Computer Science Principles

CHAPTER 8 – DATA REPRESENTATION AND COMPRESSION

Announcements

Read Chapter 8 in the Conery textbook (Explorations in Computing)

Acknowledgement: These slides are revised versions of slides prepared by Prof. Arthur Lee, Tony Mione, and Pravin Pawar for earlier CSE 101 classes. Some slides are based on Prof. Kevin McDonald at SBU CSE 101 lecture notes and the textbook by John Conery.

Data and Computers

Computers are multimedia devices, dealing with a vast array of information categories

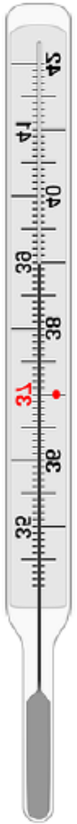
- Information is data (basic values, facts) that has been organized or processed into useful form

Computers store, present and help us modify various kinds of data: numbers, text, audio, images and graphics, video

Information can be represented in one of two ways: analog or digital

- **Analog data**: a continuous representation, analogous to the actual information it represents
- **Digital data**: a discrete representation that breaks the information up into separate elements

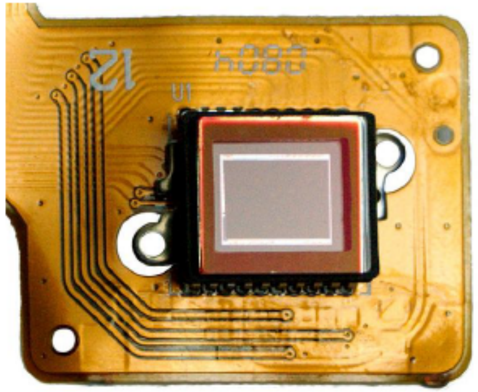
Analog vs. Digital



vs.

vs.

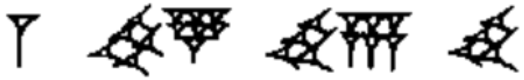
vs.



Many Ways Developed to Represent Numbers

Egyptian hieroglyphs:

Babylonian numerals:



$$= (1 * 60^3) + (57 * 60^2) + (46 * 60) + 40$$
$$= 424{,}000$$

Many others:

Arabic	0	1	2	3	4	5	6	7	8	9	10
Chinese	〇	一	二	三	四	五	六	七	八	九	十
Roman		I	II	III	IV	V	VI	VII	VIII	IX	X
Devanagari (Hindi)	०	१	२	३	४	५	६	७	८	९	१०

References:

- http://goo.gl/BSrWTH
- http://goo.gl/r8NcKF
- Wikipedia

Positional Notation

The modern Western style and some other styles of writing numbers use **positional notation**

- The position of a digit determines how much it contributes to the number's value

With **decimal** (base 10), **place-values** are powers of 10:

…, 10^3, 10^2, 10^1, 10^0, 10^{-1} , 10^{-2}, 10^{-3}, …

…, 1000s, 100s, 10s, 1/10 s, 1/100 s, 1/1000 s, …

642.15 really means $(6 \times 10^2) + (4 \times 10^1) + (2 \times 10^0) + (1 \times 10^{-1}) + (5 \times 10^{-2})$

Early computers represented numbers with base 10, but they were very unreliable. It was too hard to make the computer maintain 10 distinct voltages for the 10 digits.

Binary Numbers

Modern digital computers use **binary digits** (base-2 numbers: 0 and 1)

- The word **bit** is short for **binary digit**

The hardware determines how bits are stored

- Hard drive: magnetized spots on surface of disk
- Flash drive: presence/absence of electrons in a memory cell
- Optical disc (CD/DVD): pits and lands (flat spots)

As computational thinkers, we do not need to worry so much about how the bits are stored

- Instead, we will focus on what bits are stored

Binary Numbers

With binary we have just two digits, 0 and 1, and the place-values are powers of 2:

…, 2^3, 2^2, 2^1, 2^0

…, 8, 4, 2, 1

For example: 1100_2 written as a base-10 (decimal) number is:

$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$

$= 8 + 4 + 0 + 0 = 12_{10}$

Important to note: 1100_2 and 12_{10} are two different representations of the same quantity

Binary Numbers

Binary can also represent fractions, for example:

…, 2^3, 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}, …

..., 8s, 4s, 2s, 1s, 1/2s, 1/4s, 1/8s, …

The number 1011.011_2 written as a decimal number is:

$(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$

$= 8 + 0 + 2 + 1 + 0 + 1/4 + 1/8 = 11.375_{10}$

All data in a modern machine is stored using binary numbers

Decimal → Binary Conversion

To convert a decimal number to binary, perform these steps:

1. Repeatedly divide the decimal number by 2.
2. Set aside the remainder of each division.
3. Use the quotient for the next round of division.
4. When the quotient reaches 0, write down all of the remainders in order from last to first. This value is your answer.

This algorithm is easier to understand with examples

Decimal ➔ Binary Example #1

Convert 12_{10} to binary

12 / 2 = 6 remainder 0

6 / 2 = 3 remainder 0

3 / 2 = 1 remainder 1

1 / 2 = 0 remainder 1

Answer: 1100_2

Note that we write the remainders in the reverse order of how they are generated

Decimal → Binary Example #2

Convert 123_{10} to binary

123 / 2 = 61 remainder 1

61 / 2 = 30 remainder 1

30 / 2 = 15 remainder 0

15 / 2 = 7 remainder 1

7 / 2 = 3 remainder 1

3 / 2 = 1 remainder 1

1 / 2 = 0 remainder 1

Answer: 1111011_2

The Function dec2bin()

Function **dec2bin()** returns a string of 0s and 1s giving the binary representation of an integer

```
def dec2bin(decimal):
    binary = ""
    while decimal > 0:
        remainder = decimal % 2
        binary = str(remainder) + binary
        decimal = decimal // 2
    return binary

print(dec2bin(23))       # "10111"
print(dec2bin(100))      # "1100100"
```

data_rep.py

Encoding Data (next)

To store information in a computer's memory we have to encode it somehow: an **encoding** is a pattern of 0s and 1s

- The pattern is a representation of some real-world object, like a letter, number, sound clip, or video

Encoding is not the same as **encryption**

- Both use codes, but in this slide set we will explore standard ways of *representing* data, not *hiding* data

A set of *k* bits can represent up to 2^k items. Let's see why.

- Each bit can be 0 or 1 (two options)
- With 2 bits, we can represent $2^2 = 4$ items
- With 3 bits, we can represent $2^3 = 8$ items
- …
  With k bits, we can represent 2^k items

Representing Characters

ASCII (American Standard Code for Information Interchange) includes 7-bit and 8-bit schemes for representing characters used in the English language

Each letter, number, punctuation mark, etc. is mapped to a 7-bit number

- See ASCII.txt

Examples: capital letter “A” is 65; lowercase letter “a” is 97

A newer scheme called **Unicode** includes codes for over 100 alphabets

- Modern languages (Greek, Cyrillic, Arabic, Hebrew, Korean, Chinese, Japanese, ...) and ancient languages (hieroglyphics, runes, ...)
- Also includes technical symbols, emojis, and other symbols

Representing Characters

Here are three ways to include a Unicode symbol in a Python string:

1. Copy and paste text from an e-mail, a web page, etc.
2. Use a function named **chr** (short for "character")
   - Pass it a code number. It will return a one-letter string containing that symbol.

     Example: **chr(9829)** gives '   '
   - Find code numbers at www.charbase.com or similar websites that have lists of Unicode symbols
3. Use an escape sequence **'\uXXXX'** where **XXXX** is the 4-digit **hexadecimal** (base 16) code number

'I \u2665 cats' → 'I   cats'

See data_rep.py for more examples

Hexadecimal Numbers

In software, sometimes it's more natural to write numbers in base 16, called **hexadecimal**

With hexadecimal we have 16 digits:

- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- A-F correspond to the numbers 10-15

Same Value, Different Base

Base 10	Base 2	Base 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Base 10	Base 2	Base 16
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Hexadecimal Numbers

Place-values in hexadecimal are powers of 16:

…, 16^3, 16^2, 16^1, 16^0, 16^{-1}, 16^{-2}, 16^{-3}, …

- $51E_{16} = (5 \times 16^2) + (1 \times 16^1) + (14 \times 16^0) = 1{,}310_{10}$
- $FAD_{16} = (15 \times 16^2) + (10 \times 16^1) + (13 \times 16^0) = 4{,}013_{10}$

Changing the base of a number doesn’t change the magnitude (value) of a number

- The representation for a number gets longer as the base decreases

Hexadecimal Numbers

Hexadecimal is used widely in web design for giving colors

When giving the Unicode for a character as an escape sequence with **\u**, we always use hexadecimal

- In contrast, the **chr()** function expects the decimal representation

For creating the heart symbol

- Use **9829** (base 10) for **chr()**
- Use **2665** (base 16) for the escape sequence

The related **ord()** function returns the Unicode value of a character in decimal format:

- **ord('A')** returns **65** and **ord('x')** returns **120**

See data_rep.py for more examples

Hexadecimal Numbers

The binary representation for a string can be hard to read:

01001001 00100111 01101101 00100000 01100001

01100110 01110010 01100001 01101001 01100100

00100000 01101111 01100110 00100000 01100011

01101111 01110111 01110011 00101110

It’s a little easier to deal with codes in hexadecimal:

49 27 6D 20 61 66 72 61 69 64 20 6F 66 20 63 6F 77 73 2E

Recall that in hexadecimal we have 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and the letters A through F for ten through fifteen

- Note that each hexadecimal digit corresponds with four binary digits (bits)

Binary ↔ Hexadecimal

Base 2	Base 16
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

To convert a hexadecimal number to a binary number, simply convert each digit in the hexadecimal number into a four-digit binary number.

For example:

$\mathbf{D2B5}_{16} = \mathbf{1101001010110101}_2$

To convert a binary number to a hexadecimal, convert every four binary digits from **right to left** in the binary number into a hexadecimal digit.

For example:

$\mathbf{010001111110}_2 = \mathbf{47E}_{16}$

Groups of Bits

A **byte** is a collection of 8 bits

- A 7-bit ASCII value fits in single byte

A 32-bit integer requires 4 bytes ($32 \div 8 = 4$)

A central processing unit (CPU) operates on several bytes at a time, called a **word**

- A **word** is a collection of two or more bytes
- Typical word size are 32 bits (4 bytes) and 64 bits (8 bytes)

Memory capacity is often described in terms of *megabytes* or *gigabytes*

Error Detection

Errors in values can be caused by circumstances beyond our control

- The storage medium itself has a flaw or is deteriorating
- Data can be corrupted by interference during transfer over wires or wirelessly
- Even solar activity itself can affect electronic devices and disrupt electronic communication

The general method for detecting errors is to add extra information to the data

- Add extra data to a document before storing it in a file
- Append error checking data to a message while sending it

Error Detection

The basic procedure for enabling error-free communication:

1. Sender adds error-checking information
2. After receiving the message, the receiver analyzes the message along with the extra data to see if an error occurred
3. If an error occurred, the receiver will ask the sender to send the message again

A simple method for error-checking is to use a **parity bit**

- Add one extra bit to the end of the text
- Here, “text” means any string: an entire message or a single character

Error Detection

The value of the extra bit should make the total number of '1' bits an even number

- This property is called **even parity**

Example: parity bits for 8-bit ASCII characters

ASCII Value	Parity Bit	Total Message
A = 01000001	0 (since A has two 1's)	01000001 0
C = 01000011	1 (since C has three 1's)	01000011 1

Example for encoding the message "ATG" in ASCII

Original Message	Parity Bit	Total Message
01000001 01010100 01000111	1 (since the original message has 9 1's)	01000001 01010100 01000111 1

Error Detection

The receiver treats the parity bit like any other bit in the incoming message

- It is included in the count of the number of 1 bits
- To get the message contents, the receiver discards the last bit

Example: when sending an 8-bit ASCII 'C', the bit stream is 010000111: the digits in the code for 'C' plus a parity bit

- The receiver reads 9 bits and sees there was an even number of 1 bits; no error detected
- The receiver discards the 9th bit
- The remaining bits are the contents of the message: 01000011, which is the ASCII code for 'C’

Note: It is a very simple scheme. It can only be used to detect single or any other odd number (e.g., three, five, etc.) of errors in the output. An even number of flipped bits (errors) will make the parity bit appear correct even though the data is erroneous.

Aside: Communication Protocols

For this error-checking plan to work, the sender and receiver both have to agree on a **communication protocol**

- The protocol defines a message structure and also specifies what actions are taken during the transmission or receipt of a message
- In our simple protocol, the sender and receiver agree in advance the parity bit is the last bit

Two of the most important protocols used today are **Transmission Control Protocol (TCP)** and **Internet Protocol (IP)**

- Used extensively in Internet communication, including the Web and online video games

Computing Parity Bits

How can we write a function that computes the parity bit for a message?

First, we need to understand some different logic functions when dealing with bits

- We are familiar with **"and"** and **"or", which has the behavior shown below:**

Input		Output	
A	**B**	**AND**	**OR**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Computing Parity Bits

There is also a logic function called "**exclusive or**", abbreviated as **XOR**

- The XOR of variables **a** and **b** is true if either one of them is true, but <u>not</u> if both are true

It is uncommon in programming to use XOR in Boolean expressions (True/False expressions)

- Rather, XOR is used (almost) exclusively in bitwise operations, which are expressions that involve 0s and 1s

Input		**Output**		
A	**B**	**AND**	**OR**	**XOR**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Computing Parity Bits

Python has uses different symbols for bitwise operators

In Python:

- XOR is denoted using the caret, **^**
- Bitwise **and** is denoted by the ampersand, **&**
- Bitwise **or** is denoted by the vertical bar, or pipe, **|**

Input		Output		
A	**B**	**AND** **A & B**	**OR** **A | B**	**XOR** **A ^ B**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Computing Parity Bits

Let’s consider a function that computes a parity bit. Here’s the algorithm:

1. Initialize the return value **p** to 0.
2. Iterate over all the bits in the input code, updating **p** using the XOR operator: **p = p ^ bit**
   - If a bit is 0, it won’t change **p**.
   - But if it's a 1, it sets **p** to the opposite value.

Here’s why this works. We start with **p = 0**. Every time we see a 1, we “flip” the parity bit by replacing **p** with

p XOR 1.

For example, if the data contain three 1s, then **p** will flip three times: **p** = 0 ➔1 ➔ 0 ➔ 1, so the parity bit will be 1

The parity() Function

Given a string containing only 0s and 1s, the **parity()** function computes the parity bit

```
def parity(bits):
    p = 0
    for bit in bits :
        p = p ^ int(bit)
    return p
```

Examples:

```
parity('1000001')          # returns 0
parity('1000011')          # returns 1
```

See data_rep.py

Groups of Bits

We can define our own encoding schemes for objects

Suppose we wanted to encode DNA sequences, which are strings containing the letters A, C, G, and T

- We need only 2 bits to represent 4 different things

We could create a dictionary to map a letter to a 2-bit code

Example: "A" → 00
"C" → 01
"G" → 10
"T" → 11

We will now look at a famous algorithm for compressing data which produces a new, original binary encoding scheme for a given input data-set

Text Compression

Data compression algorithms reduce the amount of space needed to store a piece of data

A data compression technique can be:

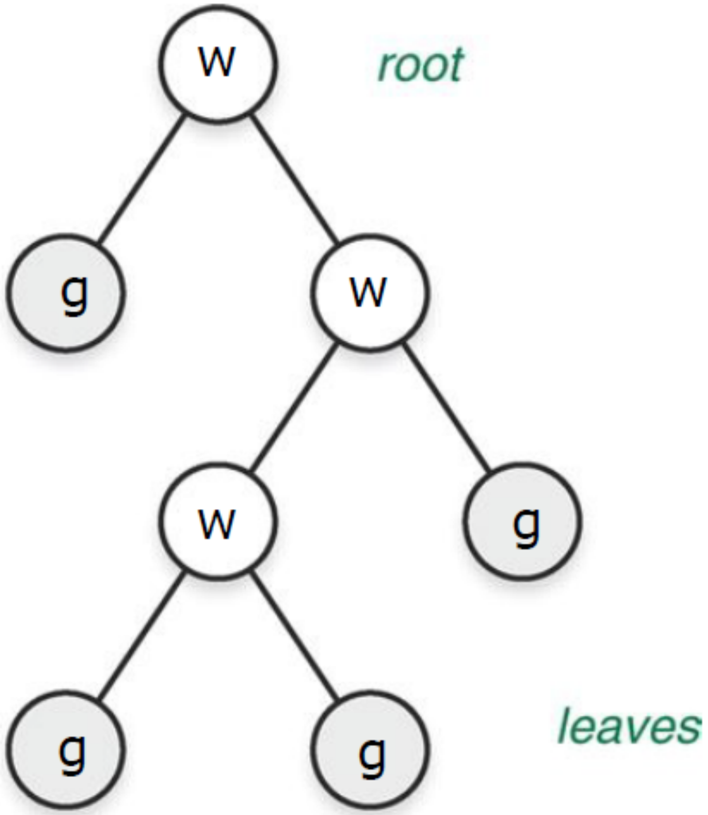
- **Lossless** (no information lost)
- **Lossy** (information lost)

There are many algorithms for compressing files (including photos, images and other types of data) but we'll focus on a lossless technique for text compression called **Huffman coding**

We will first need to explore a few data structures before we can understand how Huffman coding works

Binary Trees

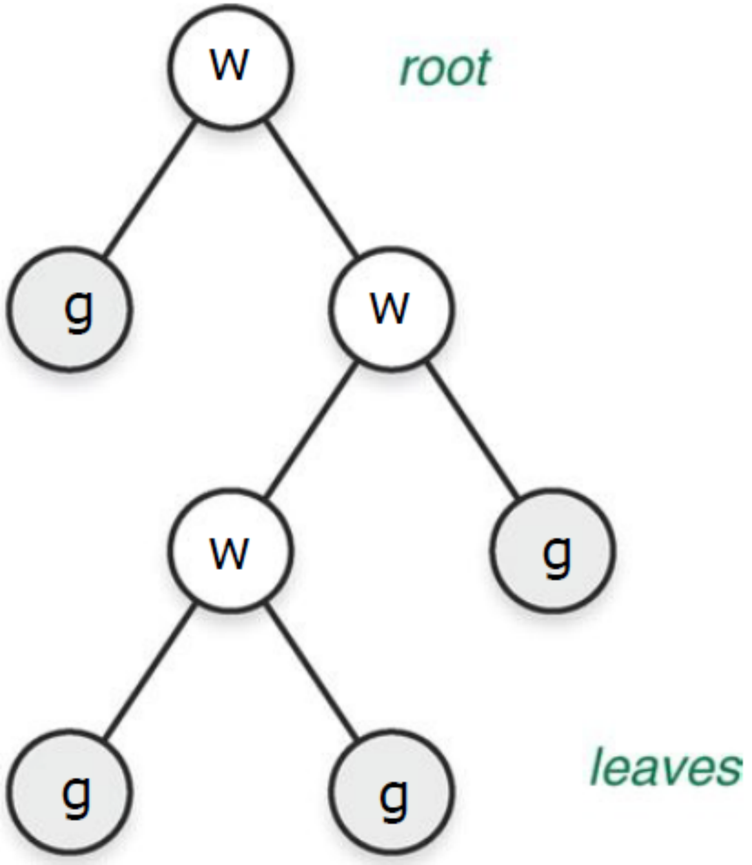
- In mathematics and computer science, a **tree** consists of data values stored at **nodes**, which are connected to each other in a hierarchical manner by **edges**
- Like a family tree, a tree shows parent-child relationships
- Each node in the tree, except for a special node called the **root**, has exactly *one* **parent node**
- Nodes can be connected to 0 or more **child** nodes immediately beneath them in the tree
- A node with at least one child is called an **interior node** (colored ‘w’hite in the figure)
- Towards the bottom of a tree we find nodes with no children; such nodes are called **leaves** (shaded ‘g’ray in the figure)



Binary Trees

In a **binary tree**, every node has either 0, 1 or 2 children

Used in this context, the word “binary” refers to the maximum number of children that a node can have. It does not refer to bits.



Huffman Coding

Huffman coding is a scheme for encoding letters based on the idea of using shorter codes for more commonly used letters

- ASCII uses 7 or 8 bits to store every letter, regardless of how often that letter is used in real text
- Imagine if we could find a way to store commonly used letters like R, S, T, N, L, E, etc. using fewer bits

For large data-sets consisting only of characters, the potential savings is huge

- This is what Huffman coding accomplishes

Huffman Coding

A **Huffman tree** is a binary tree that is at the center of Huffman coding

Inside of each node of a Huffman tree we store:

1. A letter
2. The frequency of how often that letter appears in words

The Hawaiian Alphabet

We will use the Hawaiian alphabet as part of a running example to understand how Huffman coding works

Hawaiian words are spelled with:

- Five vowels A, E, I, O, U, and
- Seven consonants H, K, L, M, N, P, W

Also, the ' symbol, called the *okina*, is used between two vowels when they should be pronounced as separate syllables

- Example: “a'a” is pronounced “ah-ah”

The Hawaiian Alphabet

The table to the right shows the frequency of each letter in Hawaiian words

We will use this knowledge to find an efficient encoding of the 13 symbols

Letter	Frequency
'	0.068
A	0.262
E	0.072
H	0.045
I	0.084
K	0.105
L	0.044
M	0.032
N	0.083
O	0.107
P	0.030
U	0.059
W	0.009

Data Structures for Huffman Coding

In an earlier lecture we learned about a special kind of list called a *priority queue*

- Every item inserted into a priority queue has a corresponding numerical priority
- The priority queue always makes sure that the item with highest priority is at the front of the list

The **PriorityQueue** class in the SpamLab implements the priority queue concept

- The **insert()** method adds an item to the priority queue
- The **pop()** method removes the item at the front of the list, which is guaranteed to be the item of highest priority

We will use a priority queue to help us build a Huffman tree

Data Structures for Huffman Coding

In the BitLab lab there is a class called **Node** we can use to build Huffman trees

When creating a **Node** object, we give the letter and the letter's frequency, as in this example:

```
from PythonLabs.BitLab import Node
leaf = Node('M', 0.032)
```

The above **Node** object creates a leaf node

The Huffman coding algorithm will take a set of such nodes, one per letter, and insert them into a priority queue

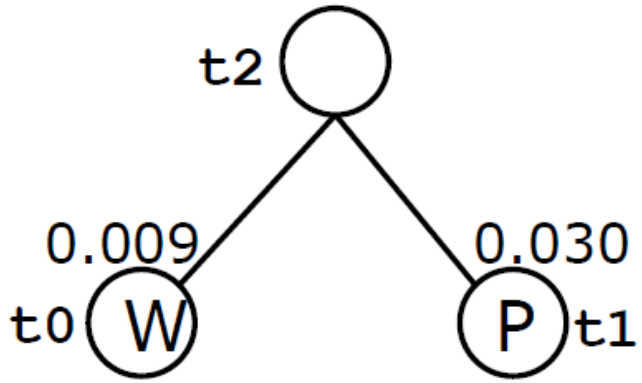
The priority queue will put the node with ***lowest*** frequency at the front of the list

- In other words, a letter's frequency will serve as its "priority", with high-frequency letters having the lowest priority

Data Structures for Huffman Coding

If we want to create an interior node, which has one or two children, we have to “tell” the **Node** object which nodes are its children, as in this example:

```
t0 = Node('W', 0.009)
t1 = Node('P', 0.030)
t2 = Node(t0, t1)
```



Huffman Coding: The Algorithm

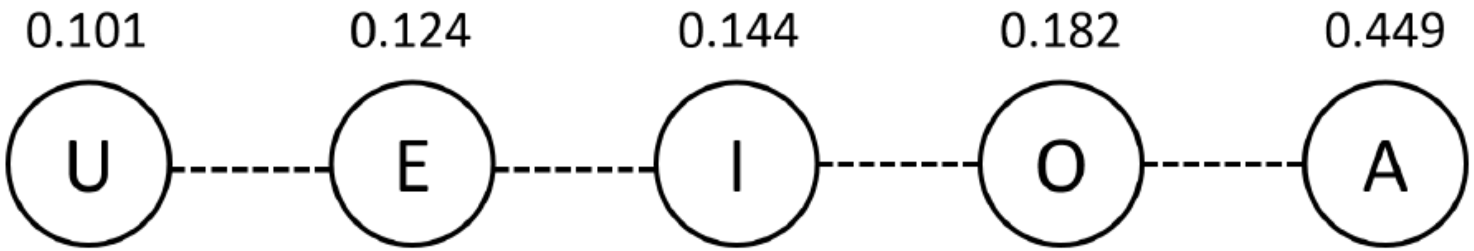
1. Make leaf nodes for every symbol in the alphabet
2. Put these nodes into a priority queue
3. Remove the first two nodes from the queue
4. Create a new interior node using these two nodes
5. Insert the new node back into the queue.
   - If there are still two more nodes in the queue, go to step 3
   - Otherwise, stop

Let’s see how this would work if we consider only the vowels (to make the example simpler)

Huffman Coding: Example #1

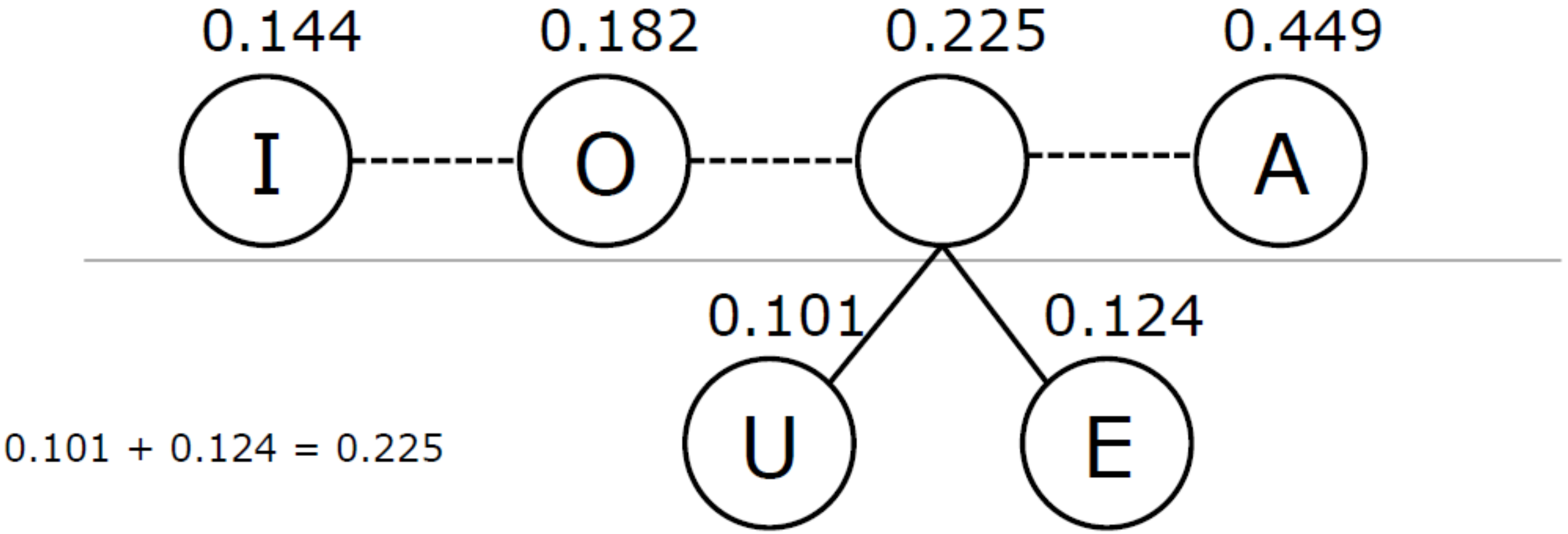
Below is the priority queue that would be created, with the front of the queue on the left:

We see that U and E are the two front nodes



So, we remove them from the queue, create a new interior node, and insert the new node into the queue, as we'll see on the next slide

Huffman Coding: Example #1

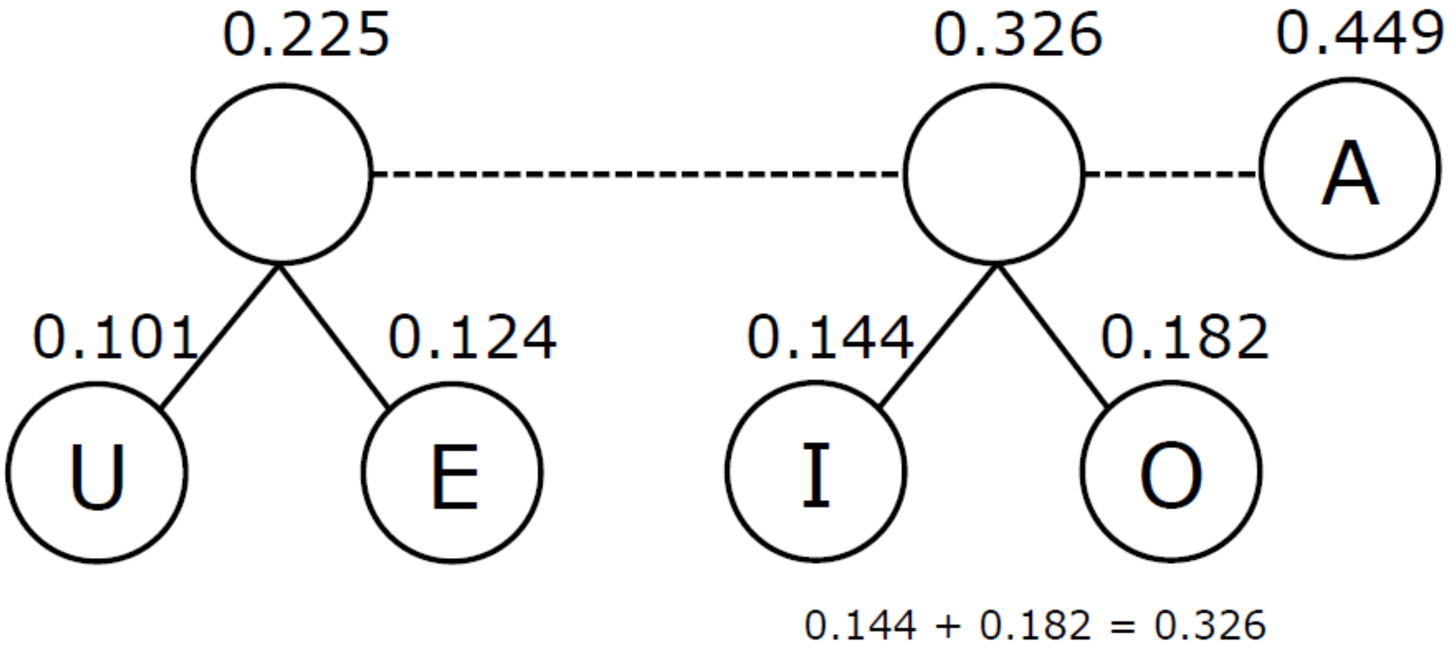


Above the horizontal line is the content of the priority queue

- Note how the queue has one fewer entry in it now

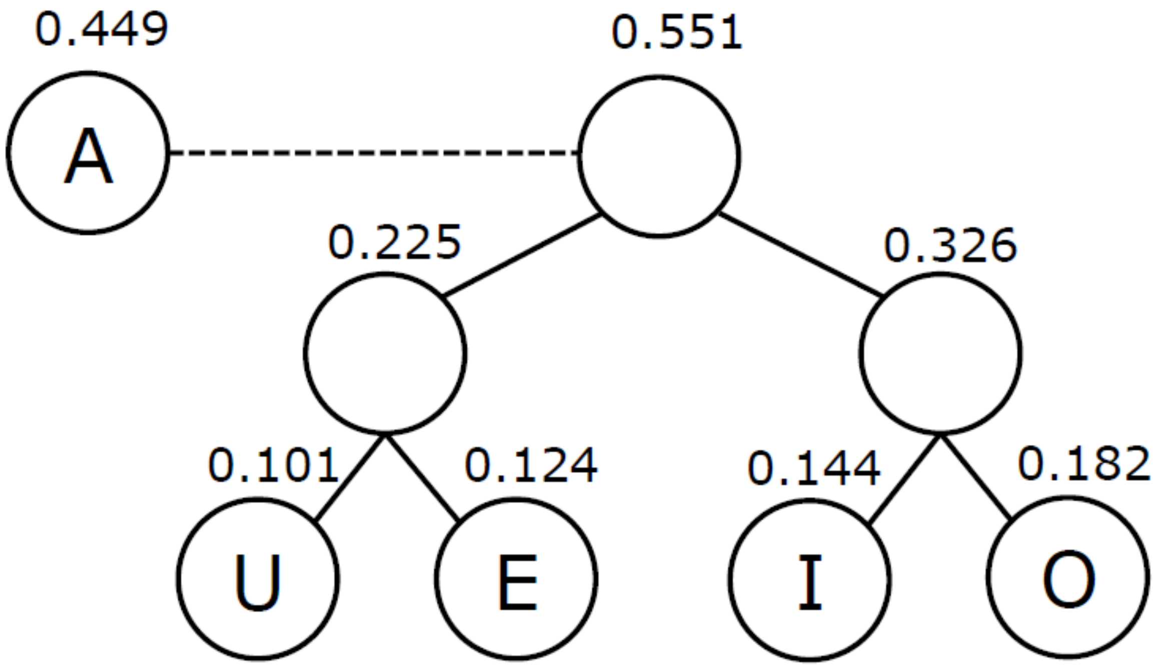
Next we'll remove the nodes for I and O, create a new node with these two nodes as children and add the new node back into the queue

Huffman Coding: Example #1



Next we'll remove the nodes with the weights 0.225 and 0.326, and combine them into a new interior node

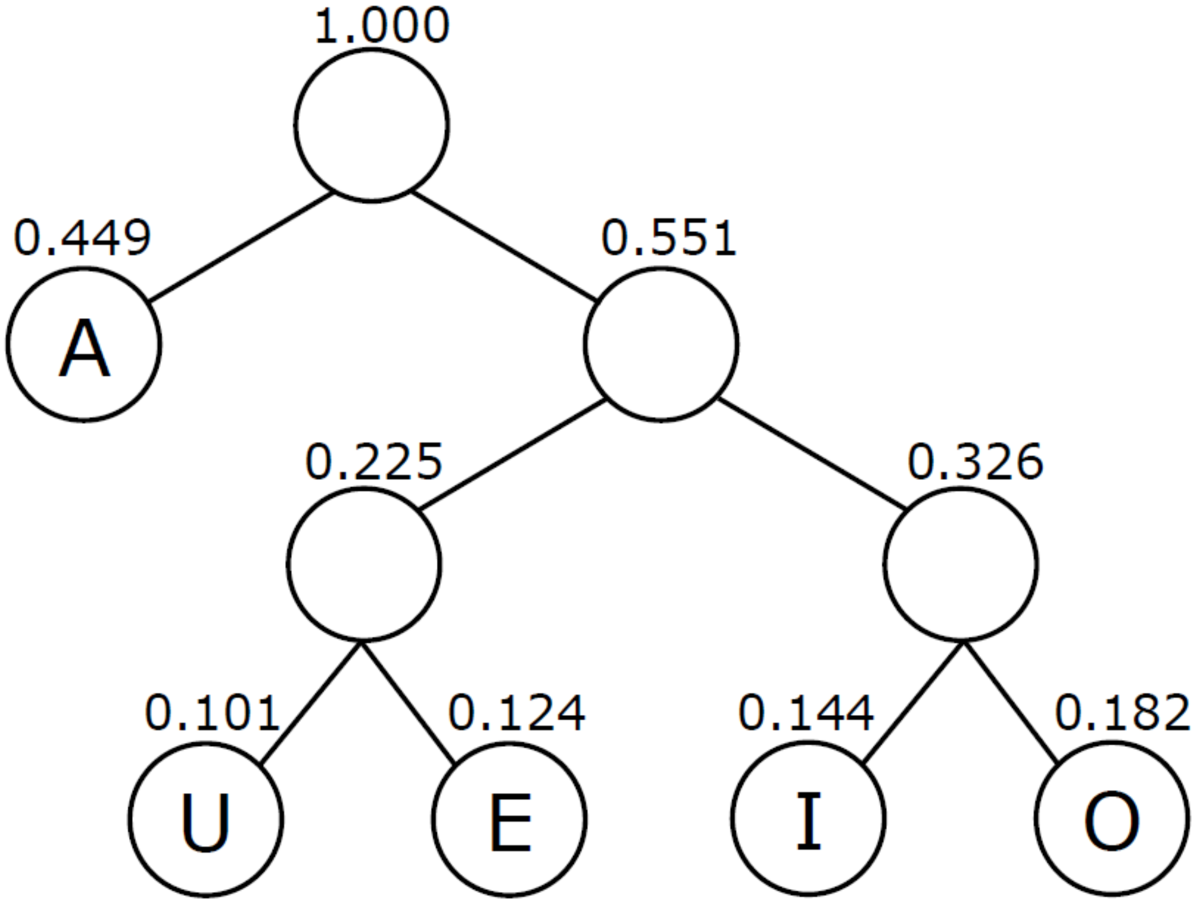
Huffman Coding: Example #1



Finally, we have only two nodes left, so we remove them both, and combine them into a new interior node

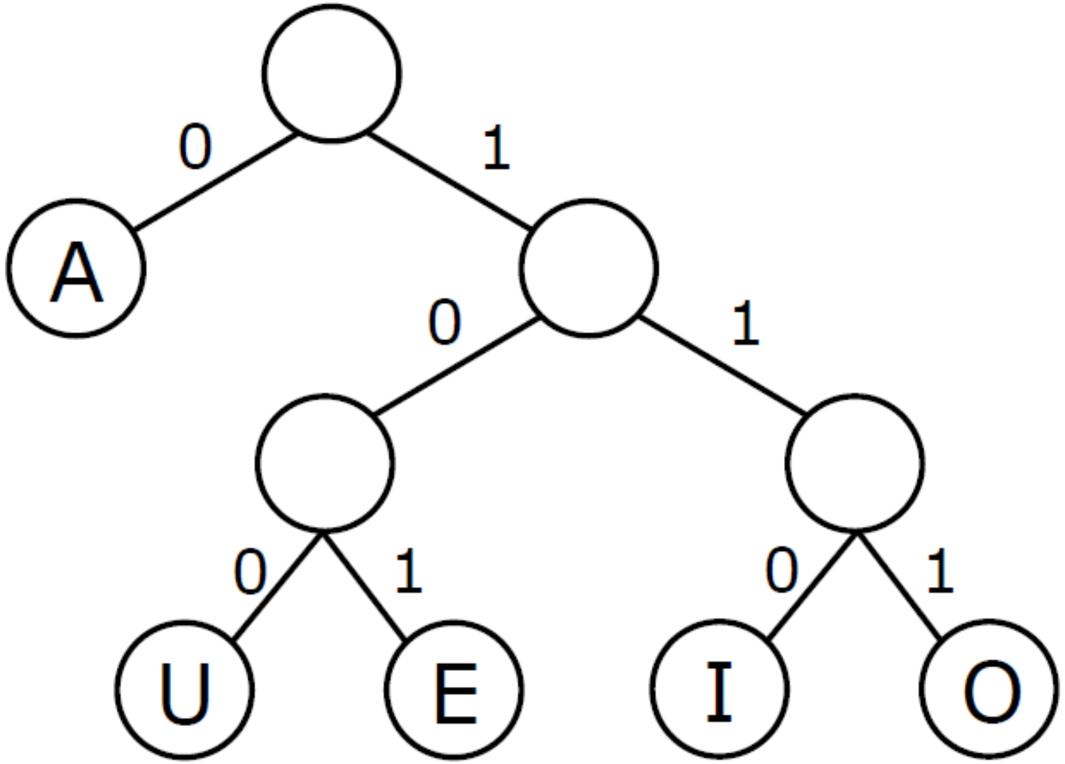
- This last node we create becomes the root of the binary tree

Huffman Coding: Example #1



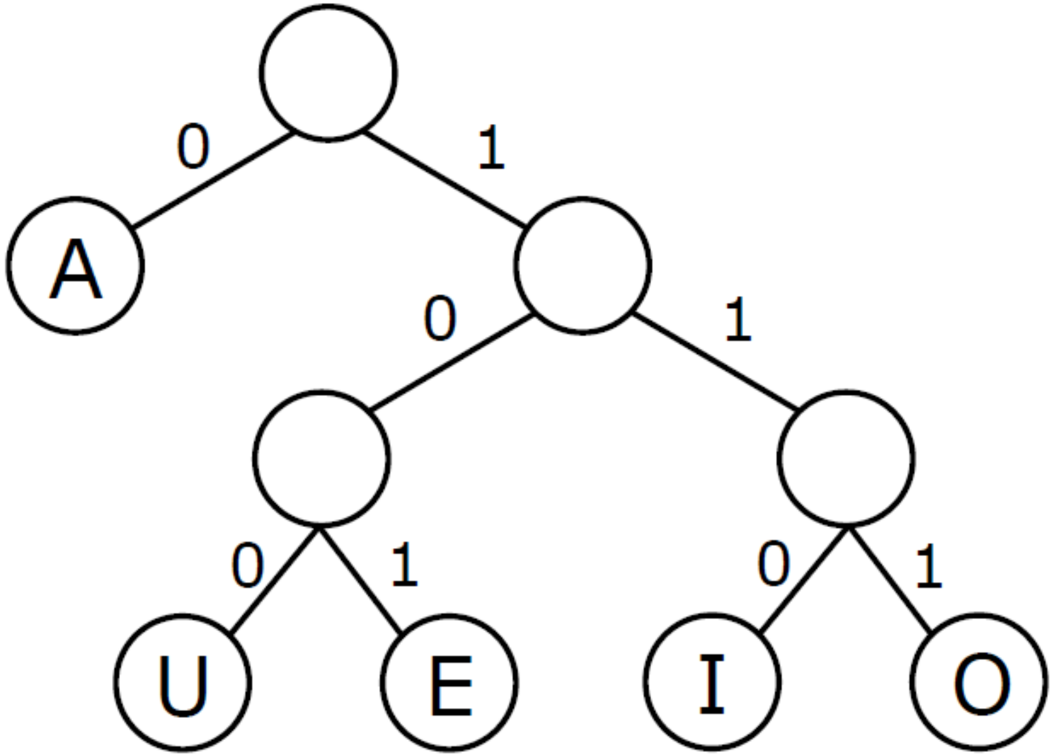
Huffman Coding: Example #1

With the tree completed, we now attach 0's and 1's to the edges connected to the **left child** and **right child** of each node, respectively



Huffman Coding: Example #1

Starting at the root, we trace the **path** from the root to each node to generate the codes for each letter:



Letter	Huffman Code
A	0
E	101
I	110
O	111
U	100

The build_tree() Function

We can now implement a function **build_tree()** that will build a Huffman tree from the list of frequencies

The function **read_frequencies()** from the BitLab module will load the frequencies stored from a file into a dictionary

The **build_tree()** function then adds the frequencies into **Node** objects, which are in turn added into the priority queue

Finally, a while loop assembles the Huffman tree by removing items two at a time from the priority queue and re-inserts the resulting “merged” pairs back into the queue

The build_tree() Function

```
from PythonLabs.BitLab import Node, read_frequencies, init_queue
def build_tree(filename):
    pq = init_queue(read_frequencies(filename))
    while len(pq) > 1:
        n1 = pq.pop()           # remove 1st element
        n2 = pq.pop()           # remove 2nd element
        pq.insert(Node(n1,n2))
    return pq[0]
```

See huffman.py

Huffman Coding: Example #1

Let’s try the function with the vowel frequencies:

```
vowel_tree = build_tree('hvfreq.txt')
print(vowel_tree)
# hvfreq.txt is available with the chapter 's programs
```

hvfreq.txt

A	0.449
E	0.124
I	0.144
O	0.182
U	0.101

Output:

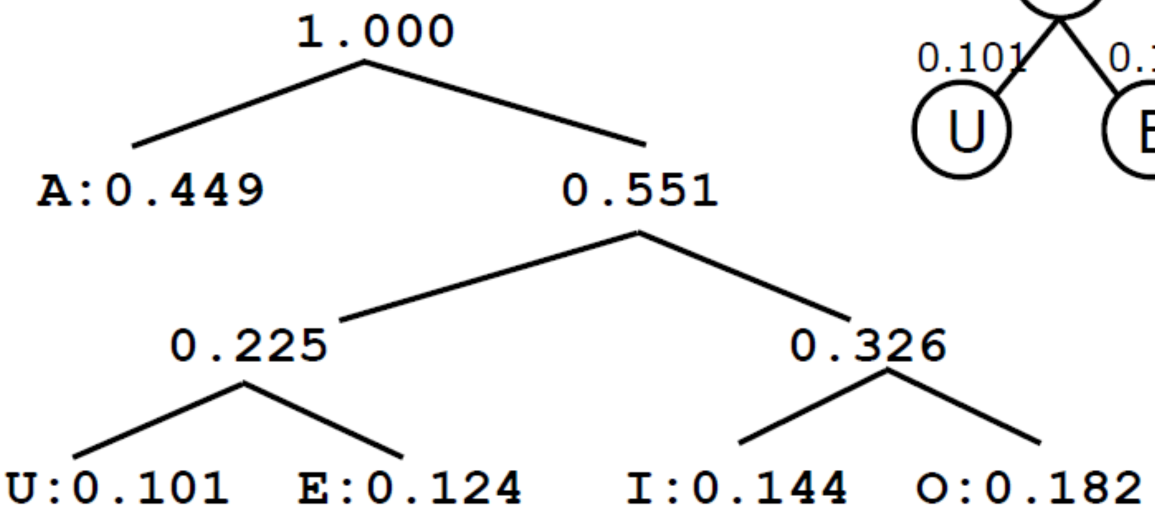
(1.000 (A: 0.449) (0.551 (0.225
(U: 0.101) (E: 0.124)) (0.326 (I: 0.144)
(O: 0.182))))

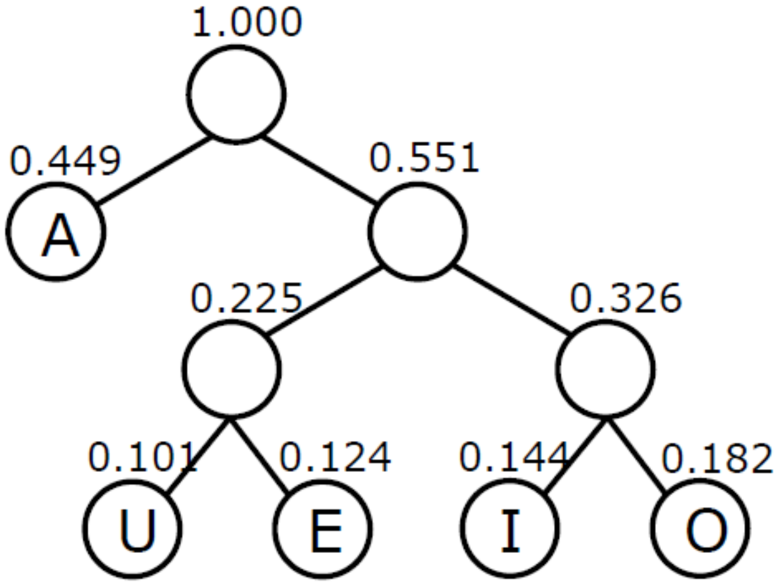
Although it may not seem like it, this is actually our tree

- Let’s reformat it a little (see next slide)

Huffman Coding: Example #1

Program output reformatted. Look familiar?





Huffman Coding: Example #1

Finally, the *recursive* function **assign_codes()** from BitLab assembled the Huffman codes from the Huffman tree:

```
from PythonLabs.BitLab import assign_codes
codes = assign_codes(vowel_tree)
print(codes)
```

Output:

```
{'A': 0, 'E': 101, 'I': 110, 'O': 111, 'U': 100}
```

Huffman Coding: Example #2

The file hafreq.txt contains the frequencies for all letters in the Hawaiian alphabet

Let’s build the Huffman tree from the frequencies:

```
alphabet_tree = build_tree('hafreq.txt')
```

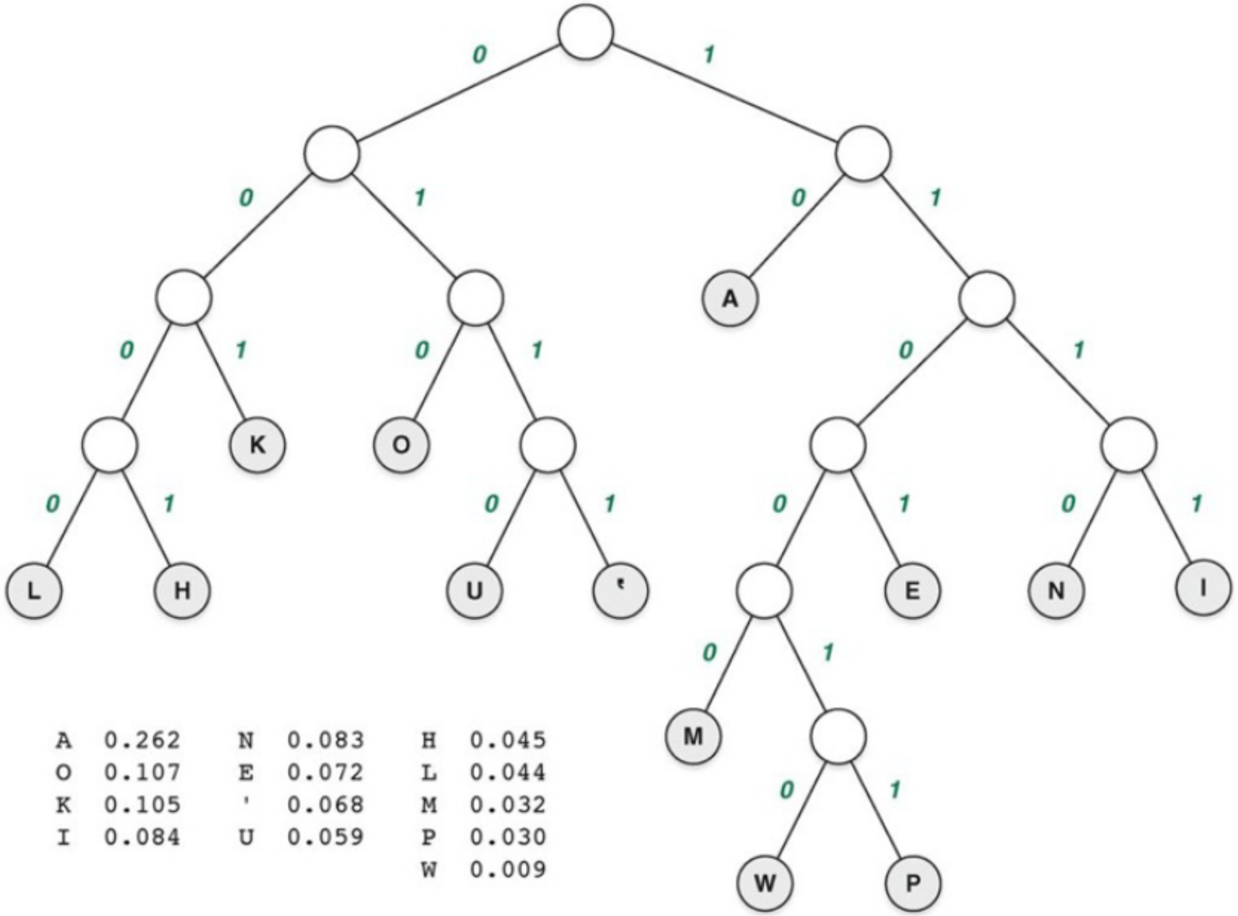
Then assign the codes:

```
codes = assign_codes(alphabet_tree)
```

Result: **{**

"'": 0111, 'A': 10, 'E': 1101,'H': 0001, 'I': 1111,
'K': 001, 'L': 0000, 'M': 11000, 'N': 1110, 'O': 010,
'P': 110011, 'U': 0110, 'W': 110010 }

Huffman Coding: Example #2



A	0.262	N	0.083	H	0.045
O	0.107	E	0.072	L	0.044
K	0.105	'	0.068	M	0.032
I	0.084	U	0.059	P	0.030
				W	0.009

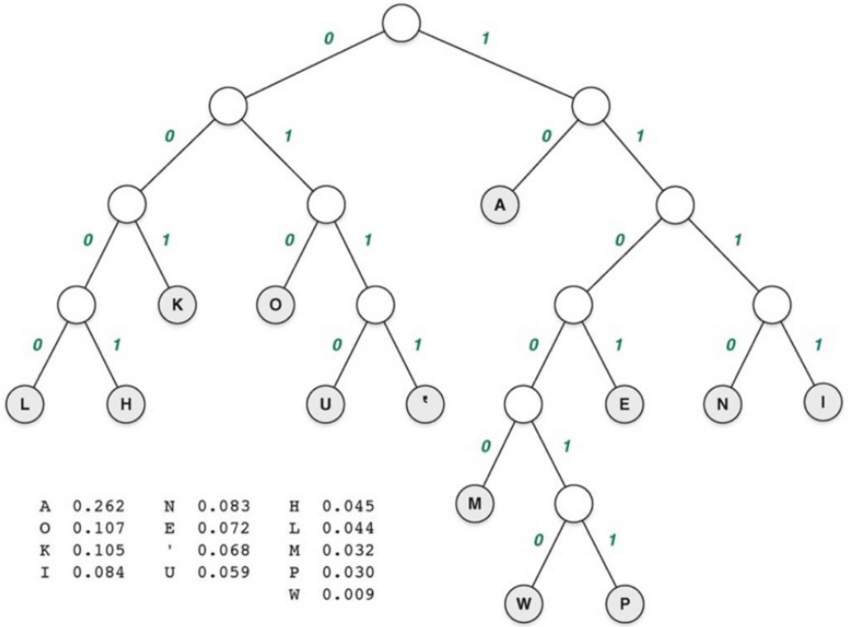
Huffman Coding: Example #2

What we find is that the most-frequently appearing letters have short codes, while the less-frequently appearing letters have longer code

Also note: no code is the prefix of another code

For example, the code for A is 10. No other code begins with 10.

- This fact is important when we want to decode a message
- Let's see now how we decode a message (next slide)



Huffman Coding: Example #2

Suppose we have the message 110001001101111

We scan the digits from left to right

- The first five digits, 11000, form the code for “M”
- The next two digits, 10, form the code for “A”
- The next four digits, 0110, form the code for “U”
- Finally, the last four digits, 1111, form the code for “I”

So, the original encoded word was “MAUI”

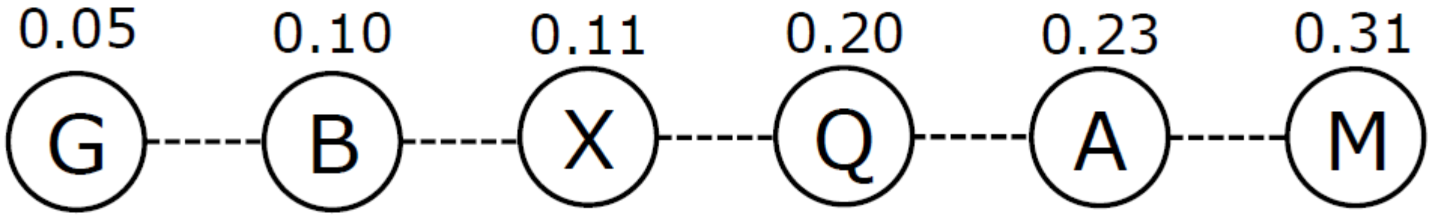
There is no other way to decode that string of bits to generate a different word

Encoding:
"'": 0111, 'A': 10, 'E': 1101, 'H': 0001, 'I': 1111, 'K': 001, 'L': 0000, 'M': 11000, 'N': 1110, 'O': 010, 'P': 110011, 'U': 0110, 'W': 110010

Huffman Coding: Example #3

Given the following letter frequencies, let's compute the Huffman coding for the letters

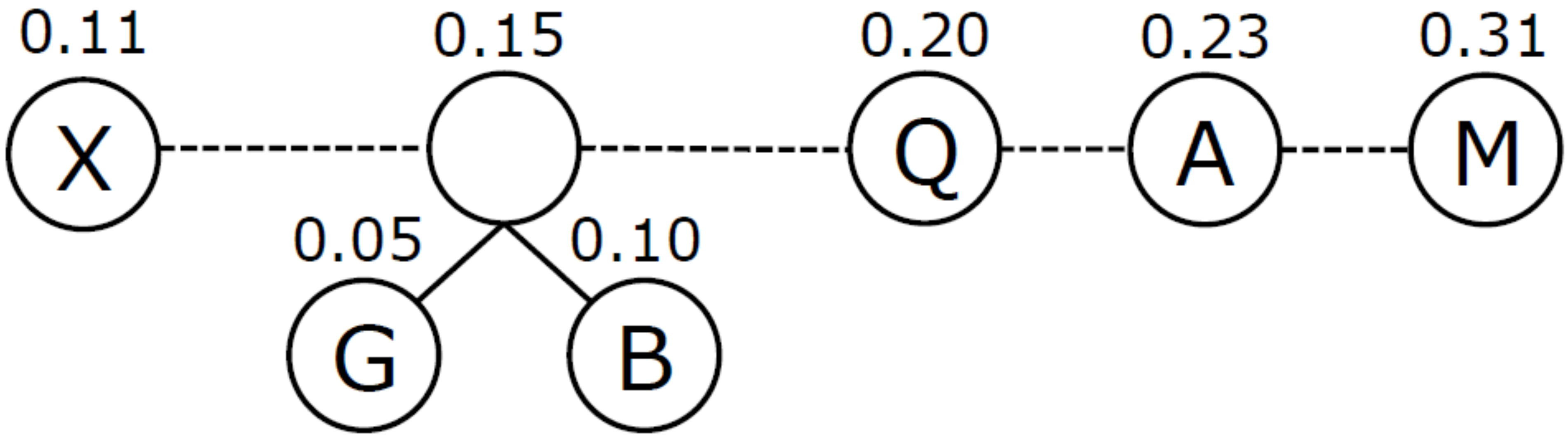
We begin by inserting the letters into a priority queue:



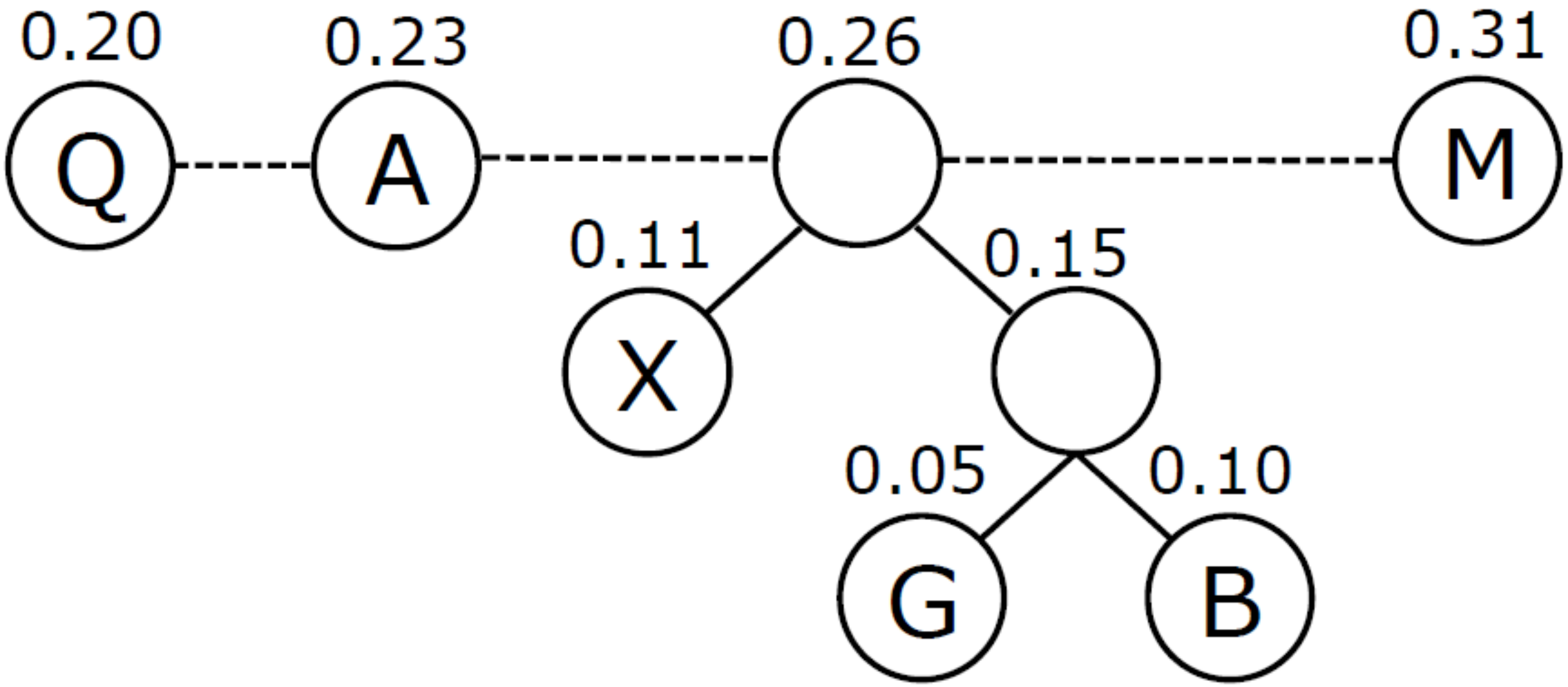
Letter	Frequency
A	0.23
B	0.10
G	0.05
M	0.31
Q	0.20
X	0.11

Now merge the first two elements in the queue until the tree is assembled (see few next slides)

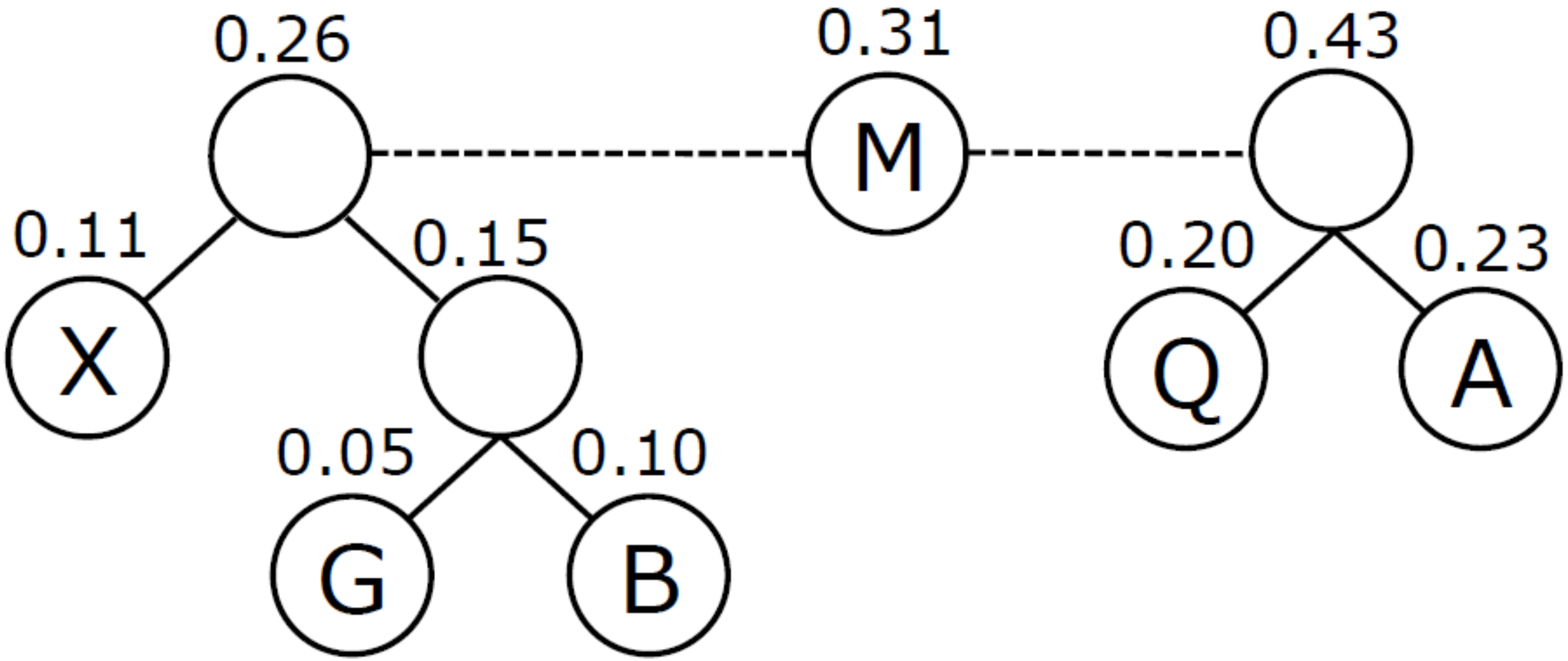
Huffman Coding: Example #3



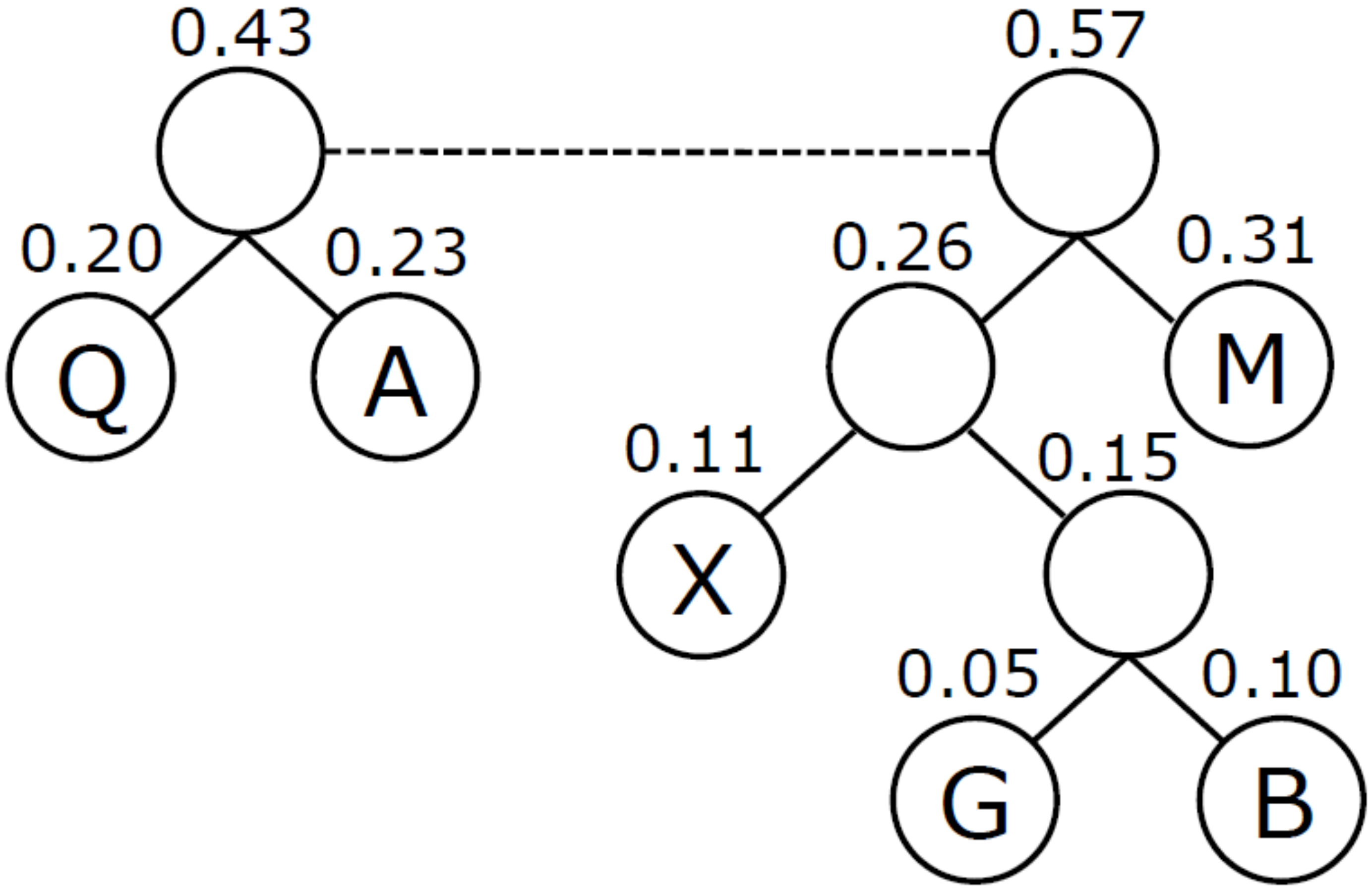
Huffman Coding: Example #3



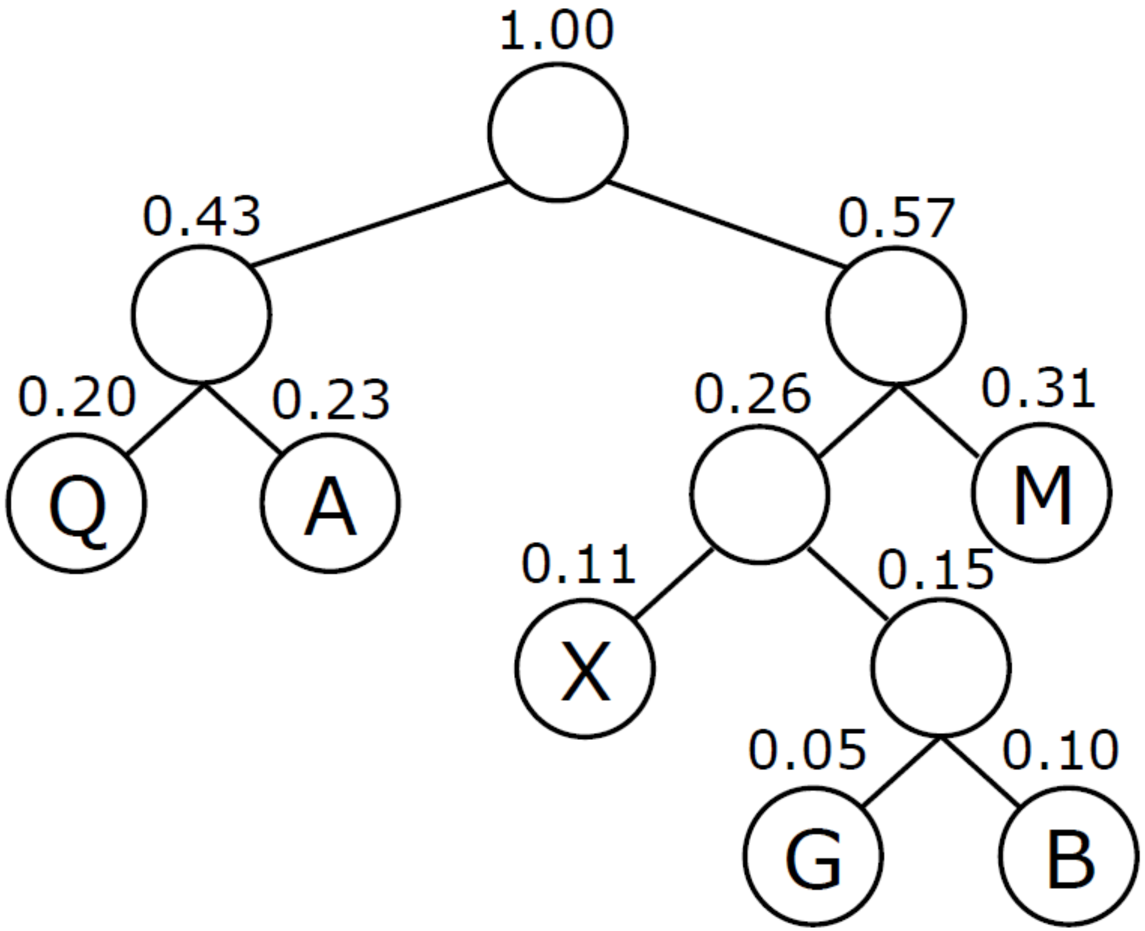
Huffman Coding: Example #3



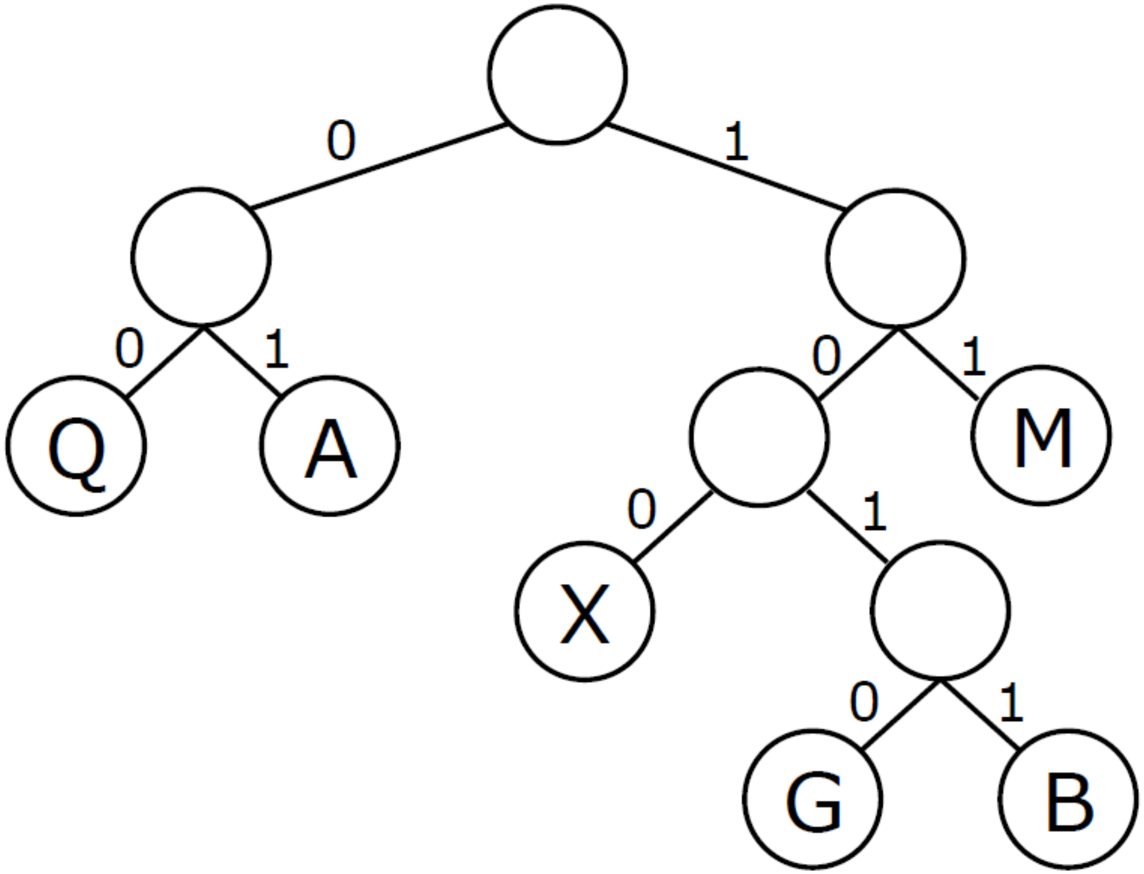
Huffman Coding: Example #3



Huffman Coding: Example #3



Huffman Coding: Example #3



Letter	Code
A	01
B	1011
G	1010
M	11
Q	00
X	100

encode()

With the dictionary for a Huffman coding assembled, it becomes very easy to encode strings:

```
huffman_codes = {
    "'": '0111', 'A': '10', 'E': '1101', 'H': '0001', 'I': '1111',
    'K': '001', 'L': '0000', 'M': '11000', 'N': '1110', 'O': '010',
    'P': '110011', 'U': '0110', 'W': '110010' }

def encode(word, encodings):
    result = ''
    for letter in word:
        result += encodings[letter]
    return result

encode('MAUI', huffman_codes)          # sample call
```

decode()

Decoding strings is a little trickier because the dictionary's key/value pairs are reversed from what we need

- The dictionary maps letters to codes, which is suitable for encoding
- For decoding we need to map codes to letters

Similar to list comprehensions, a **dictionary comprehension** lets you create a new dictionary from an existing one

Here's the code we need. It maps a value from the **huffman_codes** dictionary back to its key:

```
reversed_codes = { huffman_codes[key]: key for key in huffman_codes.keys() }
```

Would this work if values are not unique in **huffman_codes?**

decode()

We can now write the **decode()** function, where a “reversed” dictionary is given as **decodings**:

```
def decode(encoded, decodings):
    result = ''
    while len(encoded) > 0:
        for i in range(1, len(encoded) + 1):
            if encoded[:i] in decodings.keys():
                result += decodings[encoded[:i]]
                encoded = encoded[i:]
                break
    return result

decode('110001001101111', reversed_codes)    # sample call
```

See huffman.py for this code and examples

Questions?