CSE3318: Greedy Algorithms
Huffman Code
by
Dr. Bhanu Jain

All slides are based on: ***Introduction to Algorithms***, by Thomas H. Cormen, Charles E. Leiserson, Ronald E. Rivest, Clifford Stein,3rd edition (CLRS)

Greedy Algorithms: Introduction

Optimization Problems:

- Aim to find the **best solution** among all possible solutions.
- Example: Minimize production costs or maximize revenue.

Ways to Solve Optimization Problems:

1. **Greedy Method**: Makes the best choice at the moment.
2. **Dynamic Programming**: Stores solutions to subproblems and builds up the final solution.

Greedy Algorithm:

- Always makes a locally optimal choice, hoping to achieve a globally optimal solution.
- Example applications: Playing cards, investing in stocks.
- Does not always guarantee the optimal solution but works for problems with optimal substructure.

Characteristics & Advantages of Greedy Algorithms

Key Characteristics:

- **Straightforward & Optimized**: Directly selects the best available choice.
- **Fast Execution**: Finds the solution in fewer steps compared to dynamic programming.
- **Does Not Require Combining Subproblems**: Unlike dynamic programming, each step reaches an independent decision.
- **Easier to Implement**: Simple logic and efficient computation.

Examples of Greedy Algorithms:

- **Scheduling tasks** with deadlines and penalties.
- **Graph algorithms**: Minimum spanning trees (Prim's & Kruskal's), Dijkstra's shortest path.
- **CPU Scheduling**: First Come First Serve, Shortest Job First, Round Robin, Priority Scheduling.

Limitations & When To Use Greedy Algorithms

Disadvantages:

- **May lead to inaccurate results**: Always selecting the immediate best choice can sometimes overlook a better overall solution.
- **Not always optimal**: Works well only when the problem has optimal substructure.
- **Fails for certain problems**, such as the Traveling Salesman Problem (TSP).

When to Use Greedy Algorithms?

- If the problem exhibits optimal substructure (like Dynamic Programming).
- If a locally optimal choice leads to a globally optimal solution.
- If efficiency is a priority, and an approximate solution is acceptable.

Huffman Code

Greedy Algorithms

Huffman code:

- A data compression algorithm (very effective)
- A **lossless data compression** algorithm
- Do you compress data anywhere?

Algorithms used in Lossy compression: high degrees of compression result in smaller files-some original pixels/ sound waves/video frames lost forever

- Transform Coding
- Discrete Cosine Transform
- Discrete Wavelet Transform

Huffman Code

Representing **compacted file** information:

- **Fixed-length codeword**
- **Variable-length codeword**

Example:

Given:

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

- 100,000-character data file to be stored compactly.
- Characters in the file occur with the frequencies
- **6 (a-f)** different characters appear, and the character **a** occurs 45,000 times

Huffman Code

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101

Example:

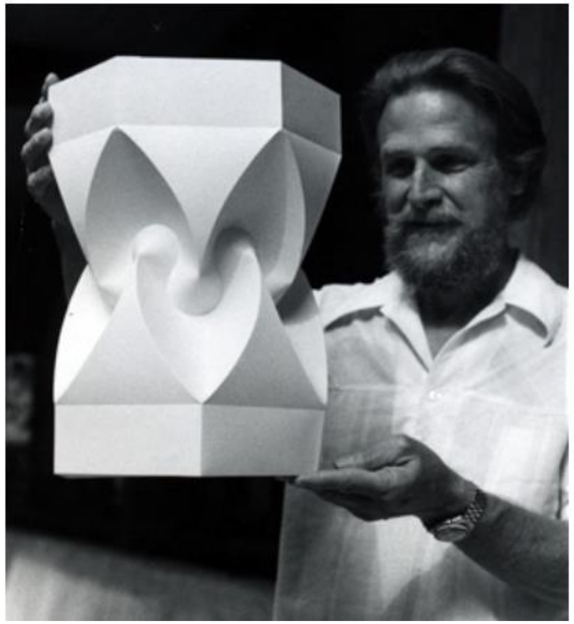
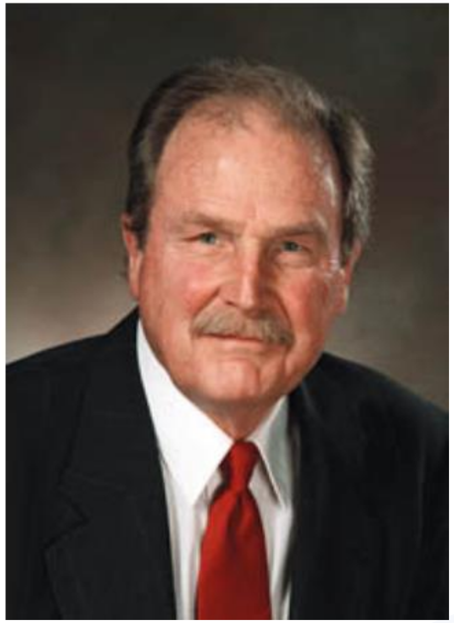
- 100,000-character data file to be stored compactly.
- Characters in the file occur with the frequencies
- **6 (a-f)** different characters appear, and the character **a** occurs 45,000 times
- Use binary character code(codeword) to represent each character by a unique binary string

Representing compacted file information:

- **Fixed-length codeword:** 3-bit codeword, we can encode the file in 300,000 bits

Final Exam vs Term Paper

[Robert Fano](#) in 1975

David Huffman in 1978 and in 1999

- Students can do better than professors. David Huffman (1925-1999) was a student in an **electrical engineering** course in **1951**.
- Professor Robert Fano, offered students a choice of taking a final exam or writing a term paper.
- Huffman did not want to take the final.
- The topic of the paper was to find the most efficient (optimal) code. (open problem)
- Huffman was ready to give up when the solution suddenly came to him.
- The code he discovered was optimal, that is, it had the lowest possible average message length.
- Huffman said that likely he would not have even attempted the problem if he had known that his professor was struggling with it"

1952 paper "A Method for the Construction of Minimum-Redundancy Codes"

Source: https://www.maa.org/press/periodicals/convergence/discovery-of-huffman-codes

Huffman Code

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Example:

- 100,000-character data file to be stored compactly.
- Characters in the file occur with the frequencies
- **6 (a-f)** different characters appear, and the character **a** occurs 45,000 times
- **Use binary character code**(codeword) to represent each character by a unique binary string

Representing compacted file information:

- **Fixed-length codeword:** 3-bit codeword, we can encode the file in 300,000 bits
- **Variable-length code**: encodes the file in only 224,000 bits:
  - 1-bit string 0 represents a, and the 4-bit string 1100 represents f.
  - Results in a savings of approximately 25% as compared to the fixed-length codeword coding.

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

Huffman Code

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Example:

Representing compacted file information:

- **Fixed-length codeword:** 3-bit codeword, we can encode the file in 300,000 bits
- **Variable-length code**: encodes the file in only 224,000 bits:
  - 1-bit string 0 represents a, and the 4-bit string 1100 represents f.
  - Results in a savings of approximately 25% as compared to the fixed-length codeword coding.

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

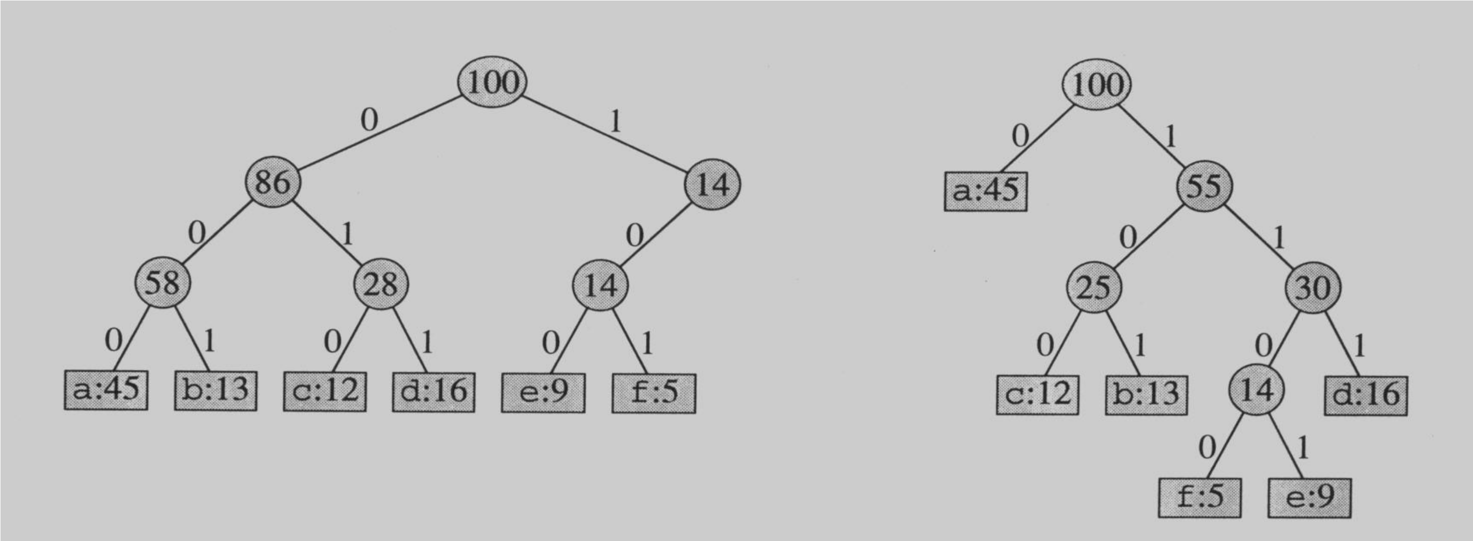
- **The basic idea**
  - Instead of storing each character in a file as an 8-bit ASCII value, store
    - the more frequently occurring characters using fewer bits and
    - less frequently occurring characters using more bits
  - On average this should decrease the file size (usually ½)

Prefix Codes

- **Prefix code** is a type of code that lets you decode a text without special markers.

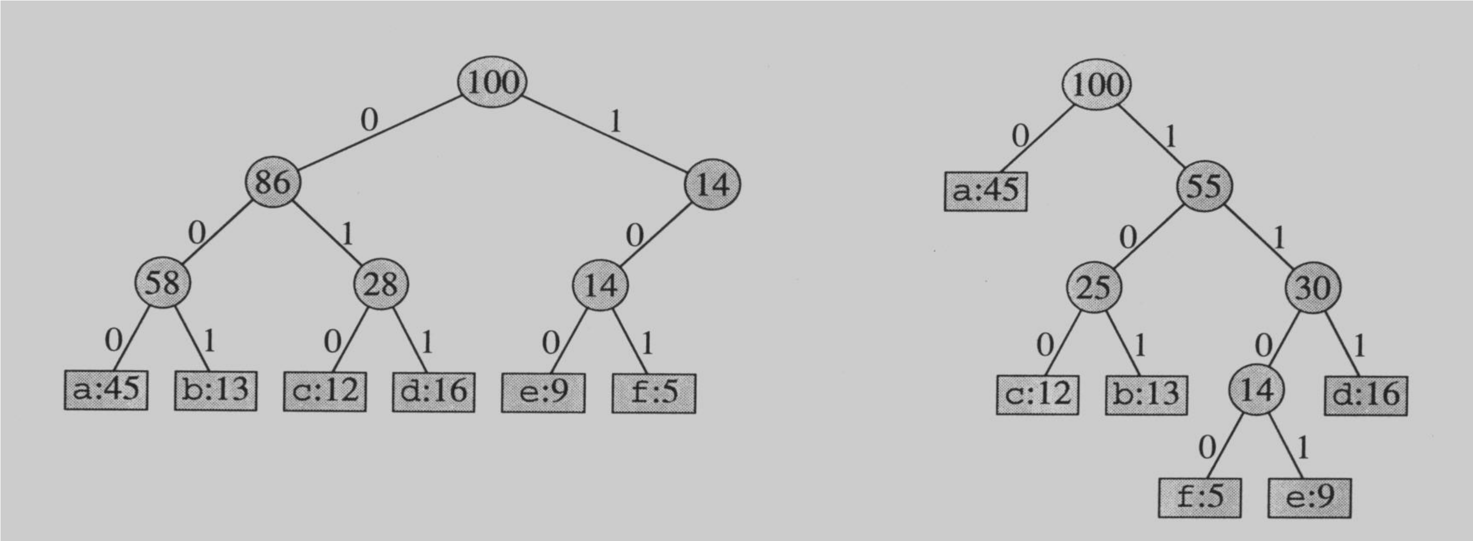
  For example, the map {a=0, b=10, c=11} is a prefix code as no marker is needed for decryption of any string

- No encoding of a character can be the prefix of the longer encoding of another character, for example, we could not encode *t* as 01 and *x* as 01101 since 01 is a prefix of 01101
- A prefix code is a code in which no codeword is a prefix of another codeword. Nor can a codeword be derived from another by appending more bits to a shorter codeword
- By using a binary tree representation, we will generate prefix codes provided all letters are leaves

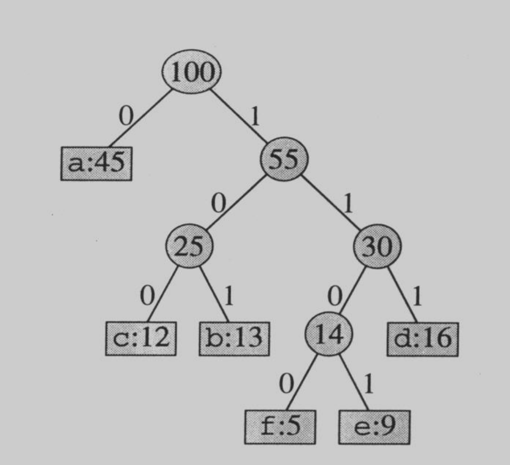


Prefix Codes

- A message can be decoded uniquely.
- Following the tree until it reaches to a leaf, and then repeat!
- Draw a few more tree and produce the codes!!!



Some Properties



- Prefix codes allow easy decoding
  - Given a: 0, b: 101, c: 100, d: 111, e: 1101, f: 1100
  - Decode 001011101 going left to right, 0|01011101, a|0|1011101, a|a|101|1101, a|a|b|1101, a|a|b|e
- 001011101 parses uniquely as 0.0 .01.1101, which decodes to aabe
- An optimal code must be a full binary tree (a tree where every internal node has two children)
- For *C* leaves there are *C-1* internal nodes
- The number of bits to encode a file is

$$\text{Average}(T) = \sum_{i=1}^{n} f_i \bullet \text{length}_T(c_i)$$

where *f(c)* is the freq of *c*, $\text{length}_T(c)$ is the tree depth of *c*, which corresponds to the code length of *c*

The Algorithm

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
      do allocate a new node z
         left[z] ← x ← EXTRACT-MIN(Q)
         right[z] ← y ← EXTRACT-MIN(Q)
         f[z] ← f[x] + f[y]
         INSERT(Q, z)
return EXTRACT-MIN(Q)                ▷ Return the root of the tree.
```

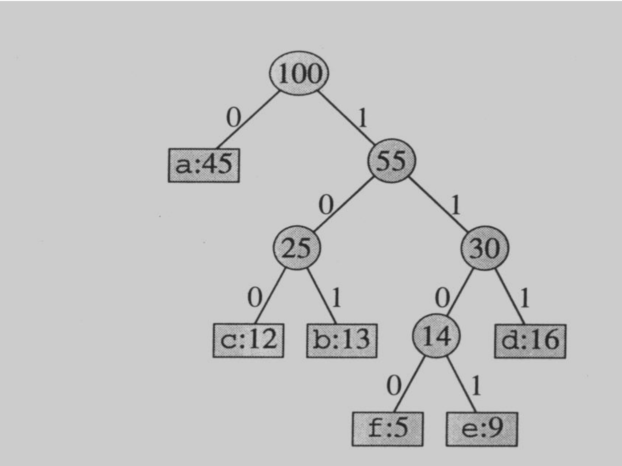
- C is a set of n characters and that each character c ϵ C is an object
- c has an attribute c.freq => frequency of c in the text.
- The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner
- Starts with a set of |C| leaves
- Performs a sequence of |C| - 1 “merging” operations to create the final tree.
- The algorithm uses a min-priority queue Q, keyed on the freq attribute, to identify the two least-frequent objects to merge together.

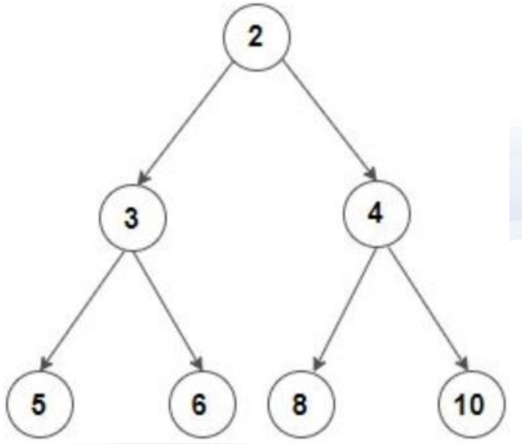
- Merging two objects, results in a new object whose frequency is the sum of the frequencies of the two merged objects

- An appropriate data structure is a binary min-heap
- Rebuilding the heap is *lg n* and *n-1* extractions are made, so the complexity is O(*n lg n*)
- The encoding is NOT unique, other encoding may work just as well, but none will work better

Running Time Analysis

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4       do allocate a new node z
5          left[z] ← x ← EXTRACT-MIN(Q)
6          right[z] ← y ← EXTRACT-MIN(Q)
7          f[z] ← f[x] + f[y]
8          INSERT(Q, z)
9  return EXTRACT-MIN(Q)          ▷ Return the root of the tree.
```





binary min-heap

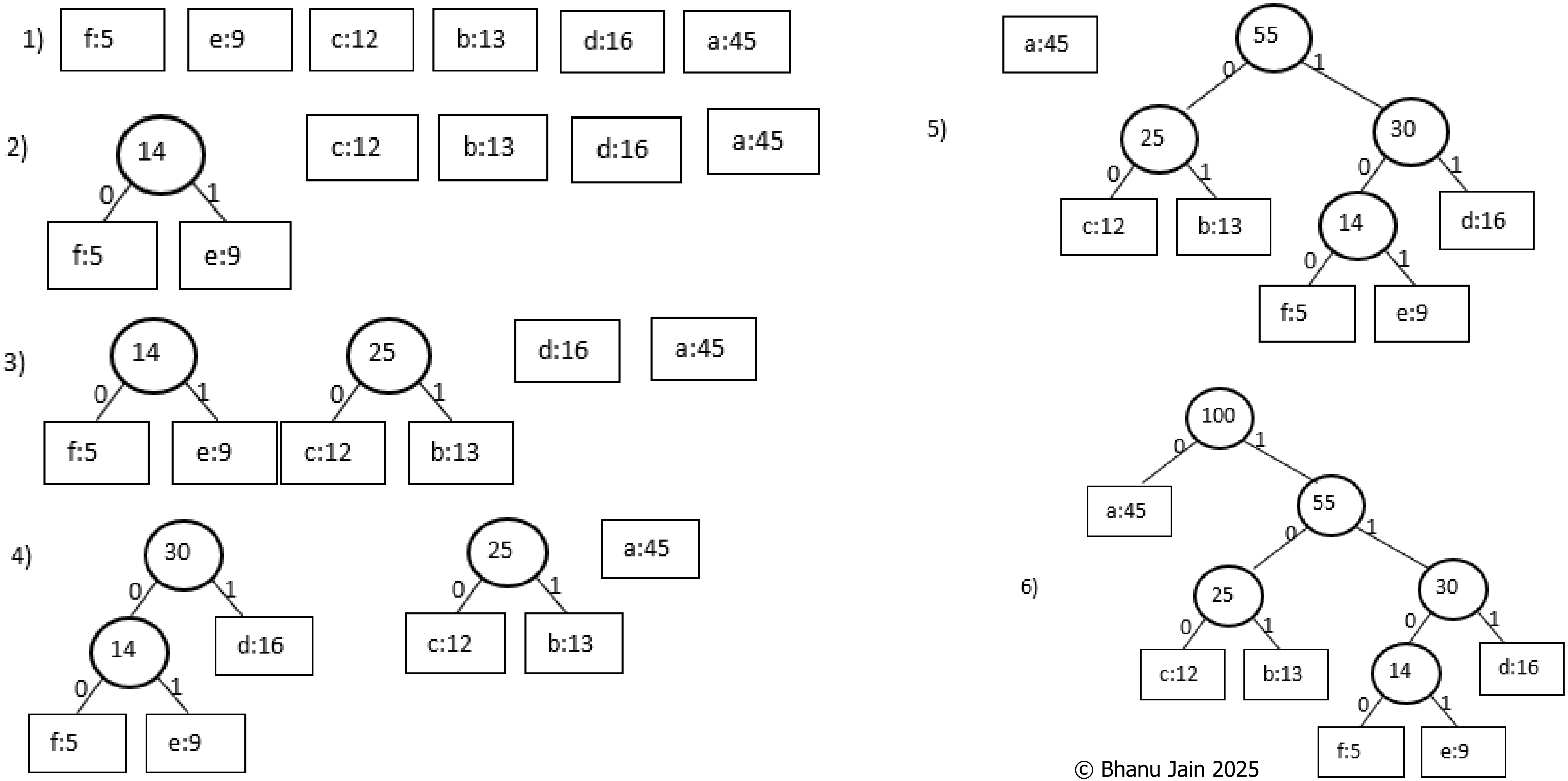
- Running time of Huffman's algorithm
- Assumption: Q is implemented as a binary min-heap
- Line 2: Initialize Q (with set C of n chars) in O(n) time using the BUILD-MIN-HEAP procedure
- Lines 3-8: The for loop exactly n – 1 times
- Since each heap operation requires time O(lg n), the loop contributes O(n lg n) to the running time.
- Thus, the total running time of HUFFMAN on a set of n characters is O(n lg n)

Running Time Analysis

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
      do allocate a new node z
         left[z] ← x ← EXTRACT-MIN(Q)
         right[z] ← y ← EXTRACT-MIN(Q)
         f[z] ← f[x] + f[y]
         INSERT(Q, z)
return EXTRACT-MIN(Q)            ▷ Return the root of the tree.
```

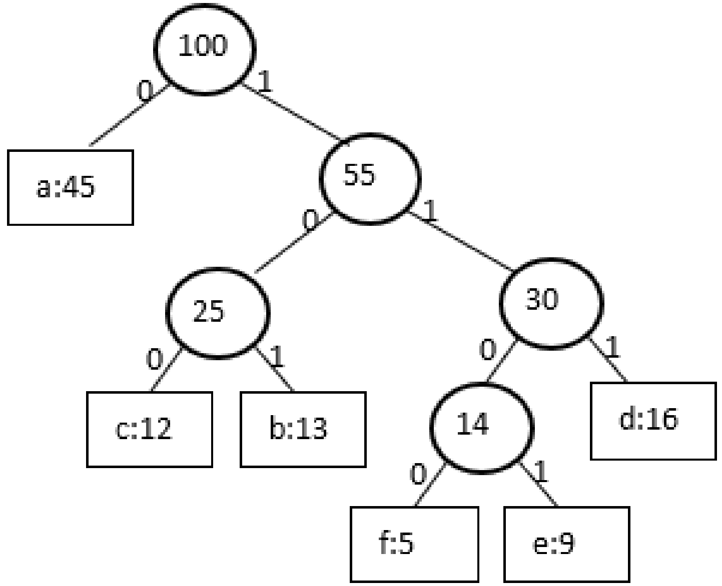
- Contents of the queue sorted into increasing order by frequency
- Leaves are shown as rectangles containing a character and its frequency
- Internal nodes are shown as circles containing the sum of the frequencies of their children.
- An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child
- The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter

Building the Encoding Tree



Building the Encoding Tree

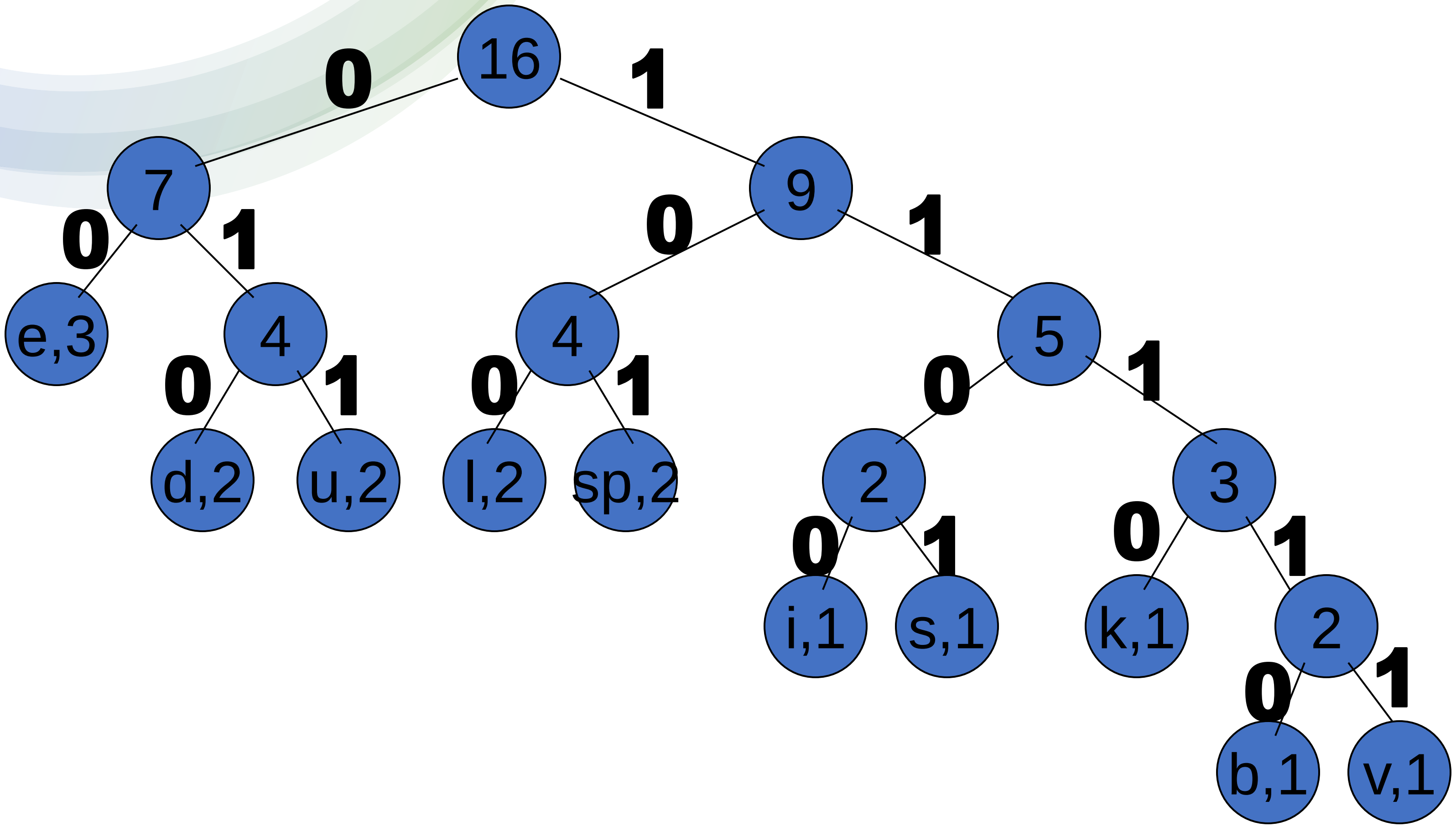
6)



a	0
b	101
c	100
d	111
e	1101
f	1100

Example by Dr. Junzhou Huang

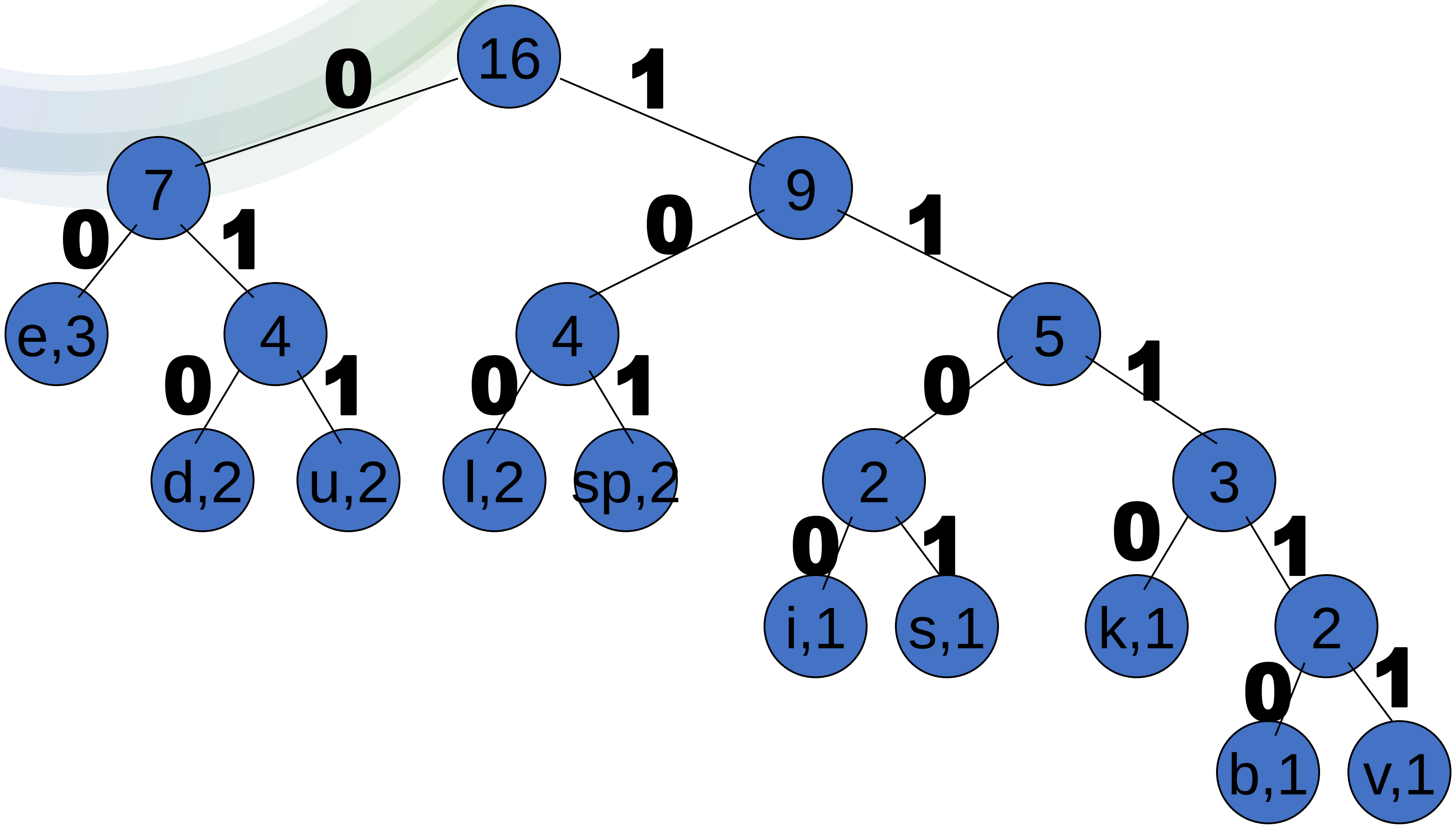
Huffman Code: Example



e	
d	
u	
l	
sp	
i	
s	
k	
b	
v	

Example by Dr. Junzhou Huang

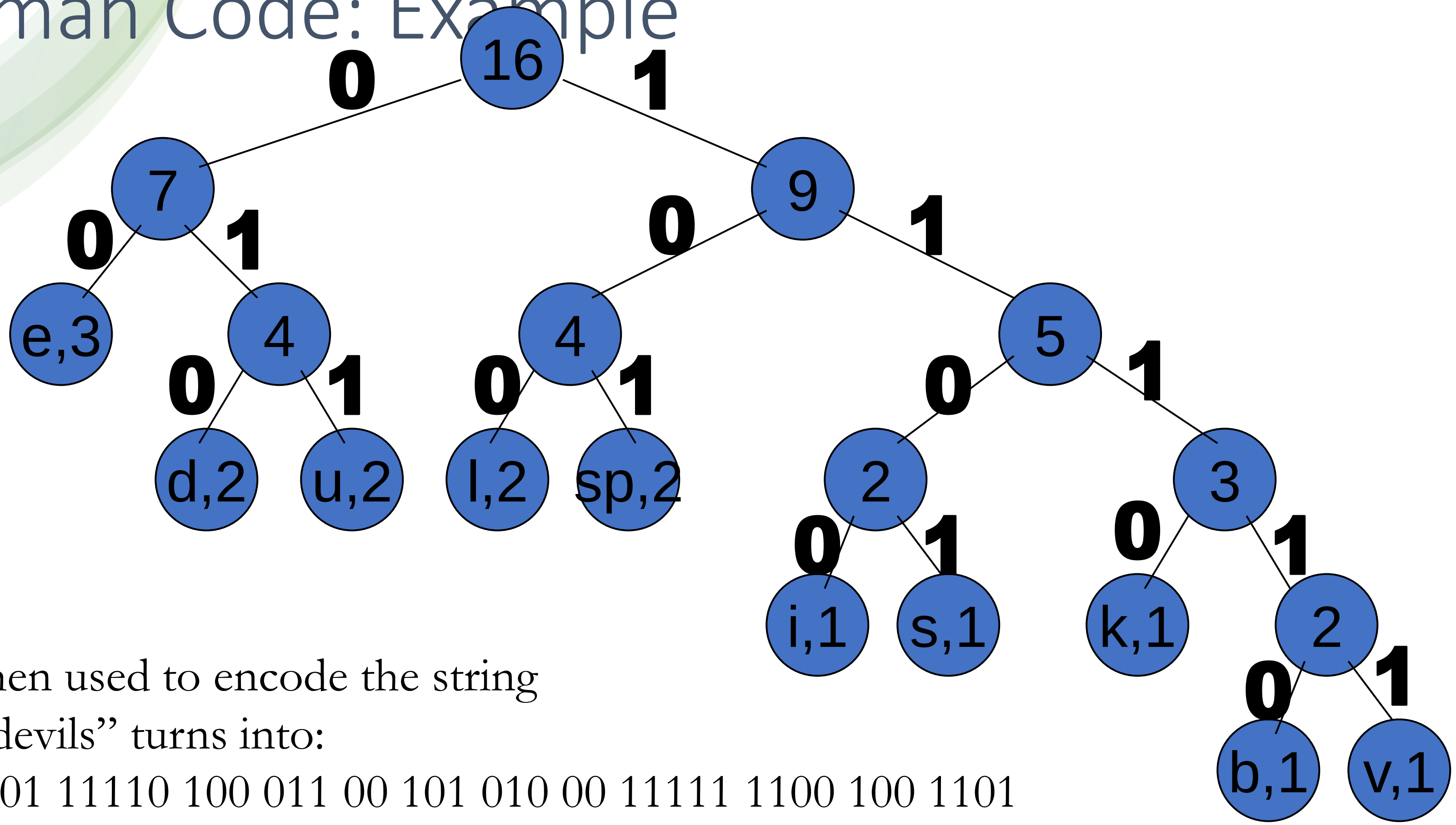
Huffman Code: Example



e	00
d	010
u	011
l	100
sp	101
i	1100
s	1101
k	1110
b	11110
v	11111

Example by Dr. Junzhou Huang

Huffman Code: Example



e	00
d	010
u	011
l	100
sp	101
i	1100
s	1101
k	1110
b	11110
v	11111

These codes are then used to encode the string

Thus, "duke blue devils" turns into:

010 011 1110 00 101 11110 100 011 00 101 010 00 11111 1100 100 1101

When grouped into 8-bit bytes:

01001111 10001011 11101000 11001010 10001111 11100100 1101xxxx

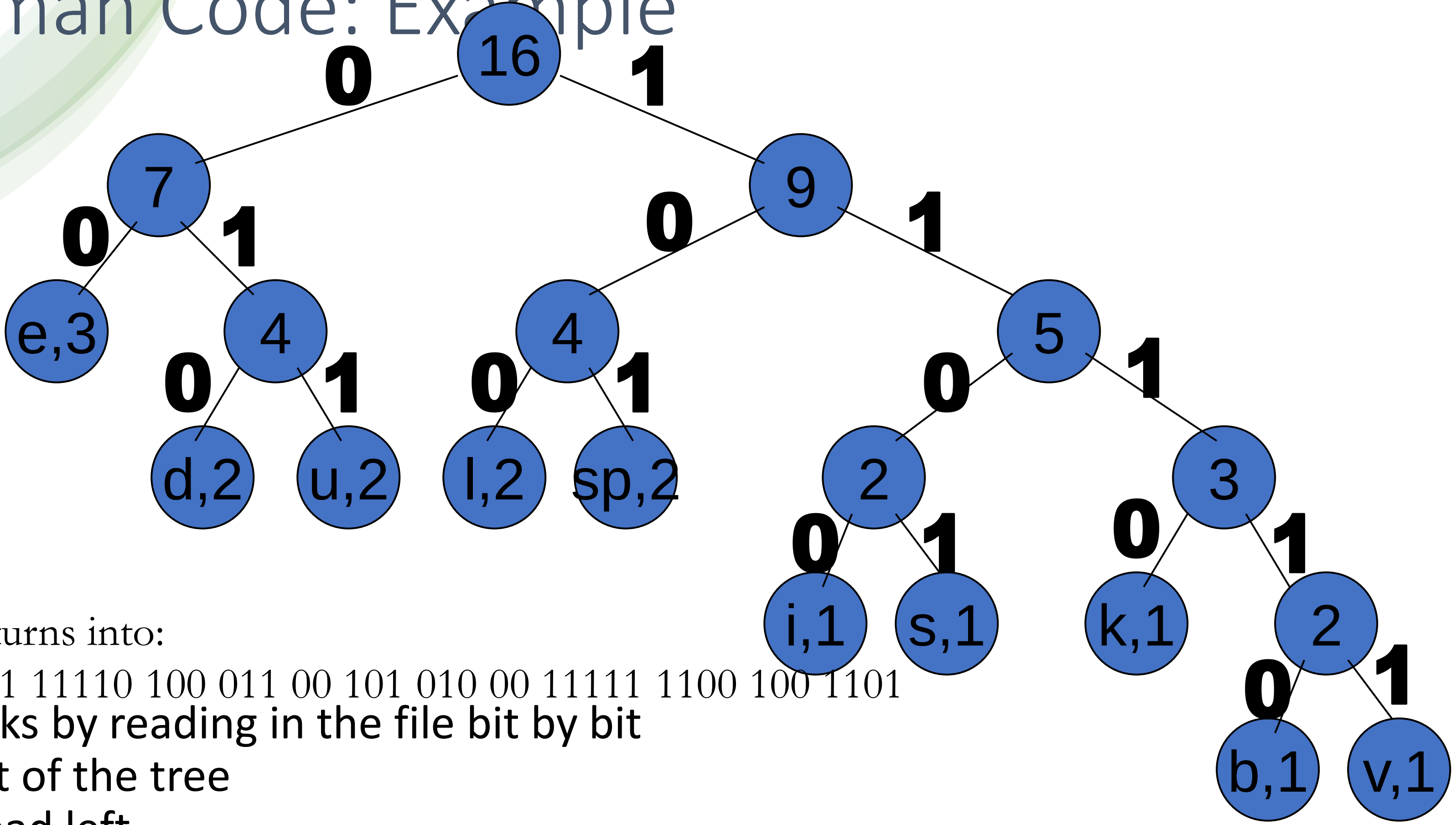
Thus, it takes 7 bytes of space compared to 16 characters * 1 byte/char = 16 bytes uncompressed

Files are stored as sequences of whole bytes, so in cases with the remaining digits of the last byte are filled with zeroes

Or Insert/encode pseudo-end-of-file-marker - indicates where the coding stopped

Example by Dr. Junzhou Huang

Huffman Code: Example



e	00
d	010
u	011
l	100
sp	101
i	1100
s	1101
k	1110
b	11110
v	11111

"duke blue devils" turns into:

010 011 1110 00 101 11110 100 011 00 101 010 00 11111 1100 100 1101

Uncompressing works by reading in the file bit by bit

- Start at the root of the tree
- If a 0 is read, head left
- If a 1 is read, head right
- When a leaf is reached decode that character and start over again at the root of the tree

Thus, we need to save Huffman table information as a header in the compressed file

- Doesn't add a significant amount of size to the file for large files (which are the ones you want to compress anywa
- Or we could use a fixed universal set of codes/frequencies

The END!