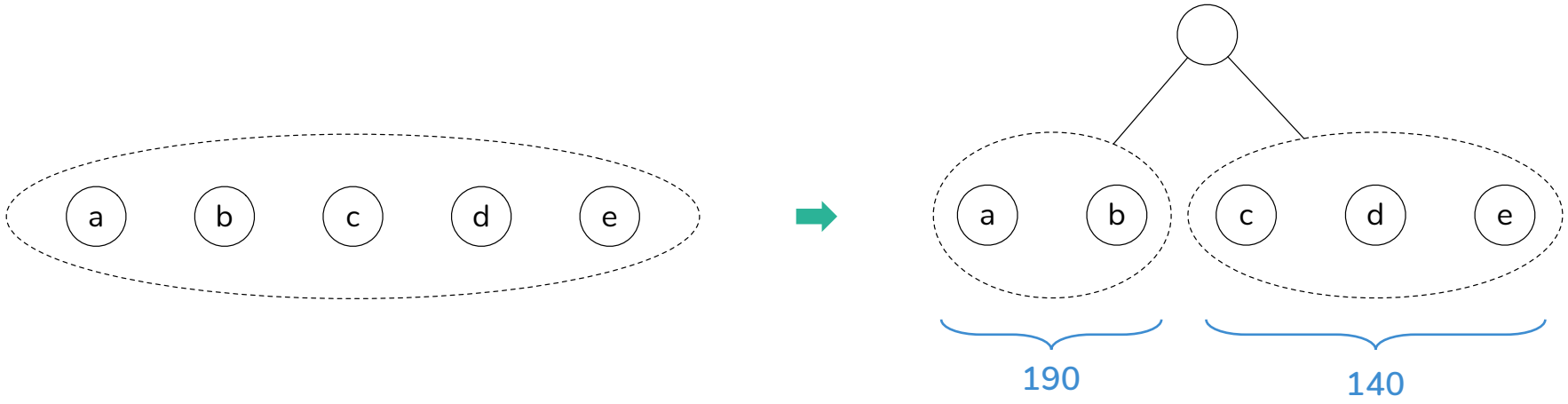
Sorted



Forest of trees

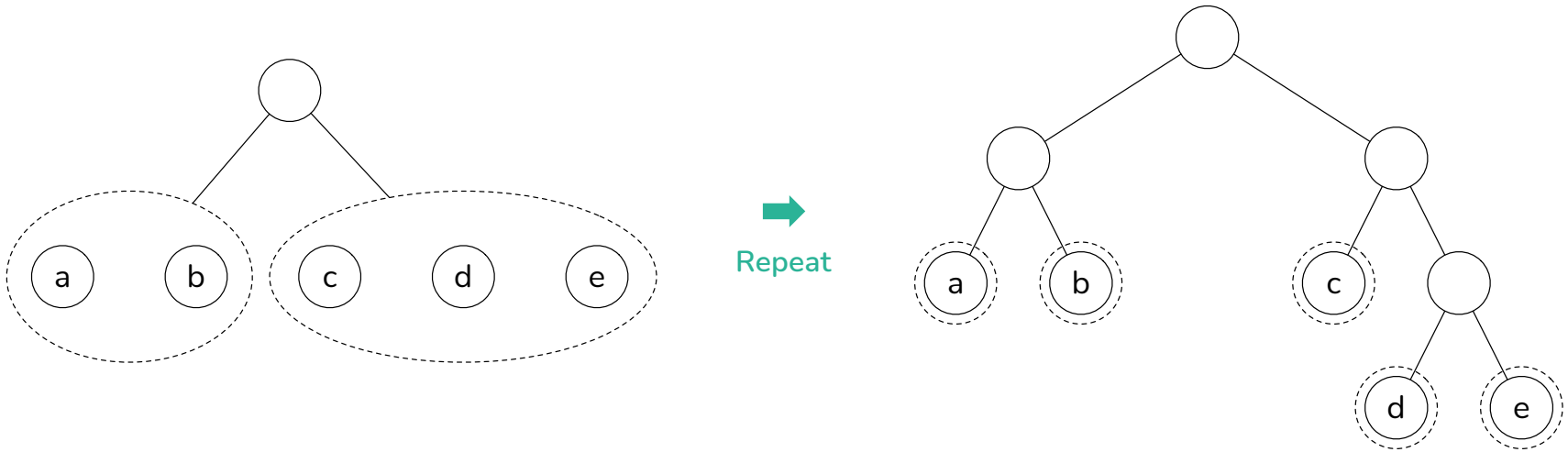
...continued

- Step 3: split the forest into two approximately equally-weighted parts. Merge the parts with a node and assign the left part a value of 0 and the right a value of 1.



...continued

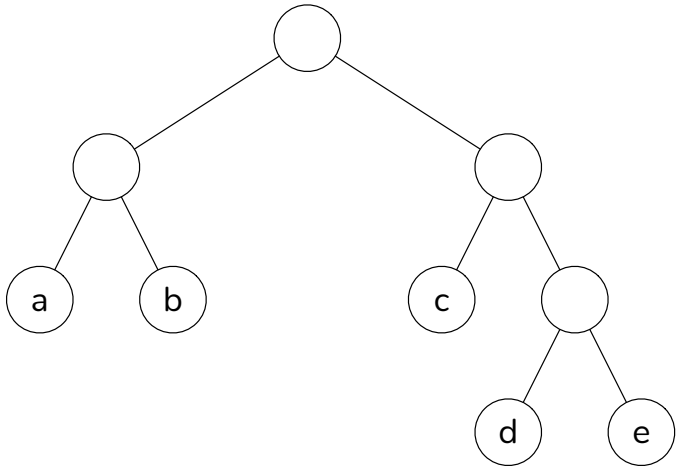
- Repeat this process recursively to every forest until each tree is alone in a forest (i.e., the symbol is a leaf in a tree).



The final prefix tree. Note that commonly occurring characters are shallower.

...continued

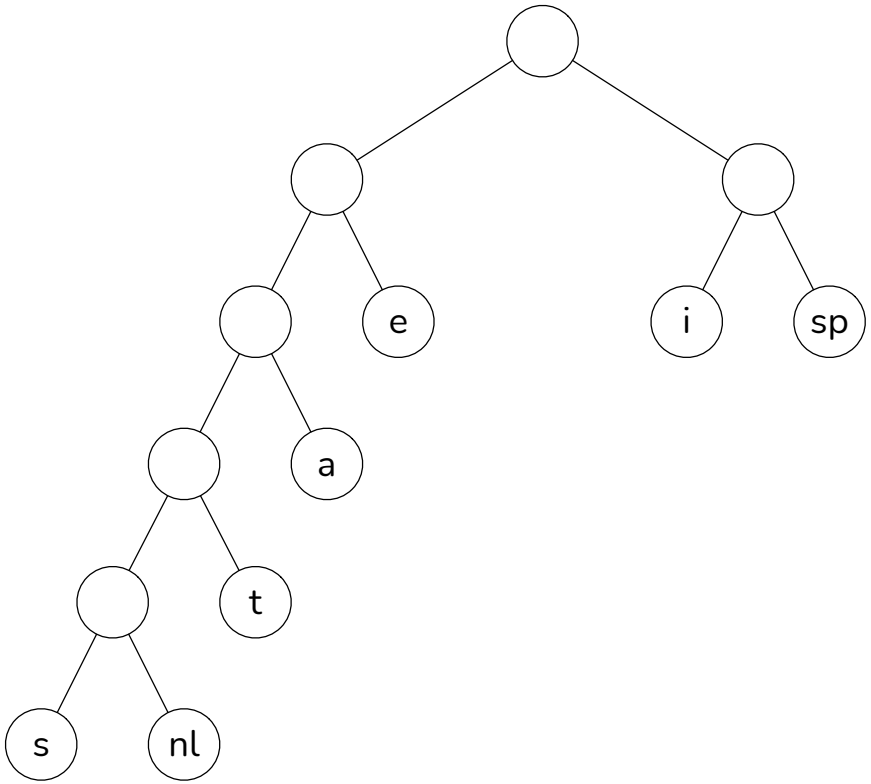
- Note that this is a basic method for creating a useful prefix tree and it does not guarantee that the prefix code is optimal.
- The final coding table we obtained:



Character	Frequency	Code
a	100	00
b	90	01
c	80	10
d	50	110
e	10	111

Commonly occurring characters have shorter codes.

The optimal prefix code tree



Character	Code	Frequency	Total bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
sp	11	13	26
nl	00001	1	5
		Total:	146

This is the file size

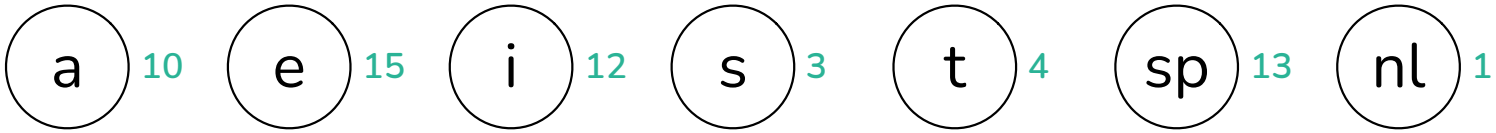
Example from Weiss

Huffman's algorithm

- An algorithm to construct an optimal prefix code tree is called Huffman's algorithm.
- It works by merging trees together (algorithm will be illustrated by example).
- Conventions:
  - A forest is a collection of trees.
  - C is the number of characters in the set.
  - The weight, w, of a tree is the sum of the frequencies of its leaves.
- In the beginning, there are C trees in the forest. Each tree is just a root/leaf node containing the character in question.
- The process iteratively merges (for $C - 1$ times) the trees until we get an optimal prefix tree.

Example

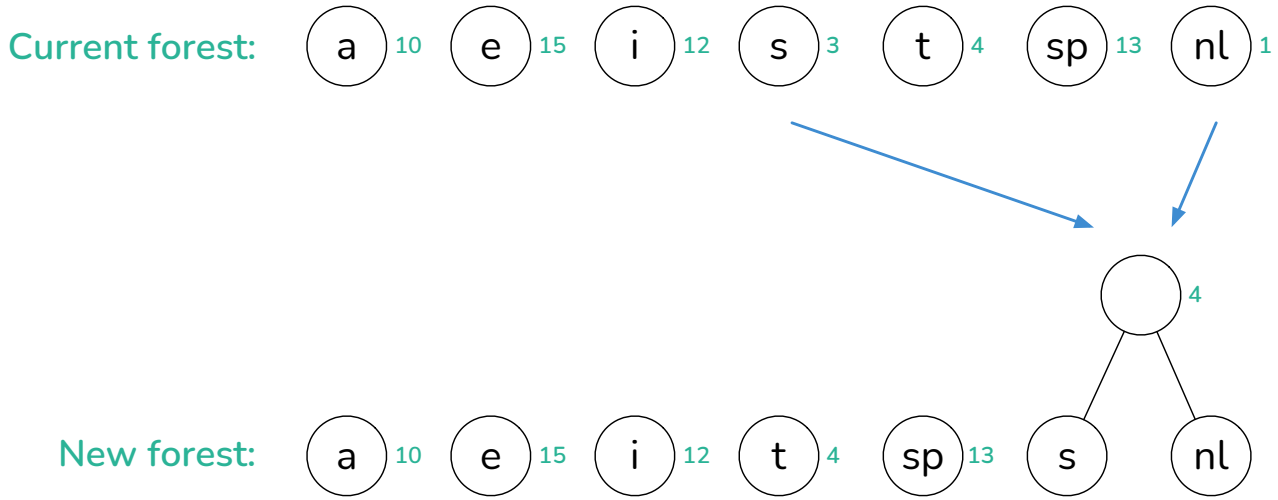
- Initial stage – forest containing C trees.
- Each tree has a single node with the character.
- The notes are annotated with the value w (weight, i.e., sum of frequencies).



Example from Weiss

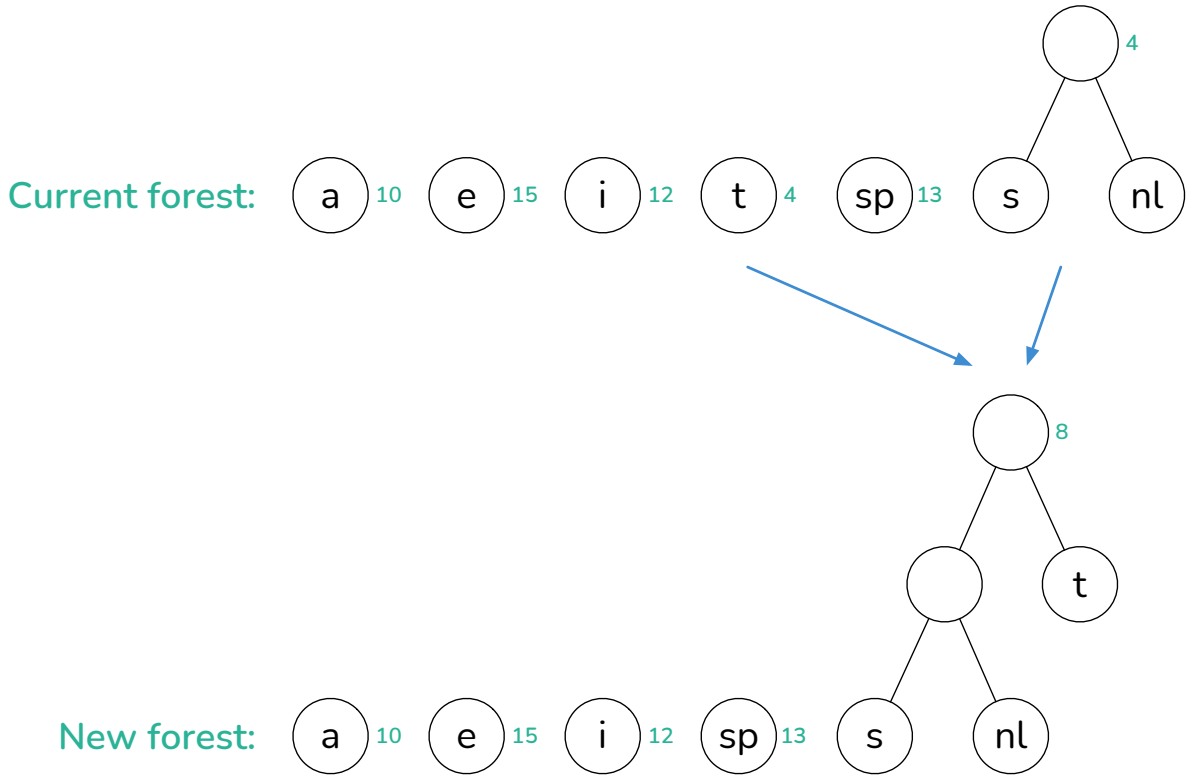
...continued

- Now we iteratively merge the two trees having the smallest weights and assign the weight of the new merged tree to the sum of the weights of the constituent trees.
- Note that ‘s’ and ‘nl’ have the smallest weights (3 and 1) so we join them with a parent node. The new tree has weight 3+1=4.



Example from Weiss

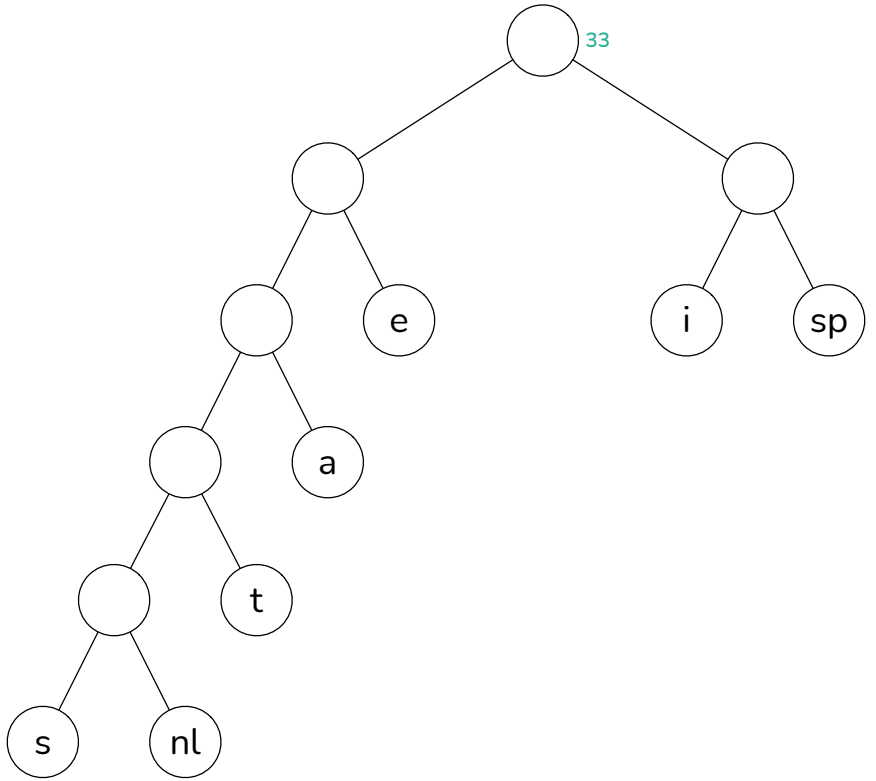
...continued



Example from Weiss

...continued

- The process is repeated until we end up with one tree. The final result is:



Example from Weiss

Huffman coding of images

- As practical example, consider Huffman encoding on 8-level grayscale images – normally we'd need 3 bits per pixel.

Source image
$128 \times 128 = 16384$

Gray level	Frequency
0	3441
1	4423
2	3932
3	1802
4	1311
5	819
6	492
7	164
Total:	16384 pixels

Gather statistics

Gray level	Huff. coding
0	11
1	01
2	10
3	001
4	0000
5	00010
6	000110
7	000111
Bits/pixel:	2.59

Huffman coding

The average bits per pixel is 2.59 vs. the 3 bits per pixel if not encoded.

Huffman coding using these statistics will give a 14% reduction in file size.

Example from http://academic.mu.edu/phys/matthysd/web226/L0423.htm

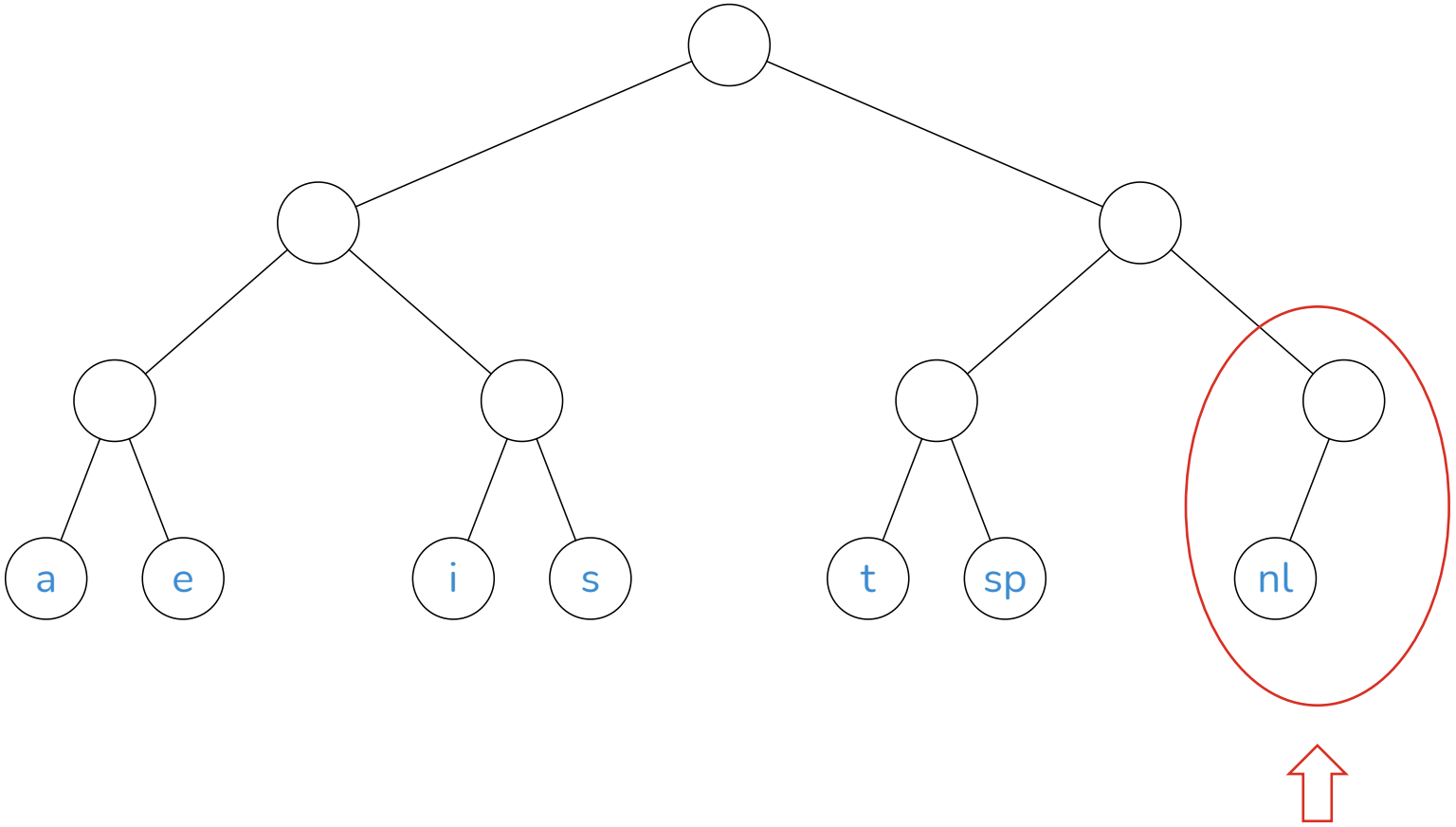
Proof of optimality

- Lemma 1:

An optimal coding tree is full – all internal nodes will have two child nodes.

Trivially shown by contradiction. Suppose that the tree T is optimal and has cost $cost(T)$. If there is an internal node N having only one child node M, we can simply remove N and replace it by the node M. This will reduce the depth of all the nodes in M's subtree, give a new subtree T' having a lower cost $cost(T')$.

…continued, lemma 1



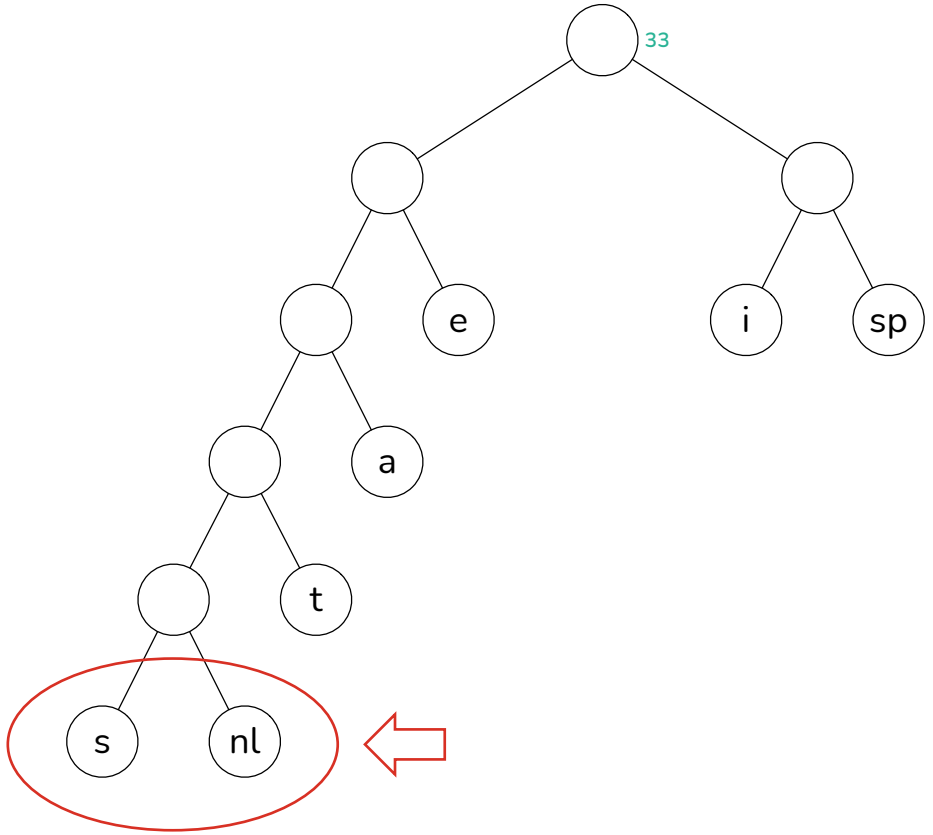
We've already seen this in this example before.

Another lemma

- Lemma 2:

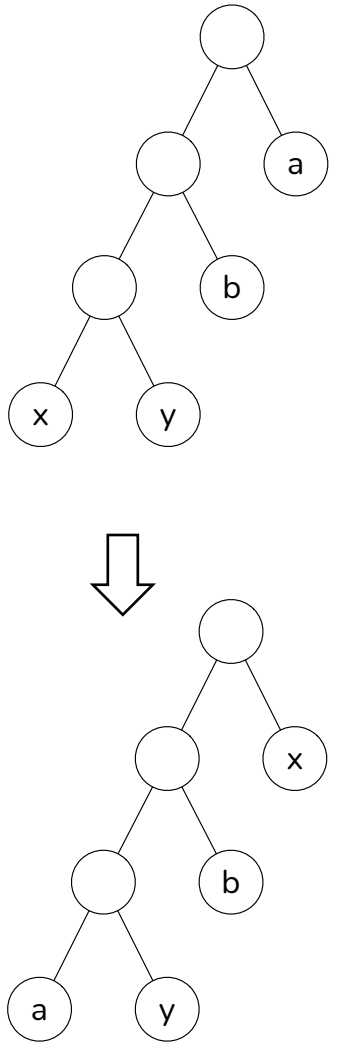
Consider two letters a and b having the smallest frequencies (least occurring).

There is an optimal code tree T in which these two letters are sibling leaves in the tree in the lowest level of the coding tree.



...continued, lemma 2

- Let T be an optimum prefix code tree having cost $cost(T)$.
- Let x and y be two siblings at the maximum depth of the tree. This case must exist due to lemma 1.
- Since, by our hypothesis, a and b have the two smallest frequencies, then $freq(a) \leq freq(x)$ and $freq(b) \leq freq(y)$.
- Since a and b are at the deepest level, $depth(a) \leq depth(x)$ and $depth(b) \leq depth(y)$.
- Case 1: swap the node for a with the node for x.
- This will give us a new tree T'.
- What is the cost of this new tree?

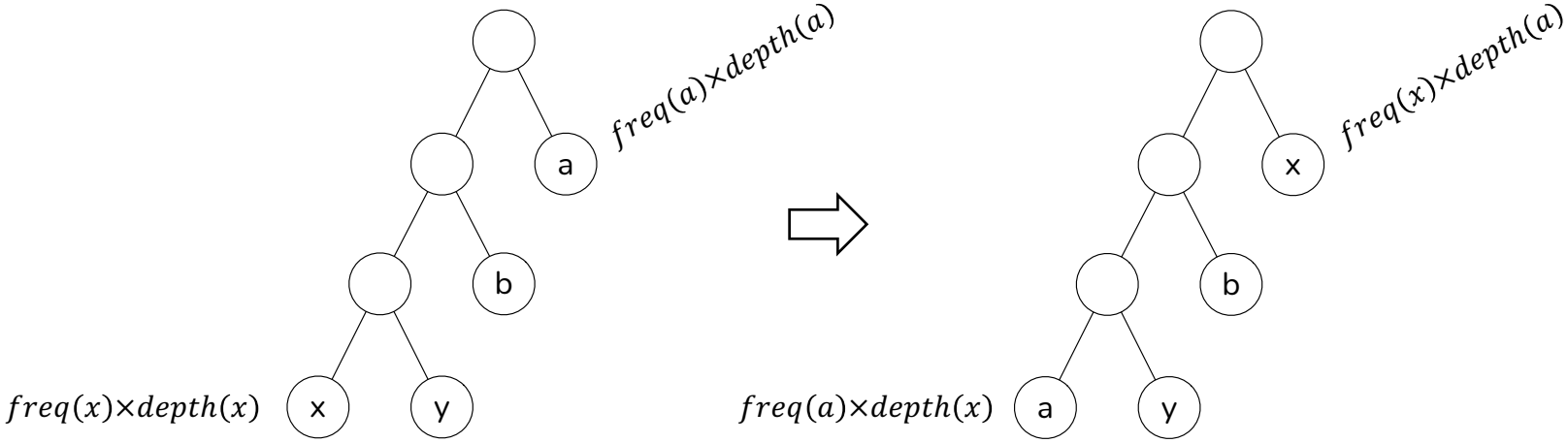


...continued, lemma 2

- The cost $cost(T')$ is given by:

$$\underbrace{cost(T')}_{} = \underbrace{cost(T)}_{} \underbrace{- freq(x)\times depth(x) - freq(a)\times depth(a)}_{} \underbrace{+ freq(x)\times depth(a) + freq(a)\times depth(x)}_{}$$

- i.e., the cost of the new tree is the cost of the previous one after removing the cost of x and the cost of a and then adding their costs back according to their new positions in the new tree.



...continued, lemma 2

- Rearranging...

$$cost(T') = cost(T) - freq(x)\times depth(x) - freq(a)\times depth(a) + freq(x)\times depth(a) + freq(a)\times depth(x)$$

- We get...

$$cost(T') = cost(T) - \big[\big(freq(a) - freq(x)\big)\times\big(depth(a) - depth(x)\big)\big]$$

Multiply/group binomials.

- So $cost(T') \leq cost(T)$
- But, by our hypothesis, $cost(T) \leq cost(T')$ since T is optimal.
- Therefore, $cost(T) = cost(T')$.
- This also holds for case 2: swap the node for b with the node for y where we get the same result.

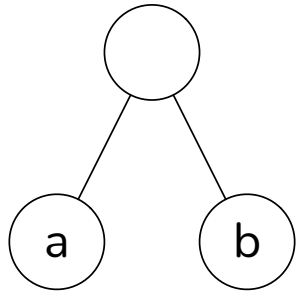
Theorem

Huffman coding has code efficiency (cost) which is lower than all prefix encodings of a given alphabet.

i.e., the algorithm is optimal.

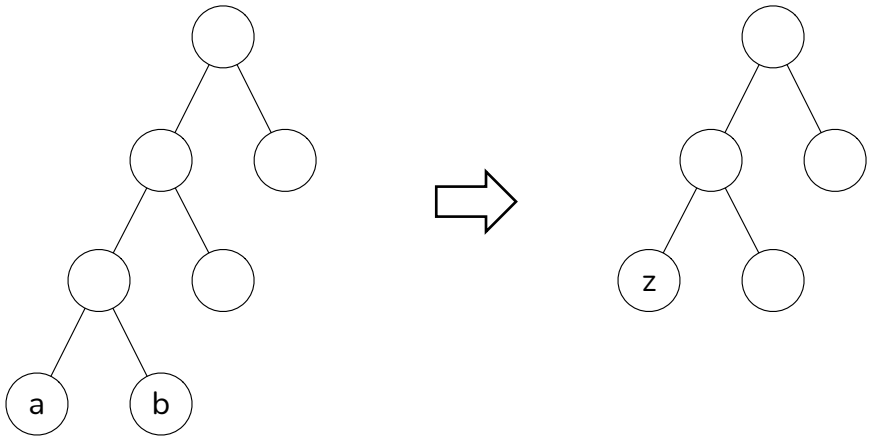
Proof

- We'll prove this by induction on the size n of the alphabet.
- Suppose that $n = 2$ (this is the smallest non-trivial alphabet size).
- In this case, the tree produced by the algorithm is obviously optimal; the tree will have one internal root note with two leaves for each symbol.



...continued

- Our inductive step will be to assume that the theorem holds true for all alphabets of size $n - 1$ and prove that it is also true for all alphabets of size n.
- Let A be an alphabet with n characters and let T be the optimal tree for A.
- T will have two leaves a and b which are siblings having minimum frequencies due to lemmas 1 and 2.
- Consider the tree T' which is a prefix code for the alphabet A' corresponding to the alphabet A after removing characters a and b then adding a new character z at their parent node in T.



...continued

- The frequency of the new character z will be $freq(a) + freq(b)$; remember that this is how Huffman 'weighs' internal nodes.
- So far, we have...
  - $depth(a) = depth(b) = depth(z) + 1$
  - Equivalently, $depth(z) = depth(a) - 1 = depth(b) - 1$
  - And $freq(z) = freq(a) + freq(b)$
  - And...

$$cost(T') = cost(T) - freq(a) \times depth(a) - freq(b) \times depth(b) + freq(z) \times depth(z)$$

...continued

- Let's clean up the previous expression for $cost(T')$ by removing all references to z in it.
- Using what we know:
  - $depth(z) = depth(a) - 1 = depth(b) - 1$
  - And $freq(z) = freq(a) + freq(b)$

$$cost(T') = cost(T) - freq(a) \times depth(a) - freq(b) \times depth(b) + freq(z) \times depth(z)$$

Becomes...

$$cost(T') = cost(T) - \big(freq(a) + freq(b)\big)$$

So...

$$cost(T) = cost(T') + \big(freq(a) + freq(b)\big)$$ ⇦ A constant!

...continued

- Let H' be the Huffman tree generated for the alphabet A'.
- Remember that H' is the tree prior to the merge that the algorithm does.
- So, by our inductive assumption, it is optimal.
- So, $cost(H') \leq cost(T')$ 1
- When we add the two symbols a and b under z, we get the Huffman tree H for the whole alphabet A.
- Remember what the algorithm is doing; when two symbols (trees in the forest) a and b are added to the Huffman tree, they are replaced by a new 'meta' symbol z (a new tree in the forest). Basically, whenever a merge is made, the size of the alphabet under consideration (the number of trees in the forest) is decreasing by one (1); minus 2 for each of a and b, and plus 1 for z.

...continued

- Using the same computation we made to calculate $cost(T)$, we get:

$$cost(H) = cost(H') + freq(a) + freq(b) \quad (2)$$

- So, from (1) and (2) ...

$$cost(H) = cost(H') + freq(a) + freq(b) \leq cost(T') + freq(a) + freq(b)$$

- We previously worked out that...

$$cost(T) = cost(T') + \big(freq(a) + freq(b)\big)$$

- So...

$$cost(H) \leq cost(T) \quad (3)$$

...continued

- Since T is optimal (by our initial hypothesis), we know that...

$$cost(T) \leq cost(H) \quad (4)$$

- Therefore, from (3) and (4) ...

$$cost(T) = cost(H)$$

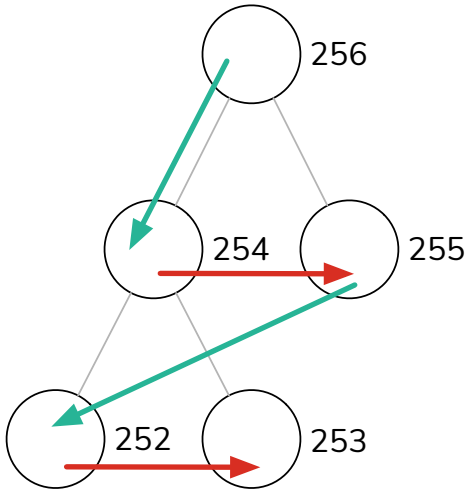
- Done.

Adaptive Huffman coding

- Does not require a priori knowledge of the character distribution (no need for a frequency table).
  - This is usually the case when data is transmitted and compressed in real time.
  - The prefix tree is rearranged as data is received.
  - Note: this makes the algorithm sensitive to transmission errors.
  - Also, useful when the computation of the statistical data may be very intensive.
- We will be describing Vitter's algorithm.

...continued

- Code is represented by a tree where:
  - Every node has a weight.
  - Every node has a unique number.
  - The unique numbers decrease downwards and increase from left-to-right.

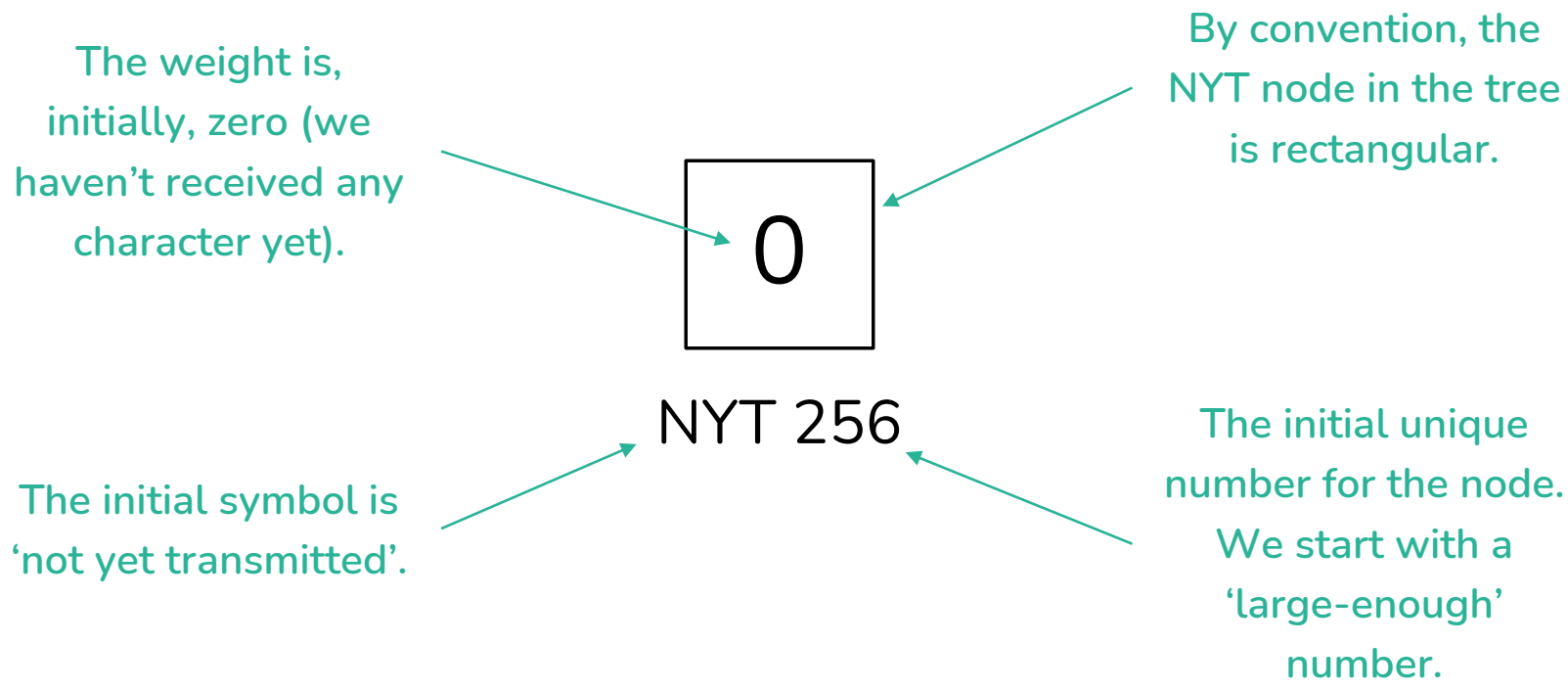


...continued

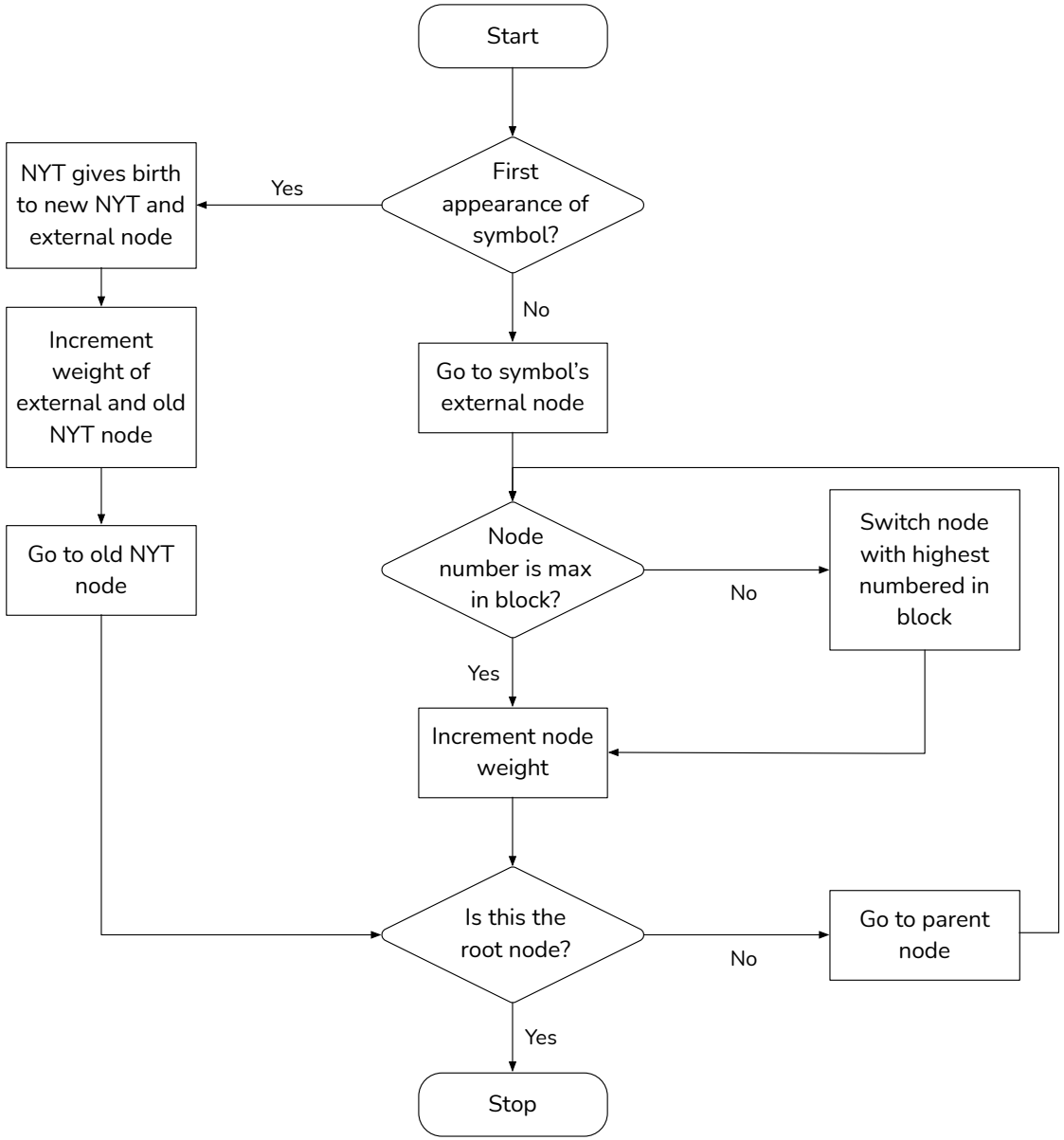
- Weights must satisfy the sibling property:
  - A node with a higher weight will have a higher number, and
  - A parent node will always have a higher node number than its children.
- The weight of a symbol is the number of times that symbol appeared so far.
- A set of nodes with the same weight is called a block.
- NYT represents the 'not yet transmitted' symbol.

...continued

- The starting point is simply a node containing the NYT symbol (in other words, this is starting prefix tree):



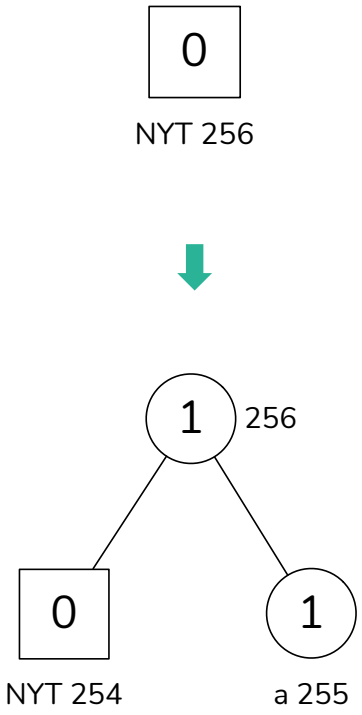
The algorithm

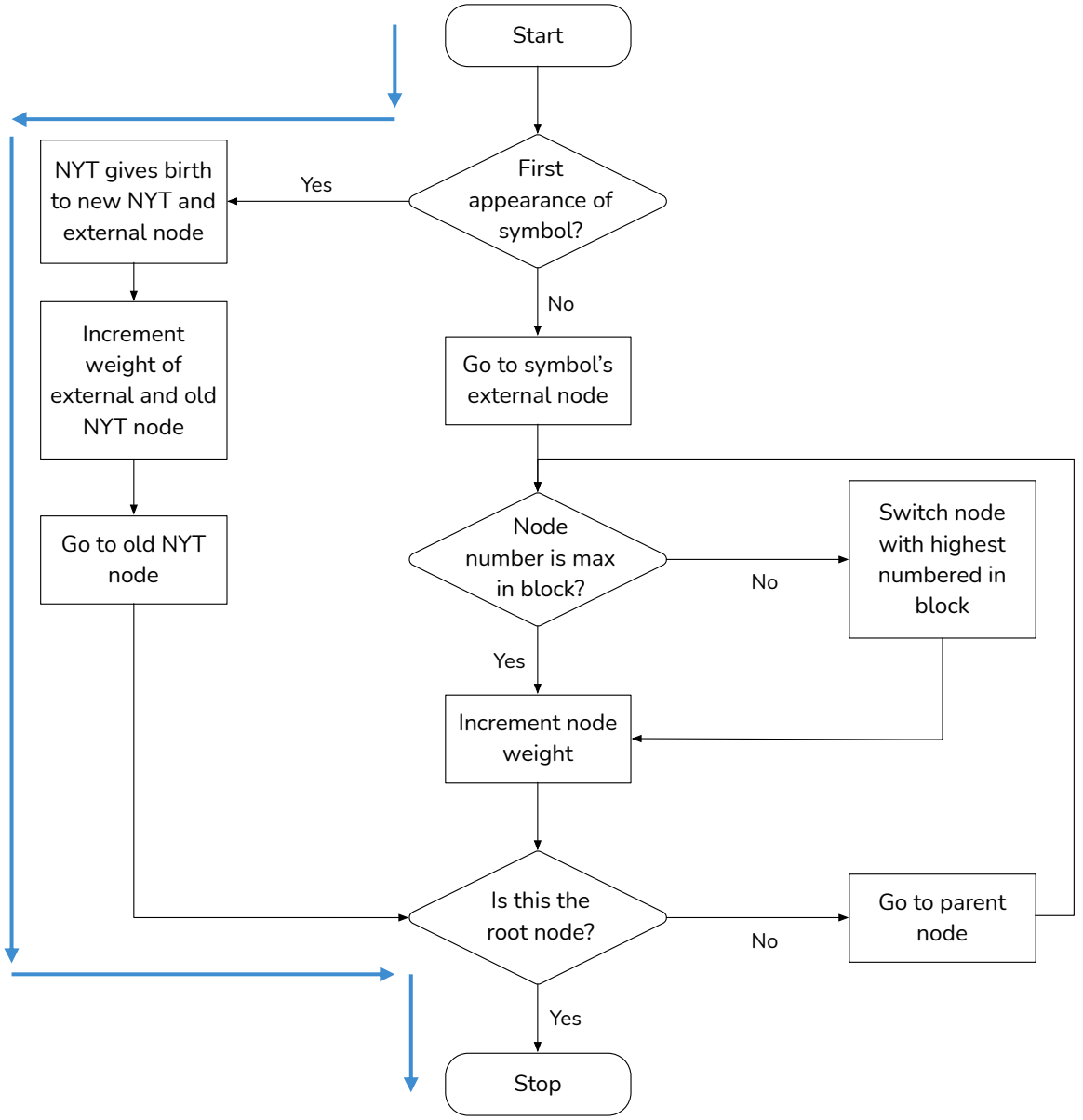


Flowchart from "Introduction to Data Compression" by by Sayood Khalid

Example

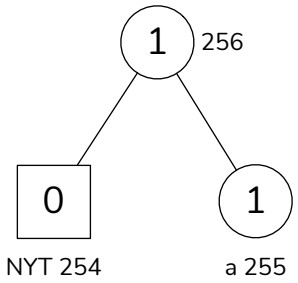
We received the symbol ‘a’

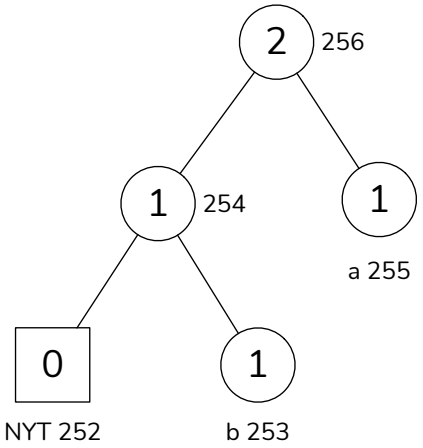


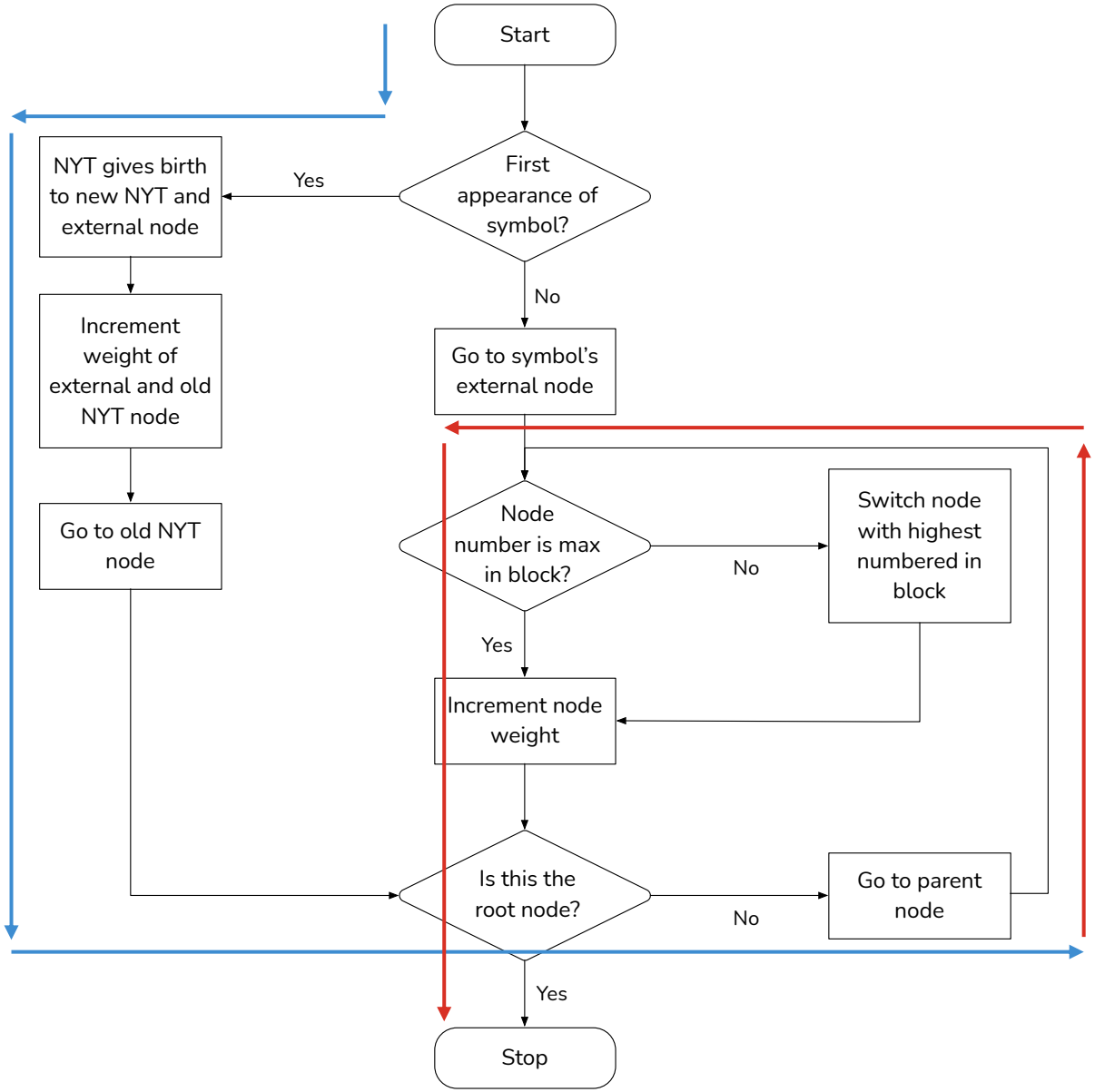


Example

We received the symbol ‘b’

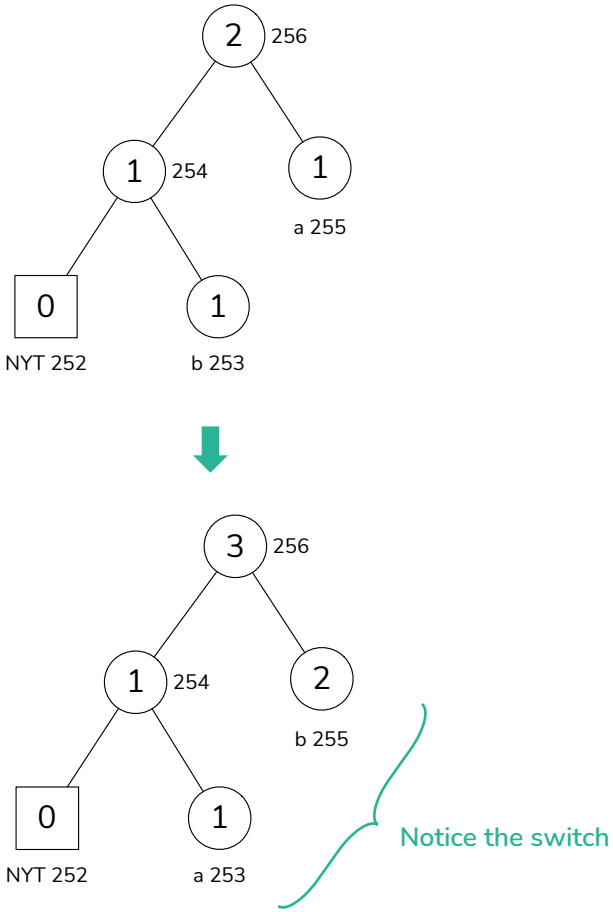


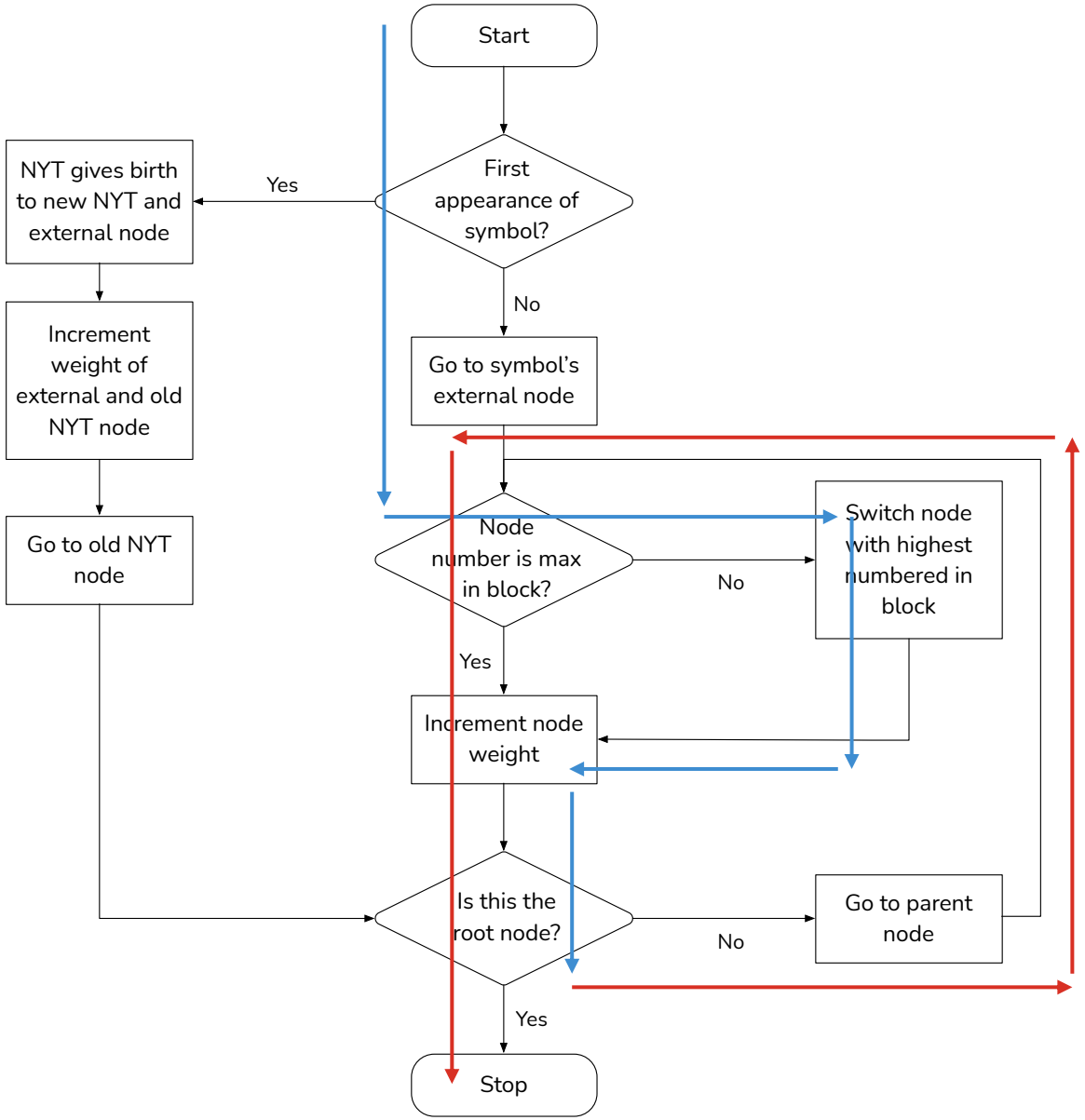




Example

We received another symbol ‘b’





Lempel-Ziv-Welch

- Lossless Data Compression Algorithm.
- It allows ‘longer’ sequence compression compared to Huffman coding which shortens (or lengthens) codes for sequences of only one character.
- In its most basic form:
  - Encode 8-bit data as 12-bit codes.
  - Codes 0 to 255 represent single character sequences and are automatically placed in the dictionary.
  - Codes 256 to 4095 make up a dictionary that is created for longer sequences encountered in the data.

Notice repeated sequences in this text...

```
<a href="/store/product/subscriptions/" class="header-highlight-link">Subscribe</a>
<div class="dropdown" id="header-search">
    <a href="/search/" class="dropdown-toggle search-toggle" aria-label="Search" aria-expanded="false">
        <span class="icon icon-search-mag-glass"></span>
    </a>
    <div class="dropdown-content">
        <form action="/search/" method="GET" id="search_form">
            <input type="hidden" name="ie" value="UTF-8">
            <input type="text" name="q" id="hdr_search_input" value="" aria-label="Search..." placeholder="Search...">
        </form>
        <a class="nav-search-close">Close</a>
    </div>
</div>
<div class="dropdown dropdown-mega" id="header-burger">
    <a href="#site-menu" class="dropdown-toggle" aria-label="Menu" aria-expanded="false">
        <span></span>
    </a>
    <div id="site-menu" class="dropdown-content">
        <section class="burger-navigate">
            <h3>
                <span class="icon icon-half-target"></span>

                    Navigate

            </h3>
            <ul>
                <li>
                    <a class="nav-link store" href="/store/">Store</a>
                </li>
                <li>
                    <a class="nav-link subscribe" href="/store/product/subscriptions/">Subscribe</a>
                </li>
                <li>
                    <a class="nav-link videos" href="http://video.arstechnica.com/">Videos</a>
                </li>
                <li>
                    <a class="nav-link section-features" href="/features/">Features</a>
                </li>
                <li>
                    <a class="nav-link section-reviews" href="/reviews/">Reviews</a>
                </li>
```

Basic method

- Read bytes and append until we meet a sequence that is not in the dictionary.
- By construction, a 1-byte sequence is always in the dictionary.
- When we append a byte that will create a sequence that is not in the dictionary we stop.
- Obviously, the sequence we read apart from the last read byte must be in the dictionary. For this, we emit the corresponding code.
- The sequence we read including the last read byte is added to the dictionary and associated with a new code.
- This last byte is then the starting point of the new scan.

A simple example

- Consider the following plain text to be compressed:
- Note the # that signals the end of transmission.
  - We call this the stop-code, and we give it the code 0.

TOBEORNOTTOBEORTOBEORNOT#

Example taken from Wikipedia

...continued

- There are 26 characters in the alphabet we are considering (A-Z), plus the stop-code.
- For these 27 combinations, we need at least 5 bits (2^5=32)
- The dictionary is initialised as follows:

Initial dictionary

Symbol	Code	Decimal
#	00000	0
A	00001	1
B	00010	
C	00011	
.		
.		
.		
Z	11010	26

...continued

- The 5 bits allow for 32 entries in our dictionary.
- If we meet/need the 33rd entry we have switch the dictionary to 6 bits.
- To encode...

Buffer input characters in a sequence ω until ω + [next character] is not in the dictionary. Emit the code for ω and add ω + [next character] to the dictionary. Start buffering again with that next character.

TOBEORNOTTOBEORTOBEORNOT#

Current Sequence	Next Char	Output Code	Output Bits	Extended Dictionary
NULL	T			
T	O	20 =	10100	27: TO
O	B	15 =	01111	28: OB
B	E	2 =	00010	29: BE
E	O	5 =	00101	30: EO
O	R	15 =	01111	31: OR
R	N	18 =	10010	32: RN
N	O	14 =	001110	33: NO
O	T	15 =	001111	34: OT

The first free slot in the dictionary.

We need to switch to a 6-bit dictionary.

Example from Wikipedia

TOBEORNOTTTOBEORTOBEORNOT#

Current Sequence	Next Char	Output Code	Output Bits	Extended Dictionary
...continued...				
T	T	20 =	010100	35: TT
TO	B	27 =	011011	36: TOB
BE	O	29 =	011101	37: BEO
OR	T	31 =	011111	38: ORT
TOB	E	36 =	100100	39: TOBE
EO	R	30 =	011110	40: EOR
RN	O	32 =	100000	41: RNO
OT	#	34 =	100010	
		0 =	000000	

Stop-code. Send current sequence and stop-code.

Results

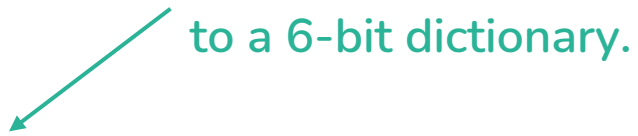
- Sending the transmission (25 characters) using 5 bits = 125 bits required.
- However, sending the compressed transmission requires 6×5 bits + 11×6 bits = 96 bits.
- We save almost 22%.

Decompression

- The decompressor only needs the initial dictionary used – the other entries can be constructed by the decompressor as well.
- In other words, the compressor and the decompressor only need to agree on the character set and how the initial dictionary is created.

Example

Input Bits	Output Code	Sequence	New Dictionary Entry Full	Conjecture
10100	= 20	T		27: T?
01111	= 15	O	27: TO	28: O?
00010	= 2	B	28: OB	29: B?
00101	= 5	E	29: BE	30: E?
01111	= 15	O	30: EO	31: O?
10010	= 18	R	31: OR	32: R?
001110	= 14	N	32: RN	33: N?
001111	= 15	O	33: NO	34: O?
010100	= 20	T	34: OT	35: T?



...continued

Input Bits	Output Code	Sequence	New Dictionary Entry Full	Conjecture
...continued...				
011011	= 27	TO	35: TT	36: TO?
011101	= 29	BE	36: TOB	37: BE?
011111	= 31	OR	37: BEO	38: OR?
100100	= 36	TOB	38: ORT	39: TOB?
011110	= 30	EO	39: TOBE	40: EO?
100000	= 32	RN	40: EOR	41: RN?
100010	= 34	OT	41: RNO	42: OT?
000000	= 0	#		

Further reading

- These notes should be supplemented by:
  - Introduction to Algorithms (Clifford Stein, Thomas H Cormen, Ronald L Rivest, Charles E Leiserson – MIT Press)
  - Data Structures and Algorithm Analysis in Java (Weiss)