Design and Analysis of Algorithms

Lecture 7

Data Compression

Topics

- Data compression
- Prefix codes
- Shannon-Fanno
- Huffman Codes
- Huffman coding algorithm
- Hu-Tucker Algorithm

Data Compression

- Consider data to be a sequence of characters
- Data is usually stored in
  - ASCII (8 bits per character)
  - Unicode (16 bits per character)
  - These are examples of binary character codes
- If there are n symbols, need $\lceil \log_2 n \rceil$ bits
- Symbols = {A, B, C, D, R} need 3 bits per symbol

  $\lceil \log_2 5 \rceil = 3$

Fixed Length Code

- File (or message) = ABRACADABRA
  - There are 11 characters

Symbol	p_i	Code word
A	5/11	000
B	2/11	001
C	1/11	100
D	1/11	101
R	2/11	111

A B R A C A D ...
000001111000100000101...

11 symbols each of 3 bits
= 33 bits
= 5 bytes

→ Better than ASCII that uses
$11 \times 8 = 88$ bits = 11 bytes

Variable Length Code

message = ABRACADABRA

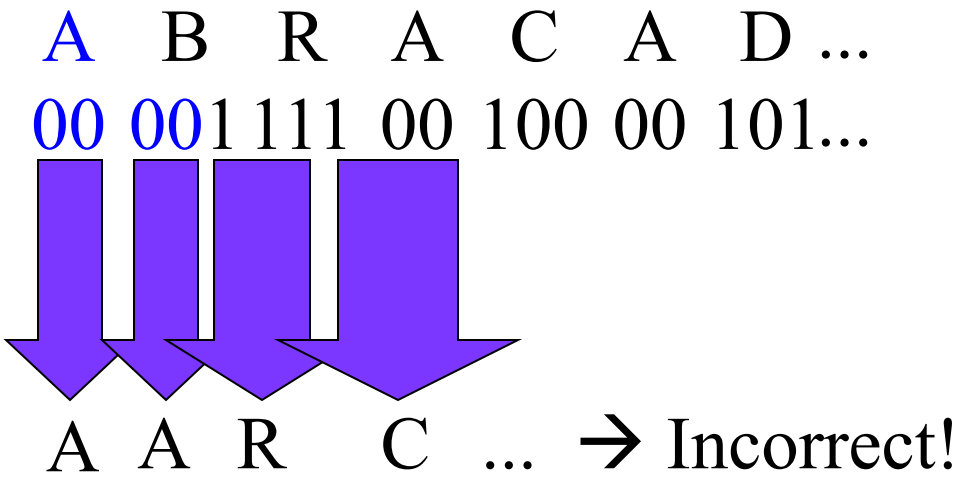
Symbol	Frequency
A	5
B	2
C	1
D	1
R	2

- Notice that symbol **A** occurs 5 times
  - ➢ so give it a shorter codeword

Non-Fixed-Length Code

So chop off one bit from A's codeword

Symbol	Codeword
A	00
B	001
C	100
D	101
R	111



Problem: codeword for A is a prefix of the codeword for B

Prefix-Free Codes

Symbol	Codeword
A	00
B	100
C	101
D	110
R	111

Solution: use a **prefix-free** code where no code word is a prefix of any other code word

How to do?

One way is Huffman coding (discussed later)

A	B	R	A	C	A	D	A	B	R	A
00	100	111	00	101	00	110	00	100	111	00

Can we go further?

So why just shorten A's codeword by one bit?

Can we do better? YES

Symbol	Codeword
A	0
B	100
C	101
D	110
R	111

Code	Bits	Bytes
ASCII	88	11
Binary	33	5
A=00	28	4
A=0	23	3
Entropy	22.44	3

Entropy

- **Shannon** proved that given a probability distribution $P = \{p_1, \ldots, p_n\}$

$$H = \sum_{i=1}^{n} \left(- p_i \log_2 p_i \right)$$

The entropy H is the smallest number of bits per symbol you can use

Entropy

- So in our example

$$11H = -\left(5\log_2\frac{5}{11} + 2\log_2\frac{2}{11} + 2\log_2\frac{2}{11} + \log_2\frac{1}{11} + \log_2\frac{1}{11}\right)$$
$$= 22.444$$

- So the last code came as close as possible
  - It has the minimum-redundancy

DAA...

- Given a list of integer frequencies, design an algorithm to produce a minimum-redundancy prefix code
- Possible approaches:
  - Brute force
  - Divide and Conquer
  - Greedy

Brute Force

- Construct all possible prefix codes, and take the one with the smallest cost.
- For n symbols, code words can range from 1 to $(n - 1)$ bits long
- Each bit can take on 2 possible values, i.e., 0 or 1
  - So that's about 2^{n^2} different codes
  - At 1 second per code = over 1 year of computation for $n = 5$.
- For a more realistic example, consider a code of 40 symbols (letters, numbers and some punctuation) at 2^{-14} seconds per code.
  - This would take far longer than the universe has left.

Divide & Conquer

- Shannon tells us that a symbol with frequency f should use $-\log_2(f/m)$ bits, where m is the sum of all the frequencies
  - So if $(f/m) = 0.5$, should use 1 bit
  - If $(f/m) = 0.25$, should use 2 bits
  - *etc.*

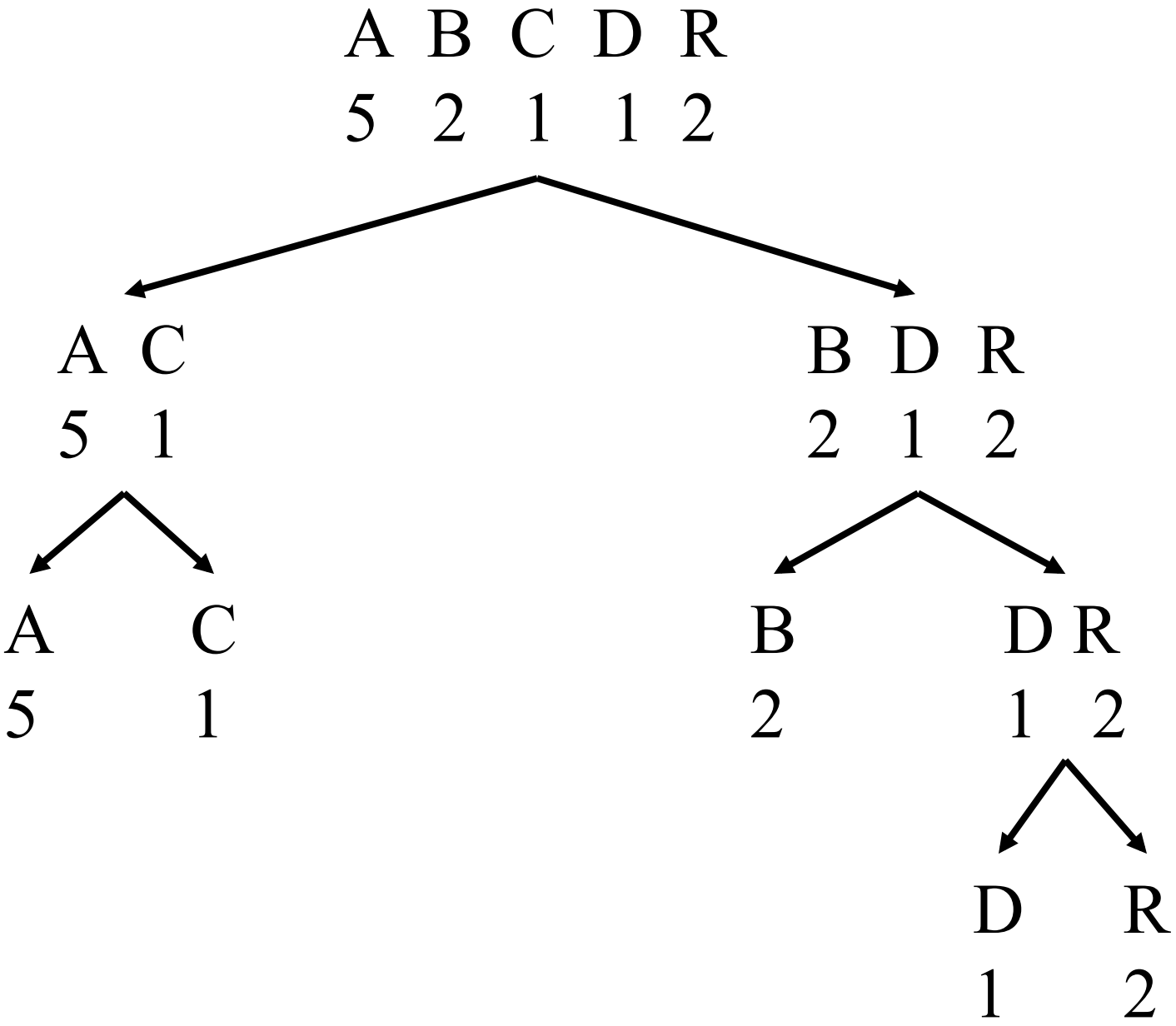
$$-\log_2 0.5 = -\log_2\left(\frac{1}{2}\right) = -\log_2 2^{-1} = --\log_2 2 = 1$$

$$-\log_2 0.25 = -\log_2\left(\frac{1}{4}\right) = -\log_2 2^{-2} = -2 \times -\log_2 2 = 2$$

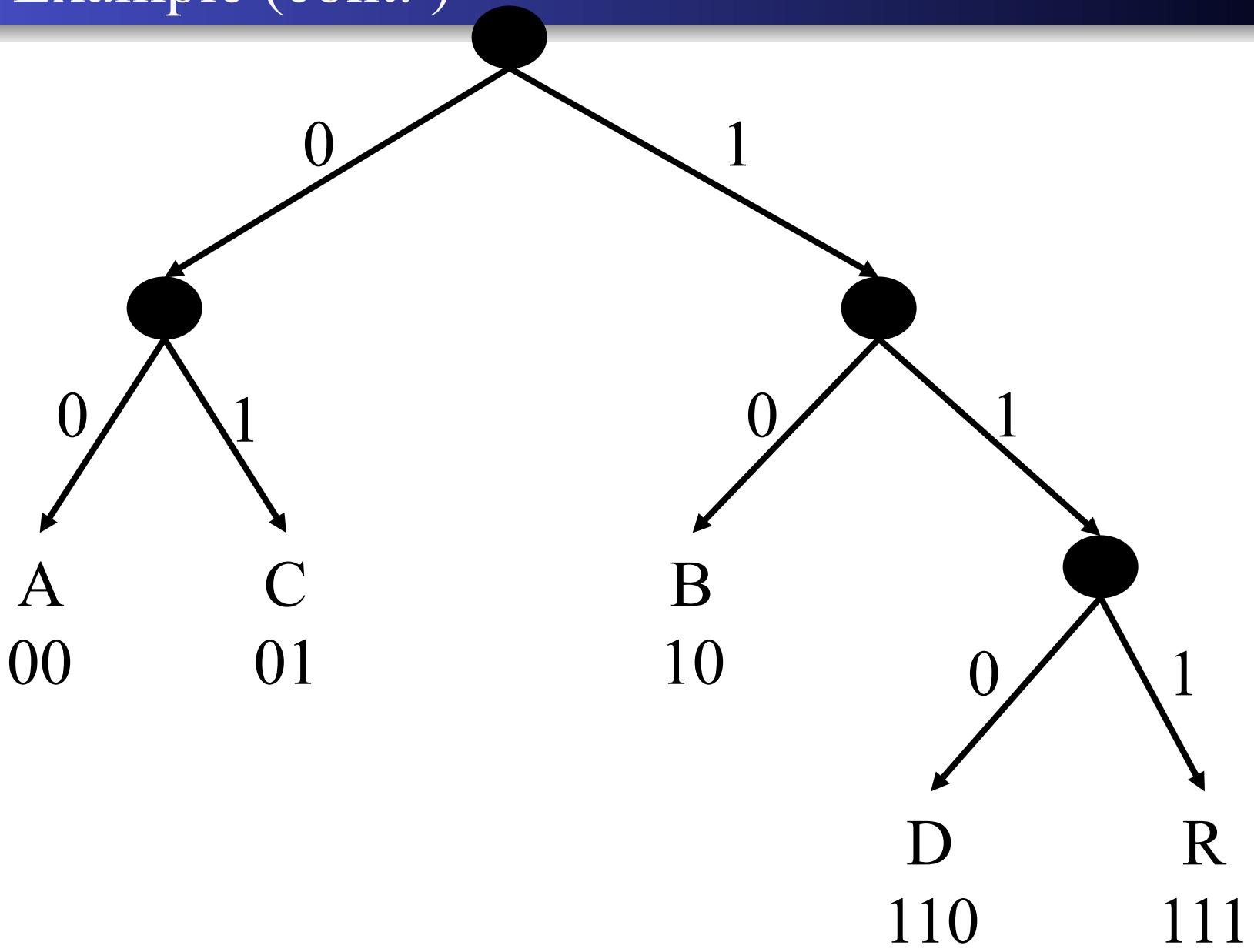
Shannon-Fano Code

- Divide symbols into two groups, with each group having as close to half the total frequency each
- Give each group one bit ($-\log_2 0.5$)

Shannon-Fano Code - Example



Example (cont.)



Shannon-Fano algorithm

1. Initialise T to be a set containing all n symbols.
2. Let T be a set such that $|T| > 1$. If T doesn't exist, go to 7.
3. Split T into T_1 and T_2 so that the sum of the weights in T_1 and T_2 are as close as possible.
4. Append a 0-bit to the codewords for symbols in T_1.
5. Append a 1-bit to the codewords for symbols in T_2.
6. Delete T and go to 2 for each T_1 and T_2.
7. Output the codeword for each of the n singleton sets.

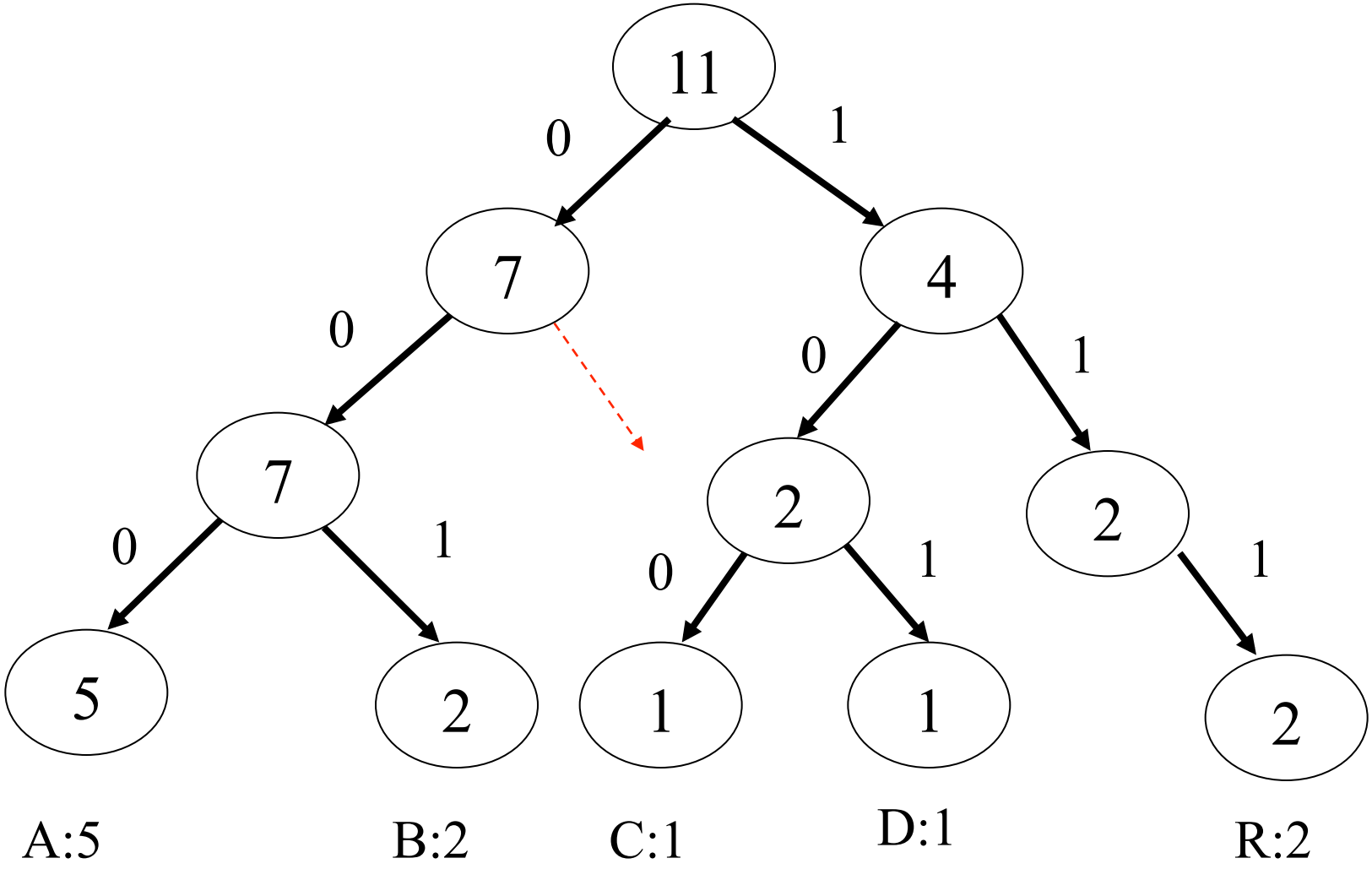
Analysis

- As an exercise...
- Hints:
  - If Step 1 is $O(n \log n)$, Step 3 can be $O(n)$
    - Sort symbols in descending order of probability → $O(?)$
    - Divide symbols into subgroups such that the sub-groups's probability are as close as possible → scan from right to left? → $O(n)$
  - Like Quicksort analysis.
    - $T(n) = T(k) + T(n - k) + n$
      - *k is the number of symbols in group.*

Code Trees

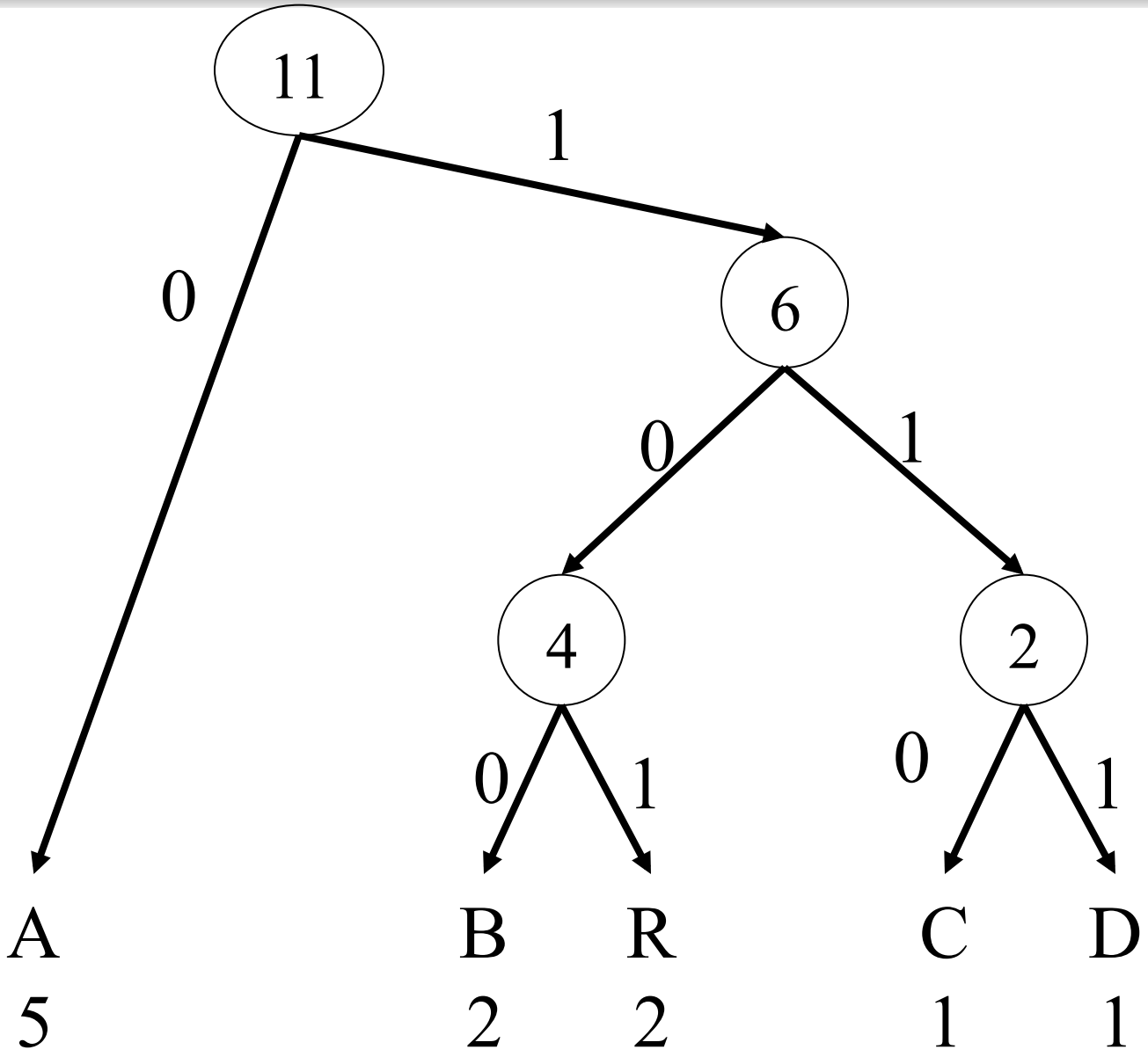
- Code trees are a good way to think about prefix codes
  - Leaves represent characters
- A binary codeword is represented as a path from the root to that character (left: 0; right:1).
- An optimal code for a file is always represented by a full binary tree → every non-leaf has two children.

Example (non-optimal)



Greedy Approach

- Rather than build the code tree top down, why not build it bottom up?
  - Make a tree leaf for each symbol with weight equal to the symbol frequency
  - Greedily choose the two smallest weight sub-trees to make a new sub-tree
  - Make the weight of the new sub-tree the sum of the child weights


11
1
0
6
0
1
4
2
0
1
0
1
A
5
B
2
R
2
C
1
D
1

Huffman algorithm

- This greedy algorithm was discovered by Huffman
- It is optimal (*i.e.*, produces a minimum-redundancy prefix code)
- Can be implemented in $O(n)$ time on sorted input
- How long for unsorted? $O(n \lg n)$

Huffman Algorithm (cont.)

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
```

Huffman Algorithm (*cont.*)

- C is a set of n characters
- Each character $c \in C$ is an object with a defined frequency $f[c]$.
- The algorithm builds the tree corresponding to the optimal code in a bottom up manner.
- The algorithm begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ merging operations to create the tree.
- A min-priority Q, keyed on f, is used to identify two least frequent objects to merge together.
- The merger result is a new object whose frequency is the sum of frequencies of the two merged objects.

Huffman Algorithm Complexity

- Assume Q is implemented as a binary min-heap for n characters.
- Q initialization (line 2) takes $O(n)$
  - Use BUILD_MIN_HEAP
- For loop in lines 3-8 is executed $n - 1$ times
- Each heap operation takes $O(\log n)$ time
  - lines 3-8 take $O(n \log n)$.
- Thus, total running time $O(n \log n)$

Modeling & Coding

- The algorithms discussed so far are ***coding*** algorithms
  - they assume that the input message has been parsed into symbols
  - they assume that each symbol has a non-zero frequency
- Parsing and estimating frequencies is the job of ***modeling*** algorithms
- In our example we used characters as symbols, and counted their occurrence to get frequencies
- Choose each symbol that can represent a sequence of characters in the message as long as possible
  - To reduce the total number of symbols used.
  - Note: Representing one symbol for each word in the dictionary requires a lot of symbols and thus is not good.

Ziv-Lempel Model

- Try to figure out what would be a good set of symbols *i.e.*, multi-character symbols
- Consider a message as a window of text
- Represent the current text as a pointer back into the window
  - Need to know how far back to point, and
  - How many characters to copy
- Used as the model in *gzip*
- Huffman code is used for coding in *gzip*

ABRACADABRA

	Distance Back to Same	Num to Copy or ASCII code
A	0	65
AB	0	66
ABR	0	82
ABRA	3	1
ABRAC	0	67
ABRACA	2	1
ABRACAD	0	68
ABRACADABRA	7	4

Gzip

- Notice that A and ABRA repeat – good symbols
- So ABRACADABRA gets transformed into the symbol stream

  (0, 65) (0, 66) (0, 82) (3, 1) (0, 67) (2, 1) (0, 68) (7, 4)

- Two Huffman codes are then built on the frequency counts of
  - distances (including the 0s which signal a new character)
  - copy lengths

Gzip (cont.)

- So in our example we have

Distance	Freq	Codeword
0	5	0
2	1	10
3	1	110
7	1	111

Lengths	Freq	Codeword
1	2	0
4	1	1

Hu-Tucker Algorithm

- Greedy like Huffman, but sub-trees must replace their left child in a sorted list
  - Only allowed to join sub-trees that are not separated by **leaves**
- Two kinds of nodes: *circular* and *square* nodes
  - Every circular node has two children
  - Every square node (i.e., leaf node) has no children
- A pair of nodes is *compatible* if there is no square nodes between them
- The *weight* of a parent node is the sum of the weights of its children.

Three phases of Hu-Tucker algorithm

Phase 1: Combination – **input:** a sequence of n square nodes

- Combine *compatible* nodes with the least total weights into a circular node; for each **tie**, select the pair from the **left** position
- Repeat the step until there is only one circular node (i.e., the root node)

Phase 2: Assignment – **input:** the tree generated in Phase 1.

- Find the *distance* (*level*) l_i of each square node i from the root in Phase 1, i.e., generate $(l_1, l_2, \ldots, l_n)$ for square nodes $(1, 2, \ldots, n)$
  - The root node is at *level* 0, each of its children is at level 1, each of its grand children at level 2, etc.

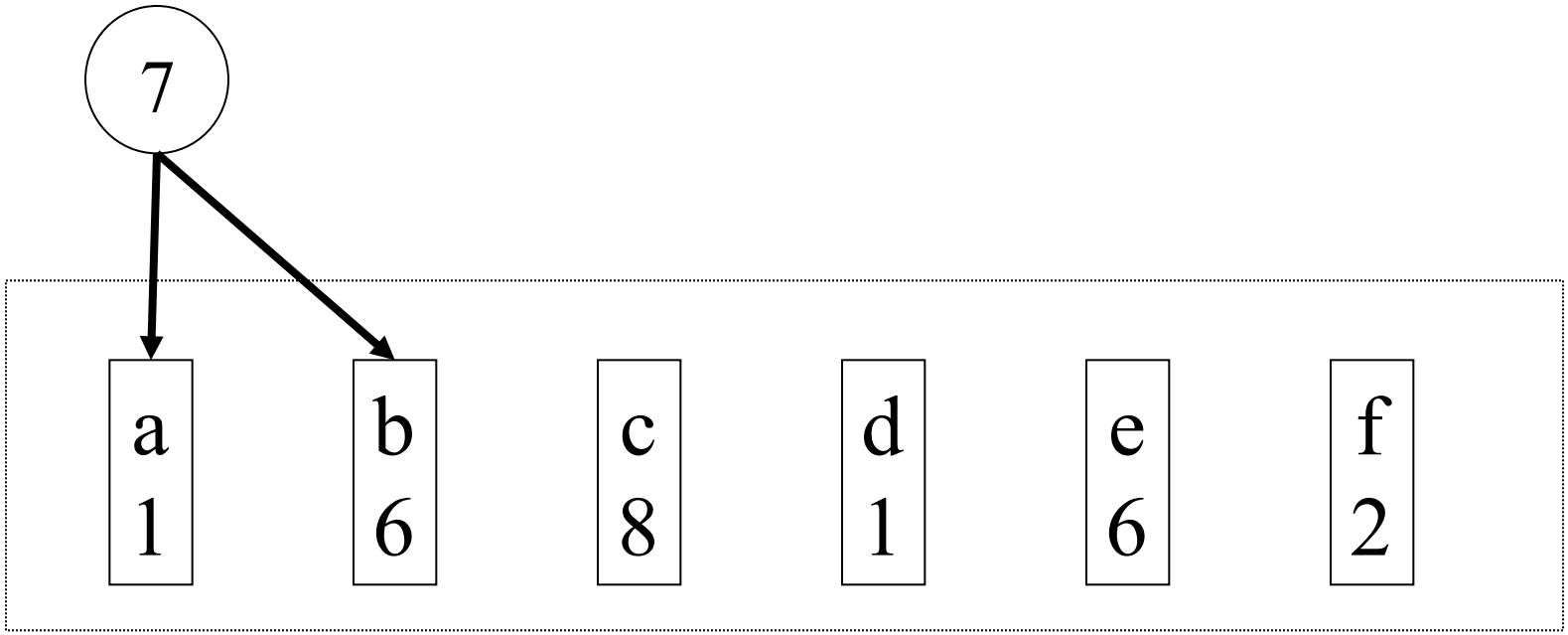
Phase 3: Reconstruction – **input:** original sequence and levels $(l_1, l_2, \ldots, l_n)$

- Phase 3 uses the levels of Phase 2 to prevent all *cross-over* links
  - a cross-over link makes a non-alphabetic code
- Combine each adjacent pair of nodes (from left to right) that have the same level number q into a parent node at level $q - 1$.
  - Repeat the step level by level starting from the maximum level until reaching level 0
  - The resulting tree is the optimal binary alphabetic tree

Hu-Tucker Algorithm (Example -1)

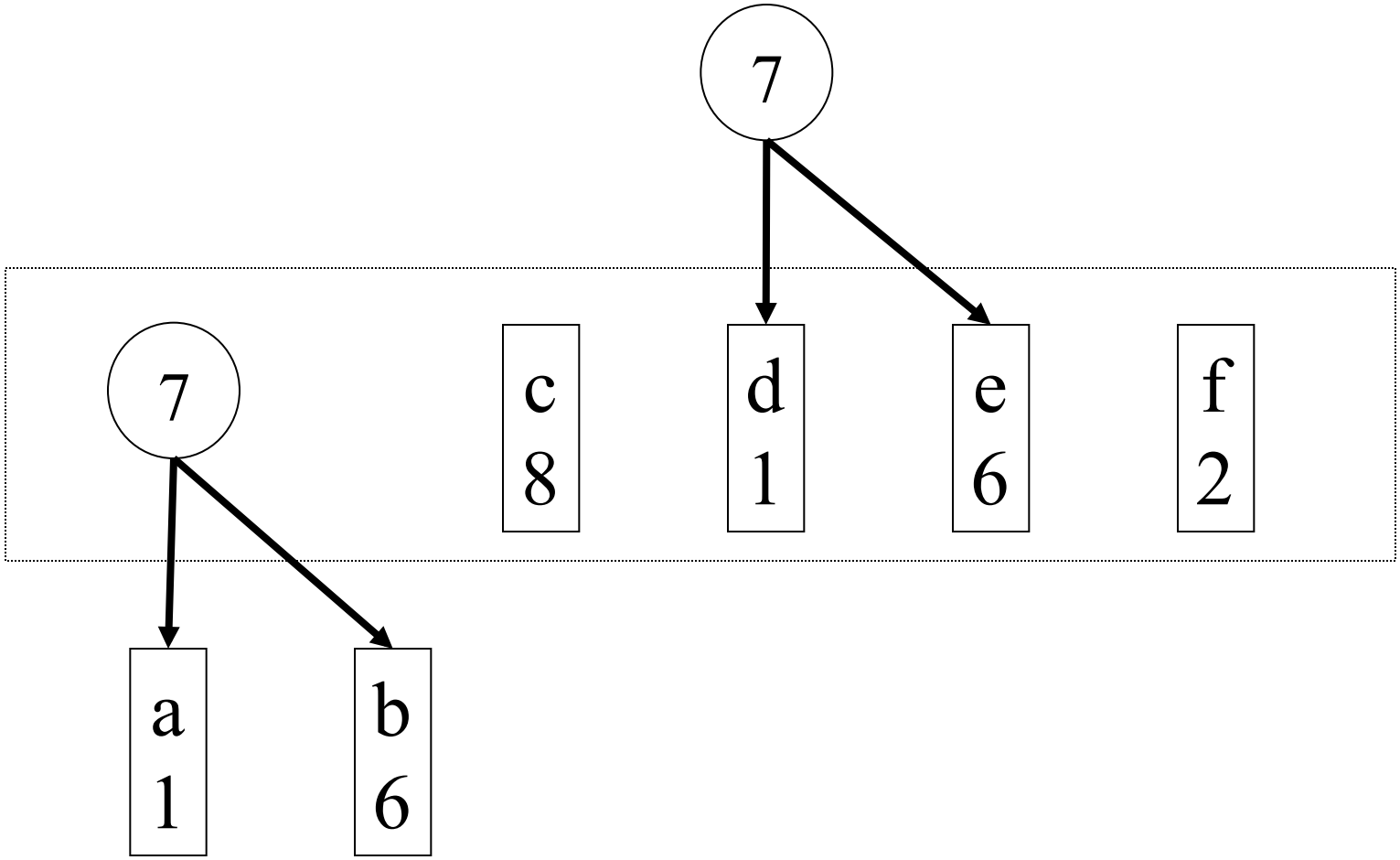
Phase-1: Combination

- Huffman's greedy algorithm says join a-d, but they are separated by a leaf, so we cannot.
  - Either a-b or d-e are available → **Choose the leftmost**.

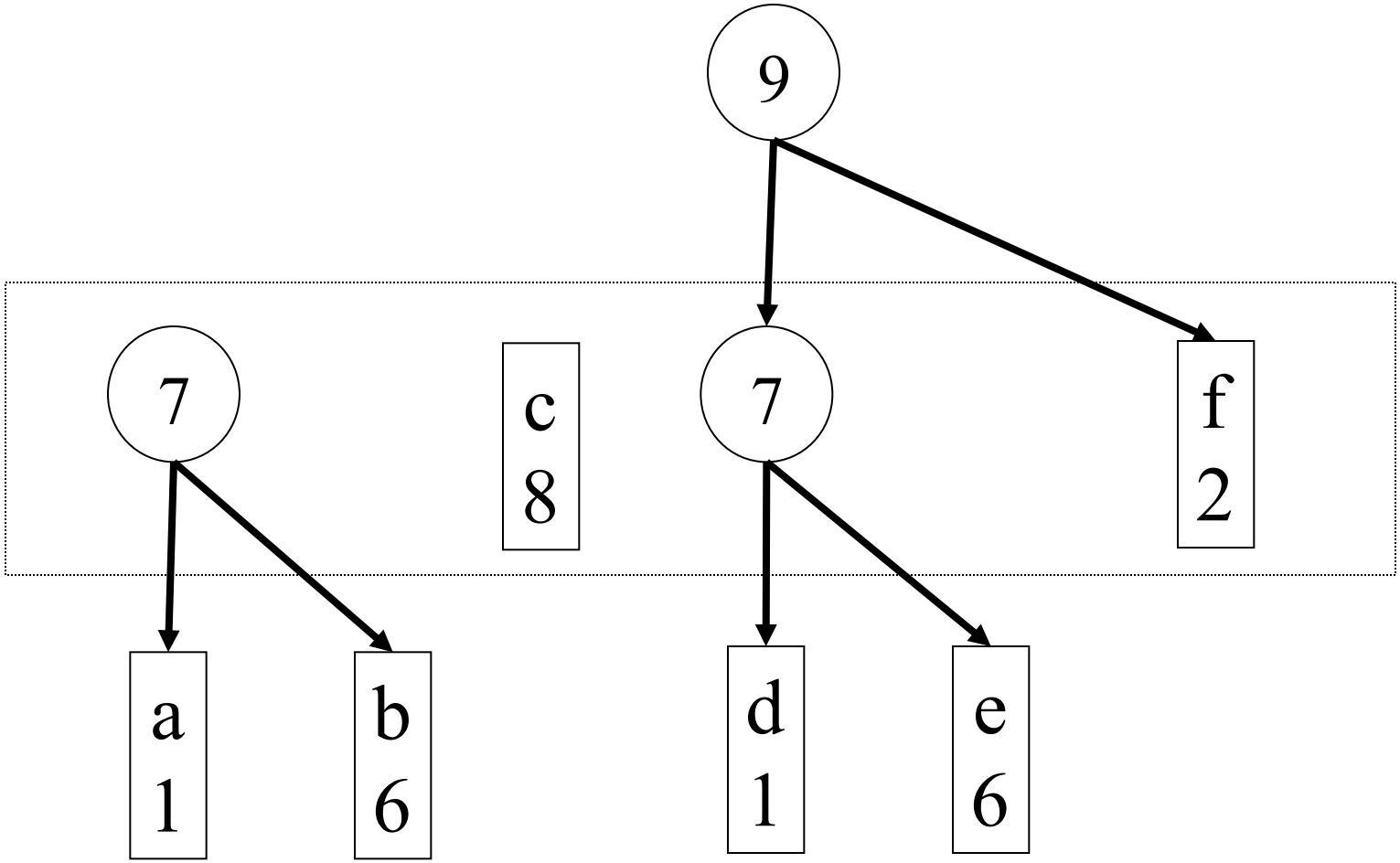


Example-1 (cont.)

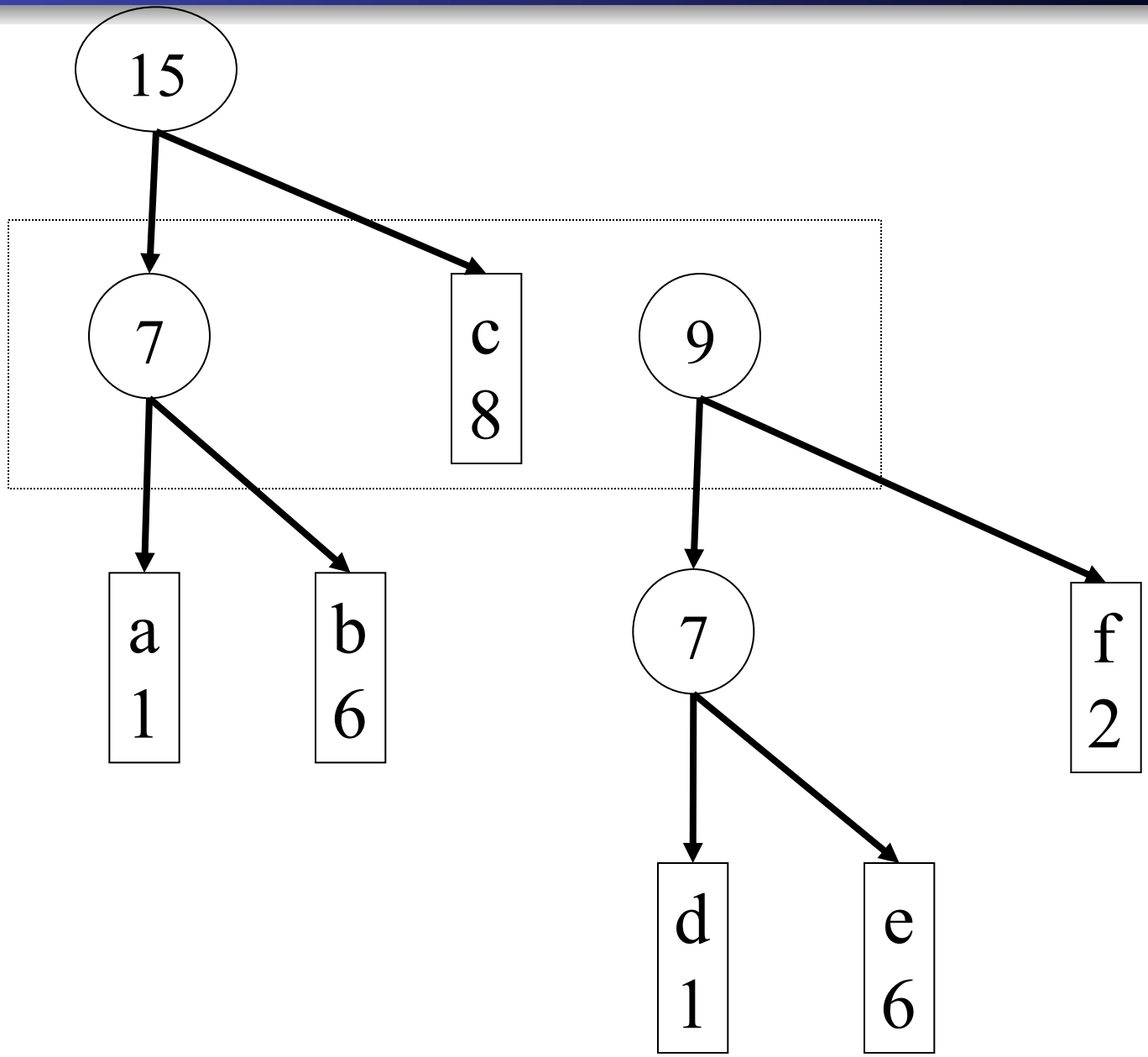
Huffman would choose d-f, but we cannot.



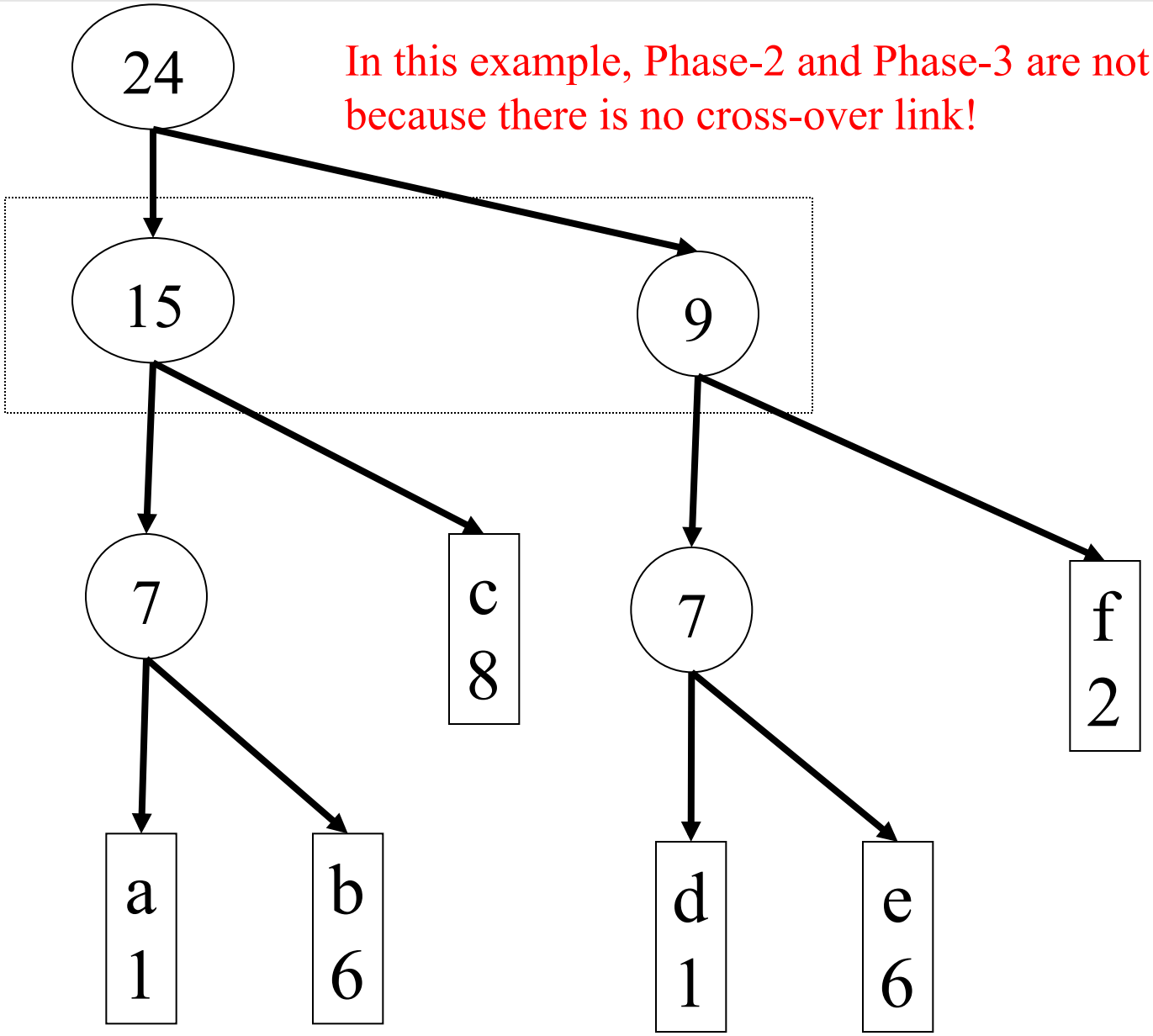
Example-1 (cont.)



Example-1 (cont.)

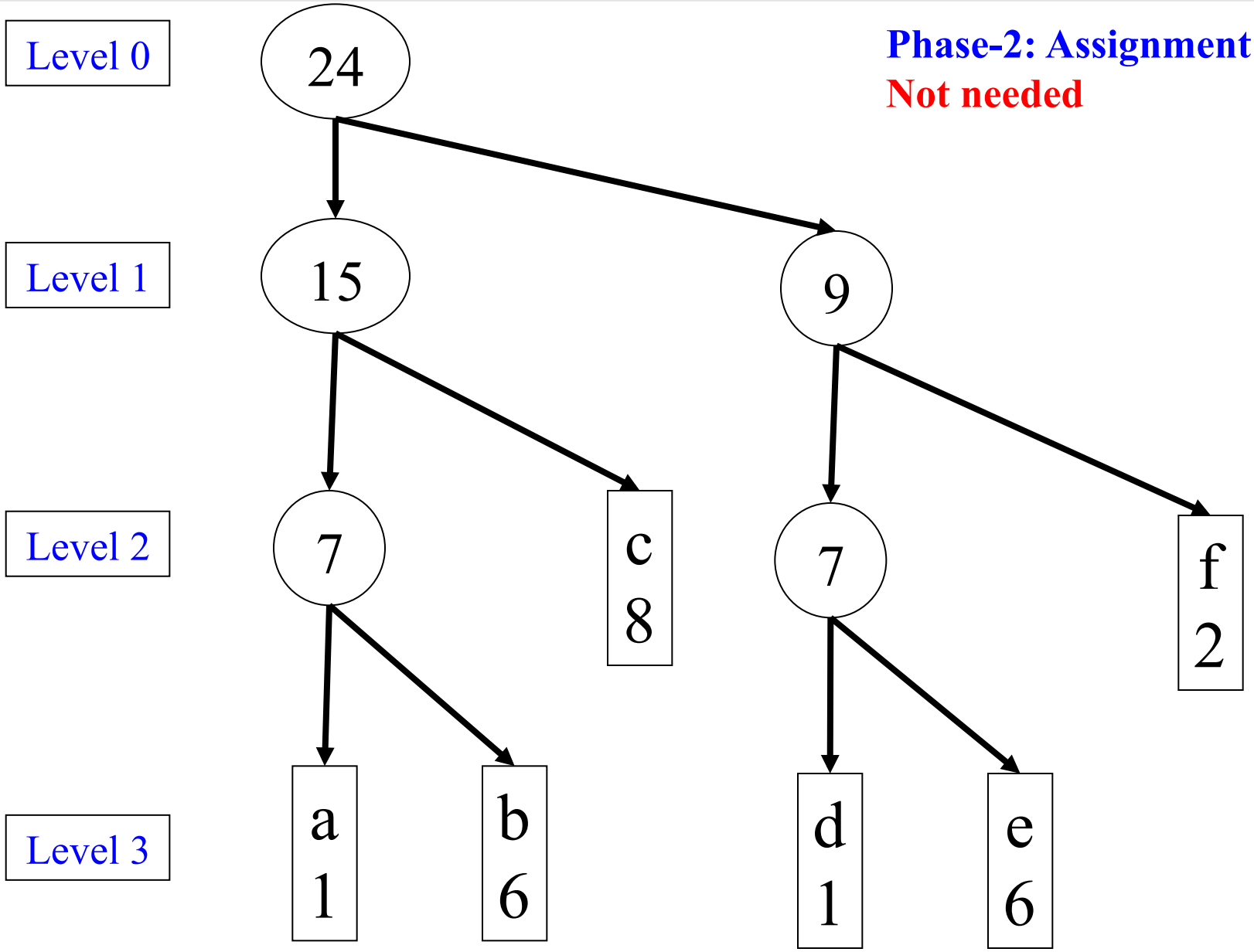


Example-1 (cont.)

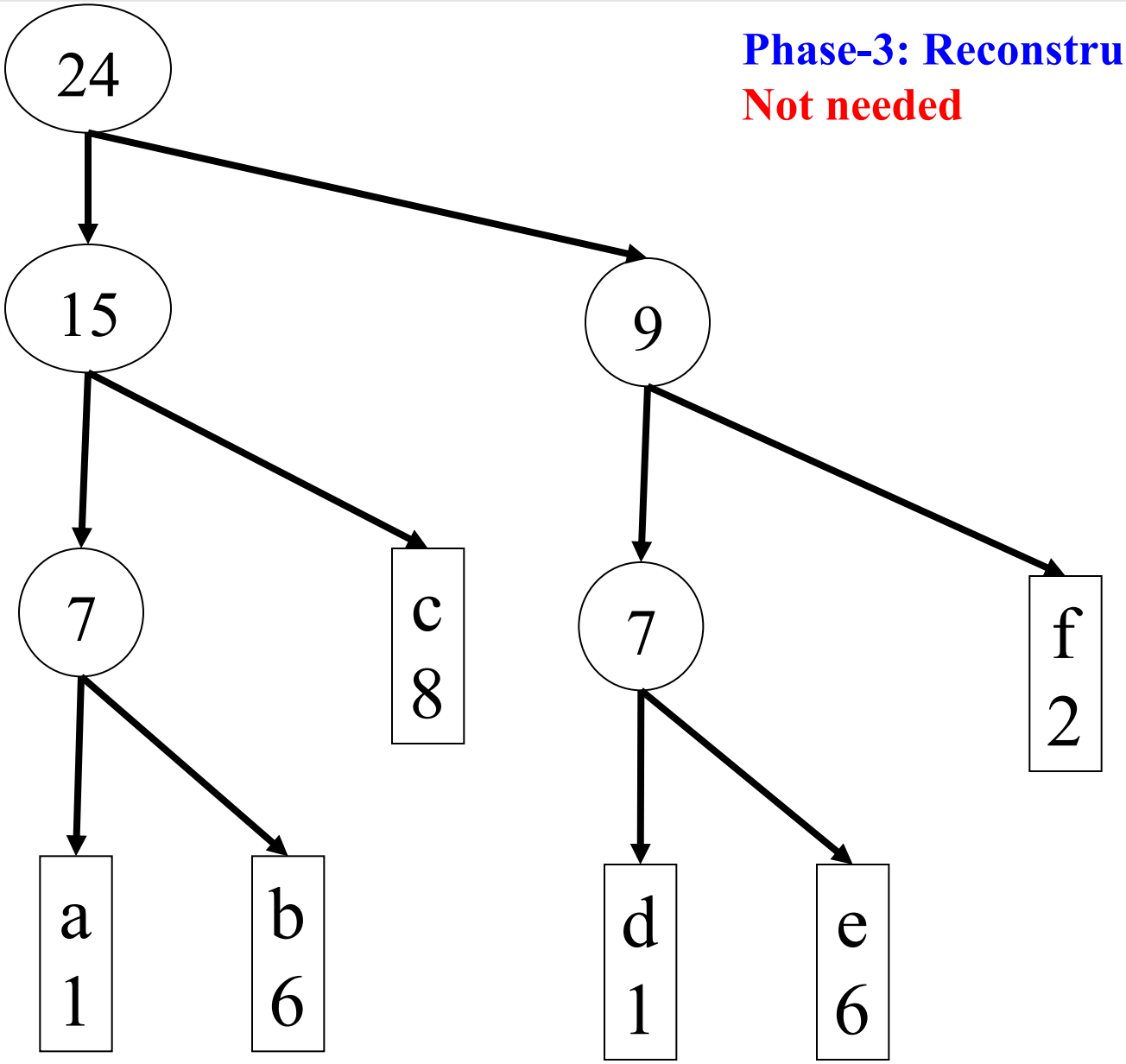


In this example, Phase-2 and Phase-3 are not needed because there is no cross-over link!

Example-1 (cont.)



Example-1 (cont.)



Phase-3: Reconstruction

Not needed

Final code

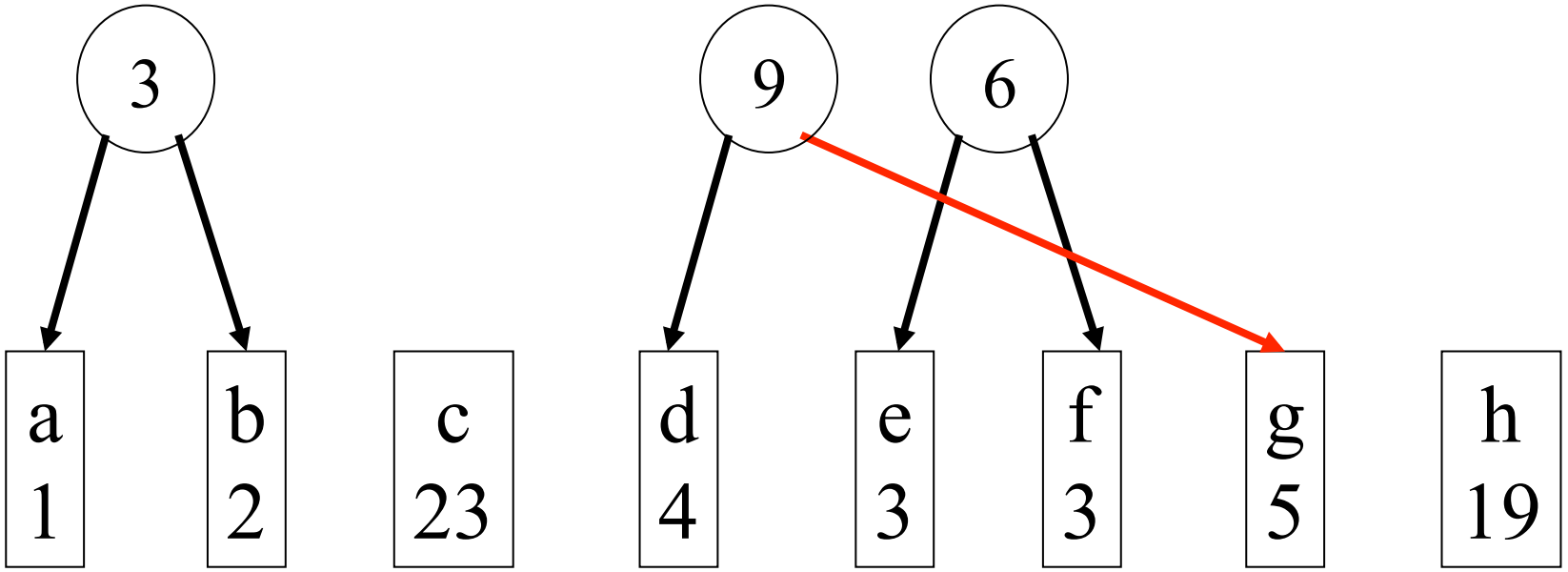
- Read off lengths from tree & assign code words

Symbol	Frequency	H-T Length	Alphabetic Code
a	1	3	000
b	6	3	001
c	8	2	01
d	1	3	100
e	6	3	101
f	2	2	11

Hu-Tucker Algorithm (Example -2)

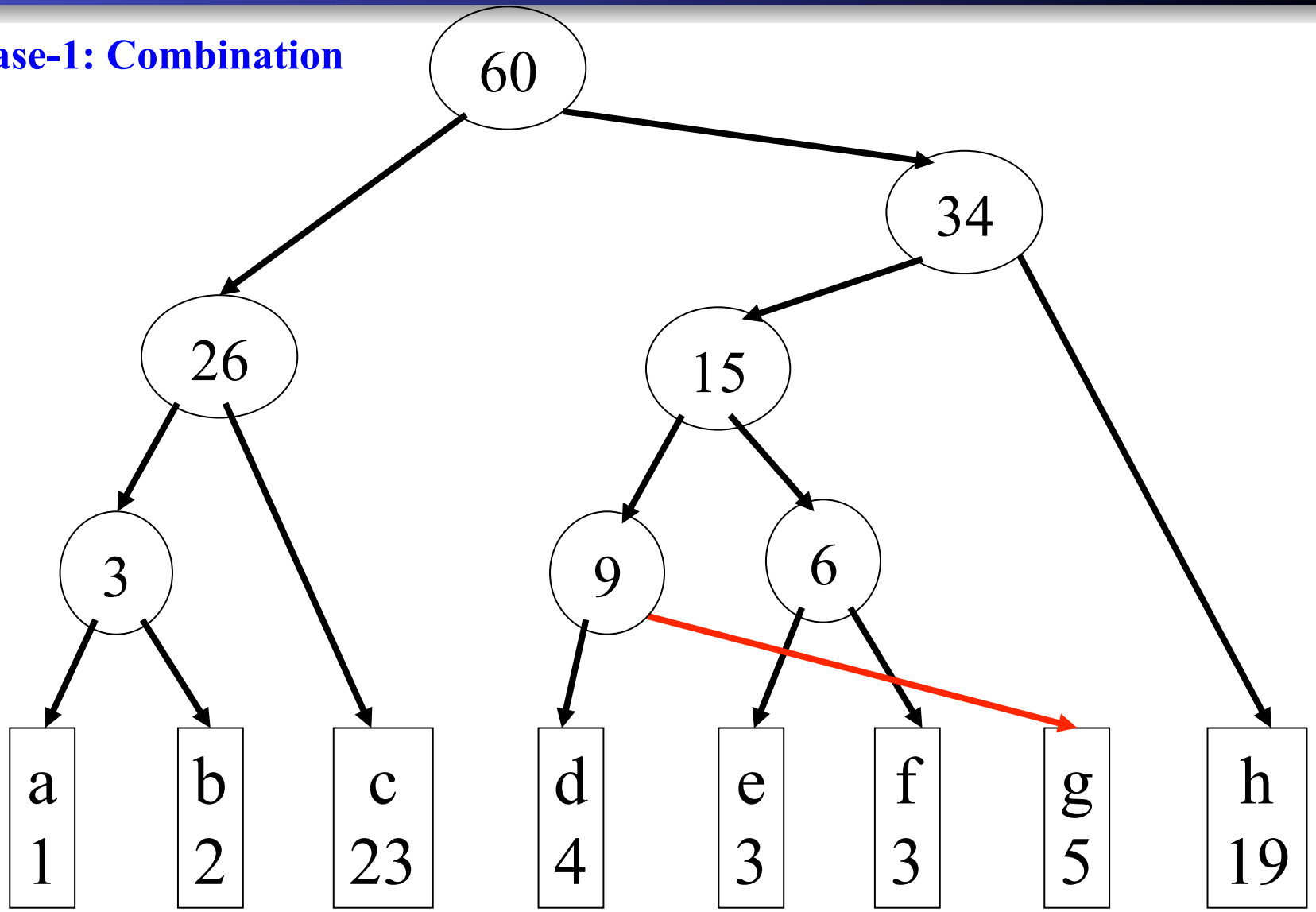
Phase-1: combination

- Circular node (3) and square node (4) are **not compatible** because they are separated by the square node (23)
- Square node (4) and square node (5) are **compatible**, because they are separated only by circular node (6)
  - However the combination creates a cross-over link (in red) that makes a non-alphabetic code
    - **Phase-3 will remove such link**



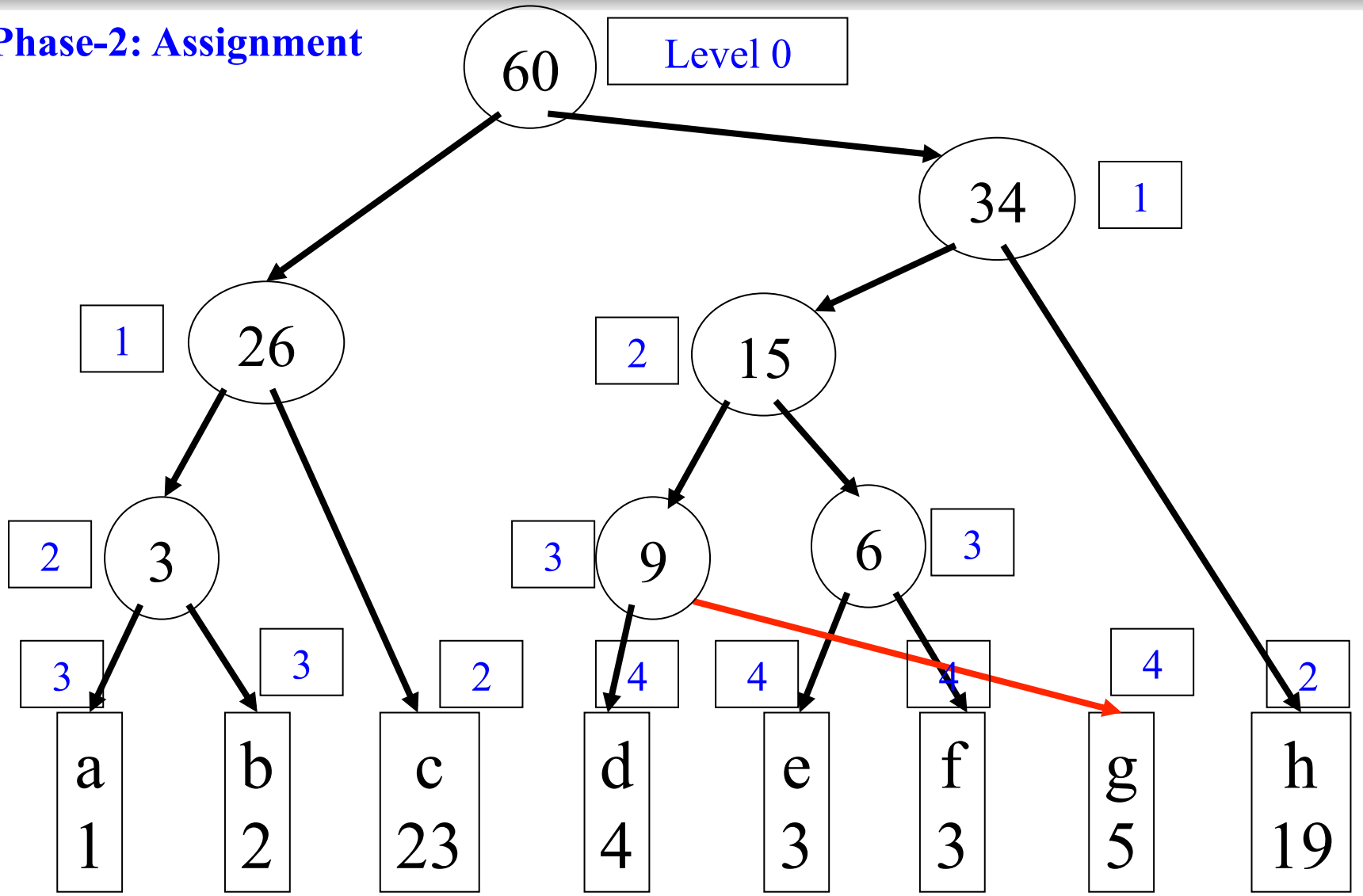
Hu-Tucker Algorithm (Example -2)

Phase-1: Combination



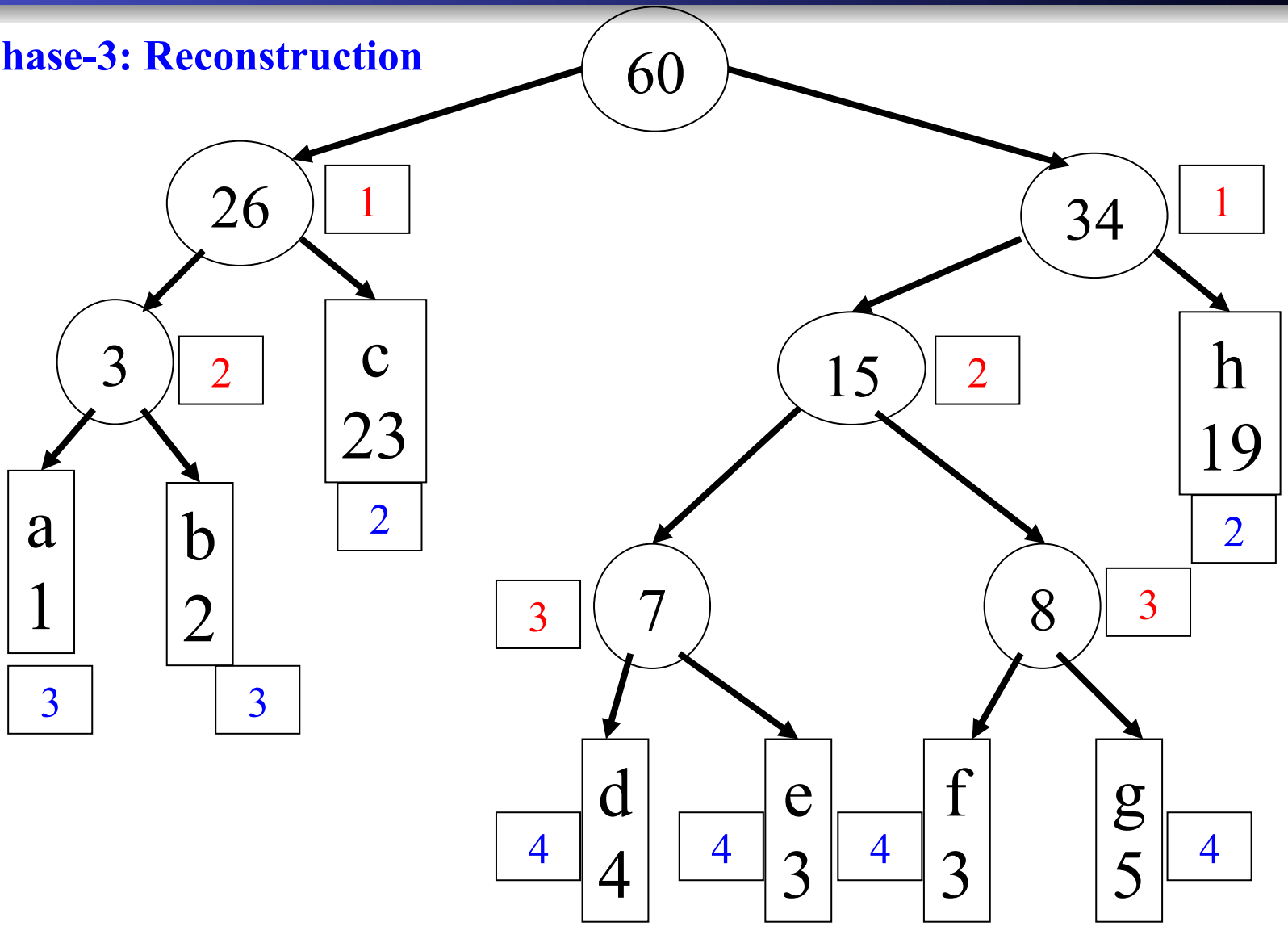
Hu-Tucker Algorithm (Example -2)

Phase-2: Assignment



Hu-Tucker Algorithm (Example -2)

Phase-3: Reconstruction



The End