

04-630

Data Structures and Algorithms for Engineers

Lecture 14: Optimal Code Trees

Adopted and Adapted from Material by:

David Vernon: vernon@cmu.edu ; www.vernon.eu

Agenda

Trees

- Types of trees
- Binary Tree ADT
- Binary Search Tree
- Height Balanced Trees
 - AVL Trees
 - Red-Black Trees
- Optimal Code Trees
- Huffman's Algorithm



Optimal Code Trees

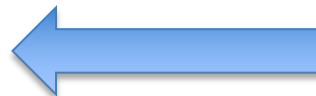
- First application: *coding* and *data compression*

Types:
* fixed-length
* block codes

Encoding



101010101010
101011111100
000010101001
010100010100
010000101010



Decoding

Text, Codes, and Compression

ASCII (code)	Character (Source)
1000100	D
1100101	e
1100011	c

- And we can decode it by chopping it into smaller strings each of 7 bits in length and by replacing the bit strings with their corresponding characters:

1000100(D)1100101(e)1100011(c)1101111(o)1100100(d)110
1001(i)1101110(n)1100111(g)0100000()
0100000()
1100101(e)1100001(a)1110011(s)1111001(y)

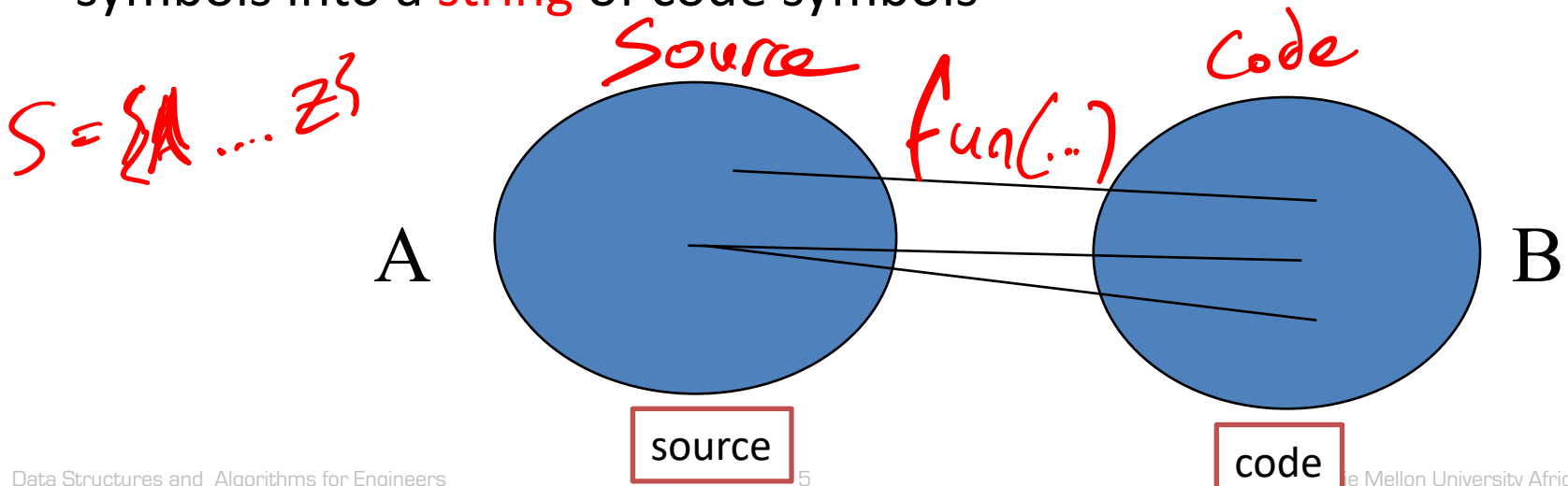
Text, Codes, and Compression

- Every code can be thought of in terms of

- a finite alphabet of **source symbols**
- a finite alphabet of **code symbols**

ASCII (code)	Character (Source)
1000100	D
1100101	e
1100011	c

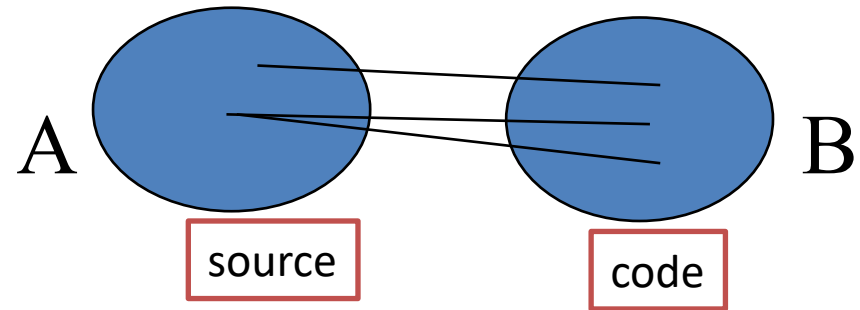
- Each code maps every finite sequence or string of source symbols into a **string** of code symbols



Text, Codes, and Compression

- Let A be the source alphabet
- Let B be the code alphabet
- An encoder/decoder f is an *injective map*

$$f: S_A \rightarrow S_B$$

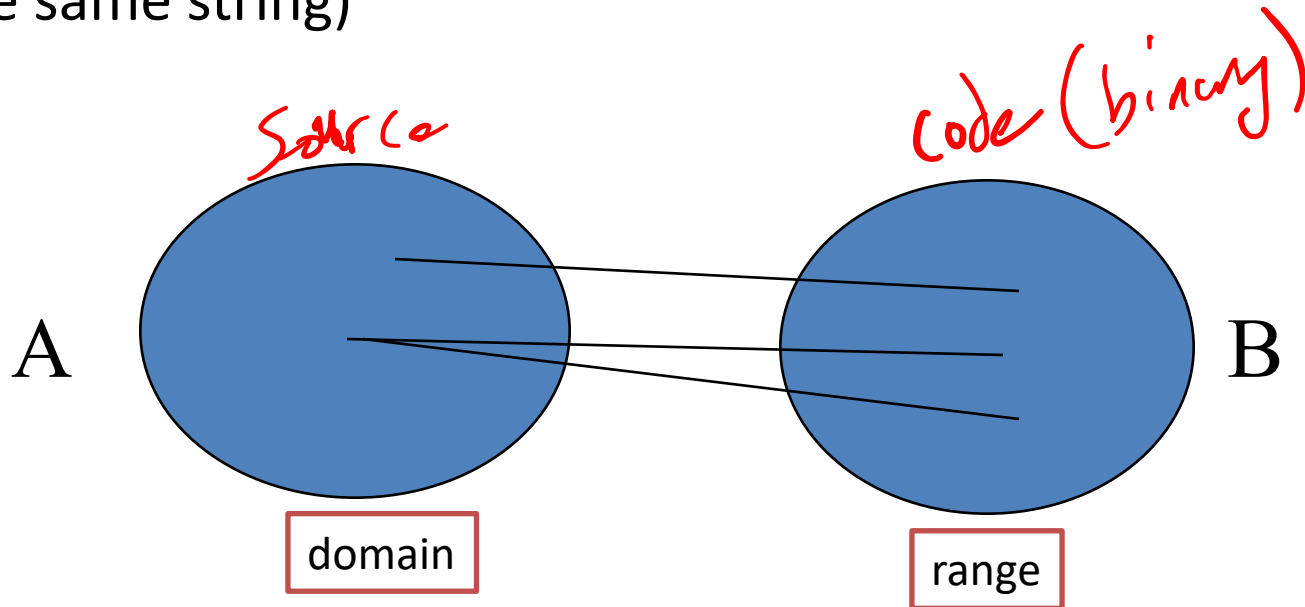


ASCII (code)	Character (Source)
1000100	D
1100101	e
1100011	c

- where S_A is the set of all strings of symbols from A
- where S_B is the set of all strings of symbols from B

Text, Codes, and Compression

- **Injectivity** ensures that each encoded string can be decoded uniquely (we do not want two source strings that are encoded as the same string)



Injective Mapping: each element in the range is related to at most one element in the domain

Problem with Block encoding

- There is a problem with block codes:
 n symbols produce nb bits with a block code of length b
- For example,
 - if $n = 100,000$ (the number of characters in a typical 200-page book)
 - $b = 7$ (e.g. 7-bit ASCII code)
 - then the characters are encoded as 700,000 bits

Enter: Variable length Encoding

- While we cannot encode the ASCII characters with fewer than 7 bits
- We can encode the characters with a different number of bits, depending on their frequency of occurrence
- Use fewer bits for the more frequent characters
- Use more bits for the less frequent characters
- Such a code is called a ***variable-length code***

Challenge with Variable Length – Decoding!!

$n * 3000$



$f_i * 3000$

- **First problem** with *variable length codes*:
 - when scanning an encoded text from left to right (decoding it)
 - How do we know when one codeword finishes and another starts?

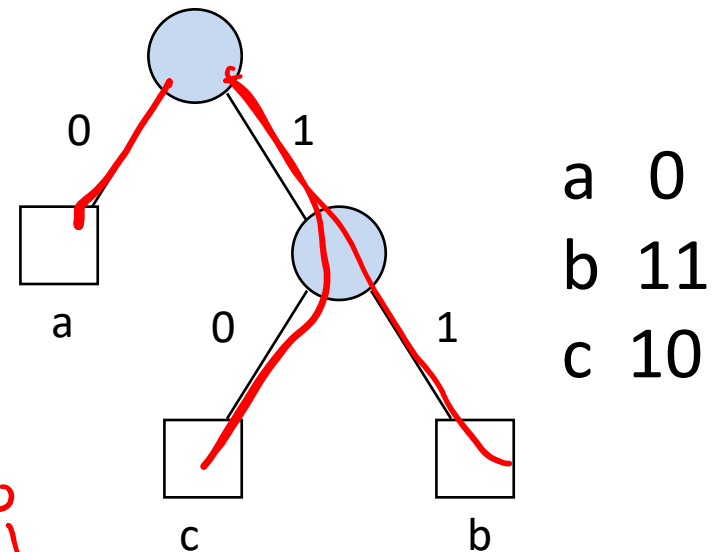
ASCII	Character
1000100	D
1100101	e
1100011	c

Frequency code	Character
1001	D
11	e
10001	c

- We require each codeword not be a prefix of any other codeword
- So, for the **binary code alphabet**, we should base the codes on **binary code trees**

Binary Tree to the Rescue!!!

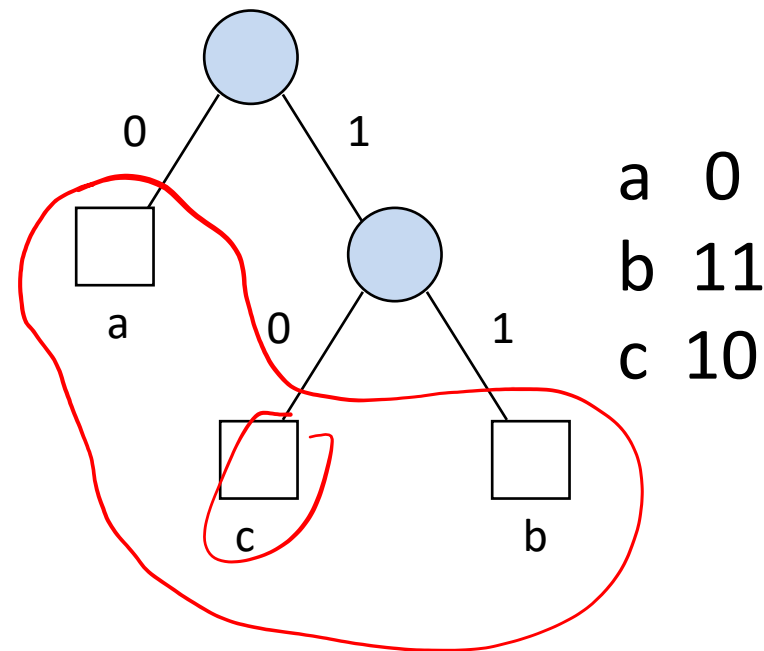
- Binary code trees:
 - *binary tree* whose external nodes are labelled uniquely with the source alphabet symbols
 - Left branches are labelled 0
 - Right branches are labelled 1



$A = \{A \dots Z\} \rightarrow 0$
 $\{A \dots Z\}$

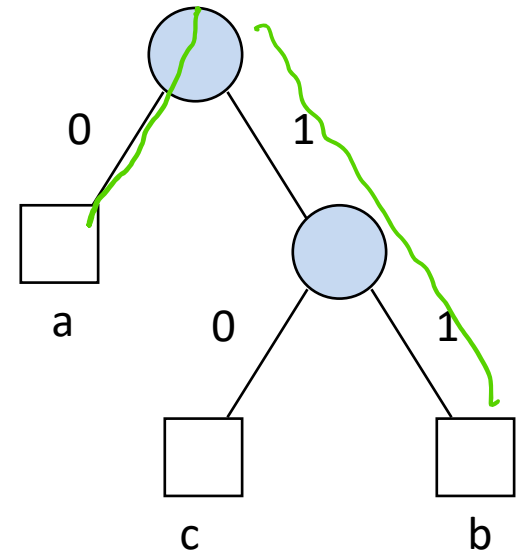
Binary code trees

- The codeword corresponding to a symbol is the bit string given by the path from the root to the external node labeled with the symbol
- Note that, as required, **no codeword is a prefix for any other codeword (prefix property).**
 - This follows directly from the fact that source symbols are only on external nodes
 - and there is only one (**unique**) path to that symbol



Prefix Codes (Binary code trees)

- Codes that satisfy the prefix property are called **prefix codes**
- Prefix codes are important because
 - we can uniquely decode an encoded text with a **left-to-right scan of the encoded text**
 - by **considering only the current bit** in the encoded text



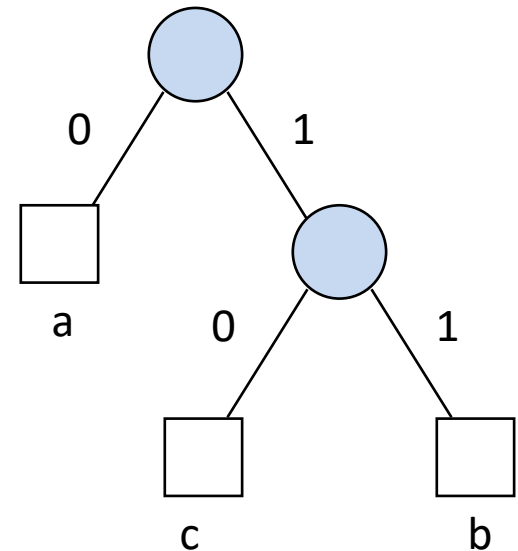
- decoder uses the code tree for this purpose

110110101000
B A B A C

~~BAC~~
BAC \Rightarrow 11 | 0 | 10

Text, Codes, and Compression

- Read the encoded message bit by bit
- Start at the root
- if the bit is a 0, move left
- if the bit is a 1, move right
- if the node is external, output the corresponding symbol and begin again at the root



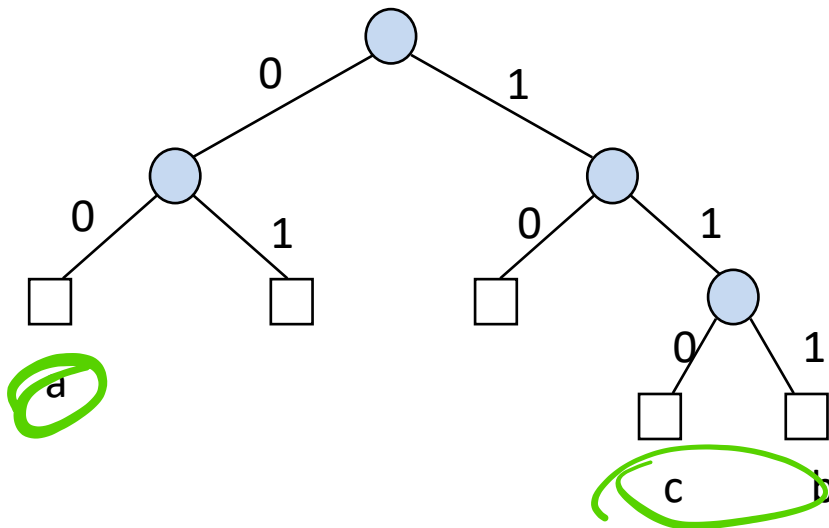
Encoded message:
0011100

Any binary tree with n external nodes labelled with the n symbols defines a prefix code

Not all prefix code trees are optimal (i.e., reduced)

a 000
b 111
c 110

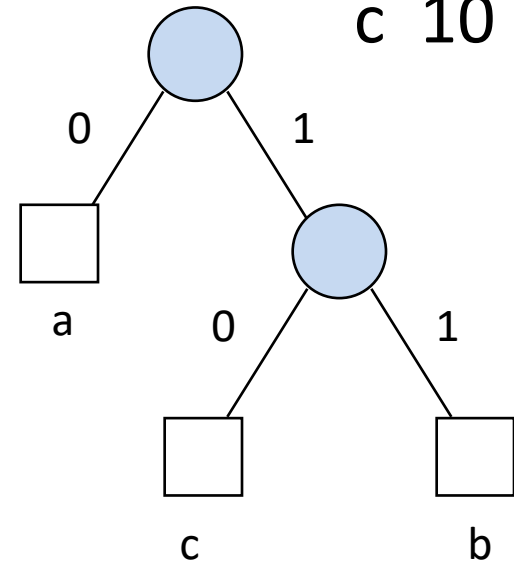
(3000)3



**Non-Reduced Prefix Code
(Tree)**

0(2)(3000)

a 0
b 11
c 10



**Reduced Prefix Code
(Tree)**

Lets Formally Define Optimality

Optimal Variable-Length Codes

- Any binary tree with n external nodes labelled with the n symbols defines a prefix code
- Any prefix code for the n symbols defines a binary tree **with at least** n external nodes
- Such a binary tree **with exactly** n external nodes is a **reduced prefix code (tree)**
- Good prefix codes are always reduced (and we can always transform a **non-reduced prefix code** into a **reduced one**)

Optimal Variable-Length Codes

- What makes a good variable length code?

$\{a, b, c\}$

- Let $A = a_1, \dots, a_n, n \geq 1$, be the alphabet of source symbols

$\{5, 10, 15\}$

- Let $P = p_1, \dots, p_n, n \geq 1$, be their probability of occurrence

- We obtain these probabilities by analysing a representative sample of the type of text we wish to encode

Optimal Variable-Length Codes

- Comparison of prefix codes - **compare the number of bits in the encoded text**

Let $A = a_1, \dots, a_n, n \geq 1$, be the alphabet of source symbols

Let $P = p_1, \dots, p_n$ be their probability of occurrence

Let $W = w_1, \dots, w_n$ be a prefix code for $A = a_1, \dots, a_n$

Let $L = l_1, \dots, l_n$ be the lengths of $W = w_1, \dots, w_n$

Handwritten example showing the encoding of the string "abcabcabc" using a prefix code. The string is written as "abcabcabc" with vertical lines separating the characters. Below each character, its binary code is written: 'a' is 10, 'b' is 11, and 'c' is 0. The total length of the encoded string is calculated as 18 bits.

Optimal Variable-Length Codes

- Given a source text T with f_1, \dots, f_n occurrences of a_1, \dots, a_n respectively
- The total number of bits when T is encoded is $\sum_{i=1}^n f_i l_i$ A =alphabet of source symbols

- The total number of source symbols is $\sum_{i=1}^n f_i$

code	len.	src	Freq.
1001	4	D	2
11	2	e	5
1000	4	c	3

- The **average length** of the W-encoding is

$$\text{Alength}(T, W) = \sum_{i=1}^n f_i l_i / \sum_{i=1}^n f_i$$

Optimal Variable-Length Codes

- For long enough texts, the probability p_i of a given symbol occurring is approximately

$$p_i = f_i / \sum_{i=1}^n f_i$$

- So the **expected length** of the W-encoding is

$$\text{E-length}(W, P) = \sum_{i=1}^n p_i l_i$$

code	len.	src	Freq.
1001	4	D	2
11	2	e	5
1000	4	c	3

Optimal Variable-Length Codes

- To compare two different codes W_1 and W_2 we can compare either
 - $\text{Length}(T, W_1)$ and $\text{Length}(T, W_2)$ or
 - $\text{Length}(W_1, P)$ and $\text{Length}(W_2, P)$
- We say W_1 is no worse than W_2 if

A = alphabet of source symbols

P = probability of occurrence

$$\text{Length}(W_1, P) \leq \text{Length}(W_2, P)$$

- We say W_1 is **optimal** if

$$\text{Length}(W_1, P) \leq \text{Length}(W_2, P)$$

for all possible prefix codes W_2 of A

$$W_1 \leq W_2$$

- ① same alphabet
- ② same frequency

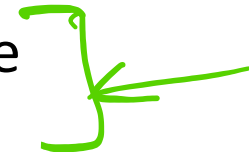
Huffman's Algorithm : Optimal Variable-Length Codes

- Huffman's Algorithm
- We wish to solve the following problem:

- Given n symbols $A = a_1, \dots, a_n$, $n \geq 1$



and the probability of their occurrence
 $P = p_1, \dots, p_n$, respectively,



construct an optimal prefix code for A and P

Difficulty Generating Optimal Variable-Length Codes

- This problem is an example of a **global optimization problem**
- Brute force (or exhaustive search) techniques are too expensive to compute:
 - Given A and P
 - Compute the set of all reduced prefix codes
 - Choose the minimal expected length prefix code

Difficulty Generating Optimal Variable-Length Codes

- This algorithm takes $O(n^n)$ time, where n is the size of the alphabet
- Why? because any binary tree of size $n-1$ (i.e. with n external nodes) is a valid reduced prefix tree and there are $n!$ ways of labelling the external nodes
- Since $n!$ is approximately n^n we see that there are approximately $O(n^n)$ steps to go through when constructing all the trees to check

Huffman Encoding

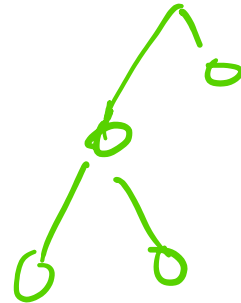
Time Complexity of Huffman

- Huffman's Algorithm is only $O(n^2)$
- This is significant: if $n = 128$ (number of symbols in a 7-bit ASCII code)
 - $O(n^n) = 128^{128} = 5.28 \times 10^{269}$
 - $O(n^2) = 128^2 = 1.6384 \times 10^4$
 - There are 31536000 seconds in a year and if we could compute 1000 000 000 steps a second then the brute force technique would still take 1.67×10^{253} years

Optimal Variable-Length Codes

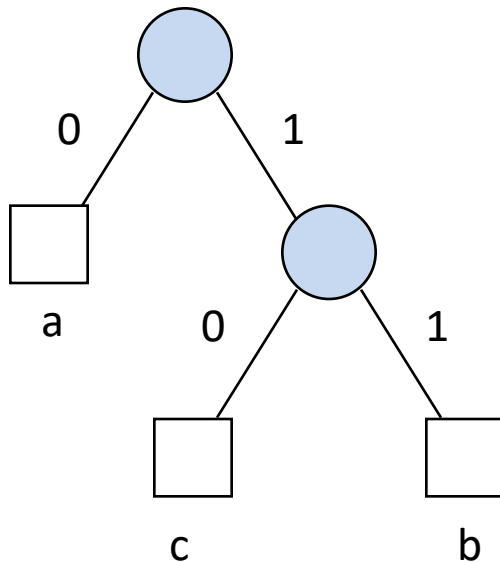
- Huffman's Algorithm uses a technique called *Greedy technique*.
- It uses local optimization to achieve a globally optimum solution
 - Build the code incrementally
 - Reduce the code by one symbol at each step
 - Merge the two symbols that have the smallest probabilities into one new symbol

Optimal Variable-Length Codes

- Before we begin, note that we'd like a tree with the symbols which have the lowest probability to be on the longest path
- Why?
- Because the length of the codeword is equal to the path length and we want
 - short codewords for high-probability symbols
 - longer codewords for low-probability symbols

Text, Codes, and Compression

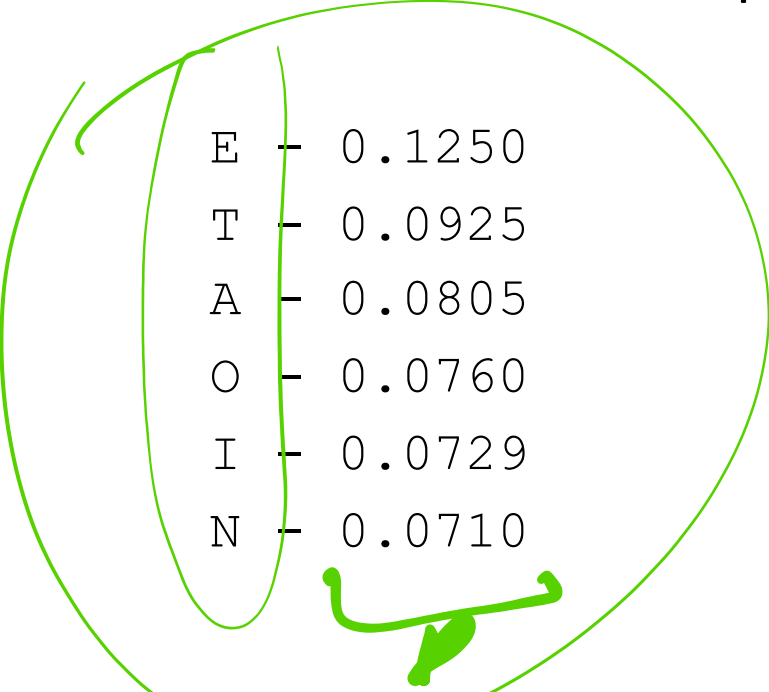
A binary code tree and its prefix code



a	0
b	11
c	10

Huffman's Algorithm

- We will treat Huffman's Algorithm for just six letters, i.e, $n = 6$, and there are six symbols in the source alphabet
- These are, with their probabilities,

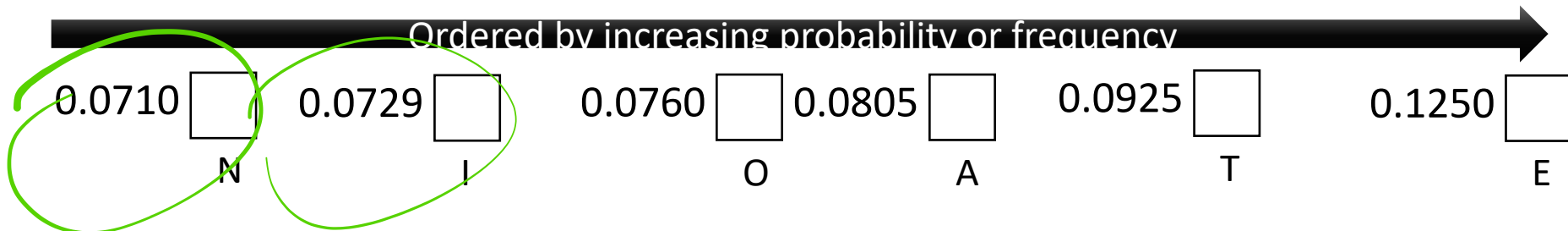


E	-	0.1250
T	-	0.0925
A	-	0.0805
O	-	0.0760
I	-	0.0729
N	-	0.0710

Huffman's Algorithm

Step 1:

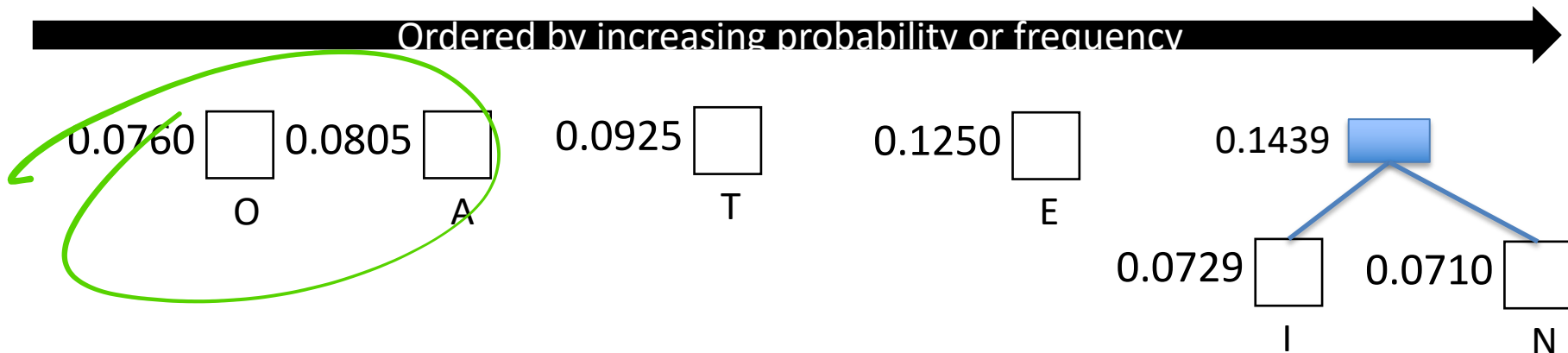
- Create a **forest** of code trees, one for each symbol
- Each tree comprises a single external node (empty tree) labelled with its symbol and weight (probability)
- Arrange the nodes in increasing order of weight (frequency or probability)



Huffman's Algorithm

Step 2:

- Choose the two binary trees, B1 and B2, that have the **smallest weights**
- **Create a new root node** with B1 and B2 as its children(right and left) and with weight equal to the sum of these two weights
- **Add** the new root node to the forest, **remove** B1 and B2, and **maintain** the ordering.

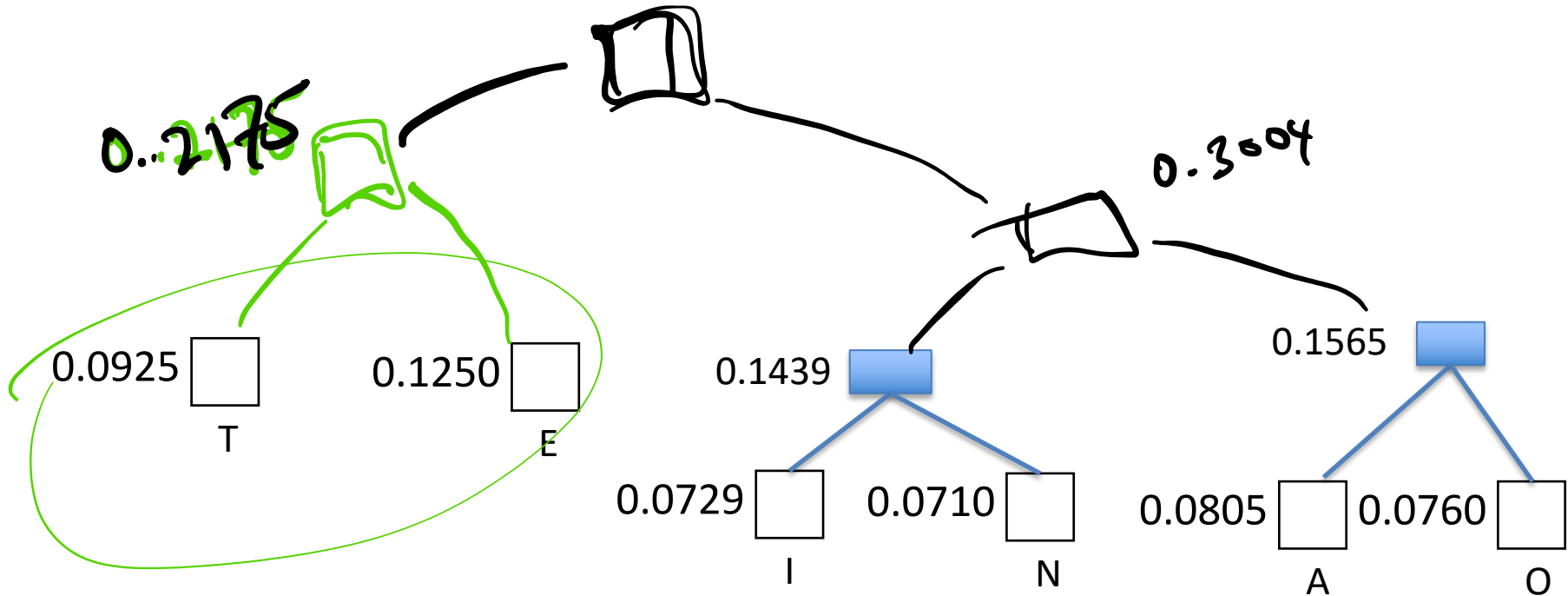


Huffman's Algorithm

Step 3:

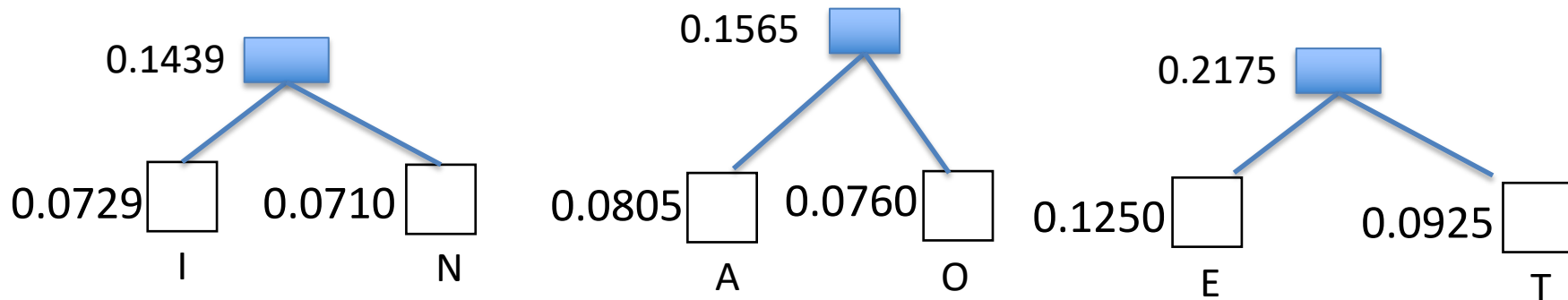
- Repeat step 2 until the forest only has **one tree**---the *Huffman tree*!

Huffman's Algorithm



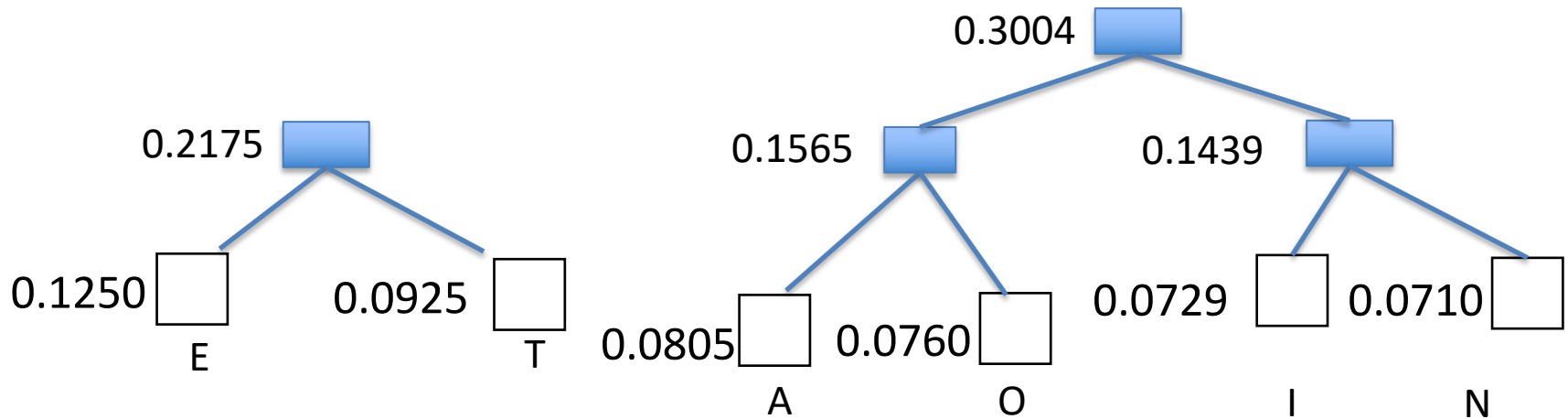
Ordered by increasing probability or frequency

Huffman's Algorithm



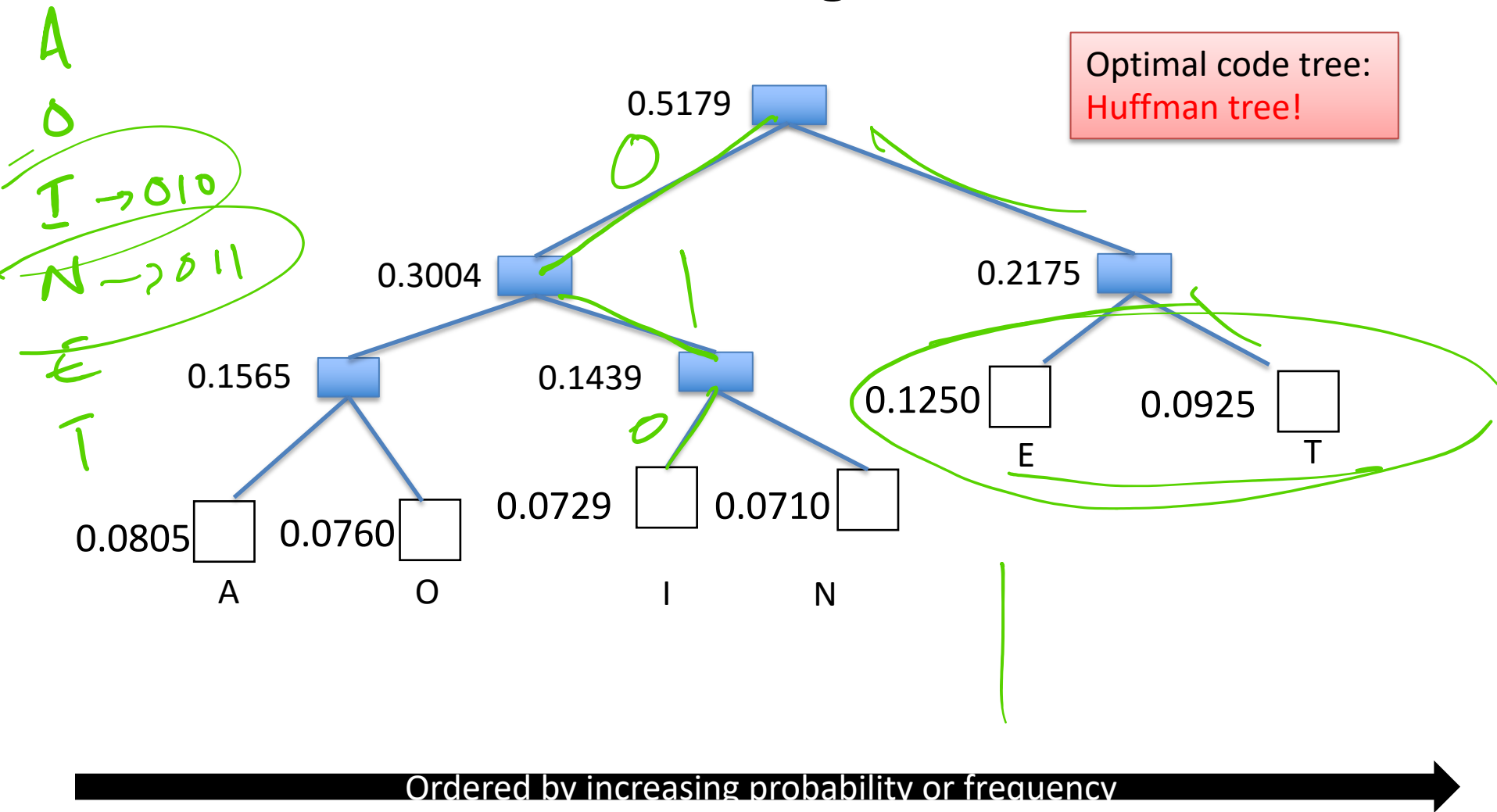
Ordered by increasing probability or frequency

Huffman's Algorithm

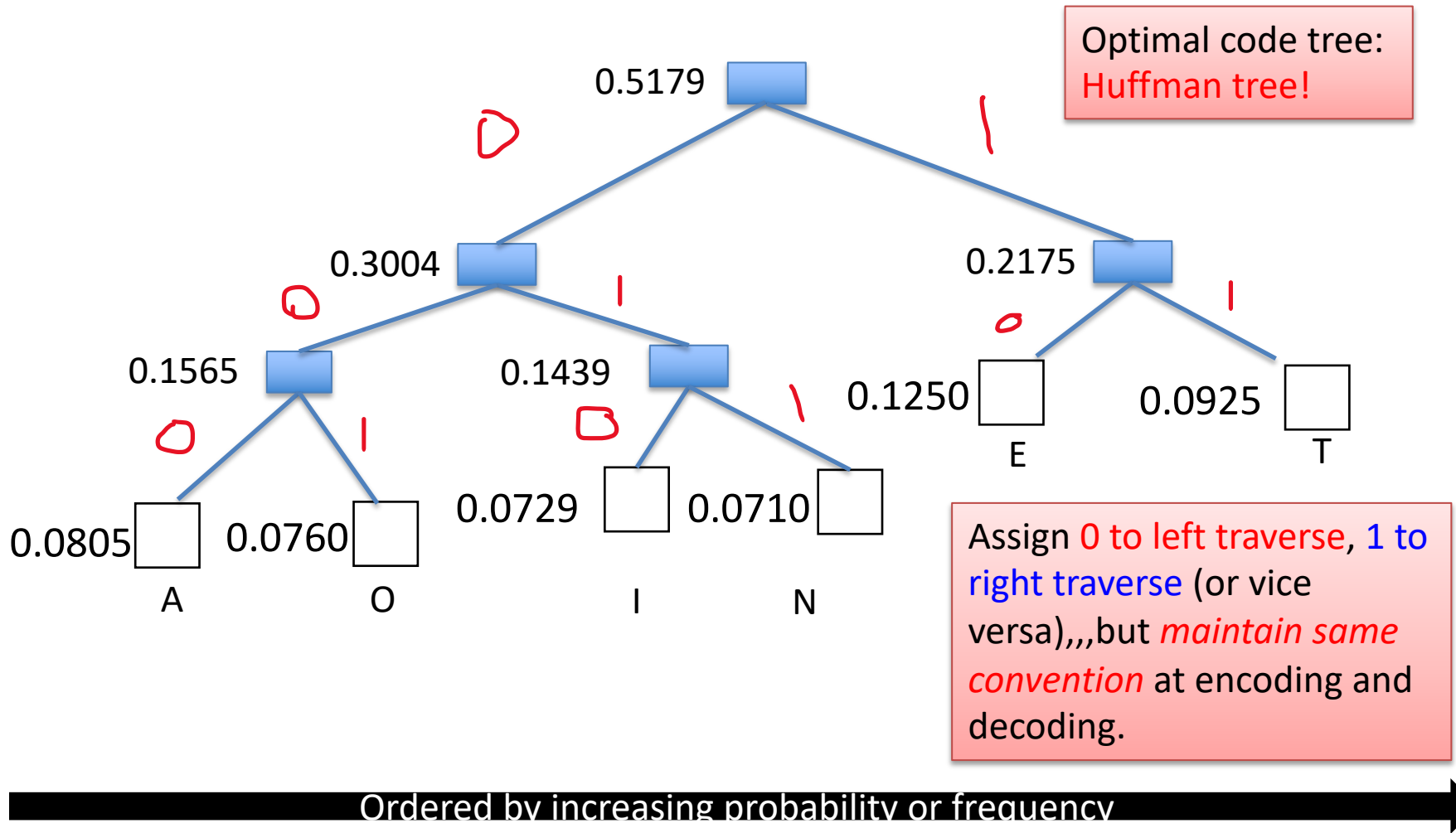


Ordered by increasing probability or frequency

Huffman's Algorithm



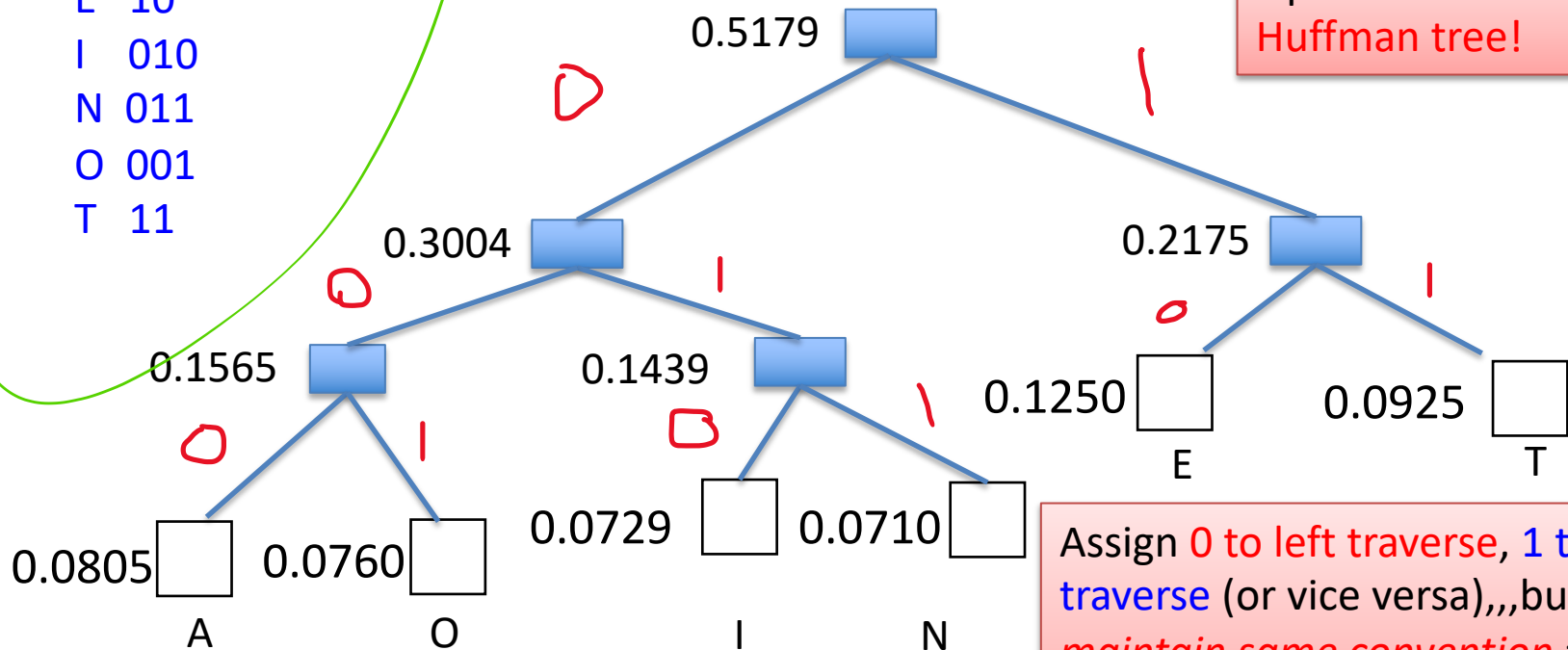
Huffman's Algorithm



Huffman's Algorithm

The final prefix code is:

A 000
E 10
I 010
N 011
O 001
T 11



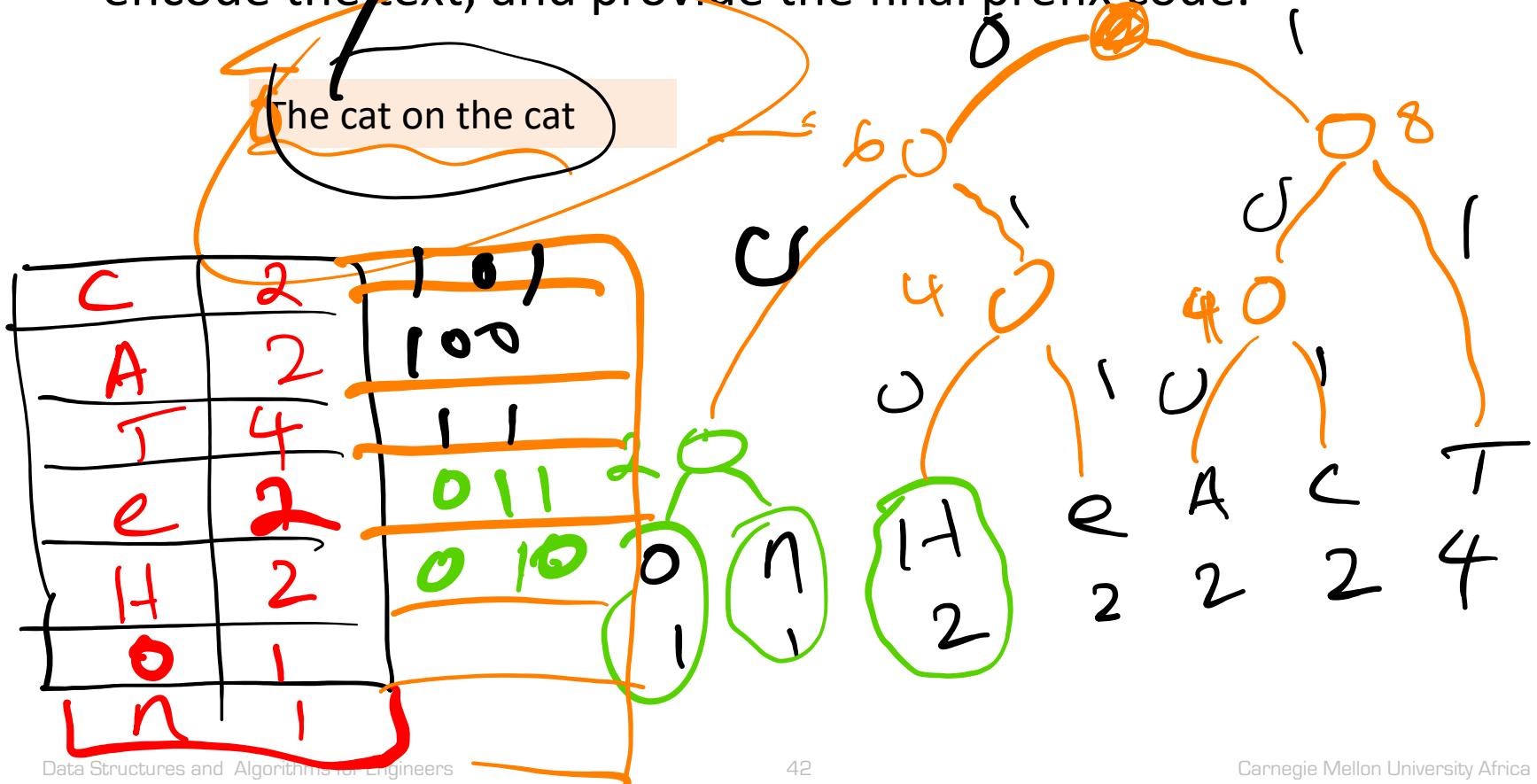
Optimal code tree:
Huffman tree!

Assign 0 to left traverse, 1 to right traverse (or vice versa),, but *maintain same convention* at encoding and decoding.

Ordered by increasing probability or frequency

Exercise

- Construct the prefix tree for the following text, compute the average code length, compute the number of bits needed to encode the text, and provide the final prefix code.



Exercise

- Construct the prefix tree for the following text, compute the average code length, compute the number of bits needed to encode the text, and provide the final prefix code.

The cat on the cat

src	frequency
t	4
h	2
e	2
c	2
a	2
n	1
o	1

Three phases in the algorithm

E - 0.1250
T - 0.0925
A - 0.0805
O - 0.0760
I - 0.0729
N - 0.0710

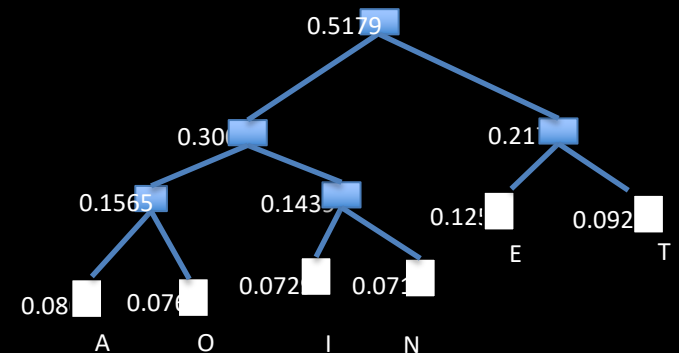
Remember must have a frequency table to do this

1. Initialize the forest of code trees
2. Construct an optimal code tree
3. Compute the encoding map

Ordered by increasing probability or frequency

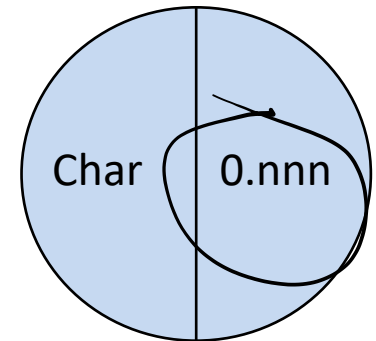
Character	Frequency
N	0.071
I	0.072
O	0.076
A	0.080
T	0.092
E	0.125

A 000
 E 10
 I 010
 N 011
 O 001
 T 11



Huffman's Algorithm

Phase 1: Initialize the forest of code trees

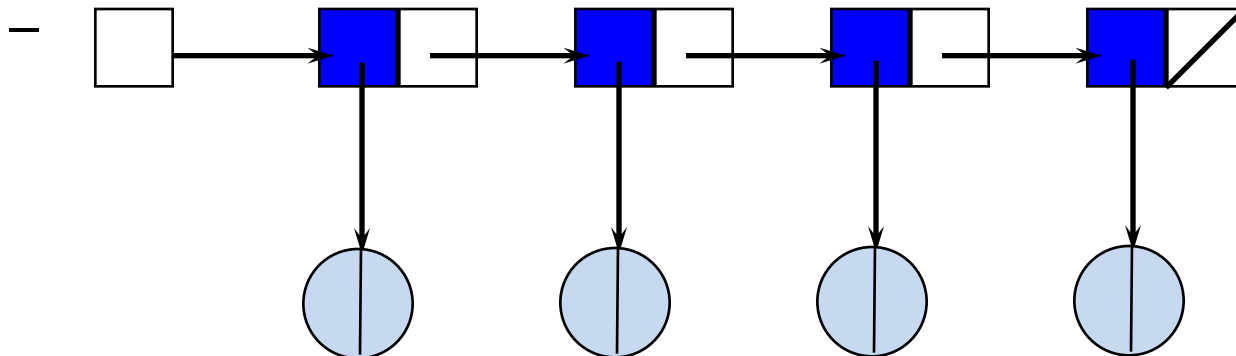


- How will we represent the forest of trees?
- Better question: how will we represent our tree ... have to store both alphanumeric characters and probabilities or frequencies?
- Need some kind of composite node
- Opt to represent this composite node as an INTERNAL node

Huffman's Algorithm

So, to create such a tree we simply invoke the following operations:

- Initialize the tree ... `tree()`
- Add a node ... `addnode(char, weight, T)`
- We must also keep track of our forest (of trees)
- Could represent it as a linked list of pointers to Binary trees ...

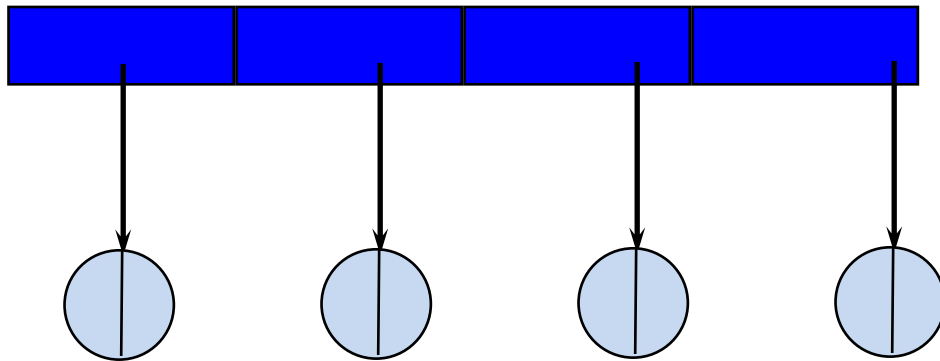


Huffman's Algorithm

- **Question:** why do we use dynamic data structures? Is there an alternative?
- **Answer:**
 - When we don't know in advance how many elements are in our data set (When the number of elements varies significantly)
- Is this the case here?
 - **No!**
- **Alternative:** An array, indexed by number, of type ...
 - *binary_tree*, i.e., each element in the array can point to a binary code tree



Huffman's Algorithm



Source alphabet	
src	frequency
t	4
h	2
e	2
c	2
a	2
n	1
o	1

- What will be the dimension of this array?
 - n , the number of symbols in our source alphabet since this is the number of trees we start out with in our forest initially

Three phases in the algorithm

E - 0.1250
T - 0.0925
A - 0.0805
O - 0.0760
I - 0.0729
N - 0.0710

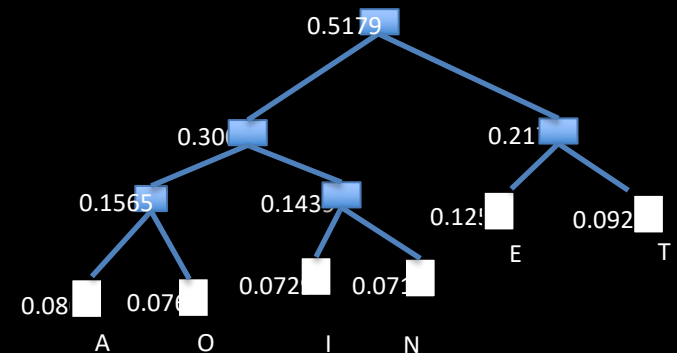
Remember must have a frequency table to do this

1. Initialize the forest of code trees
2. Construct an optimal code tree
3. Compute the encoding map

Ordered by increasing probability or frequency

0.071	0.072	0.076	0.080	0.092	0.125
N	I	O	A	T	E

A 000
E 10
I 010
N 011
O 001
T 11



Huffman's Algorithm

let n be the number of trees initially

Repeat

Find the tree with the smallest weight - A , at element i

Find the tree with the next smallest weight - B , at element j

Construct a tree, with A and B as children, with root having weight = sum of the roots of A and B

Add the tree to the array. Remove A and B .

Maintain ordering.

Until only one tree left in the array

Three phases in the algorithm

E - 0.1250
T - 0.0925
A - 0.0805
O - 0.0760
I - 0.0729
N - 0.0710

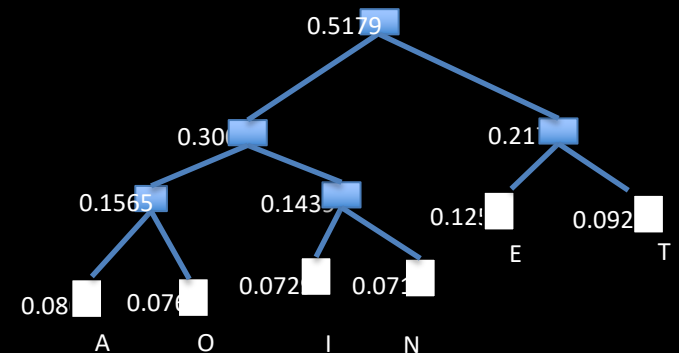
Remember must have a frequency table to do this

1. Initialize the forest of code trees
2. Construct an optimal code tree
3. Compute the encoding map

Ordered by increasing probability or frequency

0.071	0.072	0.076	0.080	0.092	0.125
N	I	O	A	T	E

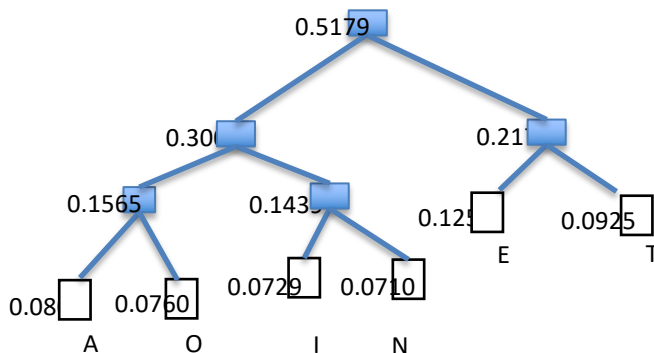
A 000
E 10
I 010
N 011
O 001
T 11



Huffman's Algorithm

Phase 3: Compute the encoding map

- We need to write out a list of source symbols together with their prefix code
 - print out the symbol (leaf) and the prefix code (path):
 - Paths → Array of binary values
- We need to **traverse** the binary code tree in some manner



A	000
E	10
I	010
N	011
O	001
T	11

Today: Optimal Code Trees



- First application: *coding and data compression*
- We will define **optimal variable-length binary codes** and code trees.
- We will study **Huffman's algorithm** which constructs them
- Huffman's algorithm is an example of a **Greedy Algorithm**, an important **class of simple optimization algorithms**