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,…,z}.
- Suppose each letter a in {a,…,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 $5\left(f_a + f_b + \cdots f_z\right)$

- **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 ½)
- 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**

a	b	c	d	e	f	
.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**

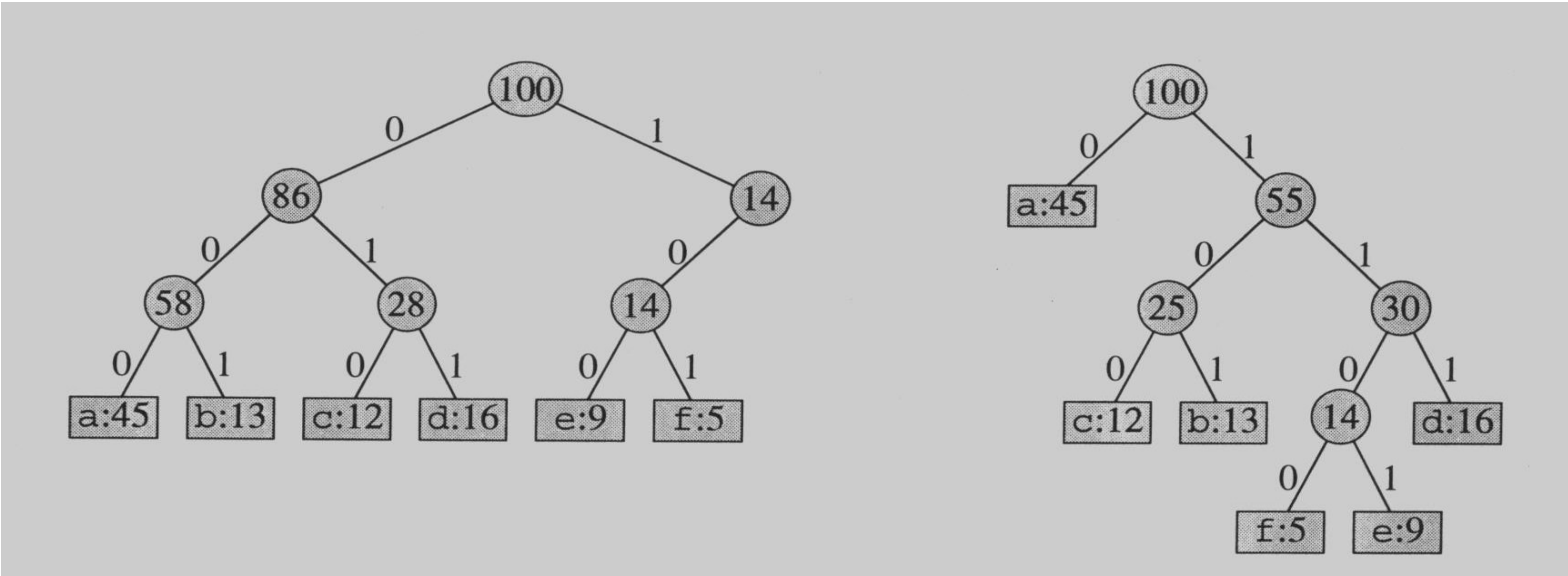
$$\left(.45\bullet 1+.13\bullet 3+.12\bullet 3+.16\bullet 3+.09\bullet 4+.05\bullet 4\right)=2.24G$$

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 ($c_1,\ldots, c_n$) with frequencies ($f_1,\ldots,f_n$).

- 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

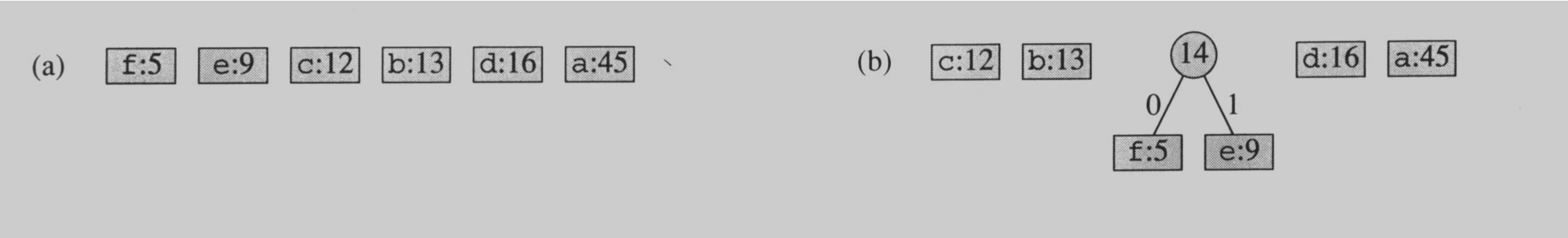
- Build the tree (code) bottom-up in a greedy fashion

- Origami aficionado

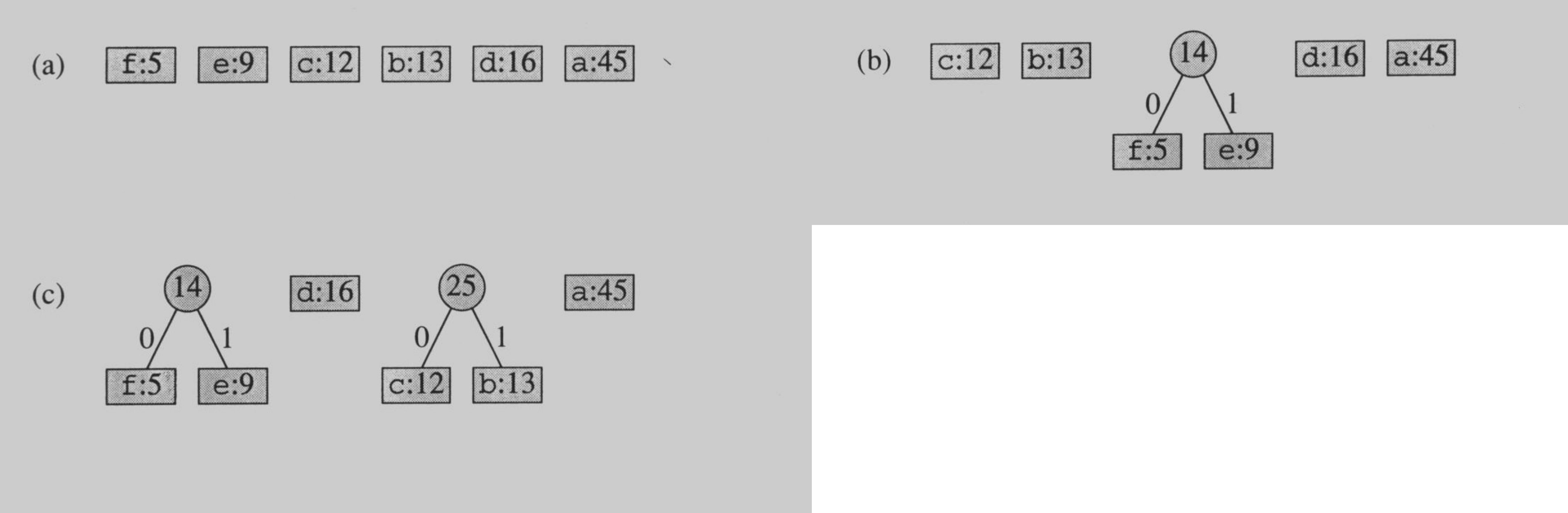
Story behind

- David Huffman was a student in an EE course in 1951. His 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 so he started working on the term paper. The topic of the paper was to find the most efficient (optimal) code.
- The fact: it was an open research problem
- Huffman spent a lot of time on the problem and was ready to give up when the solution suddenly came to him. It had the lowest possible average message length.
- Later Huffman said that likely he would not have even attempted the problem if he had known that it was an open research problem

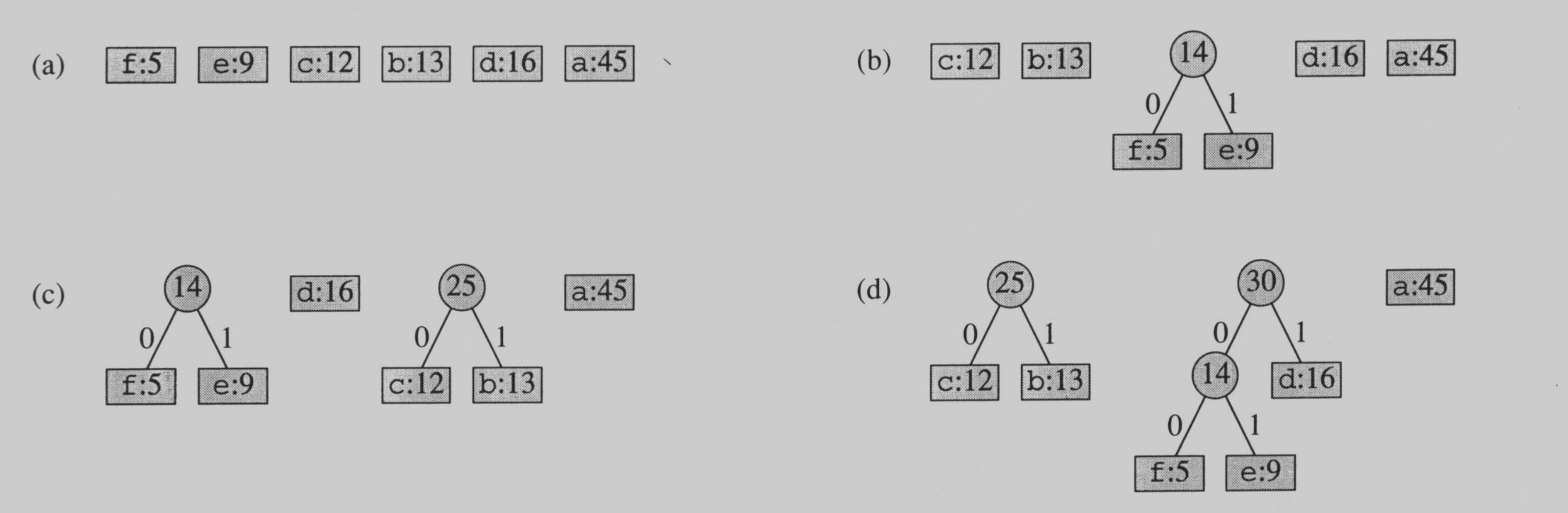
Building the Encoding Tree



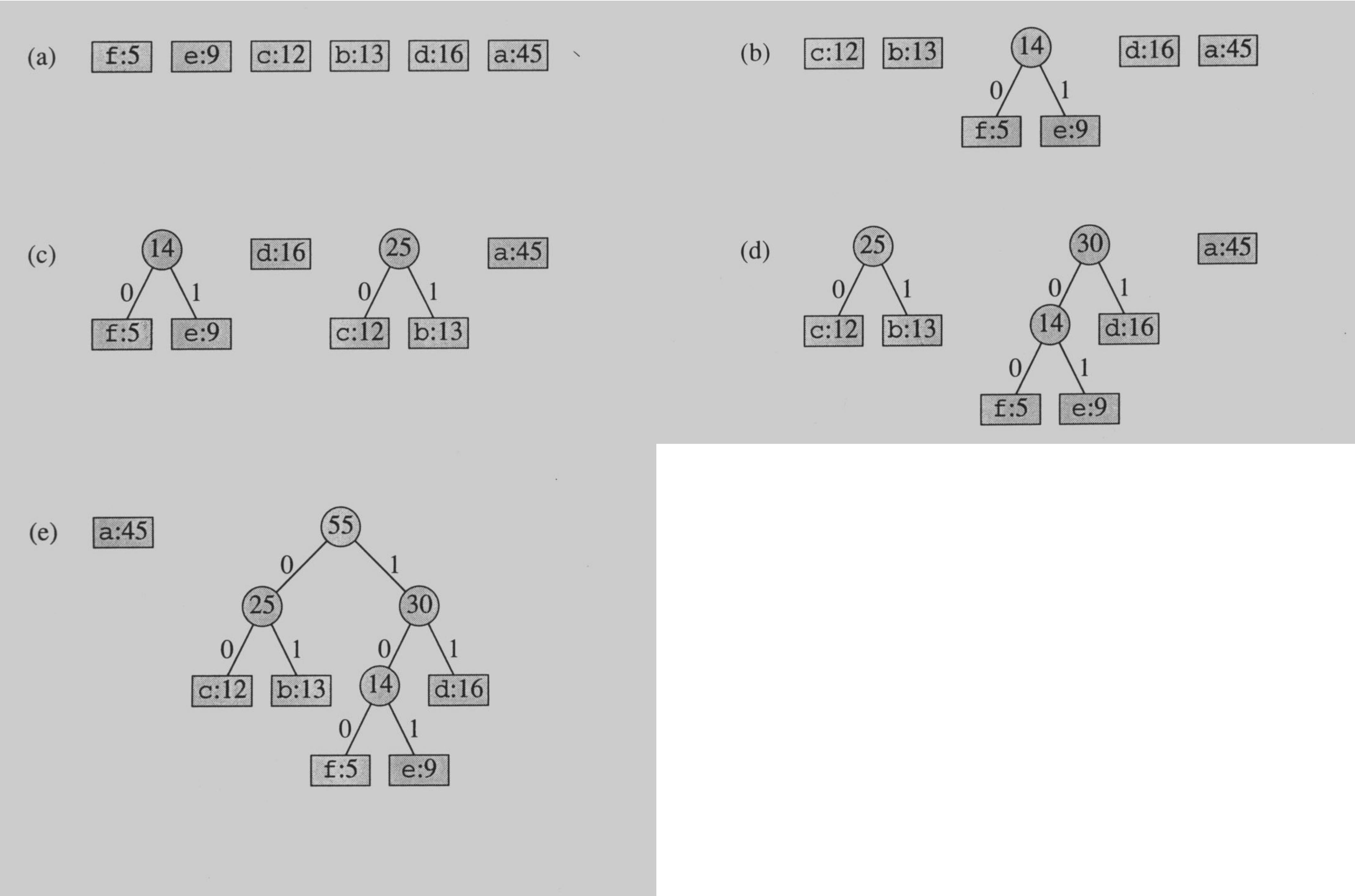
Building the Encoding Tree



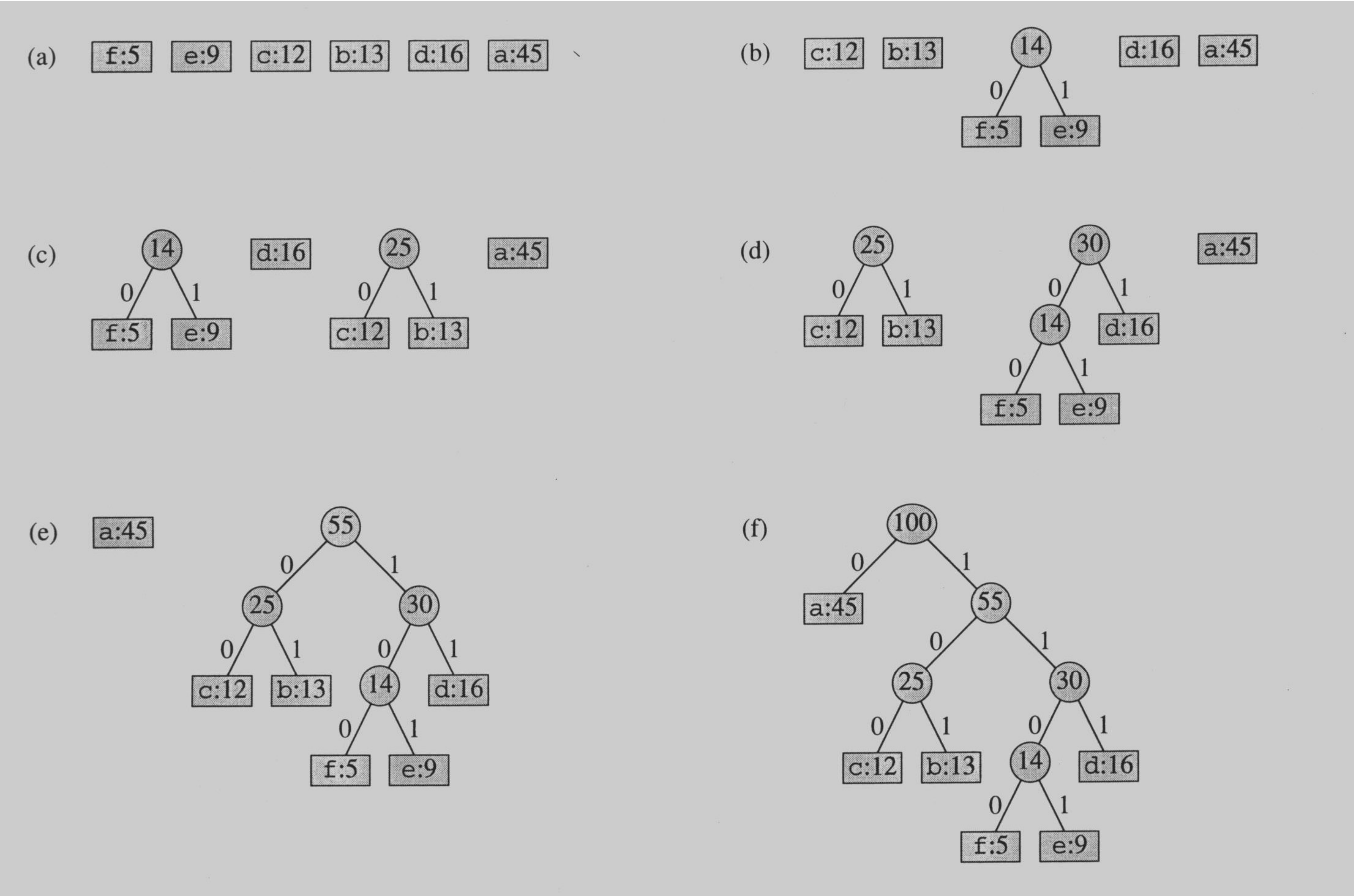
Building the Encoding Tree



Building the Encoding Tree



Building the Encoding Tree



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.
```

Q **:priority queue**

- 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

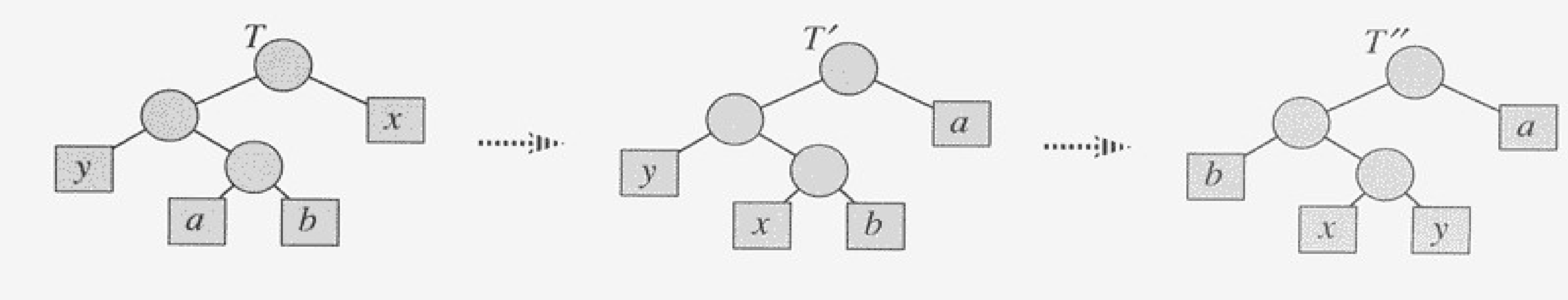
Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof:

The idea of the proof is to take the tree T representing an arbitrary optimal prefix code and modify it to make a tree representing another optimal prefix code such that the characters x and y appear as sibling leaves of maximum depth in the new tree.

If we can construct such a tree, then the codewords for x and y will have the same length and differ only in the last bit.

Lemma 16.2



An illustration of the key step in the proof of Lemma 16.2.

In the optimal tree T , leaves *a* and *b* are two siblings of **maximum depth**. Leaves x and y are the two characters with **the lowest frequencies**; they appear in arbitrary positions in T .

Assuming that x ≠ b, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T".

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

Lemma 16.2

- Let a and b be two characters that are sibling leaves of maximum depth in T. Assume $f[a] \leq f[b]$ and $f[x] \leq f[y]$
- Since f(x) and f(y) are the two lowest leaf frequencies, we assume f[x]<f[y]). Therefore, we have f[x]≤f[a]; f[y] ≤ f[b]
- As shown in the previous page, we exchange the positions in T of a and x to produce a tree T', and then we exchange the positions in T' of b and y to produce a tree T'' in which x and y are sibling leaves of maximum depth.
- The cost difference between T and T' is

$$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)$$

$$= f[x]d_T(x) + f[a]d_T(a) - f[x]d_{T'}(x) - f[a]d_{T'}(a)$$

$$= f[x]d_T(x) + f[a]d_T(a) - f[x]d_T(a) - f[a]d_T(x)$$

$$= (f[a] - f[x])(d_T(a) - d_T(x))$$

≥ 0 **Why?**

Lemma 16.2

- Term 1: f[a]-f[x] $\geq$ 0: x is a minimum-frequency leaf
- Term 2: $d_T(a) - d_T(x) \geq 0$: a is a leaf of maximum depth in T
- Therefore, B(T)-B(T')=(f[a]-f[x])($d_T(a) - d_T(x)$) ≥ 0
- Similarly, exchanging y and b does not increase the cost, then, B(T')-B(T'') ≥ 0

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

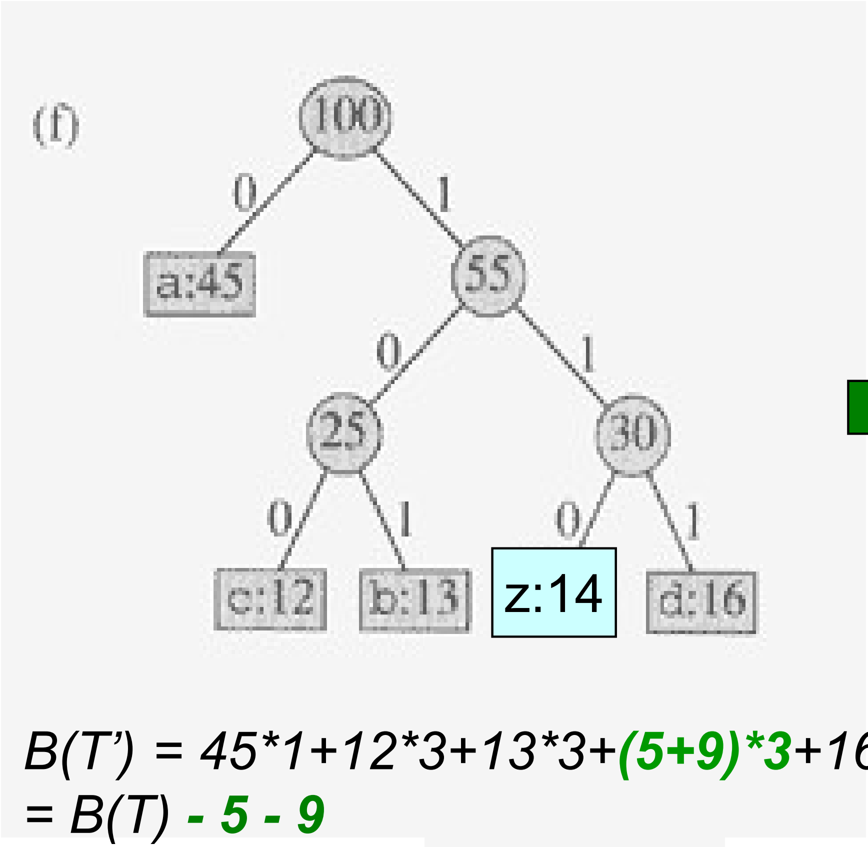
Correctness of Huffman's Algorithm

Lemma 16.3

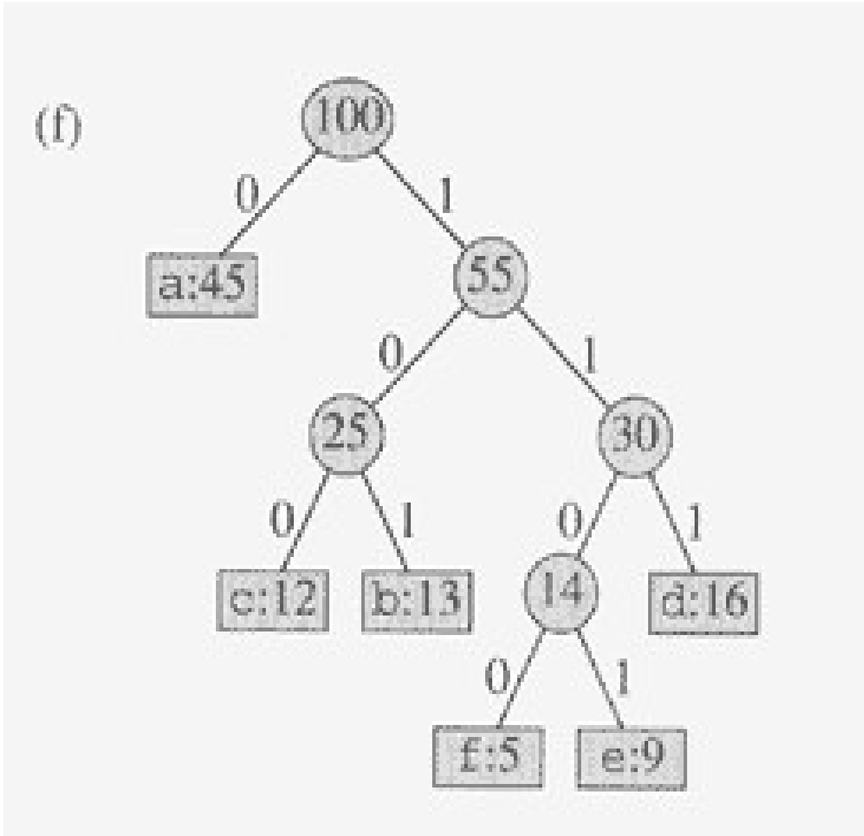
Let C be a given alphabet with frequency $f[c]$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with characters x, y removed and (new) character z added, so that $C' = C - \{x, y\} \cup \{z\}$; define f for C' as for C, except 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.

- We first show that: B(T) = B(T') + f[x] + f[y] ➔ B(T') = B(T)-f[x]-f[y]
  - For each $c \in C - \{x, y\}$ ➔ $d_T(c) = d_{T'}(c)$ ➔ $f[c]d_T(c) = f[c]d_{T'}(c)$
  - $d_T(x) = d_T(y) = d_{T'}(z) + 1$
  - $f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1)$
  - $= f[z]d_{T'}(z) + (f[x] + f[y])$

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



$B(T') = 45*1+12*3+13*3+(5+9)*3+16*3$
$= B(T) - 5 - 9$



$B(T) = 45*1+12*3+13*3+5*4+9*4+16*3$

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

$$< B(T) - f[x] - f[y]$$

$$= B(T')$$

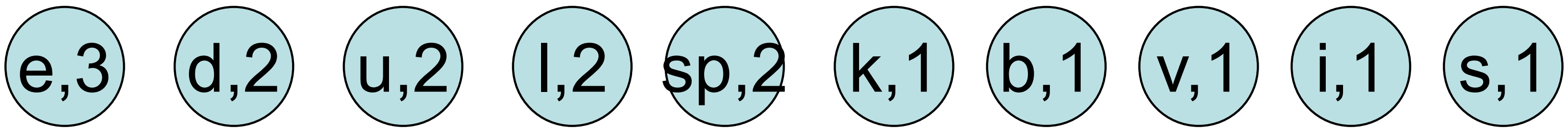
  – T''' is better than T' ➔ contradiction to the assumption that T' is an optimal prefix code for C'. Thus, T must represent an optimal prefix code for the alphabet C.

Example: Huffman Coding

- As an example, lets take the string:

  "duke blue devils"

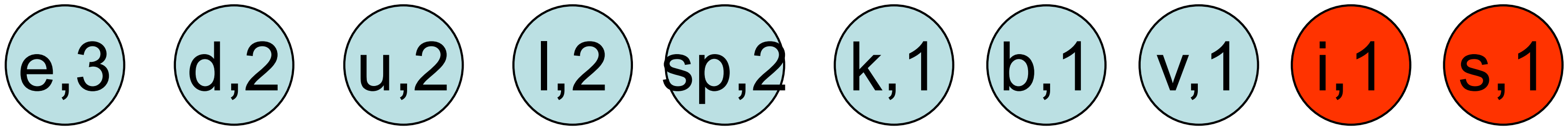
- We first to a frequency count of the characters:
  - e:3, d:2, u:2, l:2, space:2, k:1, b:1, v:1, i:1, s:1
- Next we use a Greedy algorithm to build up a Huffman Tree
  - We start with nodes for each character



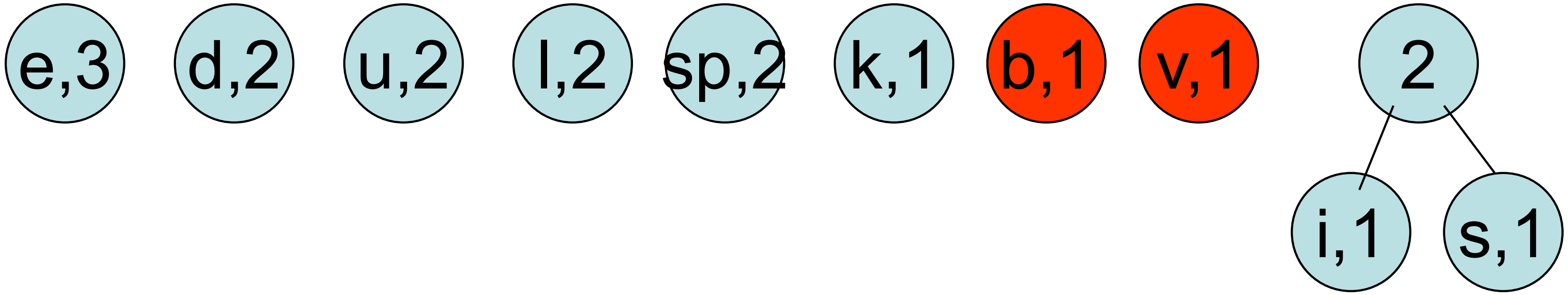
Example: Huffman Coding

- 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

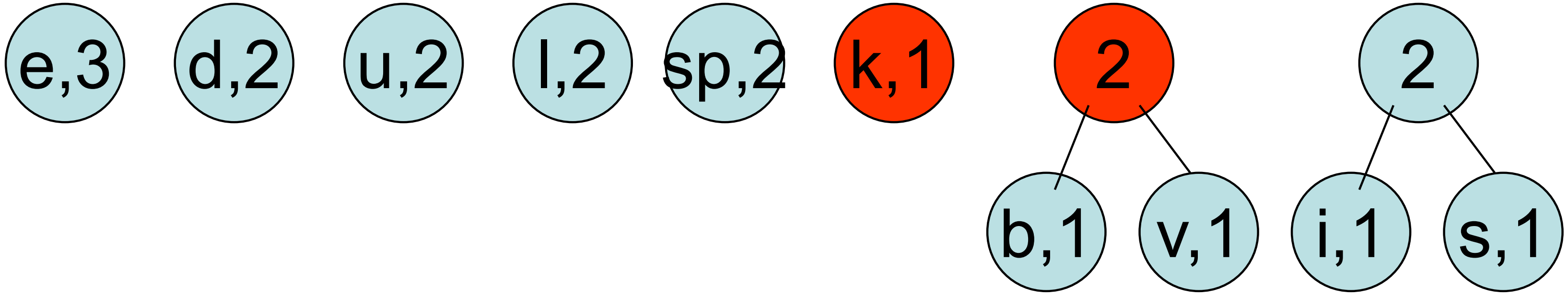
Example: Huffman Coding



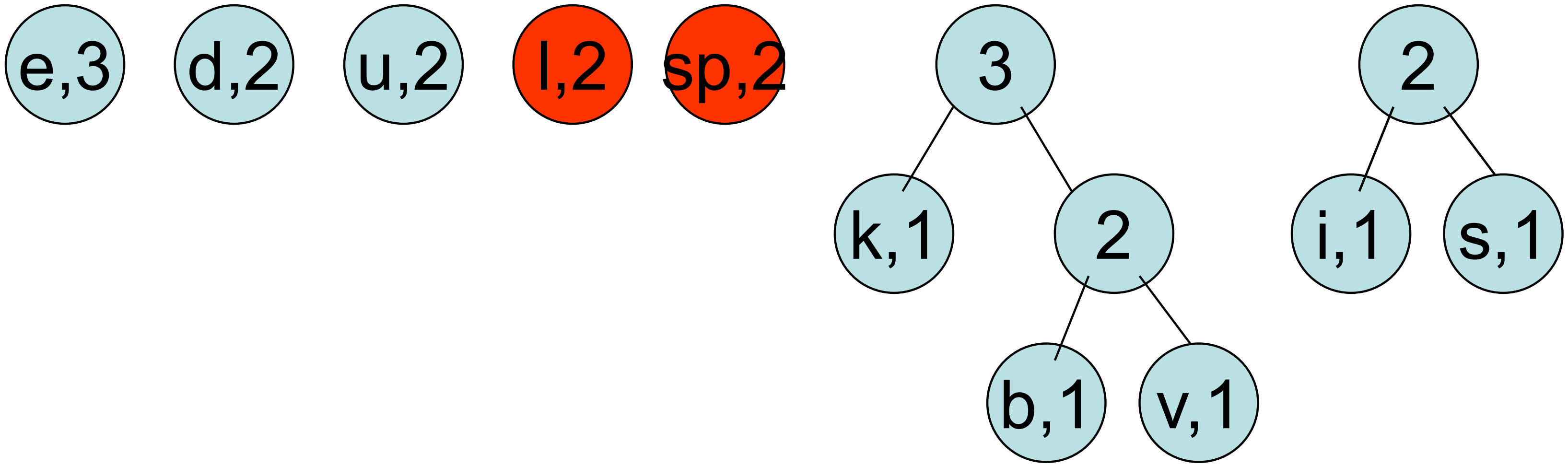
Example: Huffman Coding



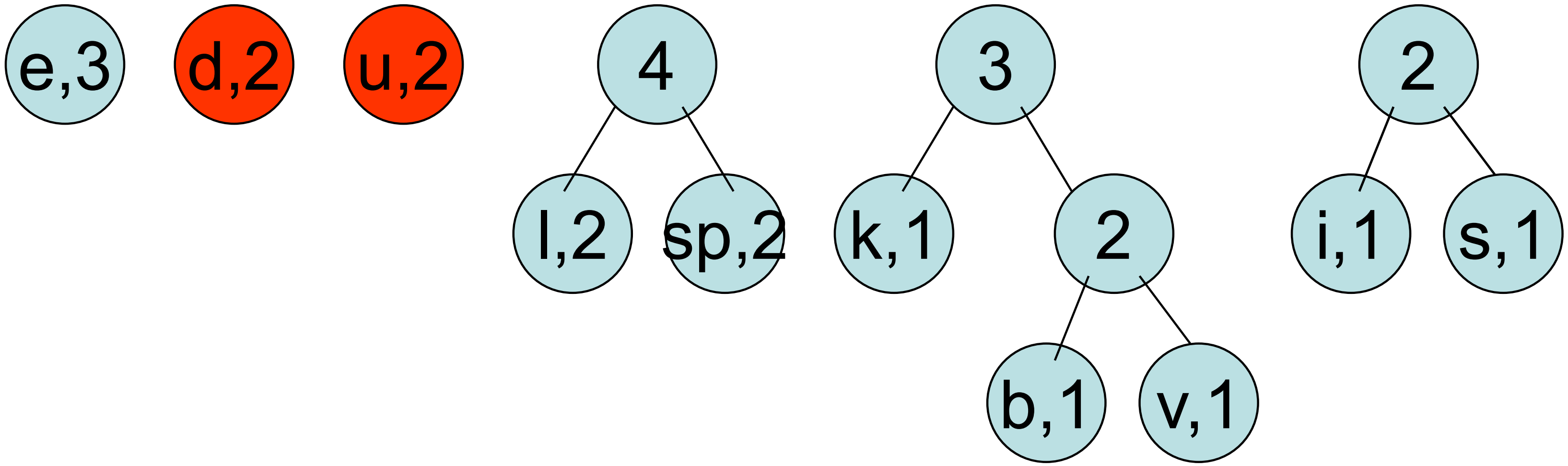
Example: Huffman Coding



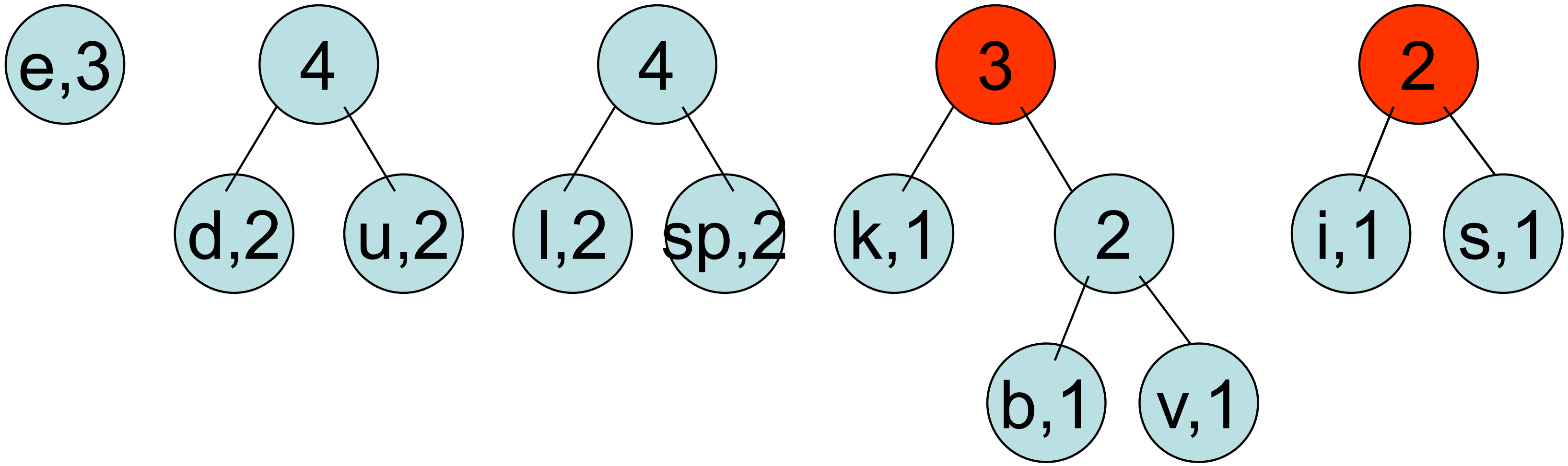
Example: Huffman Coding



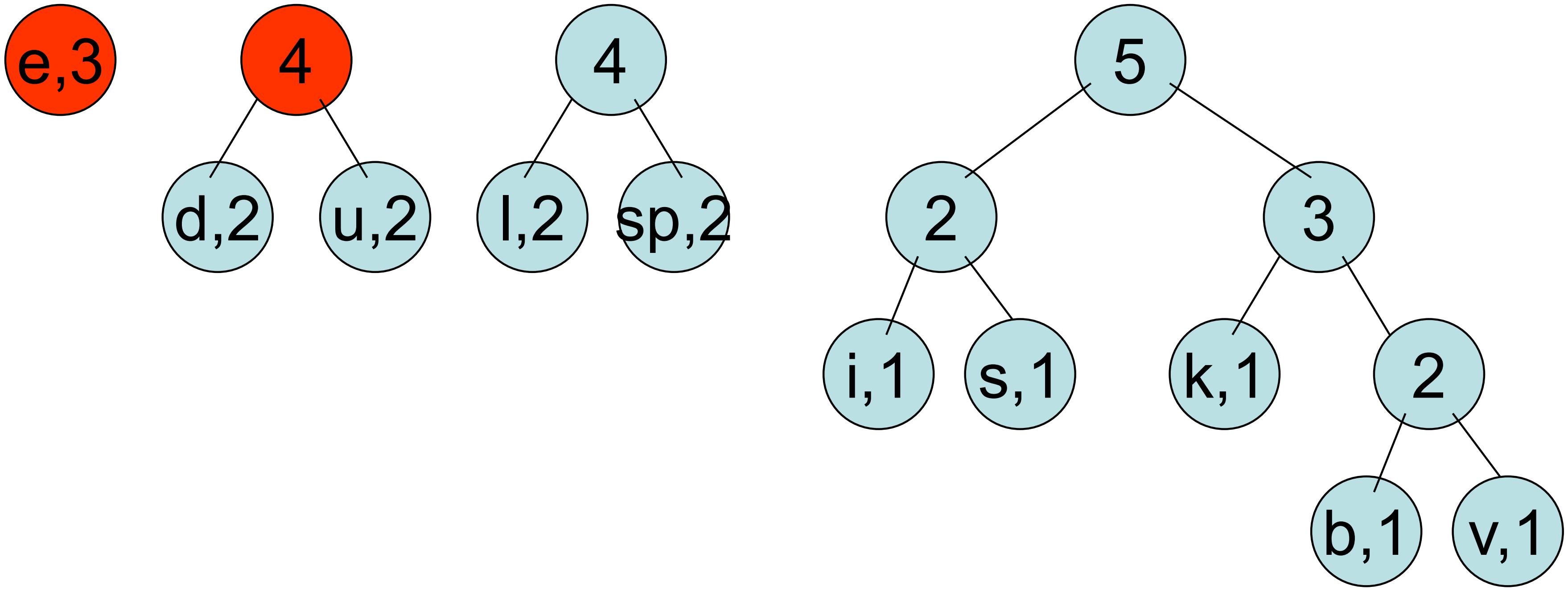
Example: Huffman Coding



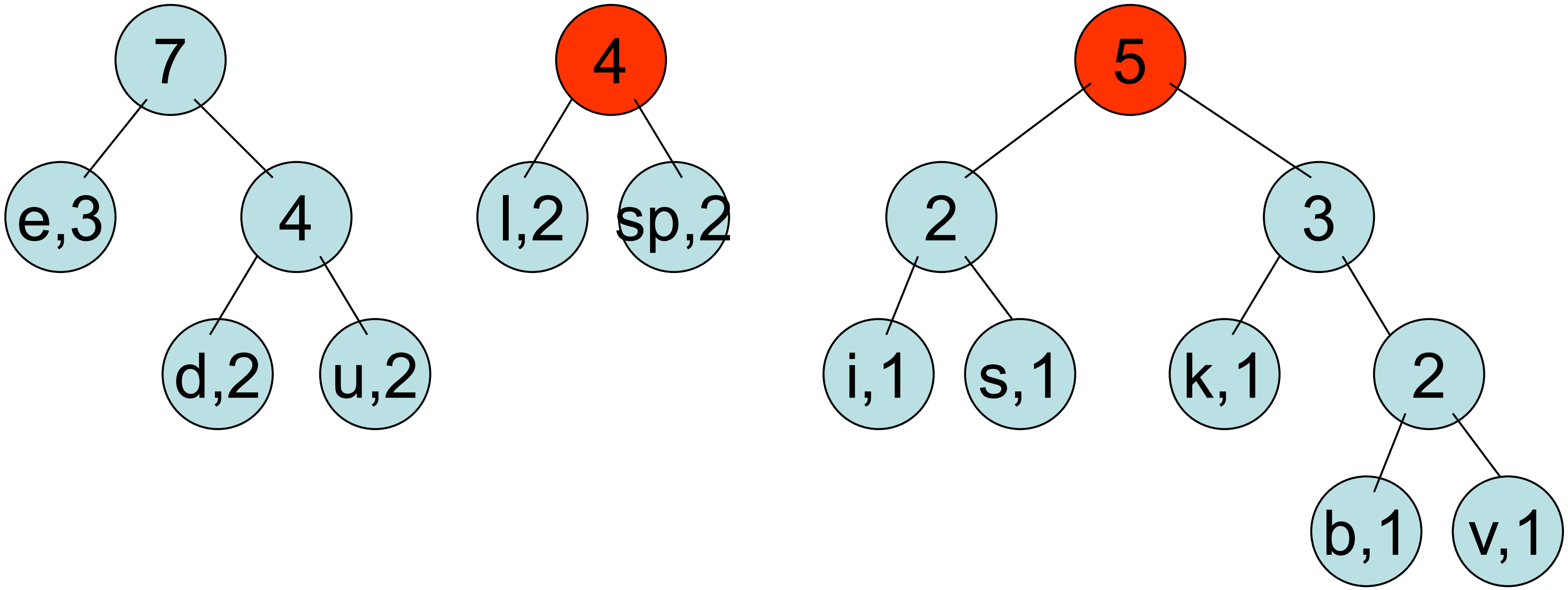
Example: Huffman Coding



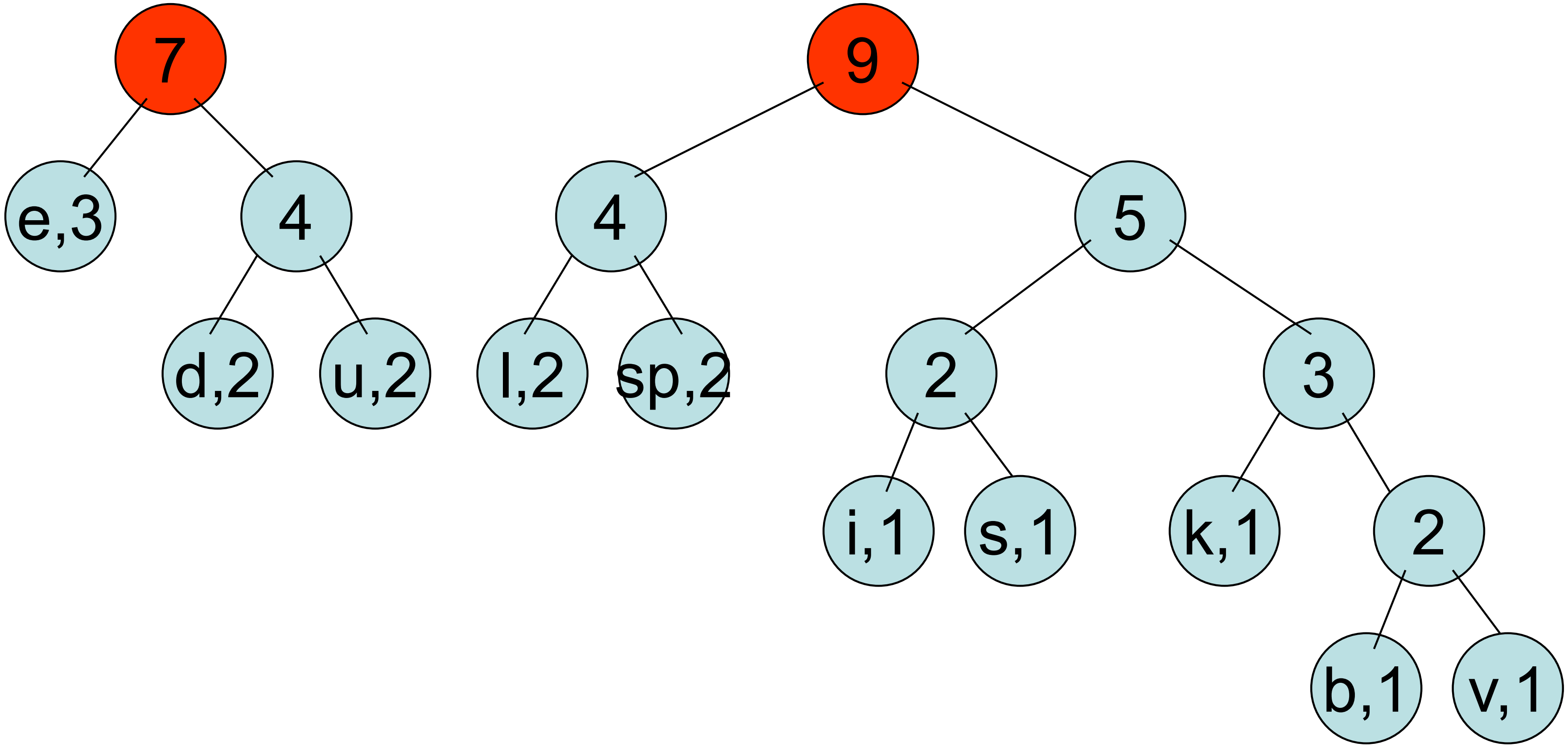
Example: Huffman Coding



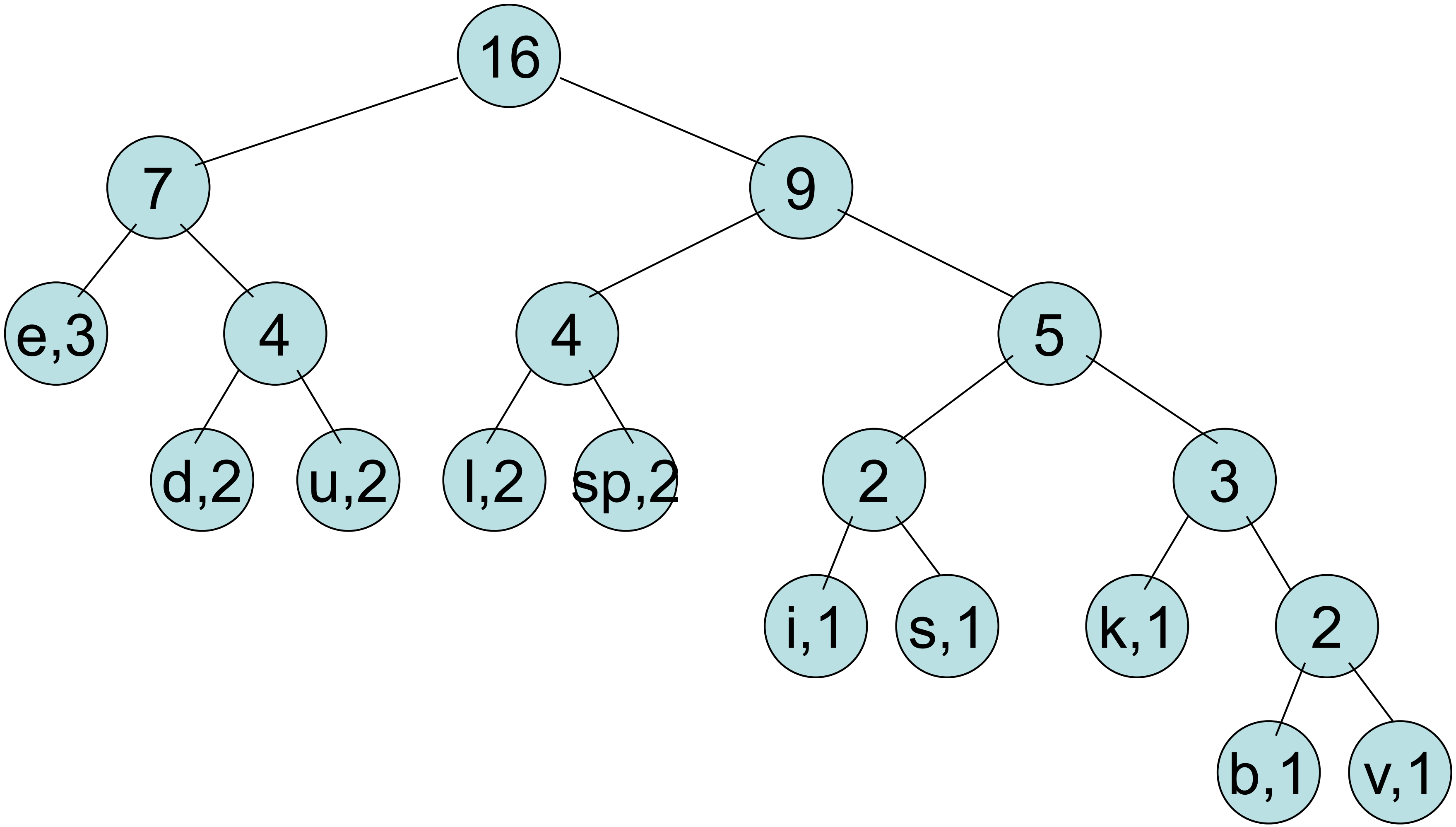
Example: Huffman Coding



Example: Huffman Coding



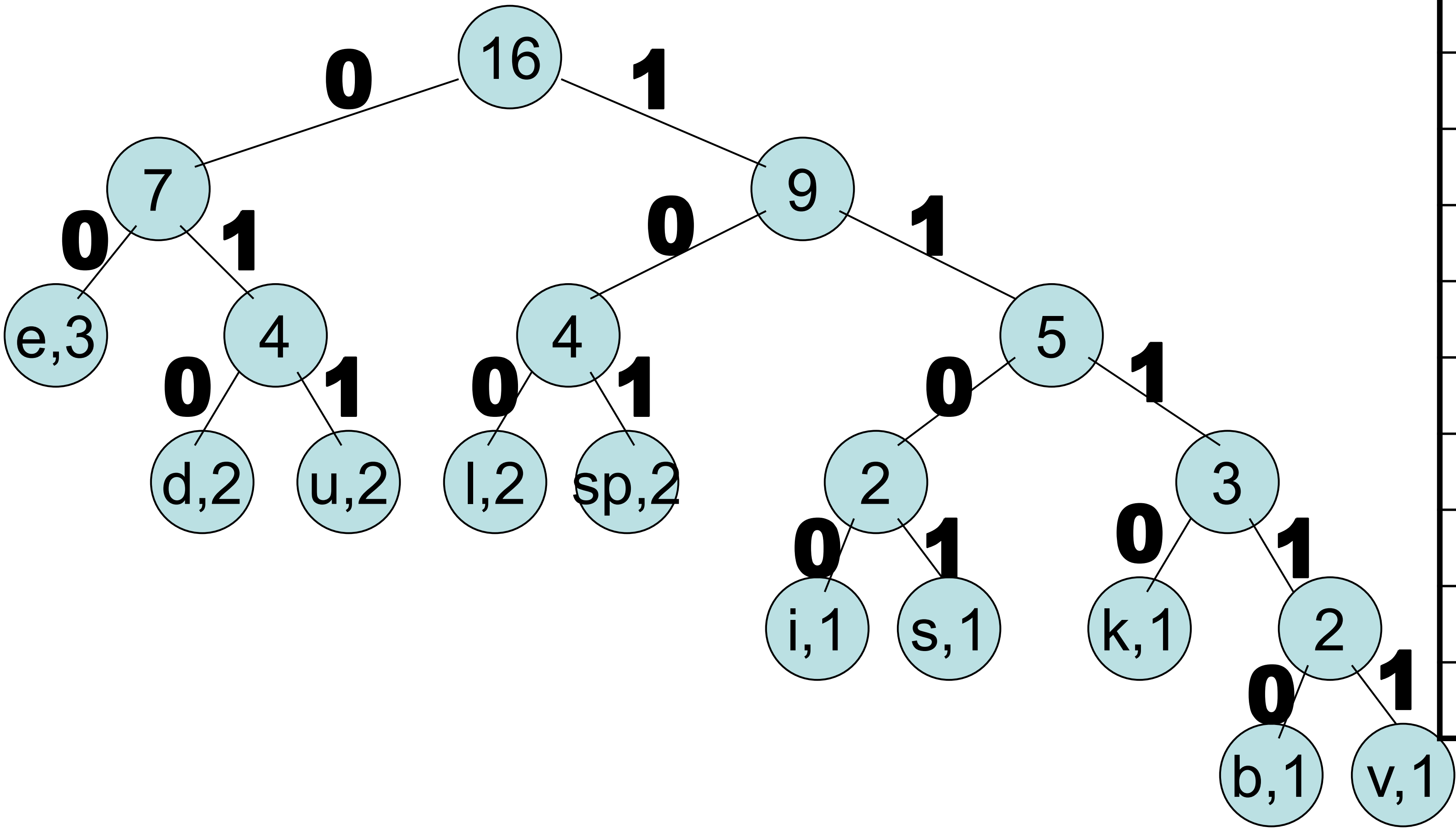
Example: Huffman Coding



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



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

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/freqencies

Exercise

- Prefix codes
  - Given a: 0, b: 101, c: 100, d: 111, e: 1101, f: 1100
  - Decode 001011101 going left to right

- Answer
  - 001011101 → 0|01011101
  - a|0|1011101 → a|a|101|1101
  - a|a|b|1101 → a|a|b|e

Exercise

- Prefix codes
  - Given I: 0, L: 101, Y: 100, X: 111, T: 1101, Z: 1100
  - Decode 01011001101 going left to right

- Answer
  - 01011001101 → 0|1011001101
  - I|101|1001101 → I|L|100|1101
  - I|L|Y|1101 → I|L|Y|T

Next

- Graph Algorithms
  - BFS & DFS
  - Topological Sort
  - Minimum Spanning Trees
  - Single Source Shortest Path
  - All-pairs Shortest Path
  - Maximum Flow
- Important
  - Please read Lecture/Textbook first