
Design and Analysis of Algorithms

CSE 5311

Lecture 17 Greedy algorithms: Huffman
Coding

Junzhou Huang, Ph.D.

Department of Computer Science and Engineering

Data Compression

- Suppose we have 1000000000 (1G) character data file that we wish to include in an email.
- Suppose file only contains 26 letters $\{a, \dots, z\}$.
- Suppose each letter a in $\{a, \dots, z\}$ occurs with frequency f_a .
- Suppose we encode each letter by a binary code
- If we use a fixed length code, we need 5 bits for each character
- The resulting message length is $5(f_a + f_b + \dots + f_z)$
- **Can we do better?**

Huffman Coding

- The basic idea
 - Instead of storing each character in a file as an 8-bit ASCII value, we will instead store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits
 - On average this should decrease the filesize (usually $\frac{1}{2}$)
- Huffman codes can be used to compress information
 - Like WinZip – although WinZip doesn't use the Huffman algorithm
 - JPEGs do use Huffman as part of their compression process

Data Compression: A Smaller Example

- Suppose the file only has 6 letters {a,b,c,d,e,f} with frequencies

<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	
.45	.13	.12	.16	.09	.05	
000	001	010	011	100	101	Fixed length
0	101	100	111	1101	1100	Variable length

- Fixed length $3G=3000000000$ bits
- **Variable length**

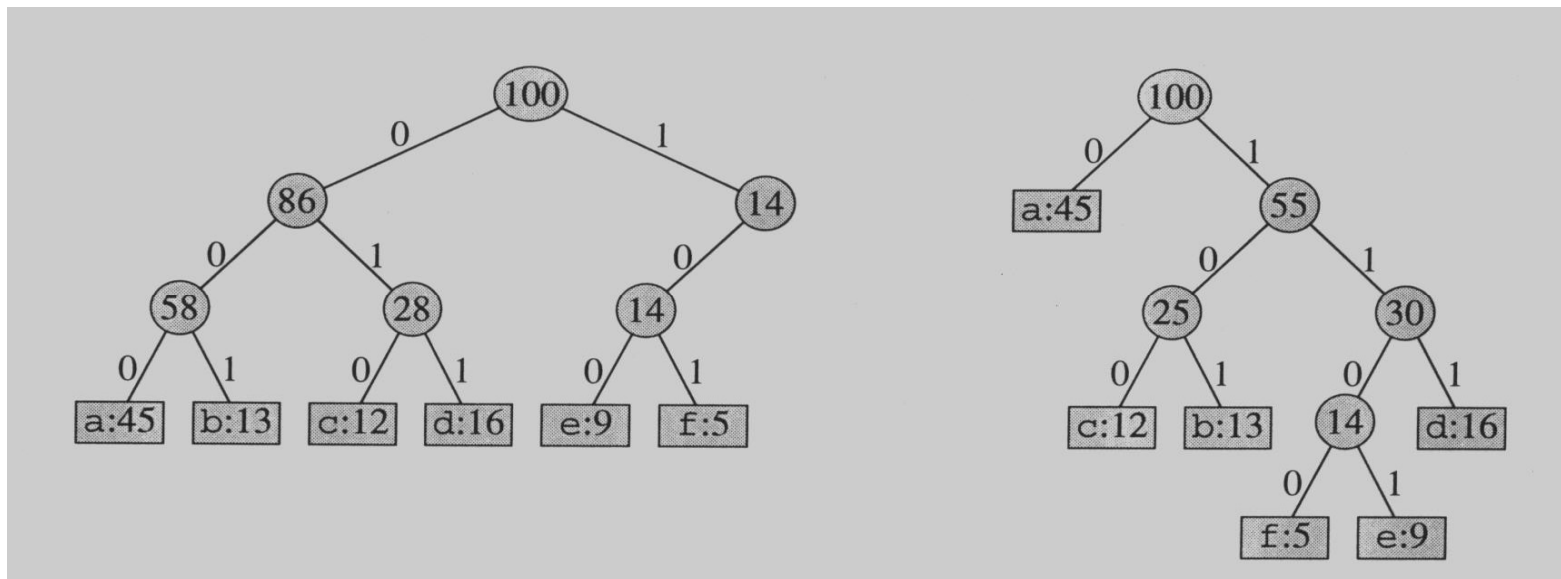
$$(.45 \cdot 1 + .13 \cdot 3 + .12 \cdot 3 + .16 \cdot 3 + .09 \cdot 4 + .05 \cdot 4) = 2.24G$$

How to decode?

- At first it is not obvious how decoding will happen, but this is possible if we use prefix codes

Prefix Codes

- 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
- 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
- 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

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

where $f(c)$ is the freq of c , $d_T(c)$ is the tree depth of c , which corresponds to the code length of c

Optimal Prefix Coding Problem

- Input: Given a set of n letters (c_1, \dots, c_n) with frequencies (f_1, \dots, f_n) .
- Construct a full binary tree T to define a prefix code that **minimizes** the average code length

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

Greedy Algorithms

- Many optimization problems can be solved using a greedy approach
 - The basic principle is that local optimal decisions may be used to build an optimal solution
 - But the greedy approach may not always lead to an optimal solution overall for all problems
 - The key is knowing which problems will work with this approach and which will not
- We will study
 - **The problem of generating Huffman codes**

Greedy algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
 - My everyday examples:
 - Driving in Los Angeles, NY, or Boston for that matter
 - Playing cards
 - Invest on stocks
 - Choose a university
 - The hope: a locally optimal choice will lead to a globally optimal solution
 - For some problems, it works
- Greedy algorithms tend to be easier to code

David Huffman's idea

- A Term paper at MIT

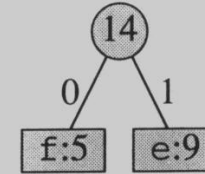


- Build the tree (code) bottom-up in a greedy fashion
- Origami aficionado

Building the Encoding Tree

(a) f:5 e:9 c:12 b:13 d:16 a:45 \

(b) c:12 b:13

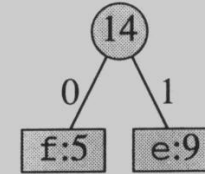


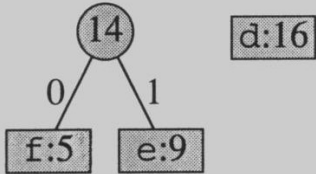
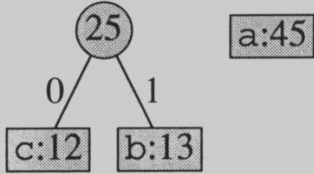
d:16 a:45

Building the Encoding Tree

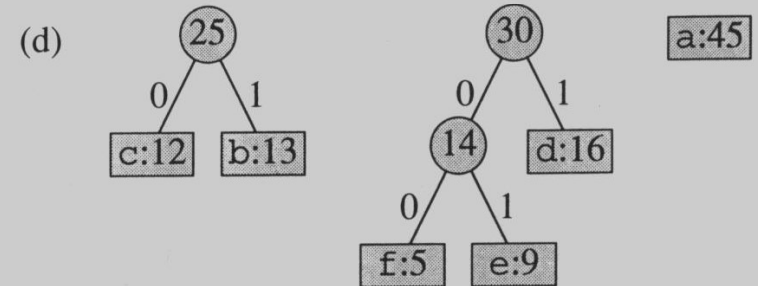
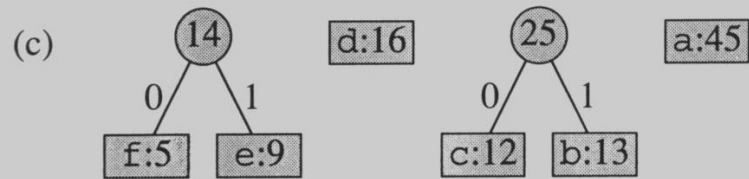
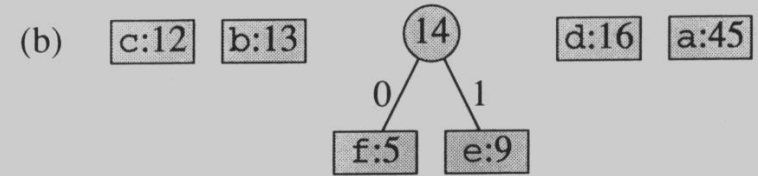
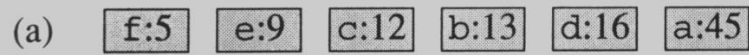
(a) f:5 e:9 c:12 b:13 d:16 a:45

(b) c:12 b:13 d:16 a:45



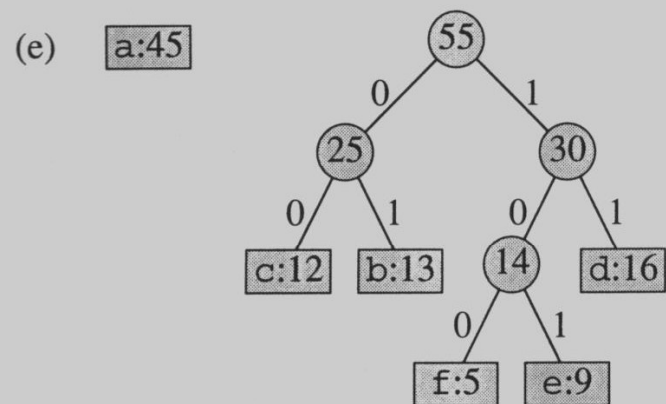
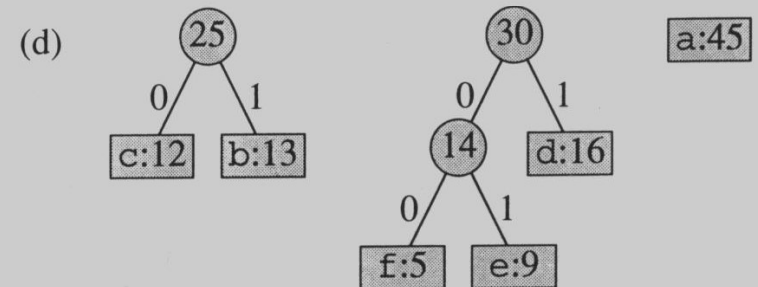
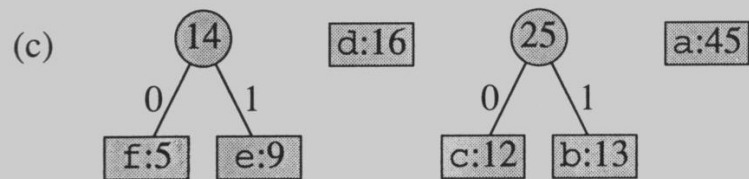
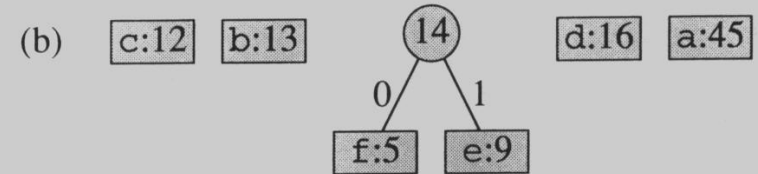
(c)  

Building the Encoding Tree

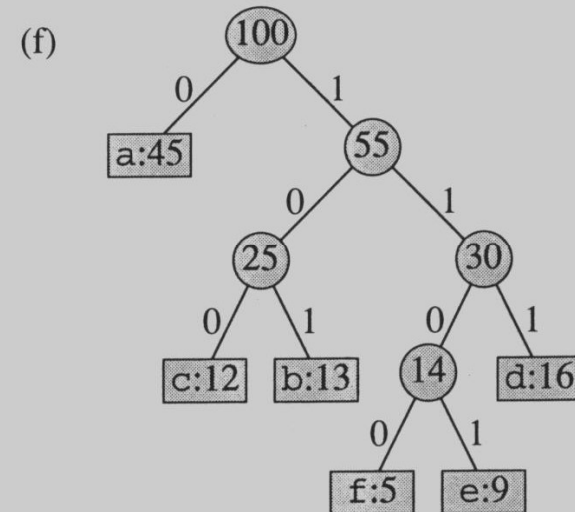
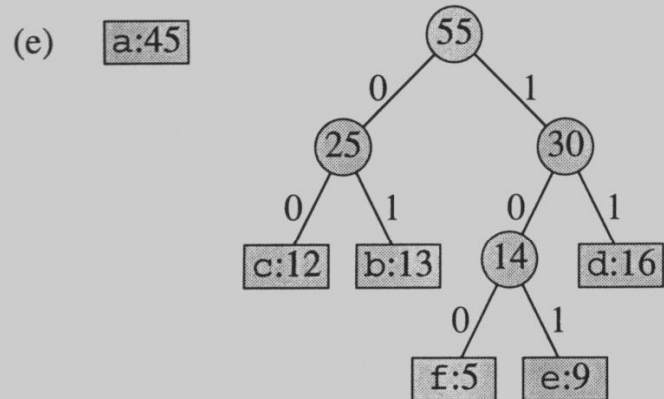
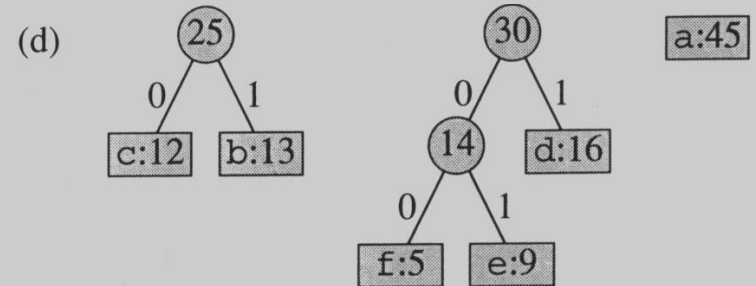
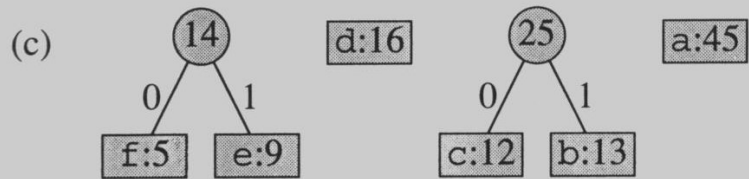
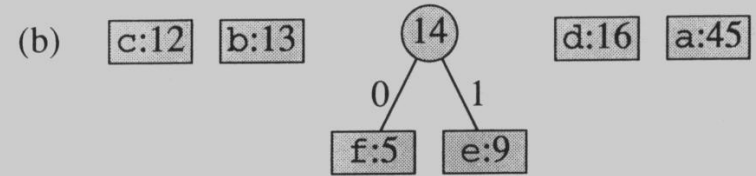
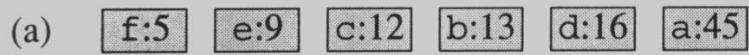


Building the Encoding Tree

(a) f:5 e:9 c:12 b:13 d:16 a:45



Building the Encoding Tree



The Algorithm

HUFFMAN(C)

1 $n \leftarrow |C|$

2 $Q \leftarrow C$

Q : priority queue

3 **for** $i \leftarrow 1$ **to** $n - 1$

4 **do** allocate a new node z

5 $left[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$

6 $right[z] \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$

7 $f[z] \leftarrow f[x] + f[y]$

8 $\text{INSERT}(Q, z)$

9 **return** $\text{EXTRACT-MIN}(Q)$

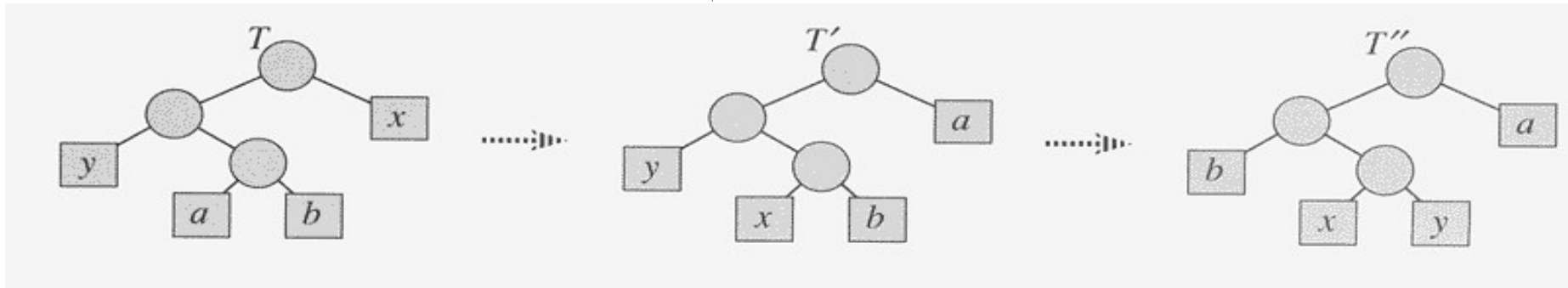
▷ Return the root of the tree.

- 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

Correctness of Huffman's Algorithm

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree

Lemma 16.2

- Without loss of generality, assume $f[a] \leq f[b]$ and $f[x] \leq f[y]$
- The cost difference between T and T' is

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C} f(c)d_{T'}(c) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\ &= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \\ &\geq 0 \end{aligned}$$

$B(T'') \leq B(T)$, but T is optimal,

$$B(T) \leq B(T'') \rightarrow B(T'') = B(T)$$

Therefore T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth

Correctness of Huffman's Algorithm

Lemma 16.3

Let C be a given alphabet with frequency $f[c]$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with characters x, y removed and (new) character z added, so that $C' = C - \{x, y\} \cup \{z\}$; define f for C' as for C , except that $f[z] = f[x] + f[y]$. Let T' be any tree representing an optimal prefix code for the alphabet C' . Then the tree T , obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C .

• Observation: $B(T) = B(T') + f[x] + f[y] \rightarrow B(T') = B(T) - f[x] - f[y]$

– For each $c \in C - \{x, y\} \rightarrow d_T(c) = d_{T'}(c) \rightarrow f[c]d_T(c) = f[c]d_{T'}(c)$

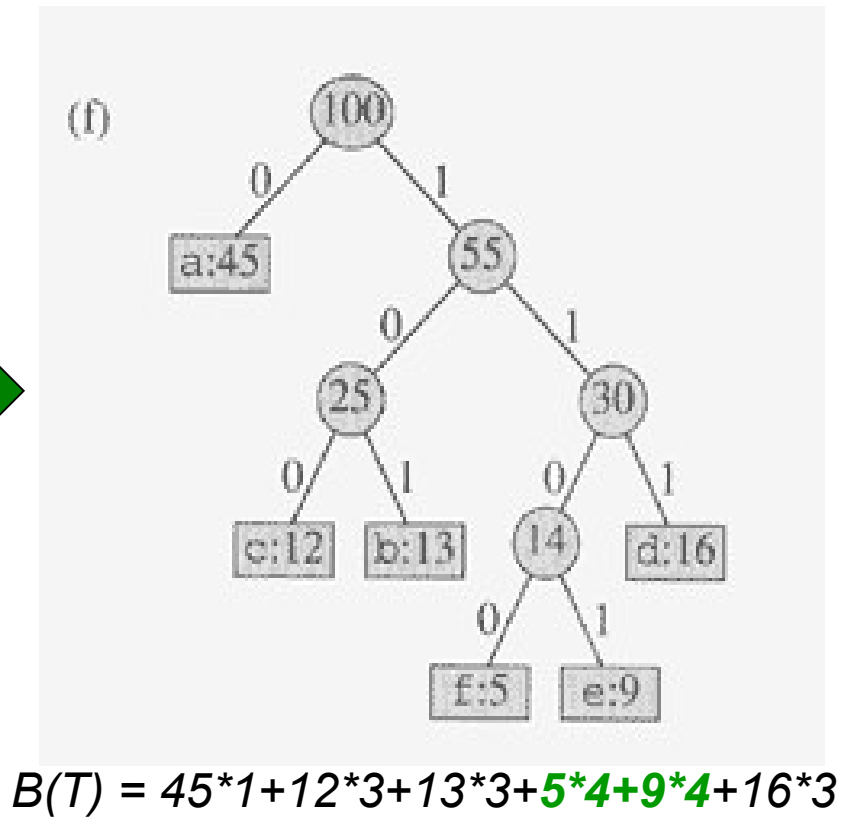
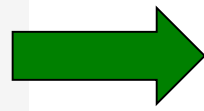
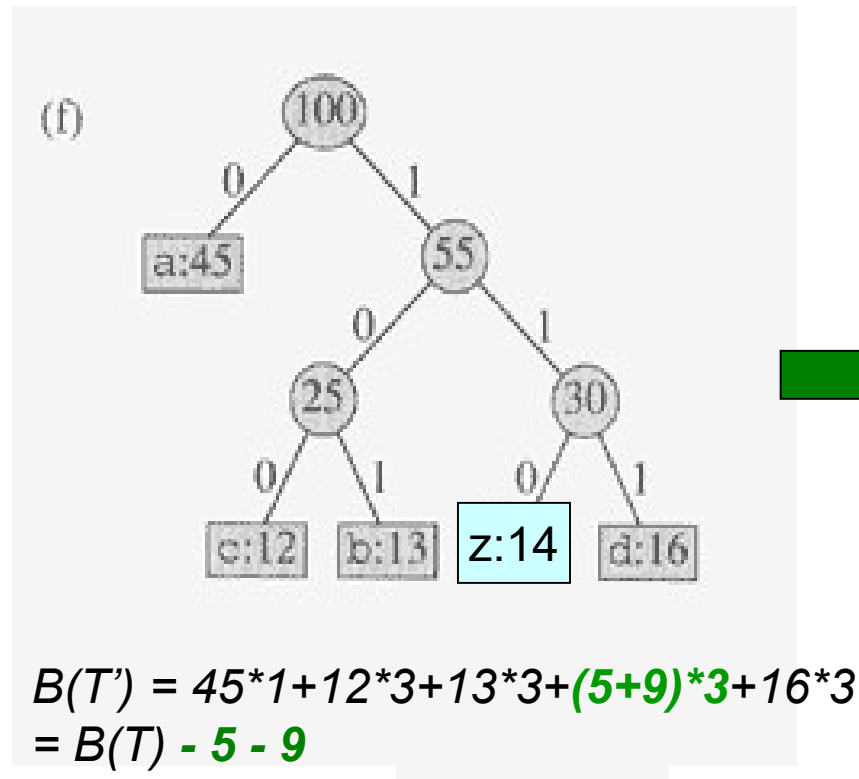
– $d_T(x) = d_T(y) = d_{T'}(z) + 1$

– $f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1) = f[z]d_{T'}(z) + (f[x] + f[y])$

Theorem 16.4

Procedure HUFFMAN produces an optimal prefix code.

$$B(T') = B(T) - f[x] - f[y]$$

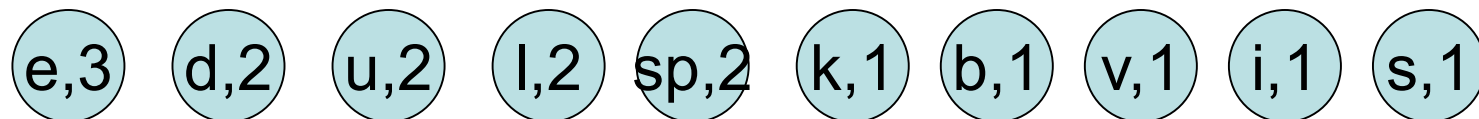


Proof of Lemma 16.3

- Prove by contradiction.
- Suppose that T does not represent an optimal prefix code for C . Then there exists a tree T'' such that $B(T'') < B(T)$.
- Without loss of generality, by Lemma 16.2, T'' has x and y as siblings. Let T''' be the tree T'' with the common parent x and y replaced by a leaf with frequency $f[z] = f[x] + f[y]$. Then
- $B(T''') = B(T'') - f[x] - f[y] < B(T) - f[x] - f[y] = B(T)$
 - T''' is better than T \rightarrow contradiction to the assumption that T is an optimal prefix code for C

Example: Huffman Coding

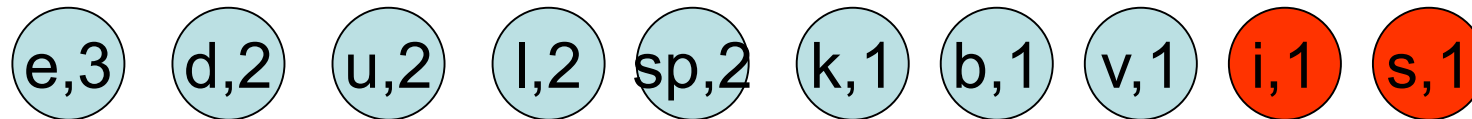
- As an example, lets take the string:
 “duke blue devils”
- We first do a frequency count of the characters:
 ➤ e:3, d:2, u:2, l:2, space:2, k:1, b:1, v:1, i:1, s:1
- Next we use a Greedy algorithm to build up a Huffman Tree
 - We start with nodes for each character



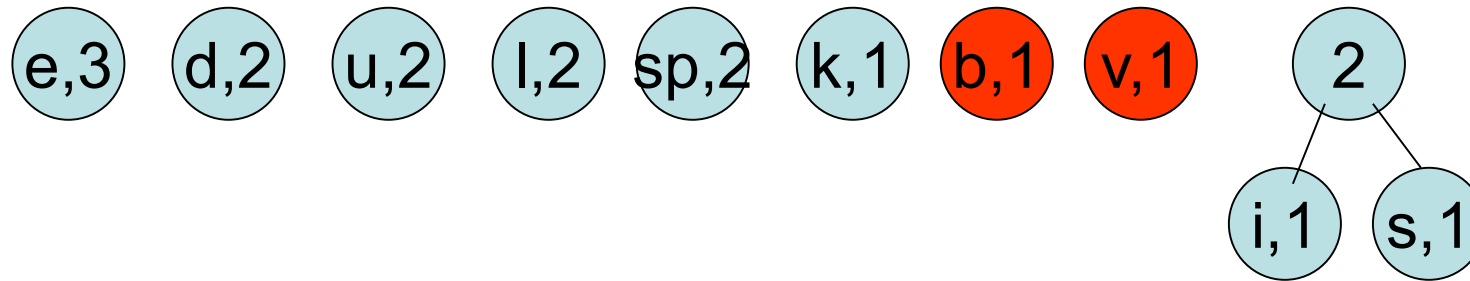
Example: Huffman Coding

- We then pick the nodes with the smallest frequency and combine them together to form a new node
 - The selection of these nodes is the Greedy part
- The two selected nodes are removed from the set, but replace by the combined node
- This continues until we have only 1 node left in the set

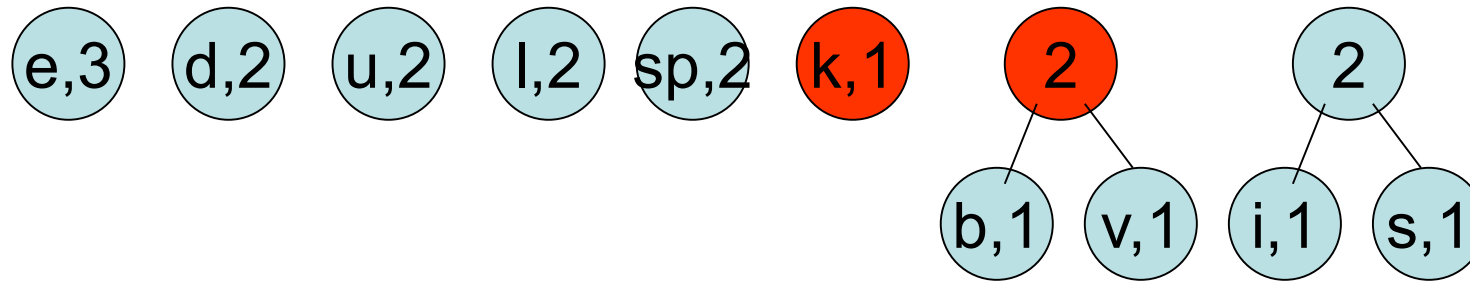
Example: Huffman Coding



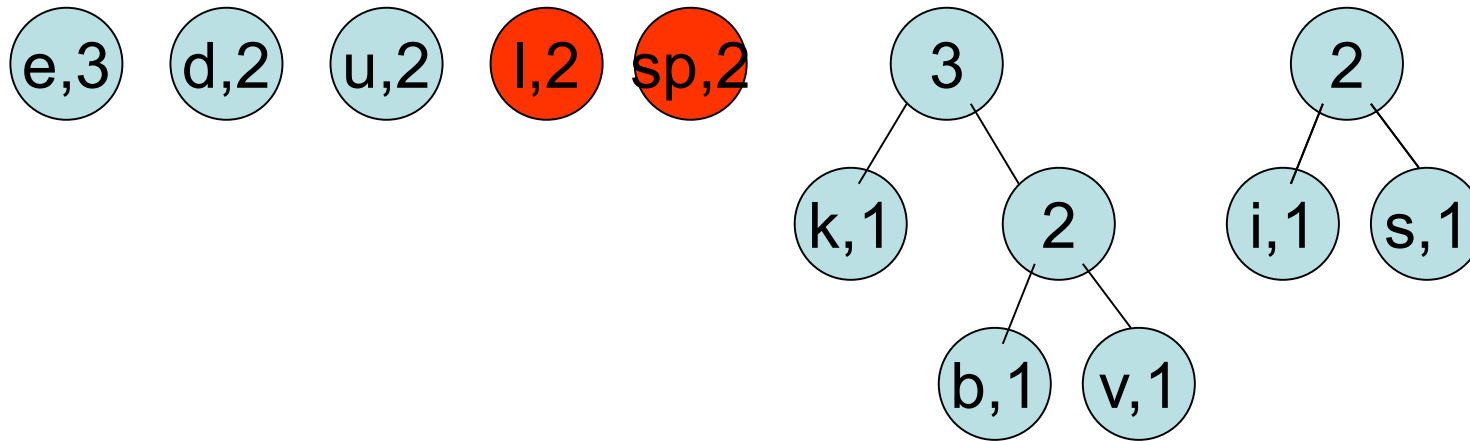
Example: Huffman Coding



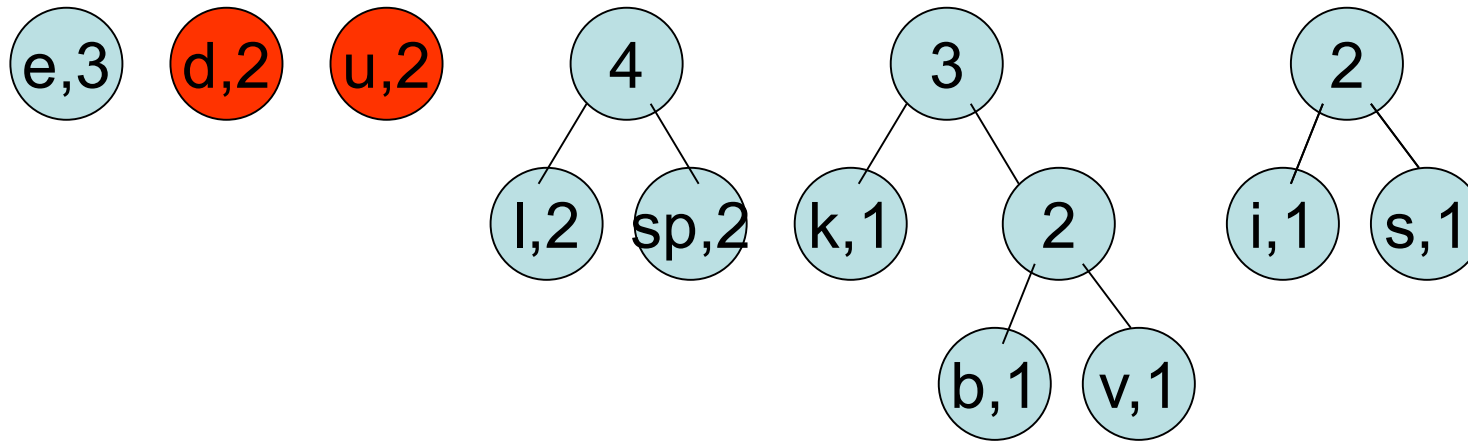
Example: Huffman Coding



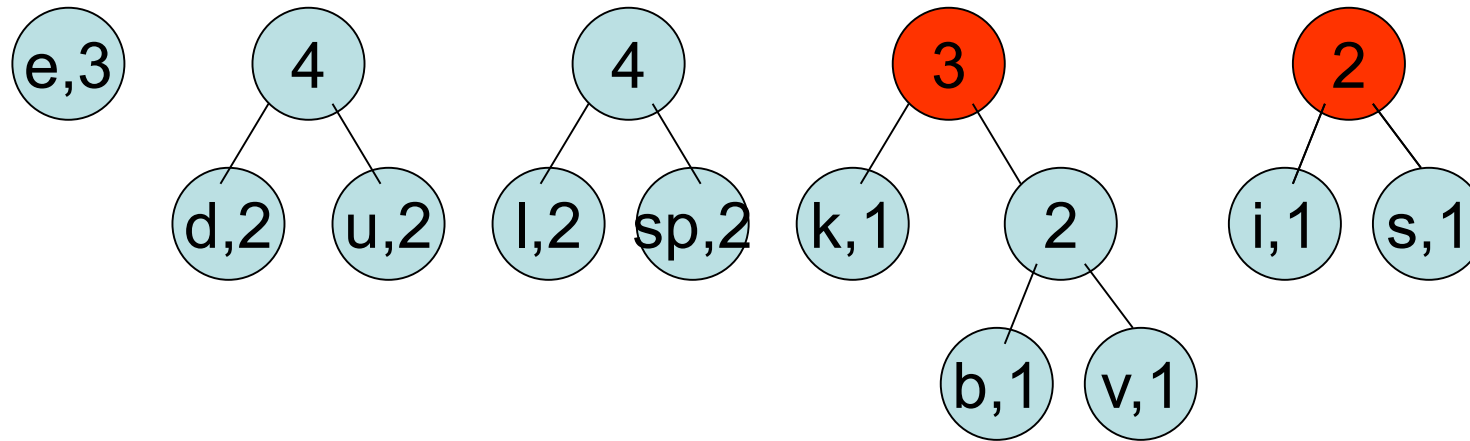
Example: Huffman Coding



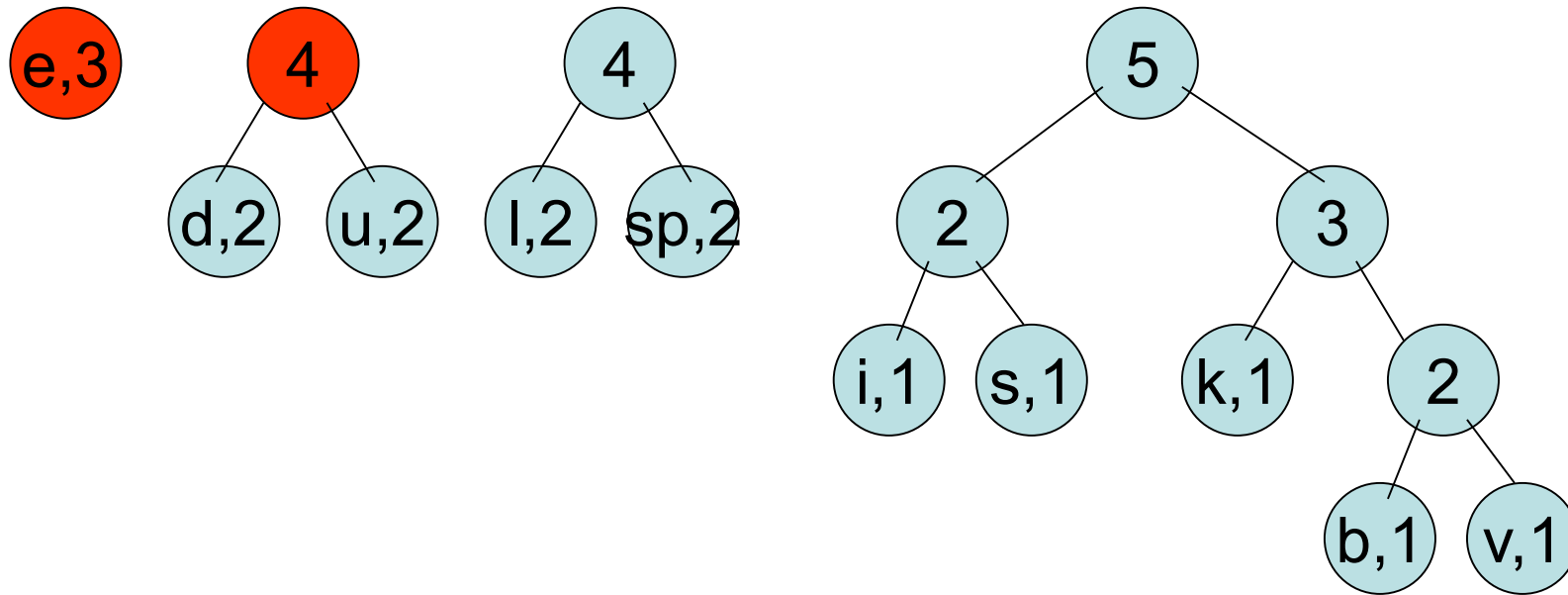
Example: Huffman Coding



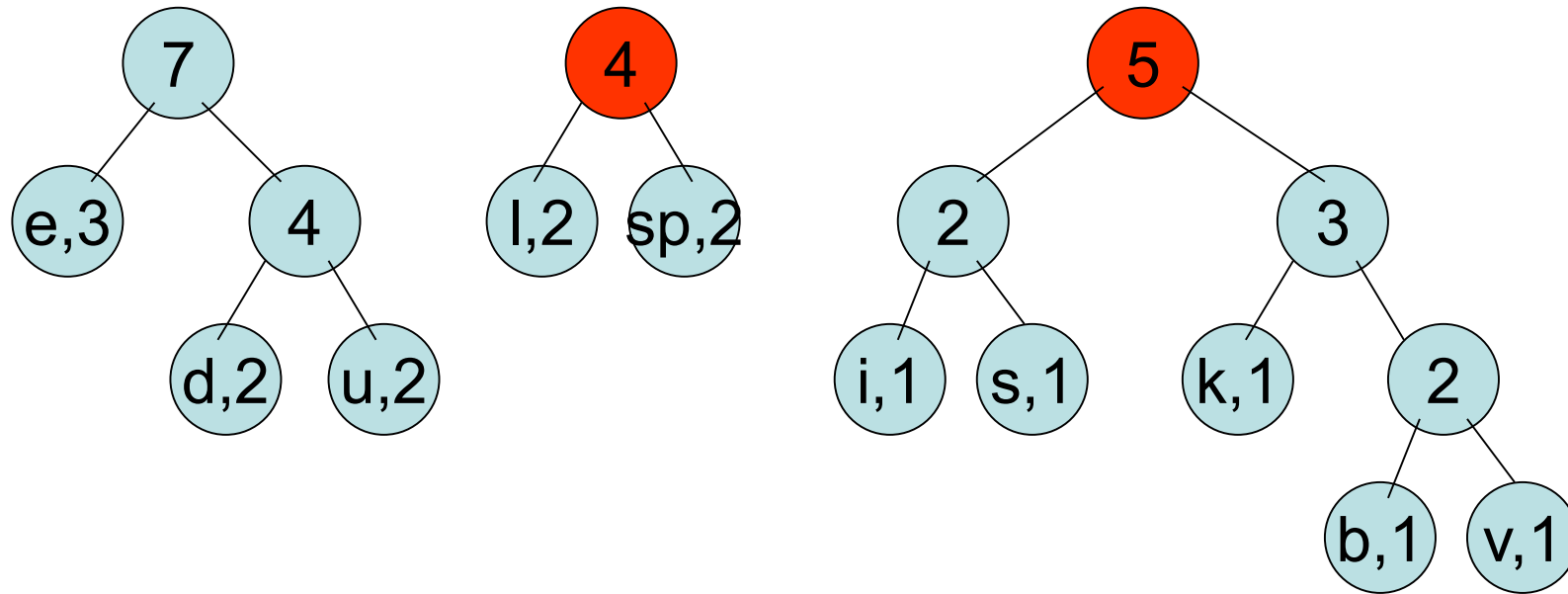
Example: Huffman Coding



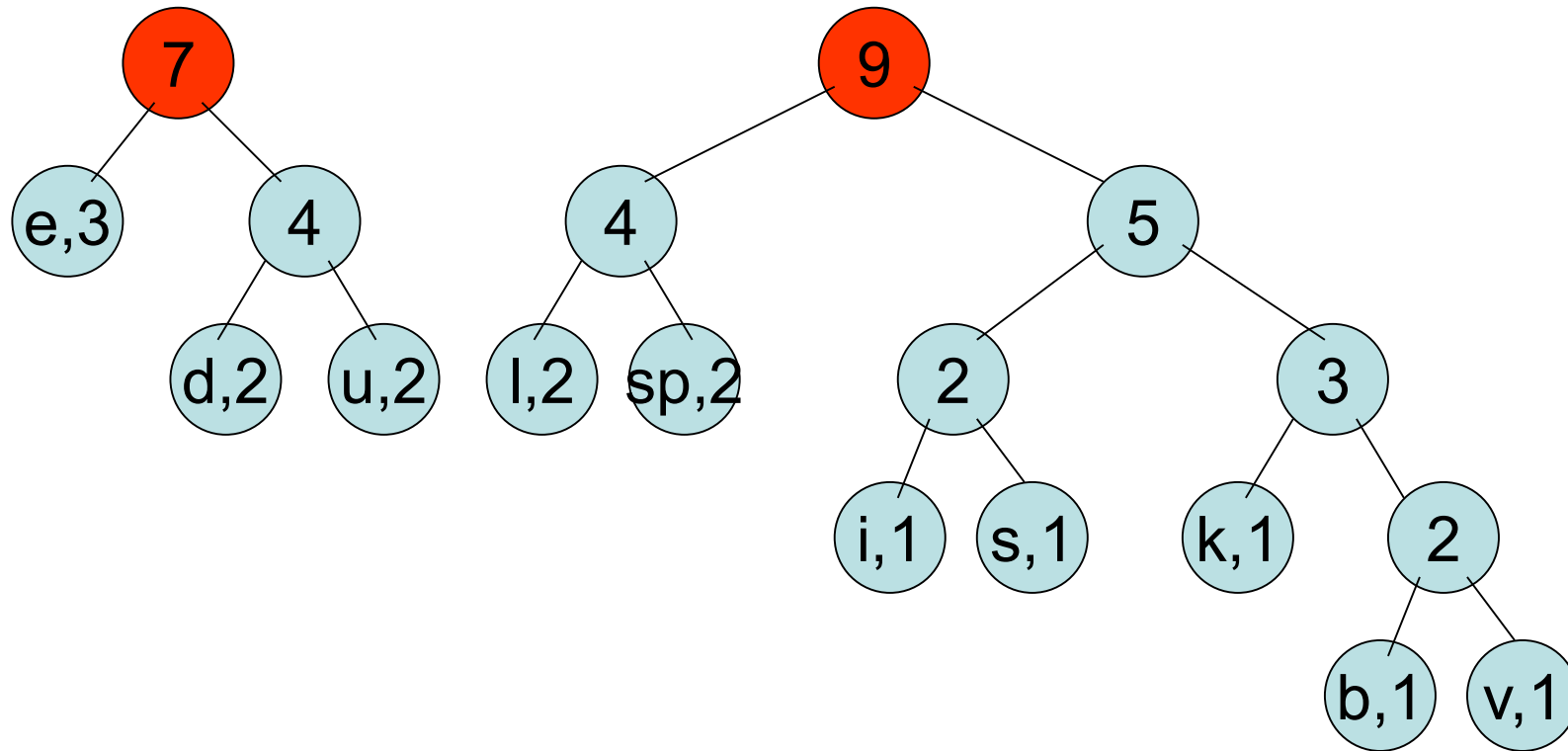
Example: Huffman Coding



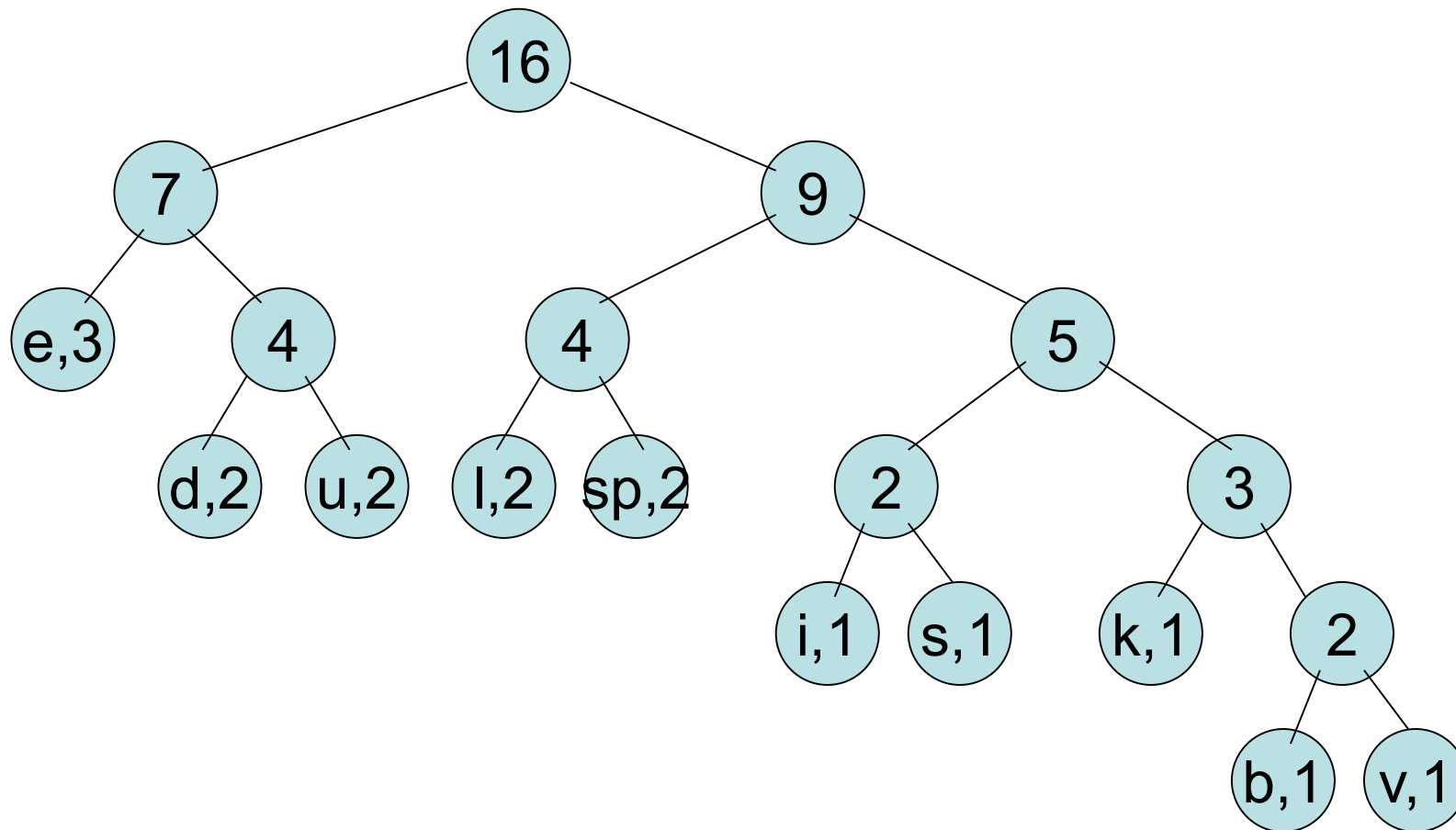
Example: Huffman Coding



Example: Huffman Coding



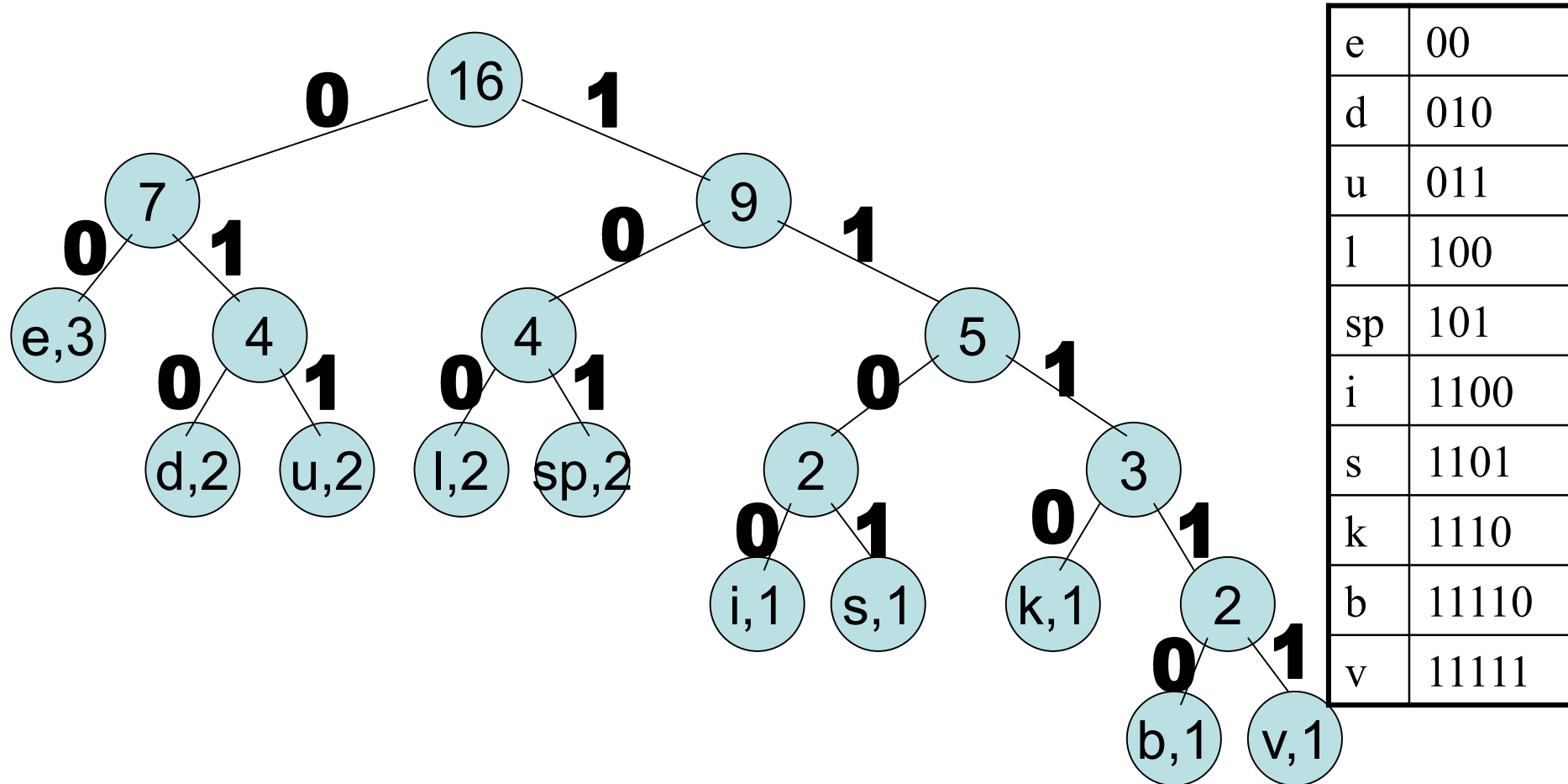
Example: Huffman Coding



Example: Huffman Coding

- Now we assign codes to the tree by placing a 0 on every left branch and a 1 on every right branch
- A traversal of the tree from root to leaf give the Huffman code for that particular leaf character
- Note that no code is the prefix of another code

Example: Huffman Coding



Example: Huffman Coding

- 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

Example: Huffman Coding

- 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 anyway)
 - Or we could use a fixed universal set of codes/frequencies