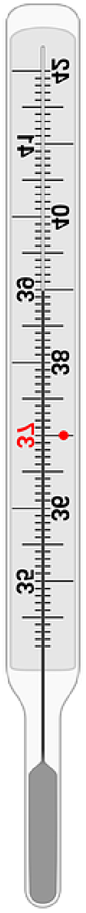
CSE 101: Introduction to Computational Thinking

Unit 8: Data Representation and Compression

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

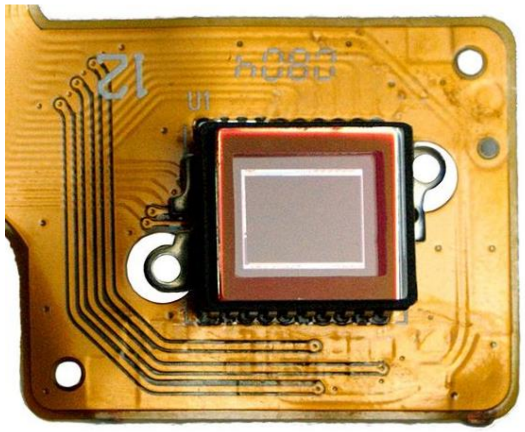
Analog vs. Digital



vs.

vs.

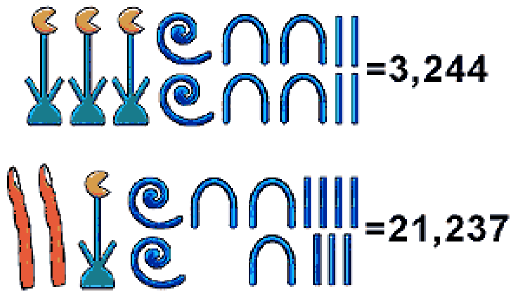
vs.



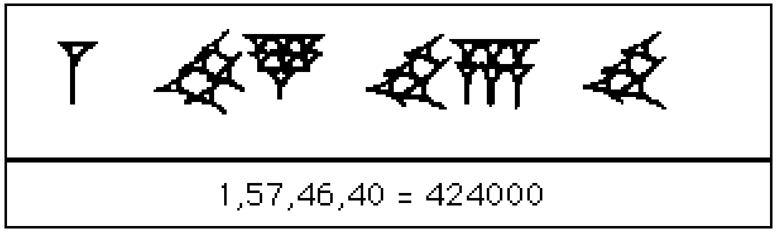
Representing Numbers

- Human beings have contrived a wide variety of ways to represent the digits that make up numbers

Egyptian hieroglyphs:

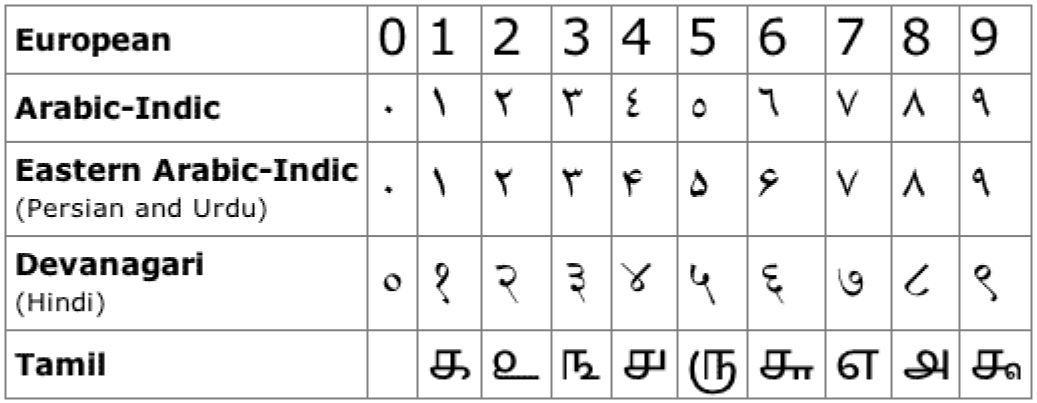


Babylonian numerals:



Roman numerals:
MMXVII = 2017

Many others!



European	0	1	2	3	4	5	6	7	8	9
Arabic-Indic	٠	١	٢	٣	٤	٥	٦	٧	٨	٩
Eastern Arabic-Indic (Persian and Urdu)	۰	۱	۲	۳	۴	۵	۶	۷	۸	۹
Devanagari (Hindi)	०	१	२	३	४	५	६	७	८	९
Tamil		௧	௨	௩	௪	௫	௬	௭	௮	௯

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, 1s, $^1/_{10}$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 cells
  - 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 concern ourselves about *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, 2^{-1}, 2^{-2}, 2^{-3}, …

  …, 8s, 4s, 2s, 1s, $^1/_2$s, $^1/_4$s, $^1/_8$s, …
- The number 1011.011_2 written in decimal is: $(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = 8 + 0 + 2 + 1 + 0 + {}^1/_4 + {}^1/_8 = 11.375_{10}$
- Important observation: 1011.011_2 and 11.375_{10} are two different representations of the same quantity
- All data in a modern machine is stored using binary numbers

Binary → Decimal Conversion

- To convert a binary number to decimal, add together the place-values where a 1 appears in the number
- Example: 101001_2
  - The place-values with these digits:

$$\begin{array}{cccccc} \underline{\;1\;} & \underline{\;0\;} & \underline{\;1\;} & \underline{\;0\;} & \underline{\;0\;} & \underline{\;1\;} \\ 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

- Therefore, $101101_2 = 2^5 + 2^3 + 2^0 = 32 + 8 + 1 = 41_{10}$

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 most easily understood by seeing some examples

Decimal → Binary Example #1

- Convert 123_{10} to binary

  123 / 2 = 61 rem. 1

  61 / 2 = 30 rem. 1

  30 / 2 = 15 rem. 0

  15 / 2 = 7 rem. 1

  7 / 2 = 3 rem. 1

  3 / 2 = 1 rem. 1

  1 / 2 = 0 rem. 1

  Answer: 1111011_2

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

Decimal → Binary Example #2

- Convert 1528_{10} to binary

1528 / 2 = 764 rem. 0

764 / 2 = 382 rem. 0

382 / 2 = 191 rem. 0

191 / 2 = 95 rem. 1

95 / 2 = 47 rem. 1

47 / 2 = 23 rem. 1

23 / 2 = 11 rem. 1

11 / 2 = 5 rem. 1

5 / 2 = 2 rem. 1

2 / 2 = 1 rem. 0

1 / 2 = 0 rem. 1

Answer: 10111111000_2

The Function `dec2bin()`

- Let's see a function **`dec2bin()`** that returns a string of 0s and 1s giving the binary representation of an integer

See unit08/data_rep.py

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

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

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	23

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	23
binary	""

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	23
binary	""

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	23
binary	""
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	23
binary	"1"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
`decimal`	`11`
`binary`	`"1"`
`remainder`	`1`

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	11
binary	"1"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	11
binary	"1"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	11
binary	"11"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
`decimal`	`5`
`binary`	`"11"`
`remainder`	`1`

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	5
binary	"11"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	5
binary	"11"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	5
binary	"111"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
`decimal`	2
`binary`	"111"
`remainder`	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	2
binary	"111"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	2
binary	"111"
remainder	0

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	2
binary	"0111"
remainder	0

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	1
binary	"0111"
remainder	0

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	1
binary	"0111"
remainder	0

Trace Execution: `dec2bin(23)`

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

Variable	Value
`decimal`	`1`
`binary`	`"0111"`
`remainder`	`1`

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	1
binary	"10111"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	0
binary	"10111"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	0
binary	"10111"
remainder	1

Trace Execution: `dec2bin(23)`

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

Variable	Value
decimal	0
binary	"10111"
remainder	1

Encoding Data

- 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 Unit 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: $2^3 = 8$ items. With 4 bits: $2^4 = 16$ items.
- With 5 bits: $2^5 = 32$ items. And so on…

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
- Examples: capital letter “A” is 65; lowercase letter “a” is 97
- A newer scheme called **Unicode** includes codes for 135 alphabets
  - Modern languages (Greek, Cyrillic, Arabic, Hebrew, Chinese, Japanese, Korean, ...) and ancient languages (hieroglyphics, runes, ...)
  - Also include technical symbols, emoji, 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' is 'I ♥ cats'`**
- See unit08/data_rep.py for more examples

Hexadecimal Numbers

- In some software development 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, and the letters A through F for ten through fifteen
- 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.

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

- Hexadecimal is used widely in web design for giving colors
  - You can use it with Python too, as we just saw
- 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 the heart symbol above, we used **`9829`** (base 10) for **`chr()`**, but **`2665`** (base 16) for the escape sequence
- The related **`ord()`** function returns the Unicode value of a character:
  - **`ord('A')`** returns **`65`** and **`ord('x')`** returns **`120`**
- See unit08/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)

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

Binary ↔ Hexadecimal

- 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:
  $\texttt{D2B5}_{16} = \texttt{1101001010110101}_{2}$
- 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: $\texttt{010001111110}_{2} = \texttt{47E}_{16}$

Decimal Fractions → Binary

- We’ve seen how to convert base-10 integers to binary representation, but what about real numbers?
- Algorithm: generate the bits in *left-to-right order*, starting from the radix point:
  - Multiply the decimal value by 2. If the product is greater than or equal to 1, the next bit is 1. Otherwise, the next bit is 0.
  - Drop the integer part to get a value less than 1.
  - Continue until 0 is reached (a terminating expansion) or a pattern of digits repeats (a non-terminating expansion)
- The resulting representation is called **fixed-point format**

Decimal Frac. → Binary Example

- Convert 0.4_{10} to binary
- $0.4 \times 2 = 0.8$ $0.8 < 1$, so write a 0 $\cong 0.0$
- $0.8 \times 2 = 1.6$ $1.6 \geq 1$, so write a 1 $\cong 0.01$
  - Drop the integer part
- $0.6 \times 2 = 1.2$ $1.2 \geq 1$, so write a 1 $\cong 0.011$
  - Drop the integer part
- $0.2 \times 2 = 0.4$ $0.4 < 1$, so write a 0 $\cong 0.0110$
- Since we arrived at a decimal fraction we have already seen, the pattern will repeat
- Final answer: $0.\overline{0110}_2$

Decimal Frac. → Binary Example

- Convert 13.85_{10} to binary
- $13_{10} = 1101_2$
- $0.85 \times 2 = 1.7$ $1.7 \geq 1$, so write a 1 $\cong 0.1$
  - Drop the integer part
- $0.7 \times 2 = 1.4$ $1.4 \geq 1$, so write a 1 $\cong 0.11$
- $0.4 \times 2 = 0.8$ $0.8 < 1$, so write a 0 $\cong 0.110$
- $0.8 \times 2 = 1.6$ $1.6 \geq 1$, so write a 1 $\cong 0.1101$
- $0.6 \times 2 = 1.2$ $1.2 \geq 1$, so write a 1 $\cong 0.11011$
- $0.2 \times 2 = 0.4$ $0.4 < 1$, so write a 0 $\cong 0.110110$
- We will get $\cong 0.1101100110\ldots = 0.11\overline{0110}$
- Final answer: $1101.11\overline{0110}_2$

What About Real Numbers?

- We did some base conversions involving real numbers but haven't seen yet how they can be represented in a computer
- A big disadvantage of the so-called **fixed-point format** or **fixed-precision encoding** of such numbers is that they have a very limited "dynamic range"
  - This format can't represent very large numbers (e.g., 2^{70}) or very small numbers (e.g., 2^{-17})
- These numbers are called "fixed point" because the decimal point (or **binary point**) is fixed, which limits accuracy
- On the other hand, they give *exact* answers (i.e., no rounding errors) as long as there is no overflow

Floating-Point Format

- Because fractions can have non-terminating representations and/or might require many digits to be represented exactly, usually real numbers can only be approximately represented in a computer
  - We have to tolerate a certain amount of **representational error**
- The industry standard way used to approximate real numbers is called **floating-point format**
  - **IEEE 754 floating-point standard**
- In this scheme the binary point is allowed to “float” (i.e., be repositioned) in order to give as accurate an approximation as possible

IEEE 754 Floating-Point Standard

- The IEEE 754 standard species floating-point representations of numbers and also arithmetic operations on these representations
- IEEE 754 is essentially a form of **scientific notation**, but written in binary: $\pm 2^{exponent} \times fraction$
- This format can be encoded using three fields: a **sign bit** (s), an **exponent** (e) and a **fraction** (f), sometimes called the **mantissa**
- IEEE 754 **single-precision format** requires 32 bits and provides about 7 decimal digits of accuracy

IEEE 754 Floating-Point Standard

1 bit	8 bits	23 bits
sign	exponent	fraction (mantissa)

- Sign bit: 0 (positive) or 1 (negative)
- Exponent: stored in **excess-127** for the 32-bit version
  - This means that instead of storing the exponent directly in the exponent field, we first add 127 to the exponent before storing it. More on this in a moment.
- Fraction: contains the digits to the right of the binary point
  - **Normalized**: the digit to the left of the point is always 1, and is not represented, giving us one bit of precision “for free”

IEEE 754 to Decimal

- Decimal value of a IEEE 754 floating point encoding is given by the formula: $(-1)^s \times 2^{e-bias} \times (1+f)$
  where:
  - s is the sign bit (0/1).
  - e is the decimal value of the exponent field
  - $bias$ is 127
  - f is the decimal value of the fraction field (regarded as a binary fraction)

Fixed-point to Floating-point Example

- Convert the fixed-point binary value 101011.1011_2 to floating-point notation
- $s = 0$
- $e = 5 + 127 = 132_{10} = 10000100_2$
- $f = 010111011$
- Answer: 0 10000100 01011101100000000000000

IEEE 754 to Decimal Example

- What decimal value has the following IEEE 754 encoding?
  10111110011000000000000000000000

  1   01111100   11000000000000000000000
- $s = 1$
- $e = (64 + 32 + 16 + 8 + 4) - 127 = -3$
- $f = 0.5 + 0.25 = 0.75$
- Answer: $-1.75 \times 2^{-3} = -0.21875_{10}$

Decimal to IEEE 754 Example

- Encode 13.4_{10} in 32-bit IEEE 754 floating-point format
- Positive number → $s = 0$
- From earlier in this Unit: $0.4_{10} = 0.\overline{0110}_2$
- 1101.0110011001100110011001100110…
- Normalize: move binary point 3 places to left
  - $e = 3$ Add 127: $3 + 127 = 130_{10} = 10000010$
- 1010110011001100110011001100110… (dropped leading 1)
- Take 23 bits: 1010 1100 1100 1100 1100 110
- Answer:

s	e	f
0	10000010	10101100110011001100110

IEEE 754 Special Values

- The smallest 000…0 and largest 111…1 exponents are reserved for the encoding of special values:
  - Zero (two encodings):
    - $s = 0$ or $1, e = 000 \dots 0, f = 000 \dots 0$
  - Infinity:
    - $+\infty: s = 0, e = 111 \dots 1, f = 000 \dots 0$
    - $-\infty: s = 1, e = 111 \dots 1, f = 000 \dots 0$
  - NaN (not a number):
    - $s = 0$ or $1, e = 111 \dots 1, f =$ non-zero
    - Can result from division by zero, $\sqrt{-1}$, $\log(-5)$, etc.

IEEE 754 Format Summary

Property	
Bits in Sign	1
Bits in Exponent	8
Bits in Fraction	23
Total Bits	32
Exponent Encoding	excess-127
Exponent Range	-126 to 127
Decimal Range	$\cong 10^{-38}$ to 10^{38}

Groups of Bits

- A **byte** is a collection of 8 bits
  - A 7-bit ASCII value fits in a 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 sizes 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
  - A = 01000001. There are two 1 bits, so attach a 0 as the parity bit (the total # of 1s remains two)
  - C = 01000011. There are three 1 bits, so attach a 1 as the parity bit (bringing the total # of 1s to four)
- Example: parity bit for a piece of DNA (using ASCII)
  - ATG = 01000001 01010100 01000111 + 1
  - The binary code for the entire string contains 9 1's, so we add one more 1 bit to make an even number of 1s

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'

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?
- Deciding whether to attach a 1 or a 0 to the end of a code is simple using a logic function called “exclusive or”, abbreviated as XOR
  - Normally, **`a or b`** is true if either **`a`** or **`b`** is true, or if both of **`a`** and **`b`** are true
  - The XOR of variables **`a`** and **`b`** is true if either one of them is true, but not if both are true
- XOR in Python is denoted using the caret, **`^`**

Computing Parity Bits

- 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
- In Python, bitwise-**`and`** is denoted by the ampersand, **`&`**, and bitwise-**`or`** is denoted by the vertical bar, or pipe, **`|`**

	AND	**OR**	**XOR**
a b	**a & b**	**a | b**	**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`**
    a. If a bit is 0, it won’t change **`p`**.
    b. 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 unit08/data_rep.py

Trace: `parity('10011')`

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

Variable	Value
`x`	`"10011"`

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	0

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	0
bit	"1"

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	1
bit	"1"

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	1
bit	"0"

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	1
bit	"0"

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	1
bit	"0"

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	1
bit	"0"

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	1
bit	"1"

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	0
bit	"1"

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	0
bit	"1"

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	1
bit	"1"

Trace: `parity('10011')`

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

Variable	Value
x	"10011"
p	1

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 maps 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

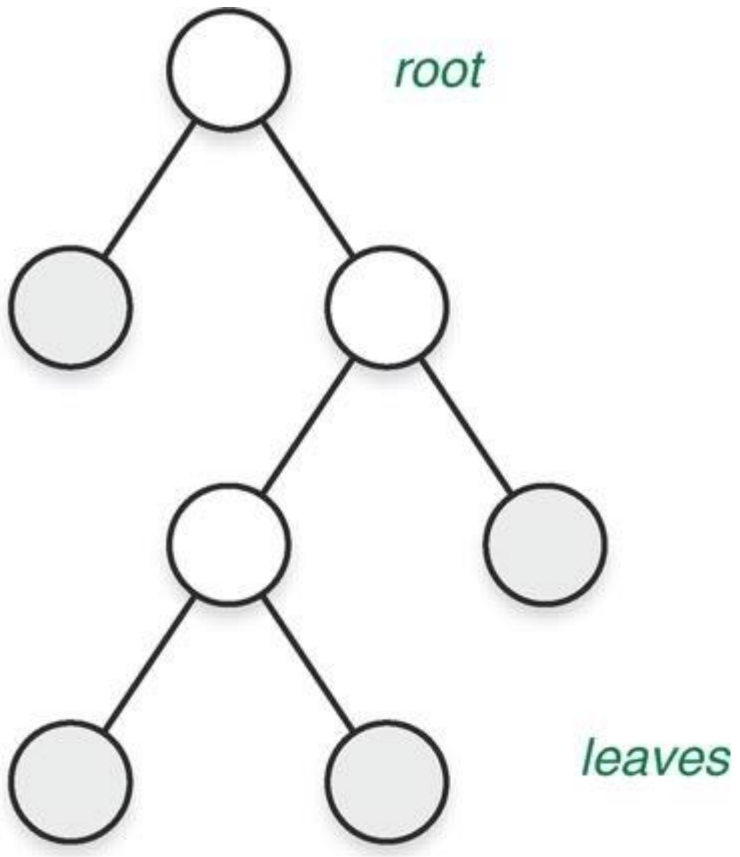
- Data compression algorithms reduce in the amount of space needed to store a piece of data
- A data compression technique can be:
  - **Lossless** (no information lost)
  - **Lossy** (information lost)
- There are many algorithms for compressing files (including photos, images and other types of data) 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 white in the figure on the next slide)
- Towards the bottom of a tree we find nodes with no children; such nodes are called **leaves** (shaded gray in the figure on the next slide)

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 heart of Huffman coding
- Inside of each node of a Huffman tree we store (i) a letter and (ii) 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 the five vowels A, E, I, O and U, and only the seven consonants H, K, L, M, N, P and W
- 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 exploit 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 Unit 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 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

Data Structures for Huffman Coding

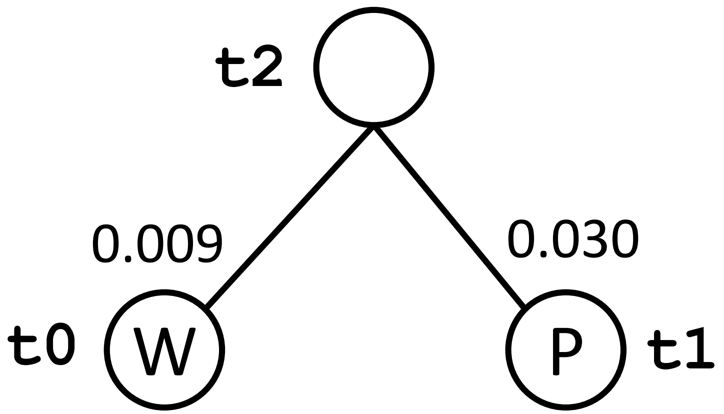
- We will use a priority queue to help us build a Huffman tree
- In the file Node.py there is a class called **`Node`** we can use to build Huffman trees
  - When creating a **`Node`** object, we give the letter and the letter's frequency, as in this example:

    ```
    from Node 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

Data Structures for Huffman Coding

- 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
- 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

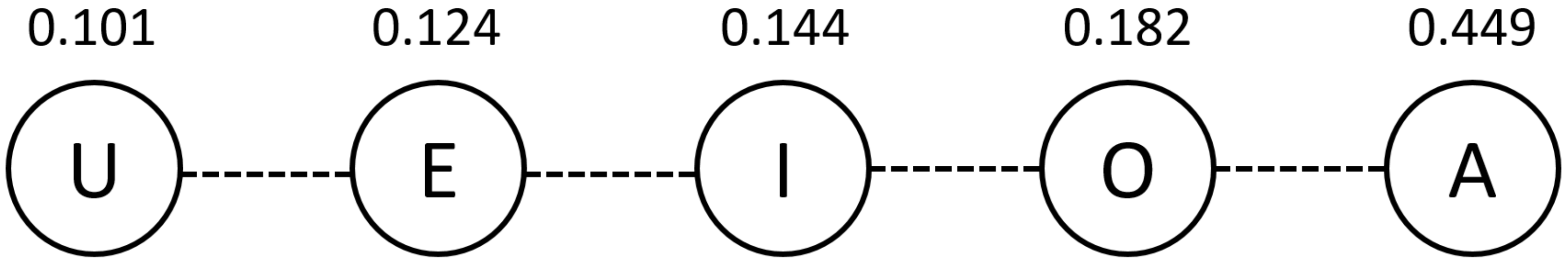
- Given the following letter frequencies, let's compute the Huffman coding for the letters

Letter	**Frequency**
A	0.449
E	0.124
I	0.144
O	0.182
U	0.101

- Now we need to insert these letters into a priority queue, as shown on the next slide

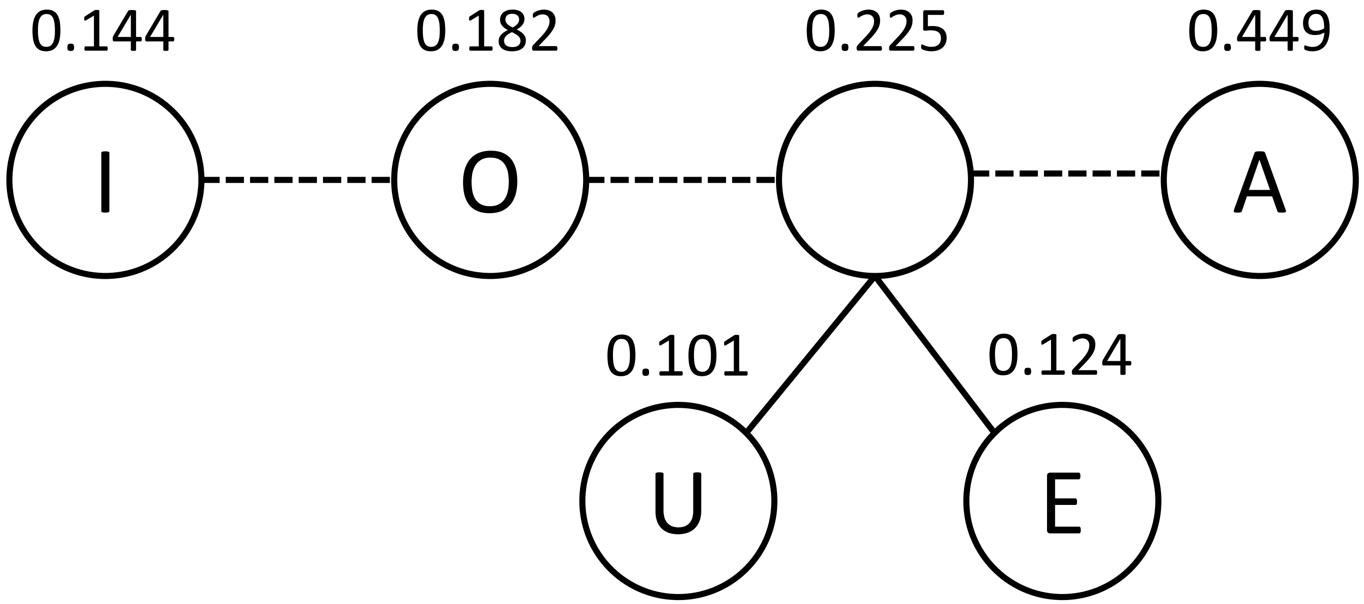
Huffman Coding: Example #1

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



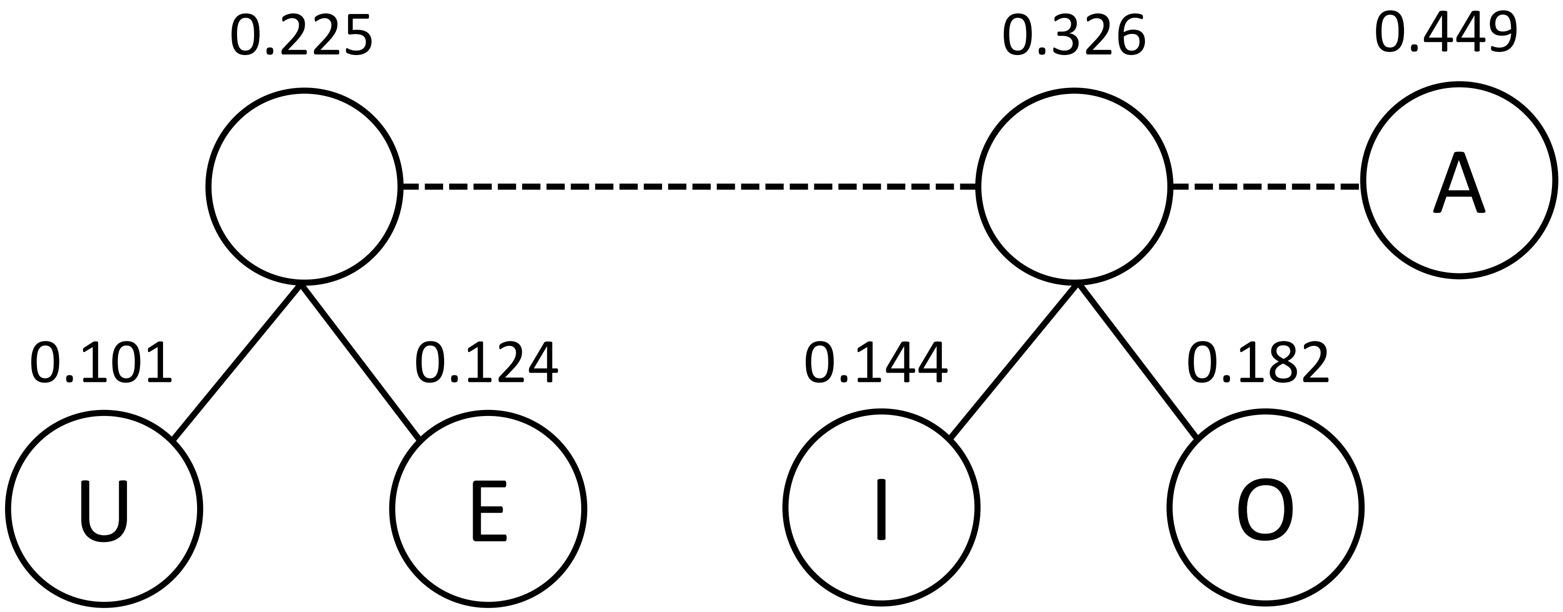
- We see that U and E are the two front nodes
- So, we remove them from the queue, create a new interior node, and insert the new node into the queue, as we'll see on the next slide

Huffman Coding: Example #1



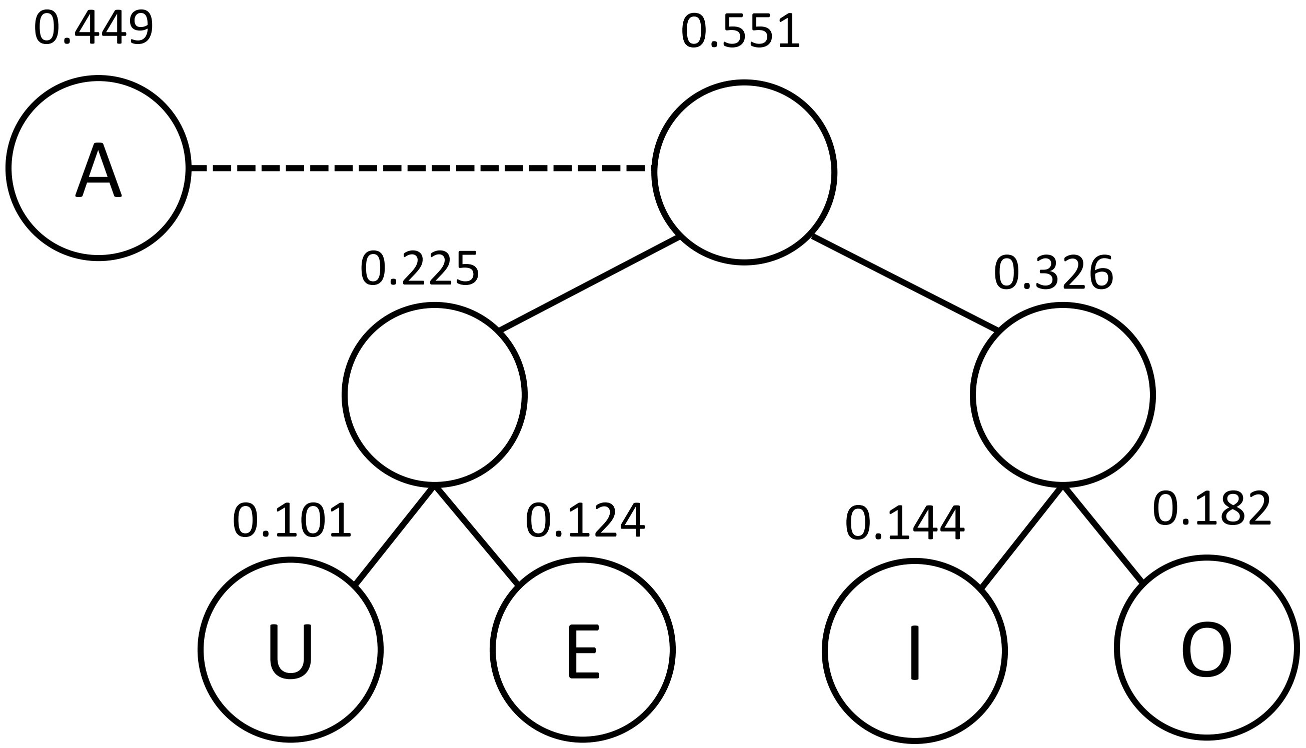
- Note how the queue has one fewer entry in it now
- Next we'll remove the nodes for I and O, create a new node with these two nodes as children and add the new node back into the queue

Huffman Coding: Example #1



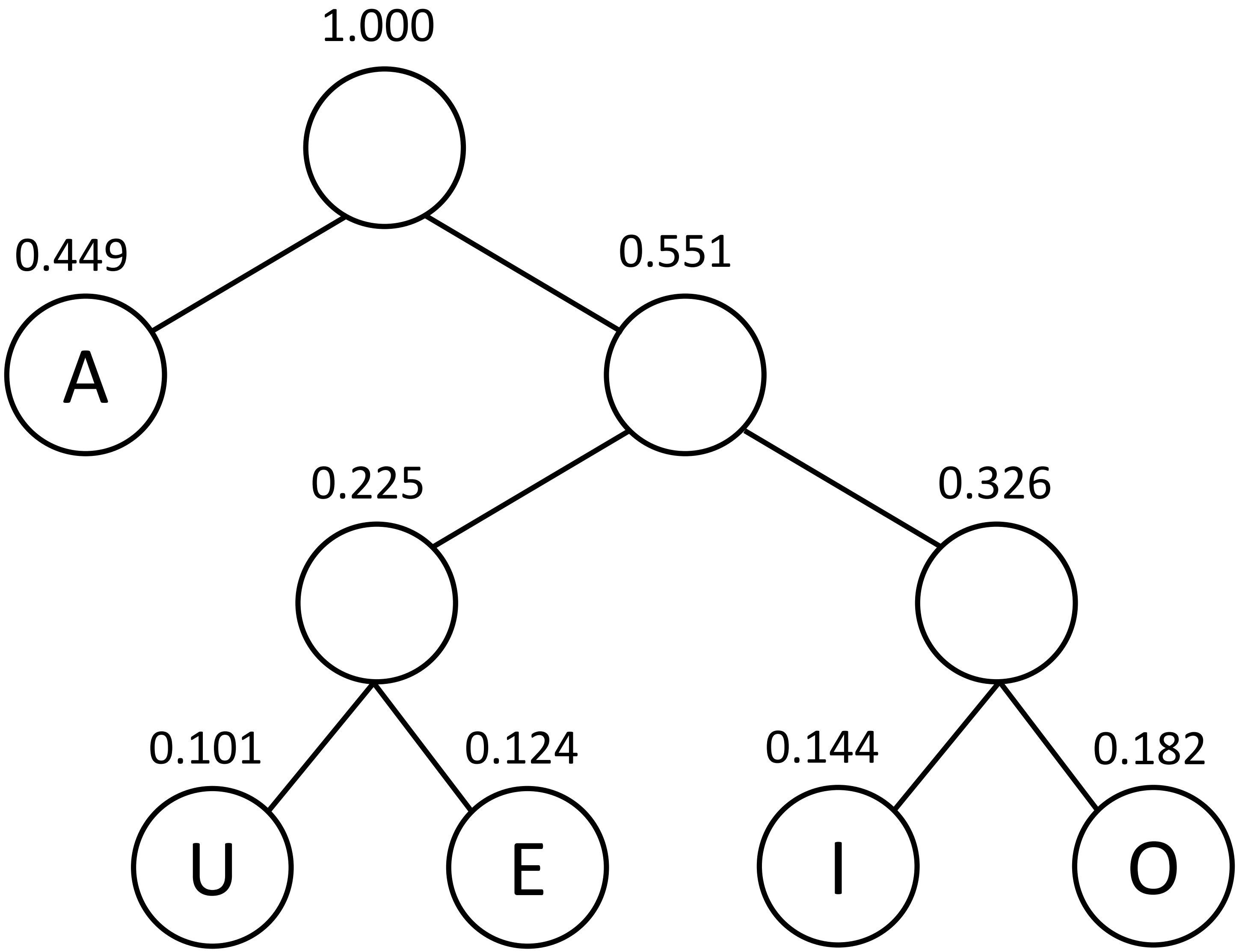
- Next we'll remove the nodes with the weights 0.225 and 0.326, and combine them into a new interior node

Huffman Coding: Example #1



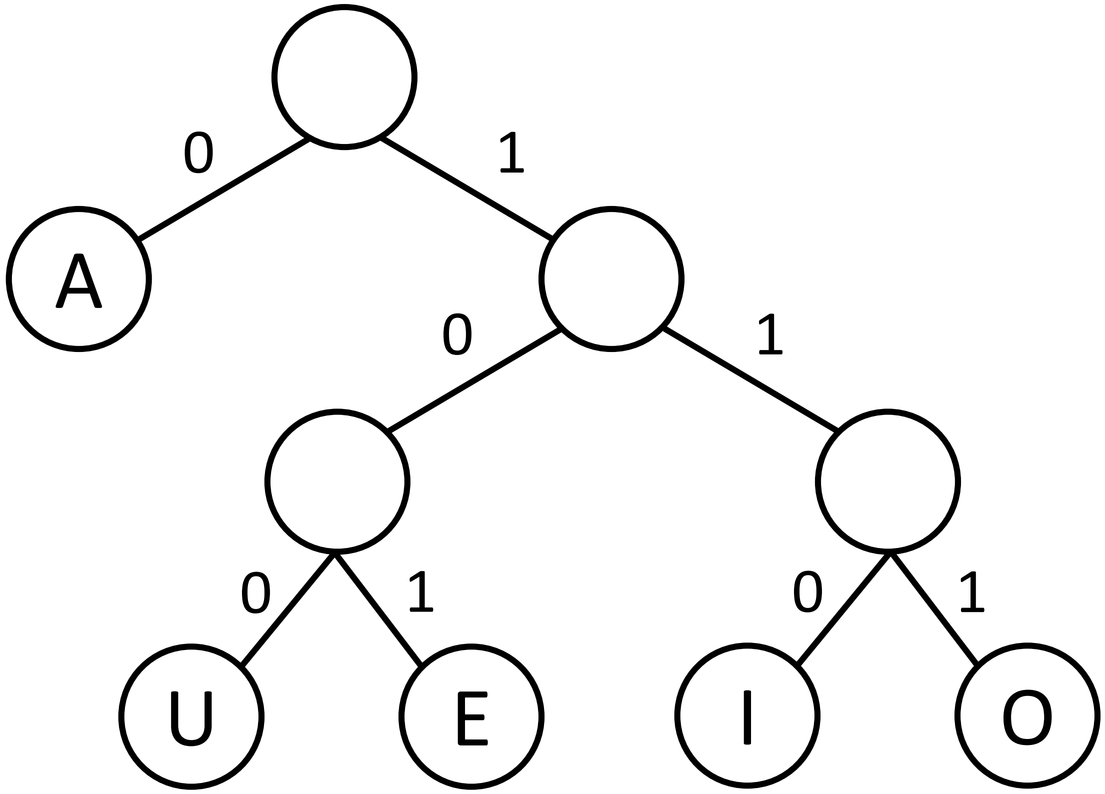
- Finally, we have only two nodes left, so we remove them both, and combine them into a new interior node
- This last node we create becomes the root of the binary tree

Huffman Coding: Example #1



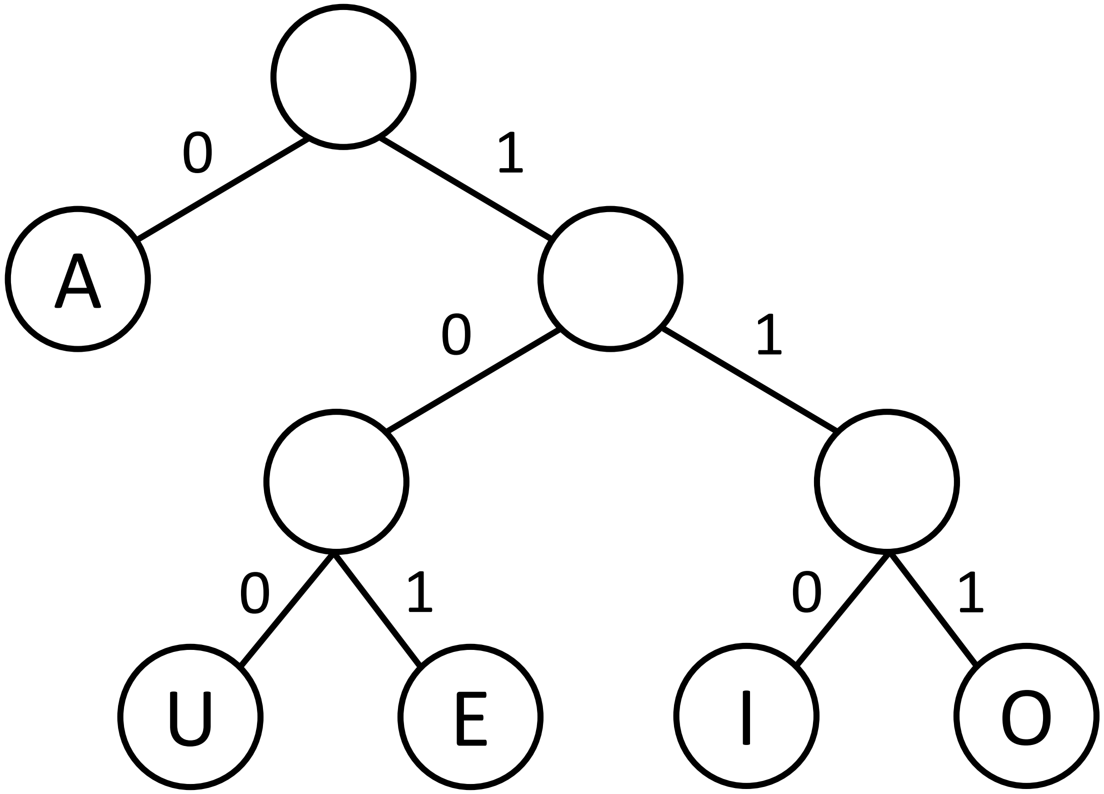
Huffman Coding: Example #1

- 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 huffman_utils.py will load the frequencies stored from a file into a dictionary
  - The dictionary maps a letter to its frequency
- Next, we call the **`init_queue()`** function, which takes this dictionary and creates a **`Node`** object that contains each letter and its frequency
  - Each such **`Node`** is then inserted into a new **`PriorityQueue`** that is returned at the end of the funcion

The `build_tree()` Function

- Then, 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()`** concludes by returning a reference to the root of the Huffman tree

The `build_tree()` Function

```
from Node import Node
from PriorityQueue import PriorityQueue
from huffman_utils import read_frequencies

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 unit08/huffman.py

Huffman Coding: Example #1

- Let's try the function with the vowel frequencies:

```
vt = build_tree('huffman/hvfreq.txt')
print(vt)
```

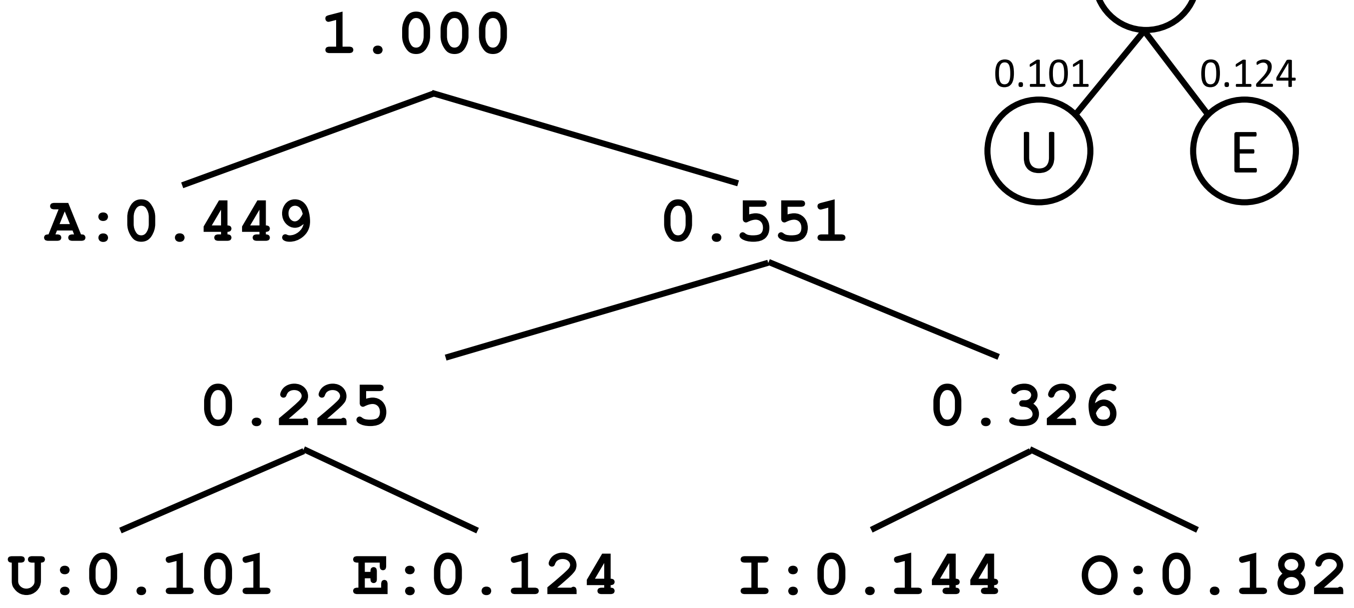
- Output:

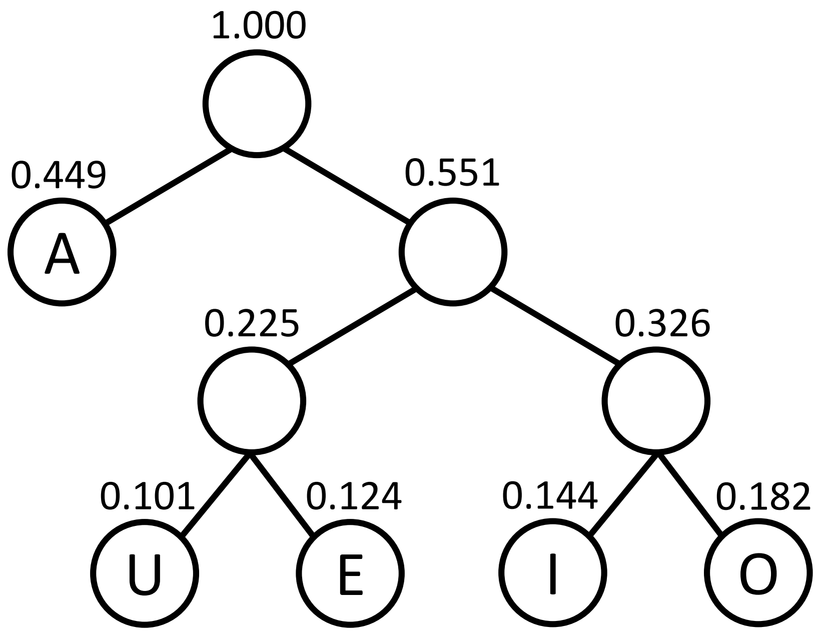
```
( 1.000 ( A: 0.449 ) ( 0.551 ( 0.225
( U: 0.101 ) ( E: 0.124 ) ) ( 0.326 ( I:
0.144 ) ( O: 0.182 ) ) ) )
```

- 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

- The *recursive* function **`assign_codes()`** assembles the Huffman codes from the Huffman tree:

```
def assign_codes(node, code={}, prefix=''):
  if node.is_leaf():
     code[node._char] = prefix
  else:
     assign_codes(node._left, code, prefix + '0')
     assign_codes(node._right, code, prefix + '1')
  return code

codes = assign_codes(vt)
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:

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

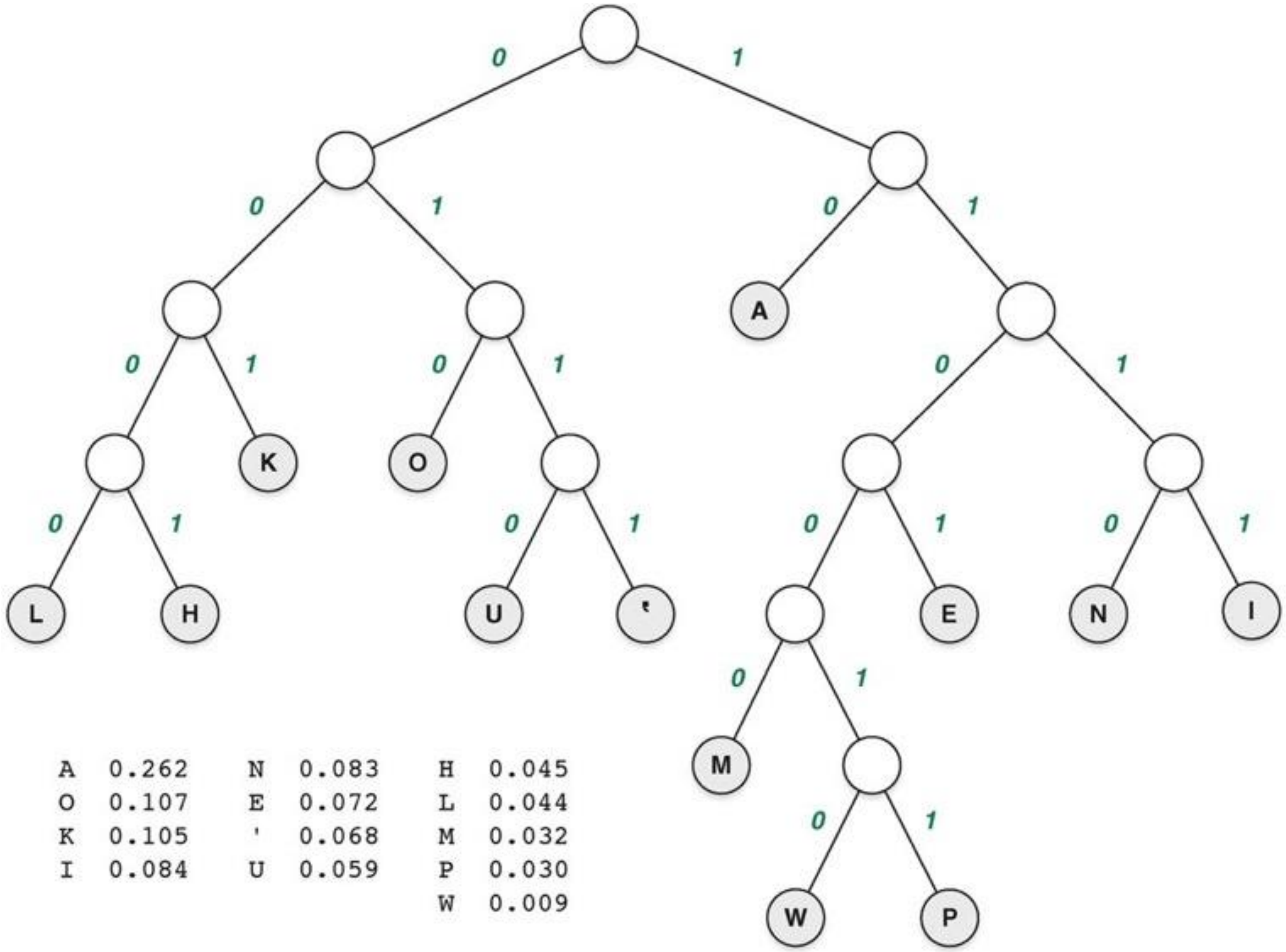
- Then assign the codes:

```
codes = assign_codes(at)
```

- 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 of 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)

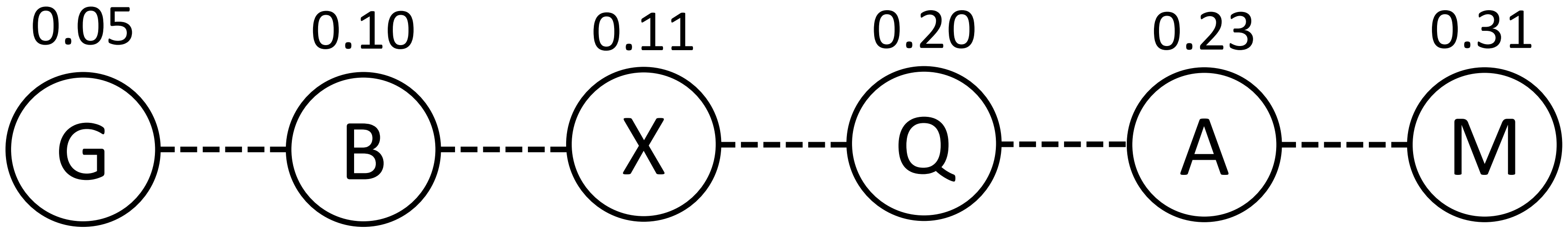
Huffman Coding: Example #2

- 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

Huffman Coding: Example #3

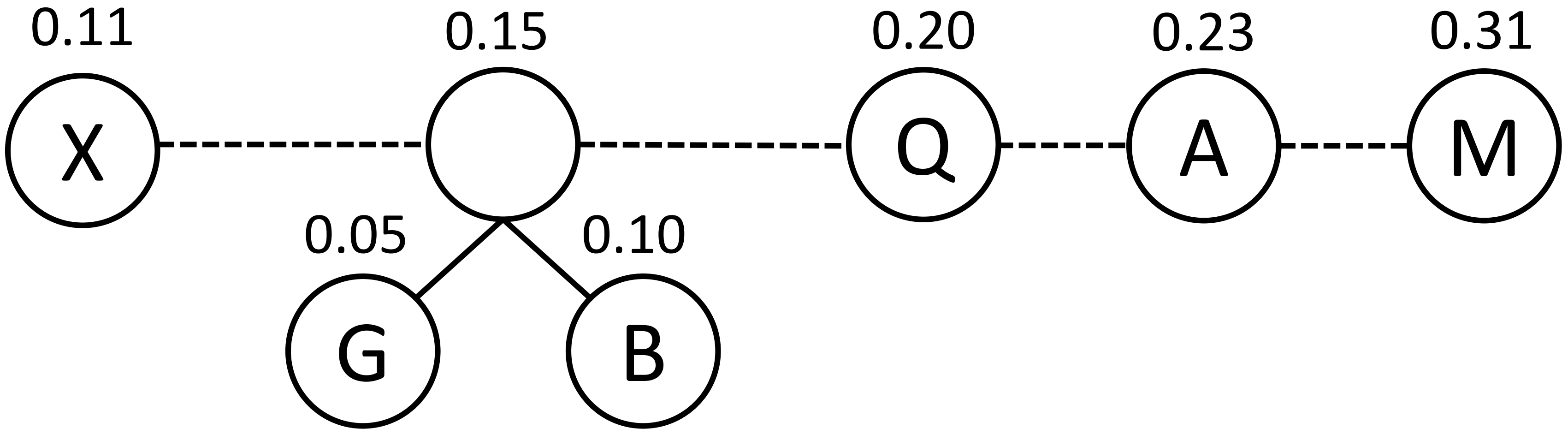
- Given the following letter frequencies, let's compute the Huffman coding for the letters
- We begin by inserting the letters into a priority queue:

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

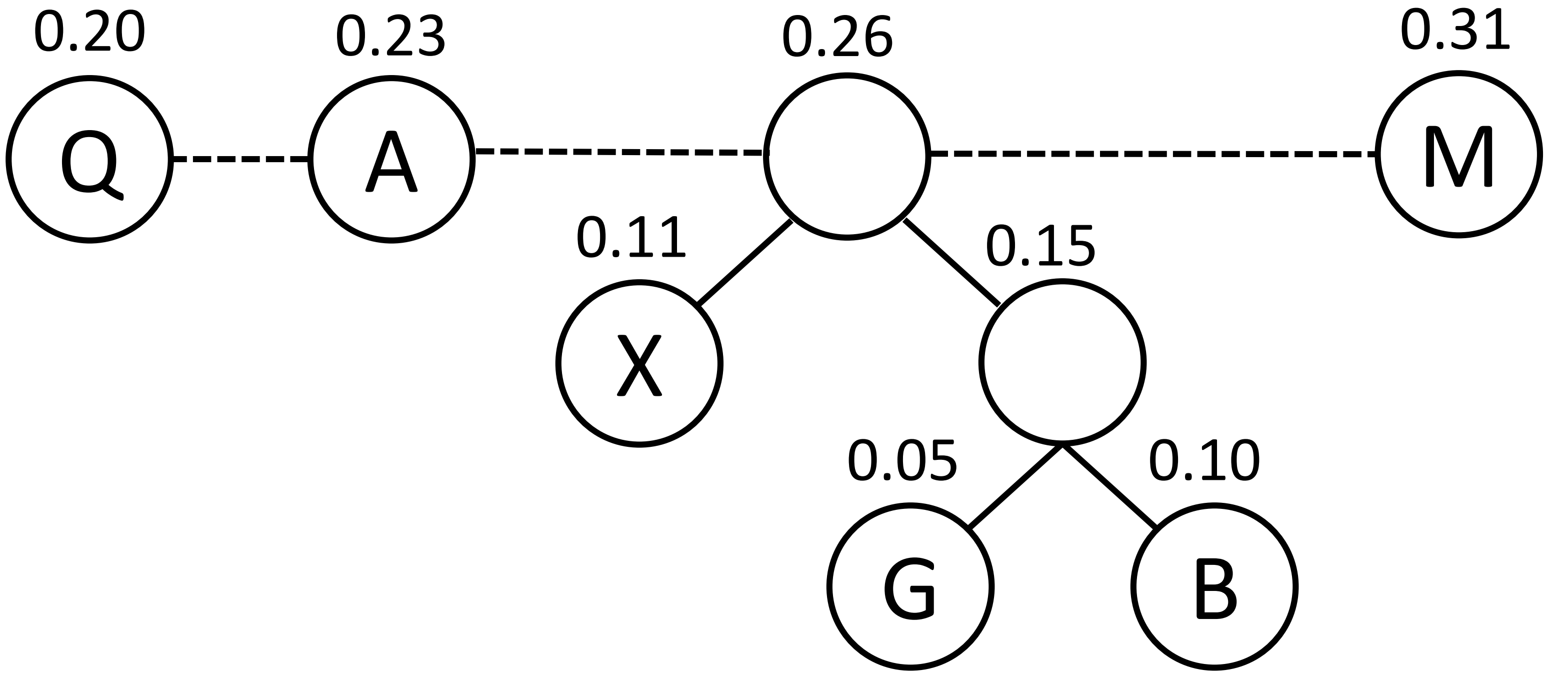


- Now merge the first two elements in the queue until the tree is assembled (see few next slides)

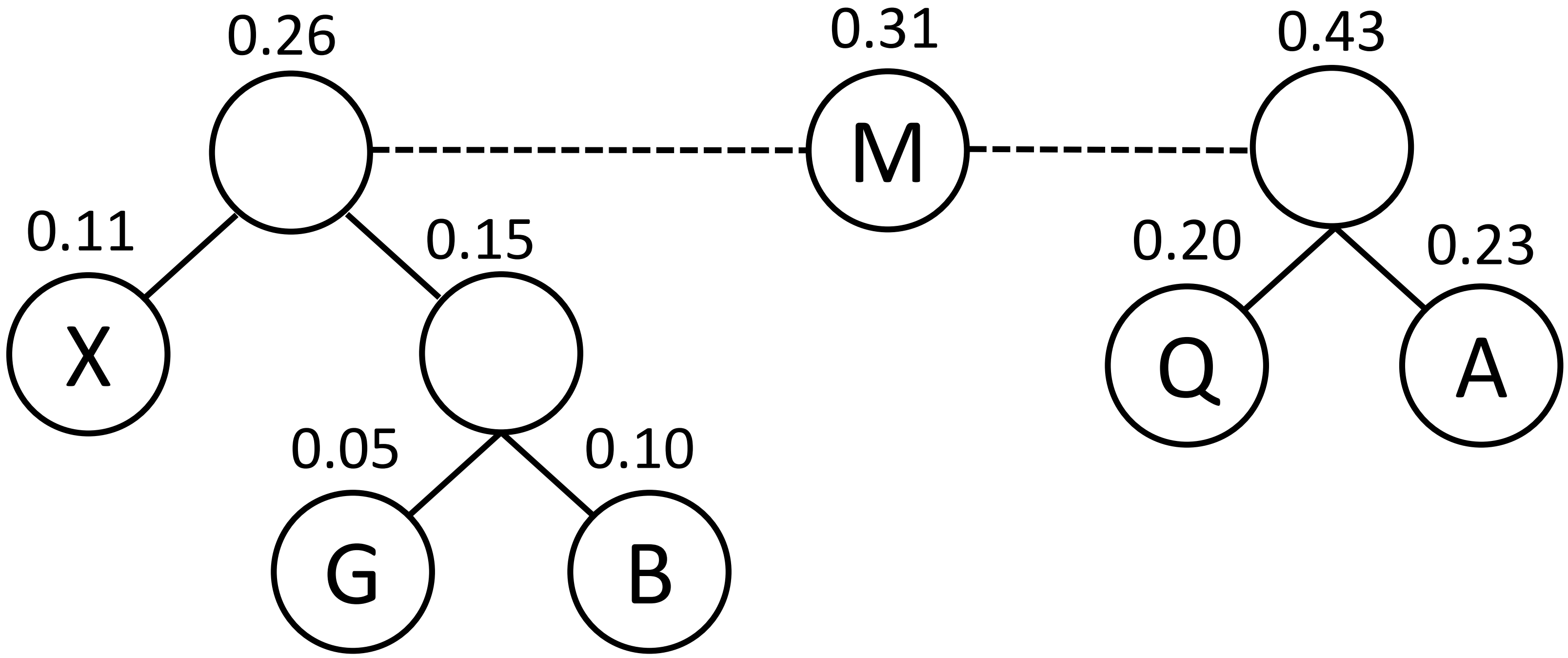
Huffman Coding: Example #3



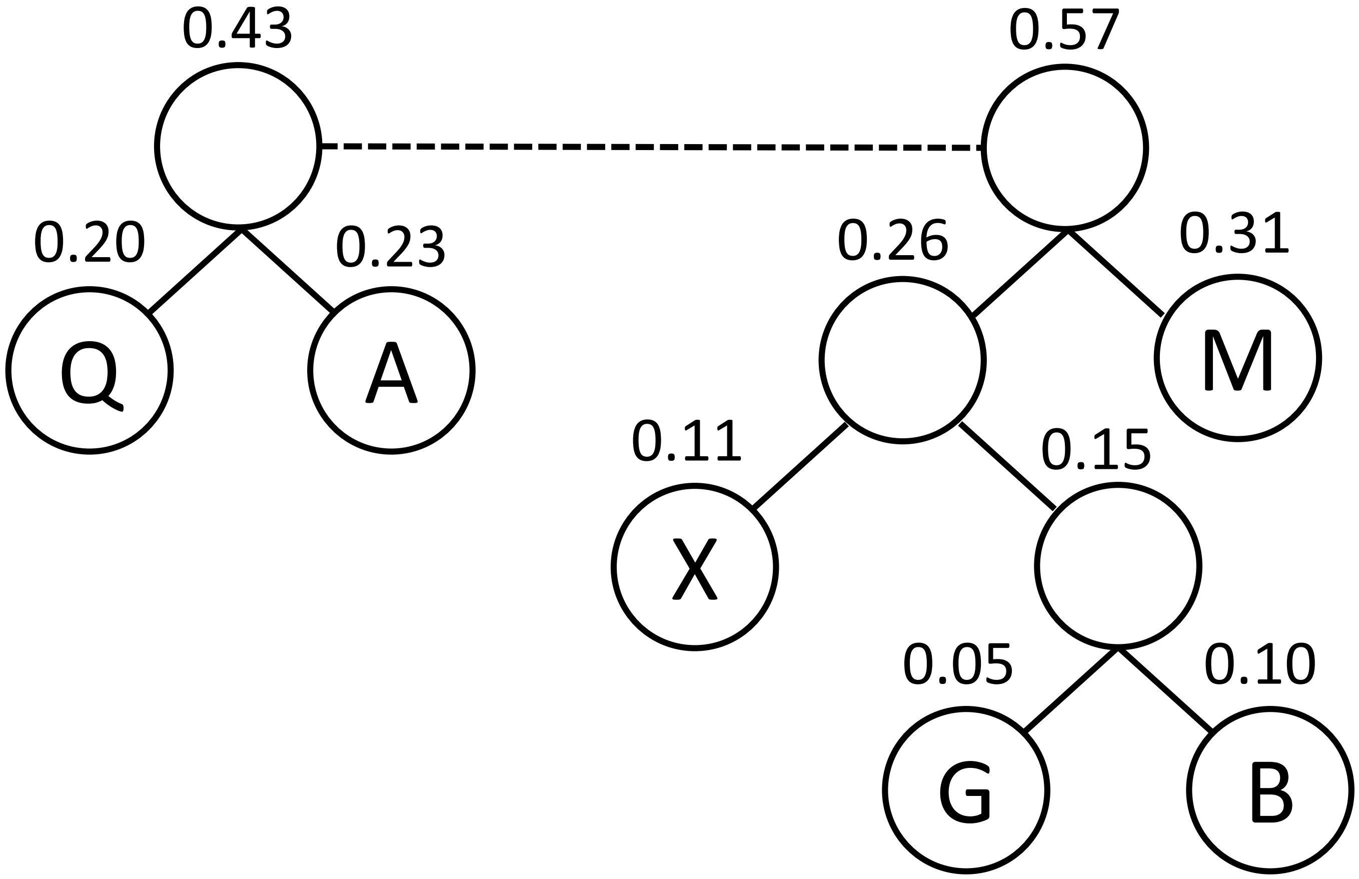
Huffman Coding: Example #3



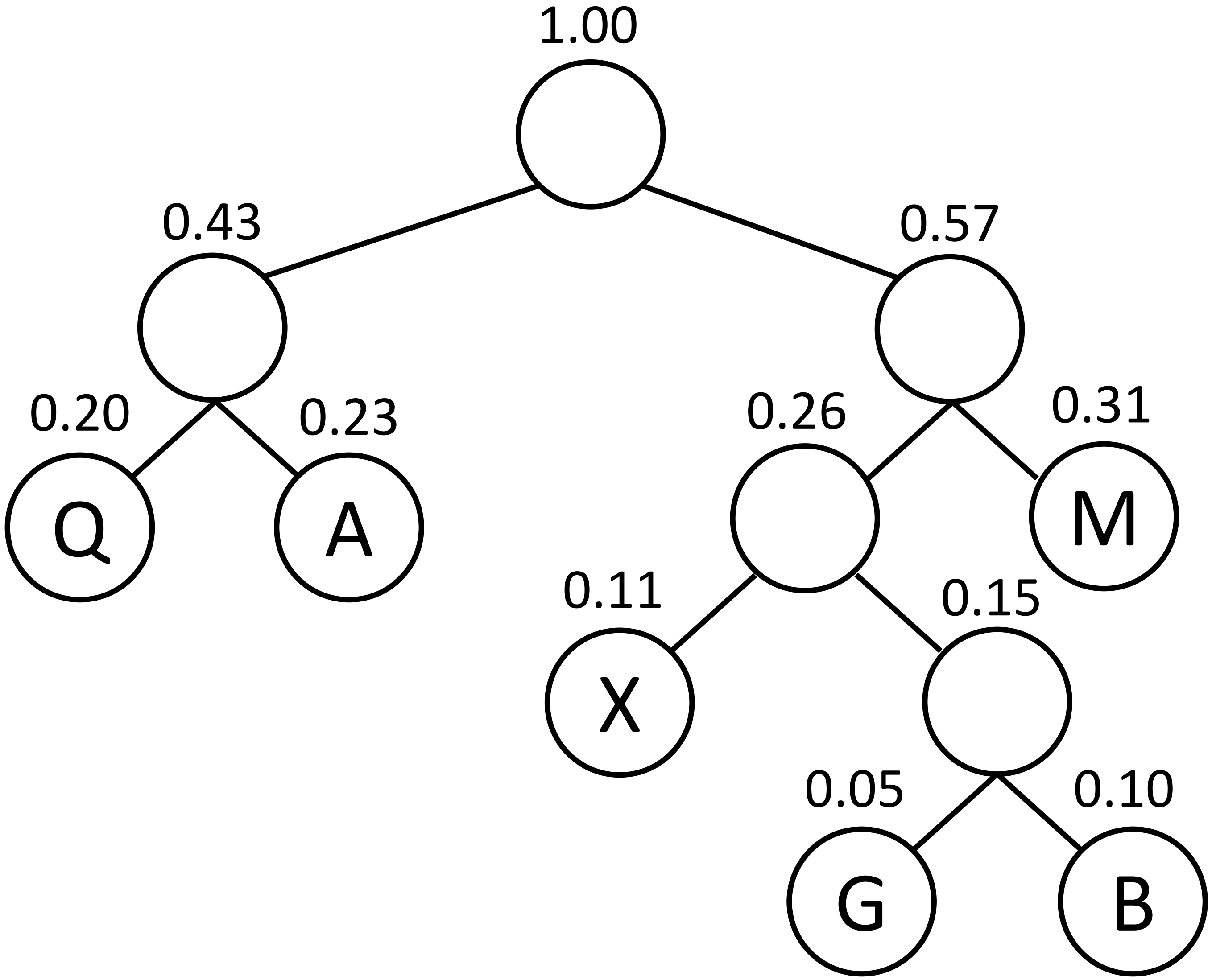
Huffman Coding: Example #3



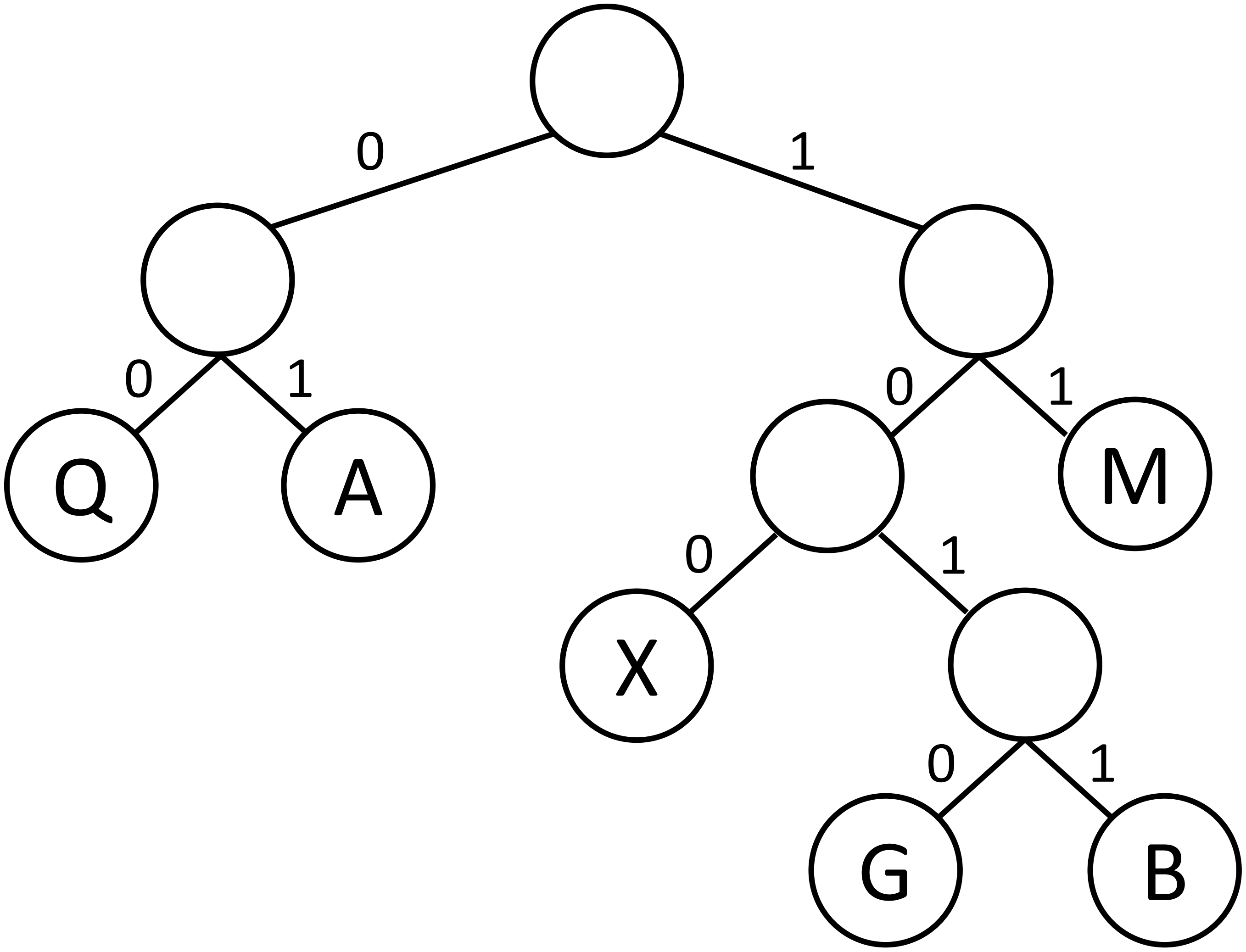
Huffman Coding: Example #3



Huffman Coding: Example #3



Huffman Coding: Example #3



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

encode()/decode()

- 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()/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()}
```

encode()/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
```

- See unit08/huffman.py for this code and examples