

UNIT 7B

Data Representation: Compression

15110 Principles of Computing,
Carnegie Mellon University - CORTINA

1

Fixed-Width Encoding

- In a fixed-width encoding scheme, each character is given a binary code with the same number of bits.
 - Example:
Standard ASCII is a fixed width encoding scheme, where each character is encoded with 7 bits.
This gives us $2^7 = 128$ different codes for characters.

15110 Principles of Computing,
Carnegie Mellon University - CORTINA

2

Fixed-Width Encoding

- Given a character set with n characters, what is the minimum number of bits needed for a fixed-width encoding of these characters?
 - Since a fixed width of k bits gives us n unique codes to use for characters, where $n = 2^k$.
 - So given n characters, the number of bits needed is given by $k = \lceil \log_2 n \rceil$. (We use the ceiling function since $\log_2 n$ may not be an integer.)
 - Example: To encode just the alphabet A-Z using a fixed-width encoding, we would need $\lceil \log_2 26 \rceil = 5$ bits:
e.g. A => 00000, B => 00001, C => 00010, ..., Z => 11001.

Using Fixed-Width Encoding

- If we have a fixed-width encoding scheme using n bits for a character set and we want to transmit or store a file with m characters, we would need mn bits to store the entire file.
- Can we do better?
 - If we assign fewer bits to more frequent characters, and more bits to less frequent characters, then the overall length of the message might be shorter.

Huffman Coding

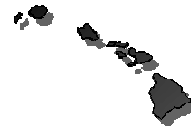
- We can use an encoding scheme named after David A. Huffman to compress our text without losing any information.
- Based on the idea that some characters occur more frequently than others.
- Huffman codes are not fixed-width.



15110 Principles of Computing,
Carnegie Mellon University - CORTINA

5

The Hawaiian Alphabet



- The Hawaiian alphabet consists of 13 characters.
 - ' is the okina which sometimes occurs between vowels (e.g. **KAMA'AINA**)
- The table to the right shows each character along with its relative frequency in Hawaiian words.

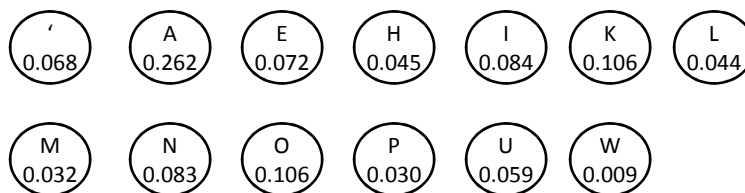
'	0.068
A	0.262
E	0.072
H	0.045
I	0.084
K	0.106
L	0.044
M	0.032
N	0.083
O	0.106
P	0.030
U	0.059
W	0.009

15110 Principles of Computing,
Carnegie Mellon University - CORTINA

6

The Huffman Tree

- We use a tree structure to develop the unique binary code for each letter.
- Start with each letter/frequency as its own node:

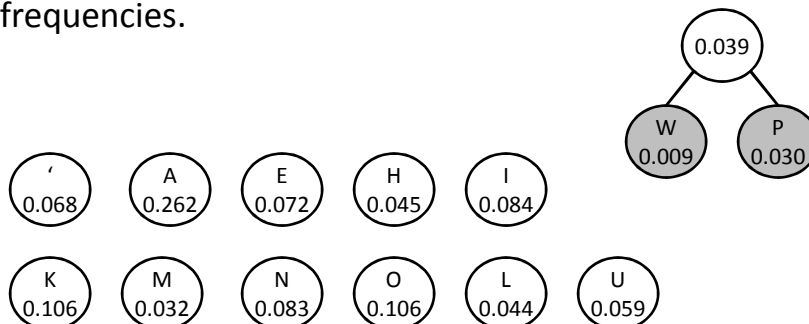


15110 Principles of Computing,
Carnegie Mellon University - CORTINA

7

The Huffman Tree

- Combine lowest two frequency nodes into a tree with a new parent with the sum of their frequencies.

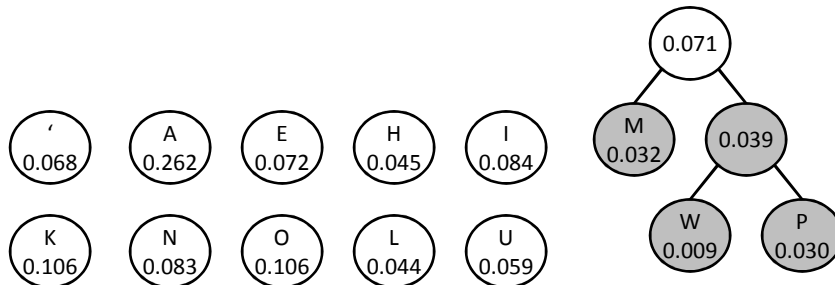


15110 Principles of Computing,
Carnegie Mellon University - CORTINA

8

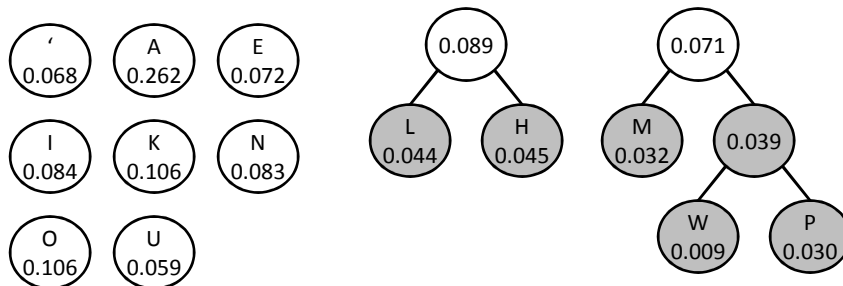
The Huffman Tree

- Combine lowest two frequency nodes (including the new node we just created) into a tree with a new parent with the sum of their frequencies.



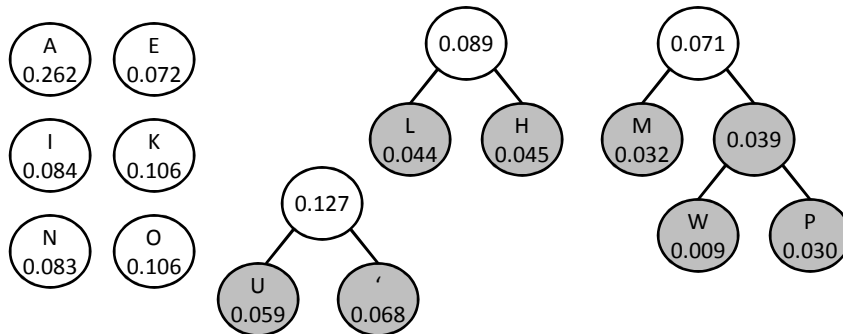
The Huffman Tree

- Combine lowest two frequency nodes (including the new node we just created) into a tree with a new parent with the sum of their frequencies.



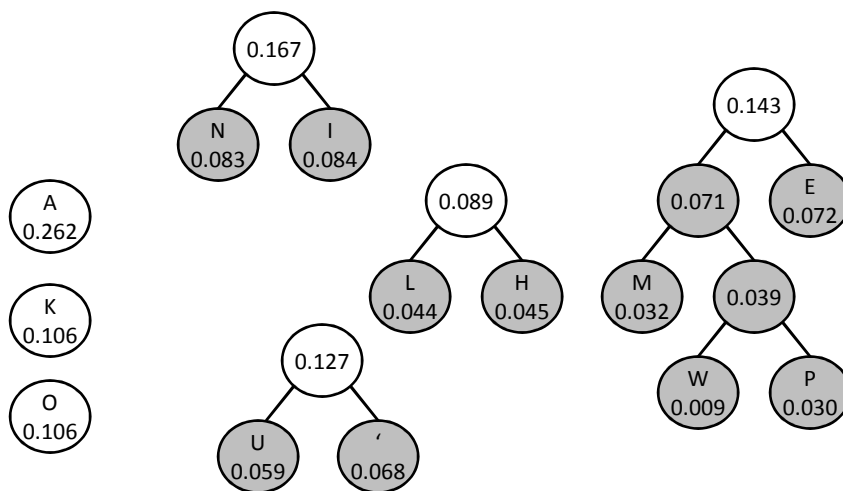
The Huffman Tree

- Combine lowest two frequency nodes (including the new node we just created) into a tree with a new parent with the sum of their frequencies...



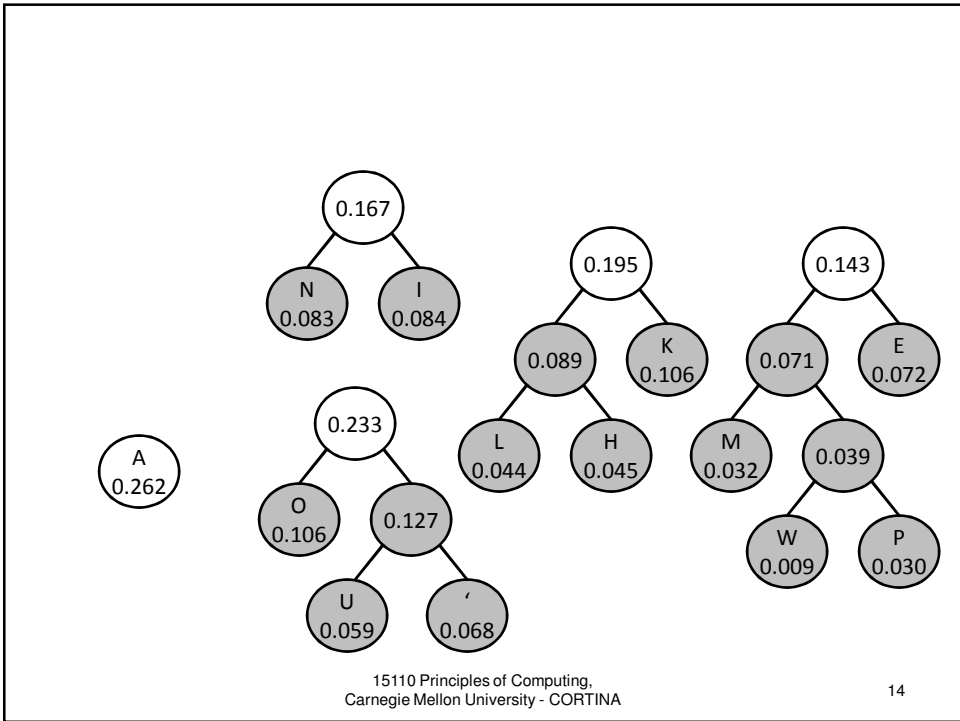
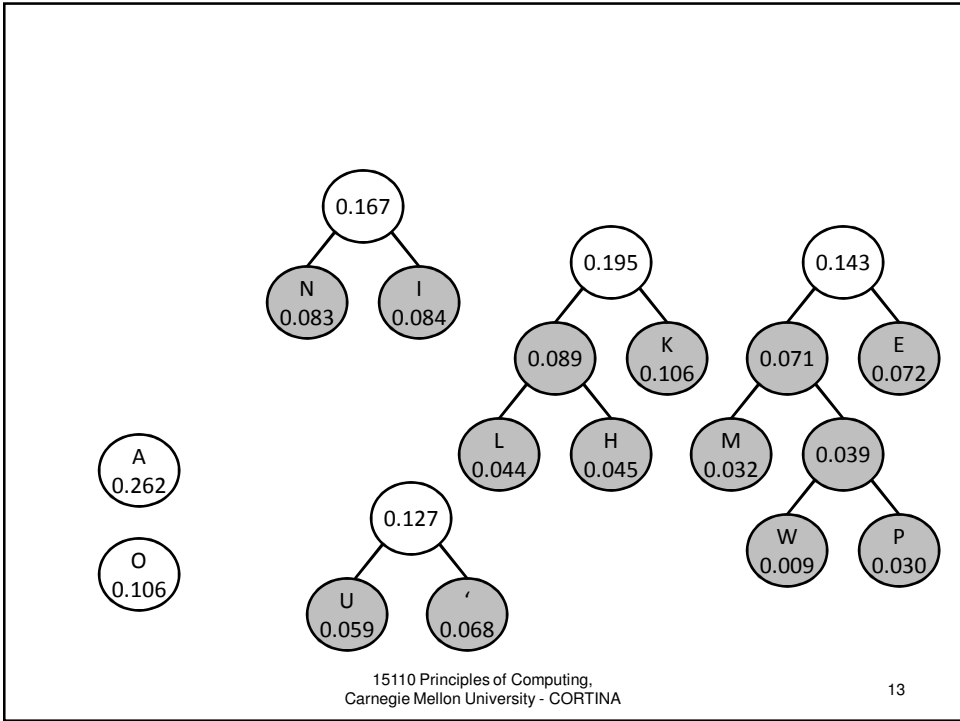
15110 Principles of Computing,
Carnegie Mellon University - CORTINA

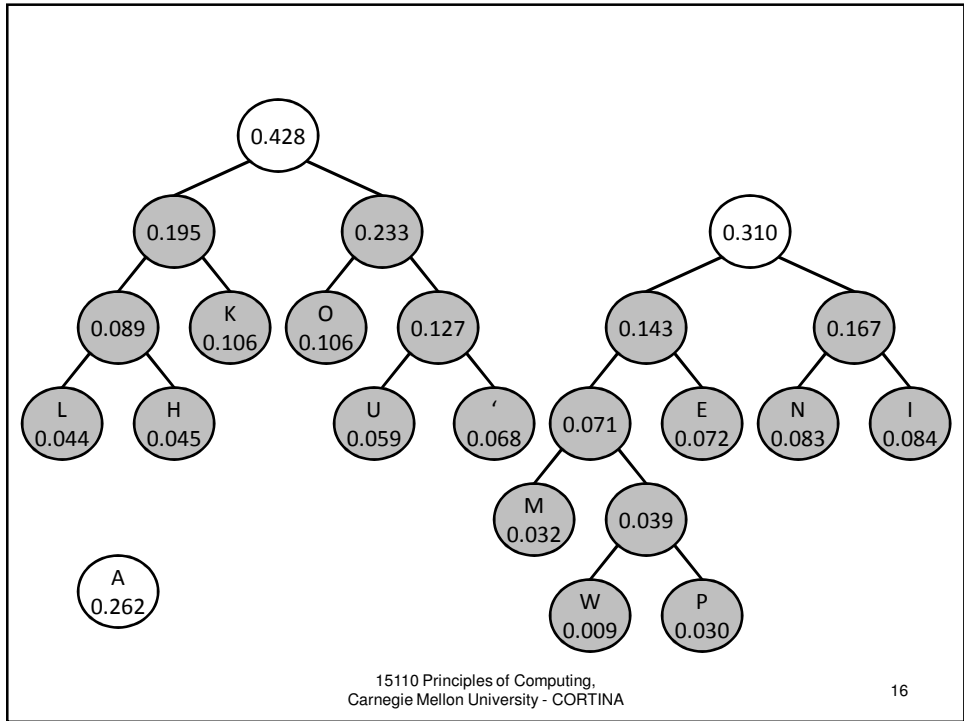
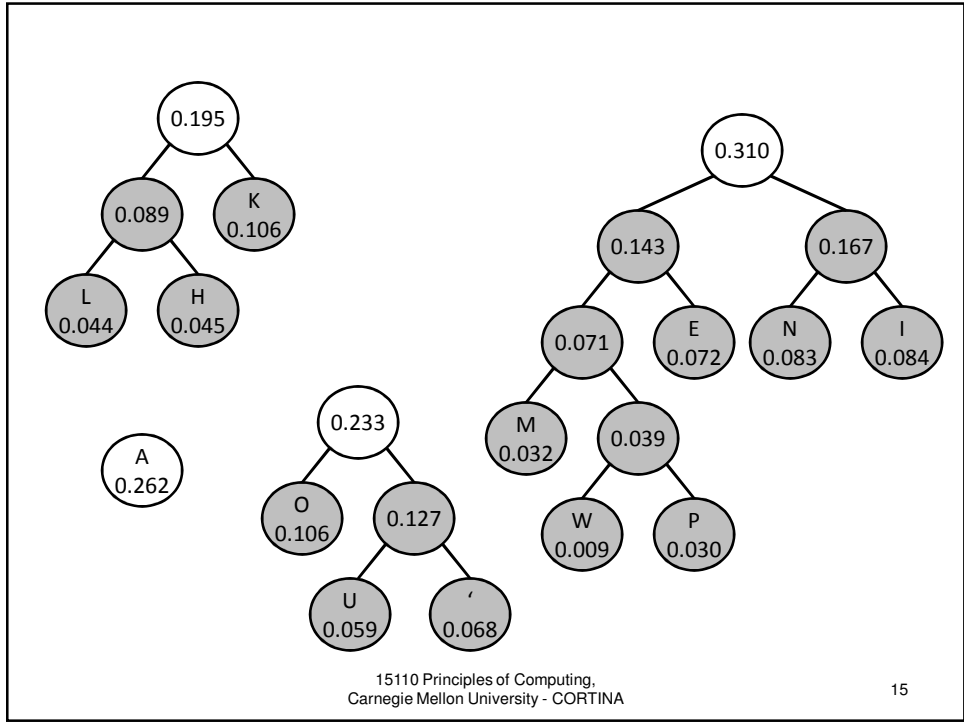
11

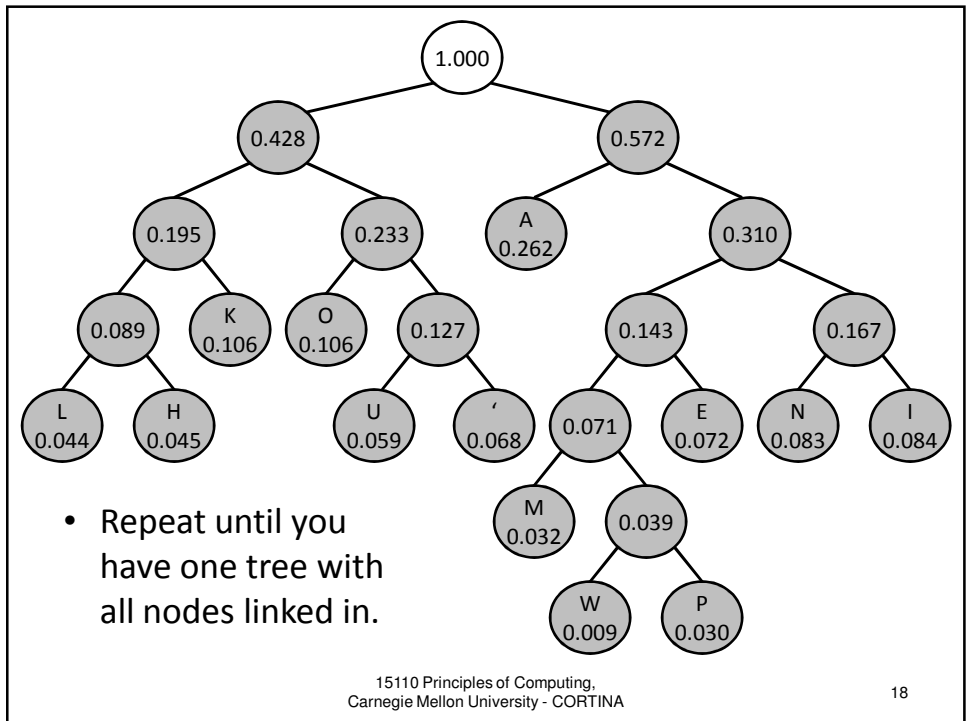
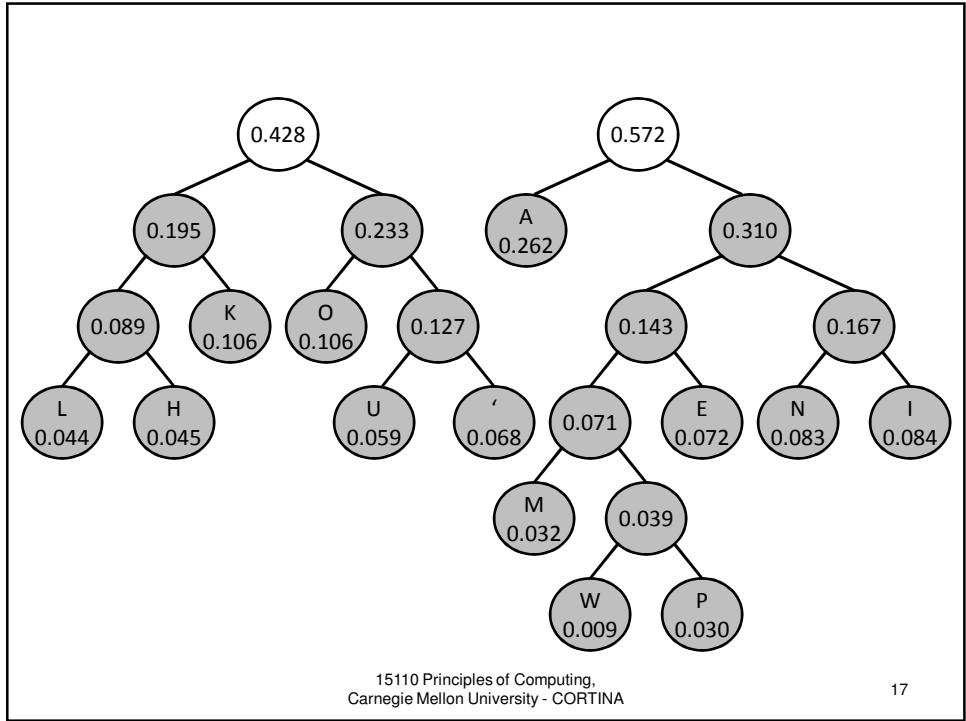


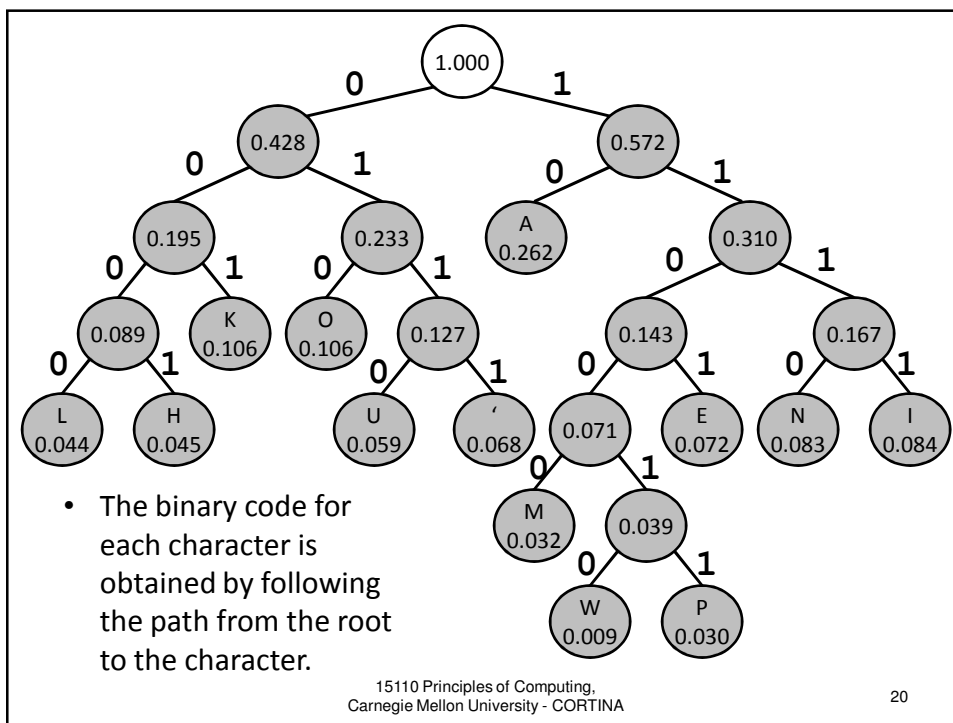
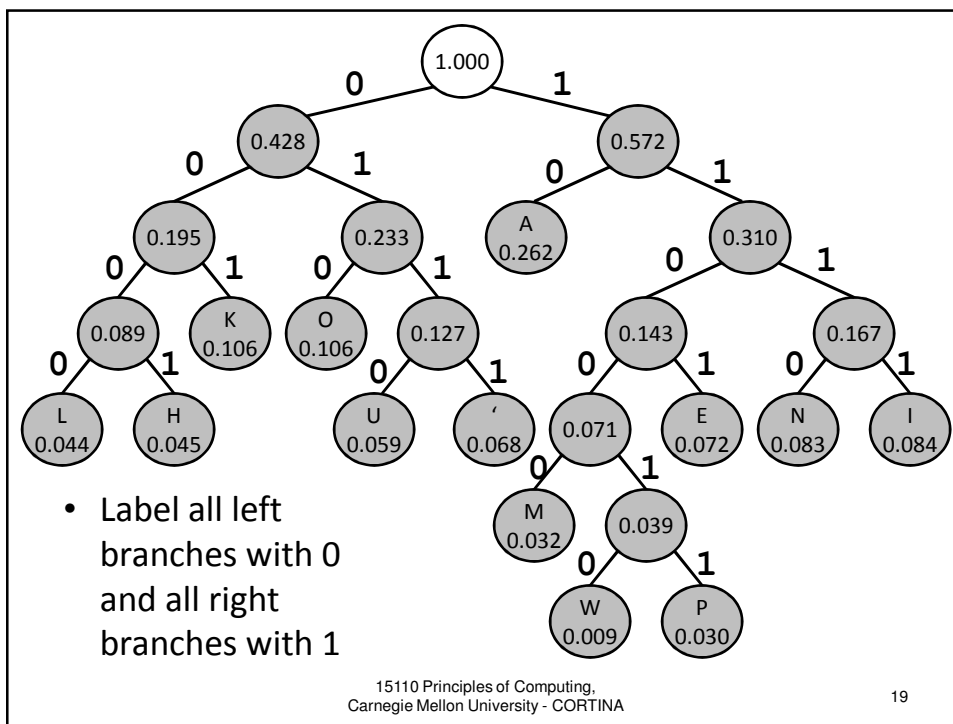
15110 Principles of Computing,
Carnegie Mellon University - CORTINA

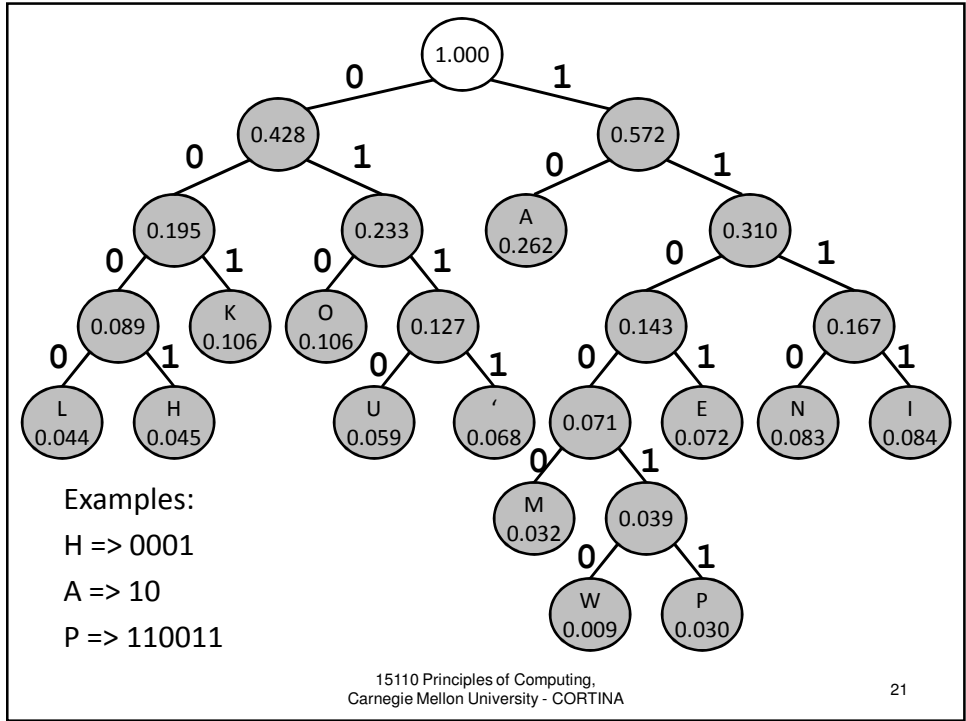
12











Fixed Width vs. Huffman Coding

<table border="0"> <tr><td>'</td><td>0000</td></tr> <tr><td>A</td><td>0001</td></tr> <tr><td>E</td><td>0010</td></tr> <tr><td>H</td><td>0011</td></tr> <tr><td>I</td><td>0100</td></tr> <tr><td>K</td><td>0101</td></tr> <tr><td>L</td><td>0110</td></tr> <tr><td>M</td><td>0111</td></tr> <tr><td>N</td><td>1000</td></tr> <tr><td>O</td><td>1001</td></tr> <tr><td>P</td><td>1010</td></tr> <tr><td>U</td><td>1011</td></tr> <tr><td>W</td><td>1100</td></tr> </table>	'	0000	A	0001	E	0010	H	0011	I	0100	K	0101	L	0110	M	0111	N	1000	O	1001	P	1010	U	1011	W	1100	<table border="0"> <tr><td>'</td><td>0111</td></tr> <tr><td>A</td><td>10</td></tr> <tr><td>E</td><td>1101</td></tr> <tr><td>H</td><td>0001</td></tr> <tr><td>I</td><td>1111</td></tr> <tr><td>K</td><td>001</td></tr> <tr><td>L</td><td>0000</td></tr> <tr><td>M</td><td>11000</td></tr> <tr><td>N</td><td>1110</td></tr> <tr><td>O</td><td>010</td></tr> <tr><td>P</td><td>110011</td></tr> <tr><td>U</td><td>0110</td></tr> <tr><td>W</td><td>110010</td></tr> </table>	'	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	<p style="text-align: center;"><u>ALOHA</u></p> <p>Fixed Width: 00010110100100110001 20 bits</p> <p>Huffman Code: 100000010000110 15 bits</p>	
'	0000																																																						
A	0001																																																						
E	0010																																																						
H	0011																																																						
I	0100																																																						
K	0101																																																						
L	0110																																																						
M	0111																																																						
N	1000																																																						
O	1001																																																						
P	1010																																																						
U	1011																																																						
W	1100																																																						
'	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																																																						

15110 Principles of Computing,
Carnegie Mellon University - CORTINA

22

Priority Queues

NOTE: For this unit, you will need RubyLabs set up and you will need to include BitLab (see p. 167)

- A priority queue (PQ) is like an array that is sorted.
`pq = PriorityQueue.new`
`=> []`
- To add element into the priority queue in its correct position, we use the `<<` operator:
`pq << "peach"`
`pq << "apple"`
`pq << "banana"`
`=> ["apple", "banana", "peach"]`

Priority Queues (cont'd)

- To remove the first element from the priority queue, we will use the `shift` method:
`fruit1 = pq.shift`
`=> "apple"`
`pq`
`=> ["banana", "peach"]`
`fruit2 = pq.shift`
`=> "banana"`
`pq`
`=> ["peach"]`

Tree Nodes

- We can store all of the node data into a 2-dimensional array:

```
table = [ ["'", 0.068], ["A", 0.262],  
          ["E", 0.072], ["H", 0.045], ["I", 0.084],  
          ["K", 0.106], ["L", 0.044], ["M", 0.032],  
          ["N", 0.083], ["O", 0.106], ["P", 0.030],  
          ["U", 0.059], ["W", 0.009] ]
```
- A tree node consists of two values, the character and its frequency. Making one of the tree nodes:

```
char = table[2].first      # "E"  
freq = table[2].last      # 0.072  
node = Node.new(char, freq)
```

Building a PQ of Single Nodes

```
def make_pq(table)  
  pq = PriorityQueue.new  
  for item in table do  
    char = item.first  
    freq = item.last  
    node = Node.new(char, freq)  
    pq << node  
  end  
  return pq  
end
```

Remember: each item in the table is a 2-element array with a character and a frequency.

Building our Priority Queue

```
pq = make_pq(table)
=> [( W: 0.009 ), ( P: 0.030 ),
    ( M: 0.032 ), ( L: 0.044 ),
    ( H: 0.045 ), ( U: 0.059 ),
    ( ' : 0.068 ), ( E: 0.072 ),
    ( N: 0.083 ), ( I: 0.084 ),
    ( K: 0.106 ), ( O: 0.106 ),
    ( A: 0.262 )]
```

← This is our priority queue showing the 13 nodes in sorted order based on frequency.

Building a Huffman Tree

(Slightly different than book version fig 7.9)

```
def build_tree(pq)
  while pq.length > 1
    node1 = pq.shift
    node2 = pq.shift
    pq << Node.combine(node1, node2)
  end
  return pq.first
end
```

← Creates a new node with node1 as its left child and node2 as its right child

Building our Huffman Tree

```
tree = build_tree(pq)
=> ( 1.000 ( 0.428 ( 0.195 ( 0.089
  ( L: 0.044 ) ( H: 0.045 ) ) ( K: 0.106 ) )
  ( 0.233 ( O: 0.106 ) ( 0.127 ( U: 0.059 )
  ( ' : 0.068 ) ) ) ) ( 0.572 ( A: 0.262 )
  ( 0.310 ( 0.143 ( 0.071 ( M: 0.032 )
  ( 0.039 ( W: 0.009 ) ( P: 0.030 ) ) )
  ( E: 0.072 ) ) ( 0.167 ( N: 0.083 )
  ( I: 0.084 ) ) ) ) ) )
```

← This is just our Huffman tree expressed using recursively nested parenthetical components:

```
( root ( left ) ( right ) )
```

15110 Principles of Computing,
Carnegie Mellon University - CORTINA

29

Assigning Codes, Encoding & Decoding

```
ht = assign_codes(tree)
```

from BitLab

takes a Huffman tree and returns a hash table that maps each letter to its binary code

```
ht["W"]
```

```
=> 110010
```

```
ht["A"]
```

```
=> 10
```

Note the [] syntax.

This returns the code associated with the character from the hash table.

```
msg = encode("ALOHA", tree)
```

```
=> 100000010000110
```

```
decode(msg, tree)
```

```
=> "ALOHA"
```

from BitLab

encode and decode functions

15110 Principles of Computing,
Carnegie Mellon University - CORTINA

30