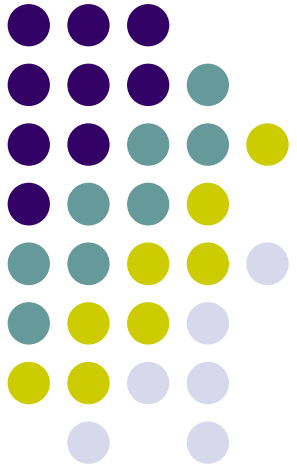
Data Compression

Huffman Coding



中央大學資工系
蘇柏齊

Fixed Length Codes vs. Variable Length Codes

- If outcomes are equally probable:
  - Fixed length code is good
  - ASCII code
- If outcomes are not equally probable:
  - Use variable length code to achieve data compression
    - Use shorter descriptions for likely outcomes
    - Use longer descriptions for less likely outcomes
  - Morse code, Huffman code
- Example: { a , b , c , d }

00 01 10 11	Fixed to fixed
0 10 110 111	Fixed to variable
	Variable to fixed?

Uniquely Decodable Code

- A variable-length code *C* is uniquely decodable iff there doesn't exist two distinct concatenations of codewords which present in the same code string
- Any sequence of codewords can be decoded in one and only one way.
-

	Code I	Code II	Code III	Code IV
S1	1	1	1	1
S2	11	10	01	10
S3	01	100	001	100
S4	00	1000	000	000

Prefix (Free) Code

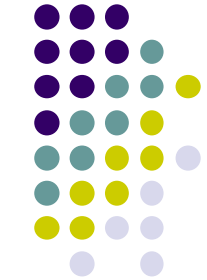
- Prefix and Suffix
  - 01, 011: 01 is called prefix of 011
  - 01, 101: 01 is called suffix of 101
- Prefix code: No codeword is a prefix of any other codeword
- Prefix code is uniquely decodable code
- Equivalent to instantaneous code
  - The code can be decoded without reference to the future codewords

Test for Uniqueness

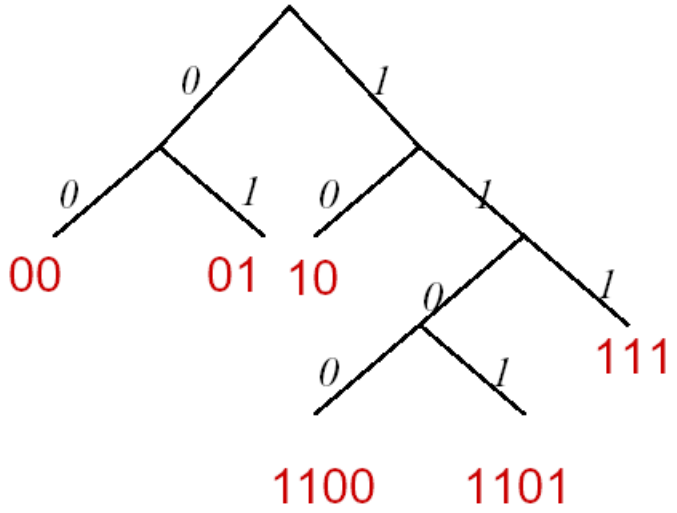
1. Construct a list of all codewords (original list)
2. Examine all pairs of codewords to see if any codeword is a prefix of another
3. Whenever we find such a pair, add the dangling suffix to the list (unless it's already added)
4. Repeat step2 and 3 until
   - Get a dangling suffix that is a codeword (in the original list) →Not uniquely decodable.
   - No more unique dangling suffix → uniquely decodable

- Examples:
  - {0,01,11} Uniquely decodable
  - {0,01,10} Not uniquely decodable

Binary Trees and Prefix Codes



- Each binary tree can be converted into a prefix code by traversing the tree from root to leaves.
- A prefix code can be expressed as a tree and codewords are leaves



$$3 \cdot 2^{-2} + 2 \cdot 2^{-4} + 2^{-3} = 1$$

Unique Decodability: McMillan and Kraft Conditions

- McMillan and Kraft Conditions
  - Necessary condition for unique decodability *[McMillan]*

$$\sum_{x \in \mathcal{A}_X} 2^{-\|c_x\|} \leq 1$$

  - Given a set of code word lengths $\|c_x\|$satisfying McMillan condition, a corresponding prefix code (uniquely decodable code) always exists *[Kraft]*
  - Hence, the inequality is both necessary and sufficient.
  - Also known as Kraft inequality or Kraft-McMillan inequality.
- Applicable to prefix code
  - no loss by only considering prefix codes.
- So restate the theorem:
  - For any uniquely decodable code, (of course including prefix code, instantaneous code), the codeword lengths $l_1, l_2, \ldots, l_m$, must satisfy the inequality
  - Conversely, given a set of codeword lengths that satisfy this inequality, there exists a prefix code (of course, a uniquely decodable code) with these word lengths.
- Each prefix code corresponding to a complete binary tree meet McMillan condition with equality.

Theorem *Let $\mathcal{C}$ be a code with N codewords with lengths $l_1, l_2, \ldots, l_N$. If $\mathcal{C}$ is uniquely decodable, then*

$$K(\mathcal{C}) = \sum_{i=1}^{N} 2^{-l_i} \leq 1.$$

Proof:
$$\left[\sum_{i=1}^{N} 2^{-l_i}\right]^n = \left(\sum_{i_1=1}^{N} 2^{-l_{i_1}}\right)\left(\sum_{i_2=1}^{N} 2^{-l_{i_2}}\right)\cdots\left(\sum_{i_n=1}^{N} 2^{-l_{i_n}}\right)$$

$$= \sum_{i_1=1}^{N}\sum_{i_2=1}^{N}\cdots\sum_{i_n=1}^{N} 2^{-(l_{i_1}+l_{i_2}+\cdots+l_{i_n})}.$$

The exponent $l_{i_1} + l_{i_2} + \cdots + l_{i_n}$ is simply the length of n codewords from the code $\mathcal{C}$. The smallest value that this exponent can take is greater than or equal to n, which would be the case if all codewords were 1 bit long. If

$$l = \max\{l_1, l_2, \ldots, l_N\}$$

then the largest value that the exponent can take is less than or equal to nl. Therefore, we can write this summation as

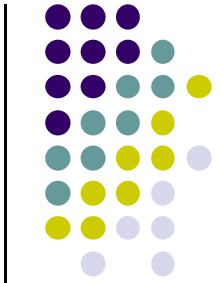
$$K(\mathcal{C})^n = \sum_{k=n}^{nl} A_k 2^{-k}$$

where A_k is the number of combinations of n codewords that have a combined length of k. Let's take a look at the size of this coefficient. The number of possible distinct binary sequences of length k is 2^k. If this code is uniquely decodable, then each sequence can represent one and only one sequence of codewords. Therefore, the number of possible combinations of codewords whose combined length is k cannot be greater than 2^k. $A_k \leq 2^k$.

$$K(\mathcal{C})^n = \sum_{k=n}^{nl} A_k 2^{-k} \leq \sum_{k=n}^{nl} 2^k 2^{-k} = nl - n + 1.$$

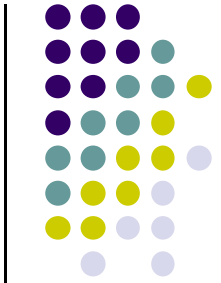
But if $K(\mathcal{C})$ is greater than one, it will grow exponentially with n, while $n(l-1)+1$ can only grow linearly. So if $K(\mathcal{C})$ is greater than one, we can always find an n large enough that the inequality is violated.

Shannon-Fano Code



- $H(X) <= L_{s\text{-}f}(c) < H(X)+1$
- Procedures:
  1. From the data source, calculate the probability of the occurrence of each symbol.
  2. Sort the symbols according to their probability.
  3. Divide the sorted symbols with their prob. into two groups (upper and lower) with approximately equal prob.
  4. Assign 0 to the upper group and assign 1 to the lower group (symbols in the upper group start with 0).
  5. Repeat step 3 and 4 until each group has only one symbol

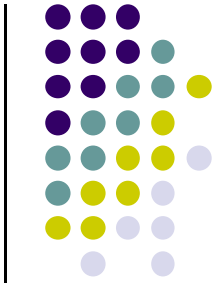
Shannon-Fano Code Example



S	Prob.	Cum.	Code
A	0.39	1.00	00
B	0.18	0.61	01
C	0.15	0.43	10
D	0.15	0.28	110
E	0.07	0.13	1110
F	0.06	0.06	1111

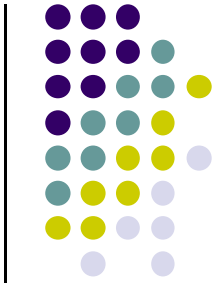
- Entropy: 2.3083
  Bits average: 2.41
  Redundancy: 0.1017

Huffman Coding



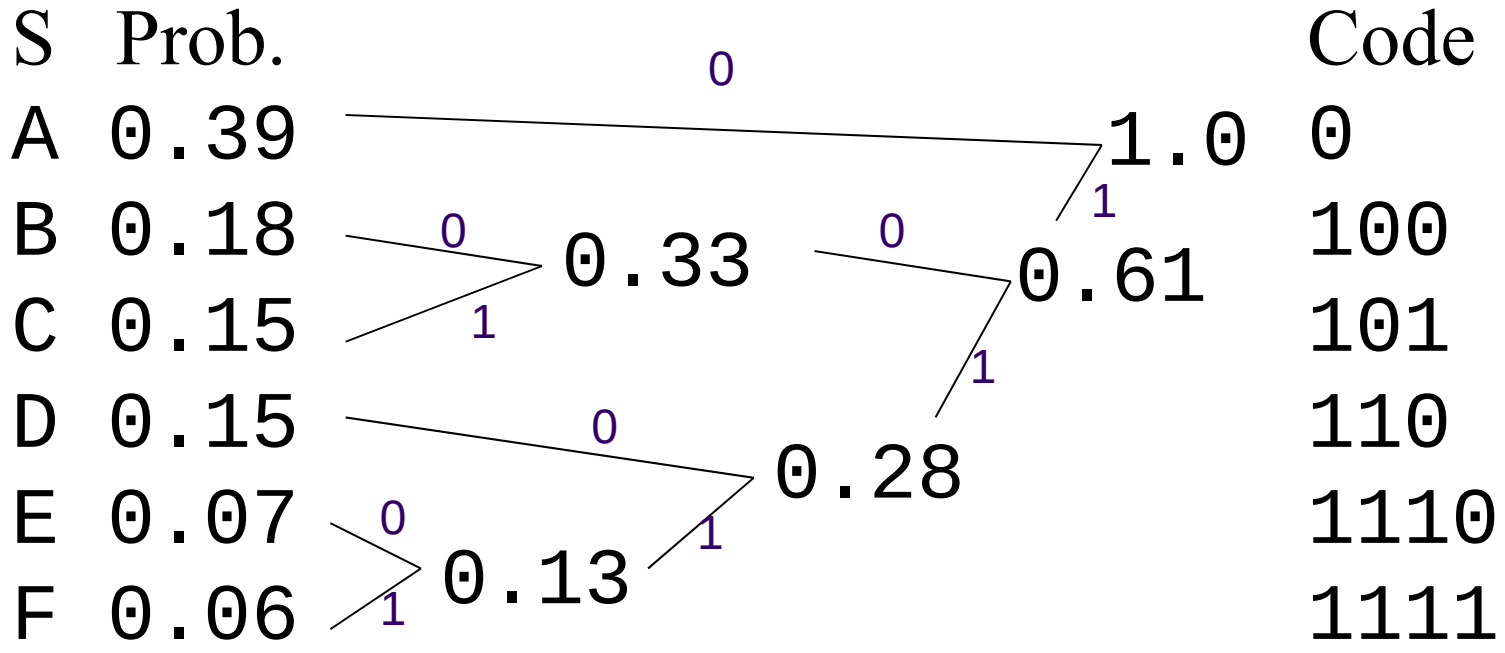
- Background:
  - Developed by David Huffman as part of a class assignment. The class was taught by Robert Fano at MIT
- Huffman coder is a prefix-free coder.
- Huffman coder is an optimal coder for a given probability model.
  - Based on the two observations on an optimal prefix coder:
    - Symbol that occur more frequently will have shorter codeword than symbol that occur less frequently.
    - The two symbols that occur least frequently will have the same codeword length.
      - Because Huffman coder is a prefix coder, it is not necessary to let the codeword of least prob. symbol longer than that of second least prob. symbol.

Huffman Coding



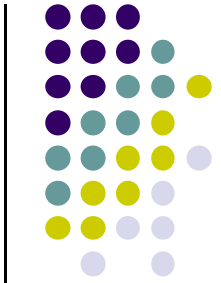
- Recursive procedures:
  1. Sort each symbol by its probability and add the node for each symbol into the coding list.
  2. Combine the two nodes with least prob. to form one new node with the prob. equal to the sum of two prob.
  3. Assign ‘0’and ‘1’arbitrarily to the branches of these two nodes with least prob.
  4. Drop these two nodes from the coding list.

Huffman Coding



- Entropy=2.3083
- Bits average=2.35, Redundancy=0.0417

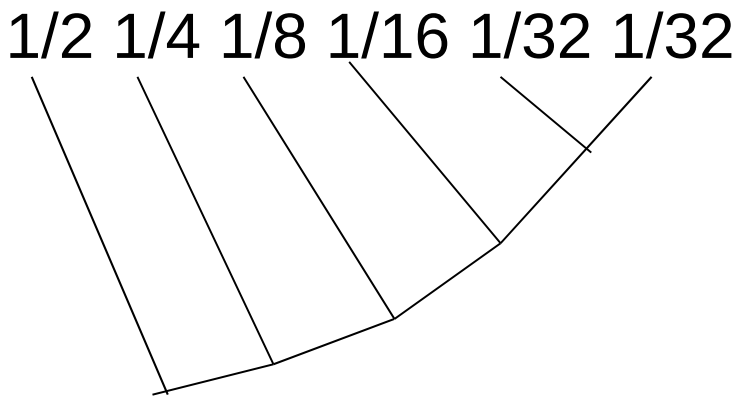
Huffman Coding



- Comments:
  - Unlike Shannon-Fano coder, which is encoded from root to leaf, Huffman coder encodes from leaf to root.
    - S-F starts with MSB for each symbol.
      Huffman starts with LSB for each symbol.
  - Huffman code is optimal if each symbol is assigned one codeword.
  - Huffman code is not unique
    - Minimum Variance Huffman code
      - Merge nodes with less depth

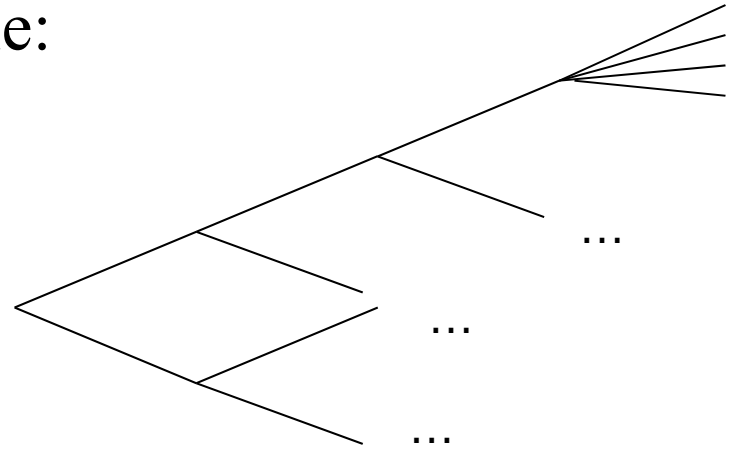
Modified Algorithm

- Disadvantages:
  - Large computation complexity for a large amount of symbols case.
  - Static (fixed) Huffman codeword table can severely affect the coding performance.
  - Not suitable for binary symbols with skewed prob.
  - Unequal decoding time for each symbol
- Modified method:
  - Truncated Huffman
  - Extended Huffman
  - Adaptive Huffman



Truncated Huffman Codes

- Only the most probable K symbols are generated by Huffman codes. The rest symbols are represented by a prefix code followed by a fixed length code
- Example:



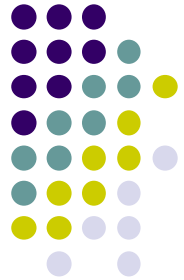
- JPEG, MPEG

Extended (Vector) Huffman Code

- Huffman coding very inefficient for *H(X)<<1* bit/symbol
- Remedy
  - Combine *n* successive symbols to a new "block-symbol"
- Huffman code for block-symbols
- Redundancy
  - $H(\mathbf{X}) <= R^n < H(\mathbf{X})+1$, $H(\mathbf{X})/n <= R < H(\mathbf{X})/n+1/n$
  - iid: $H(X) <= R < H(X)+1/n$
- Disadvantage: exponentially growing alphabet size
- Example:
  - *X*= {0, 1}, P (0) = 0.1, P(1) = 0.9 , H(X) = 0.469, L(Cn) = 1
  - 00 0.01 } 0.10 } 0.19 } 1.00 111
    01 0.09 110
    10 0.09 10
    11 0.81 0

    L' (Cn) = 1.29 / 2 = 0.645

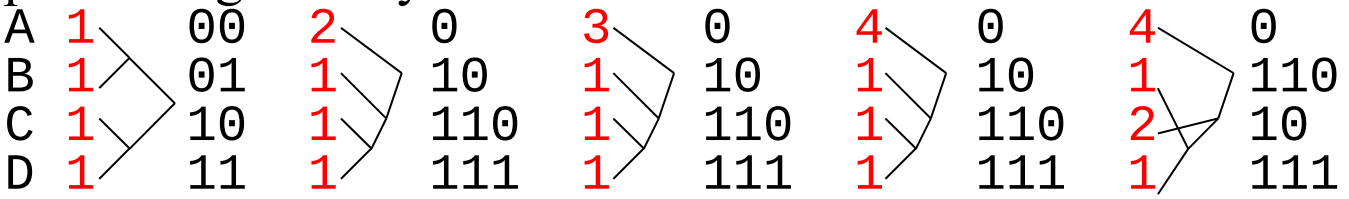
Adaptive Huffman Coding



- A fixed estimated probability table performs poorly for a large amount of data. Adaptive estimated probability model can achieve better results.
  - No need to transmit the codebook
- Based on the causal data information (no knowledge about the future statistics), adaptive Huffman coder can proceed on the fly.
- COMPACT in UNIX uses adaptive Huffman coding

Adaptive Huffman Coding

- Basic idea by example:
- {A, B, C, D}
  To encode A A A C…
  - Both encoder and decoder initially assume that each symbol appears once (each with prob. 1/4).
  - Synchronized update of the tree by encoder and decoder after processing each symbol

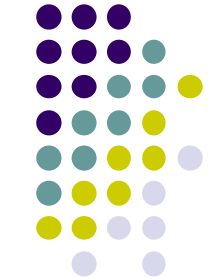


  - Output: 00 0 0 110…
- Problem?
  - How to find an efficient way for updating the tree?
  - 256 symbols 8bits->7bits->6bits->5bits…

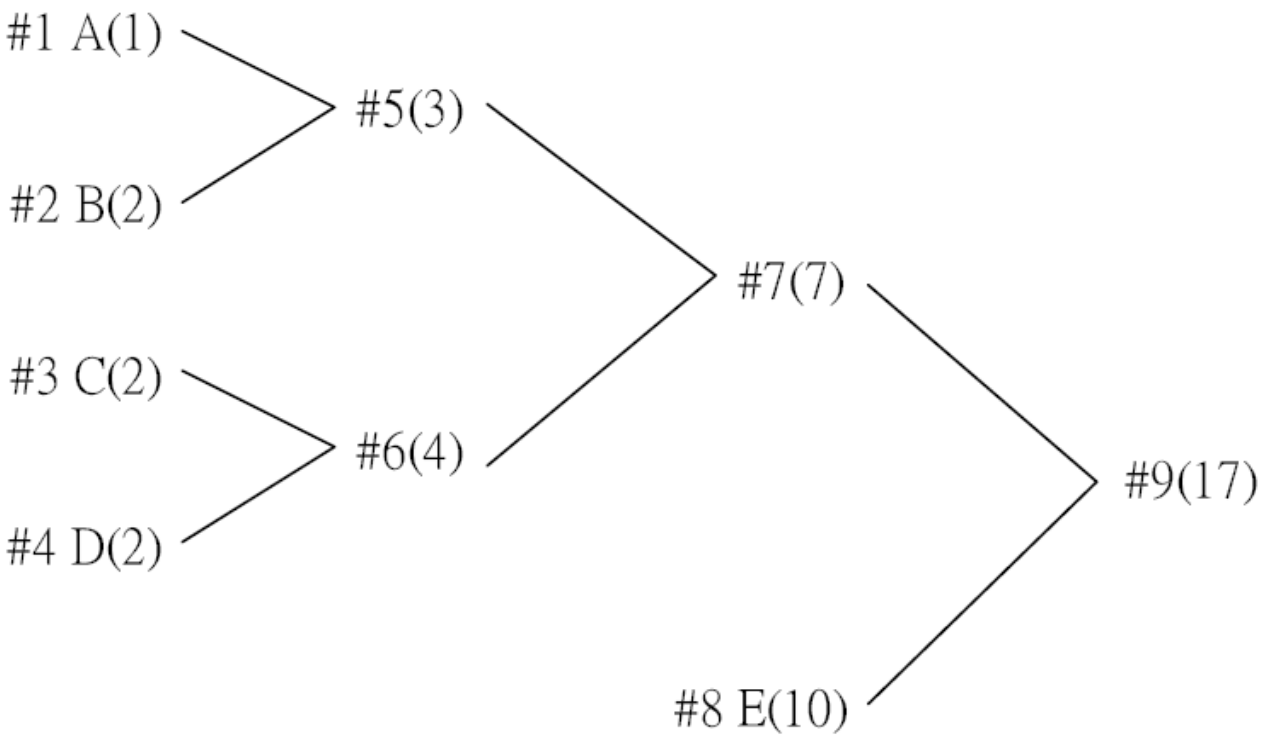
Adaptive Huffman Coding

- Observation of a Huffman tree:
  - It's a binary tree with a weight assigned to each node where the weight is proportional to the probability.
  - Sibling property:
    Each node can be listed in an order with increasing weight from left to right and from button to top. (If we sort all the nodes according to their weights, each node will be adjacent to its sibling.)
- If a tree possesses the sibling property, it is a Huffman tree.
- We can use a numbered list of nodes and make sure that the correct order is maintained. (The larger the node number, the larger the weight.)

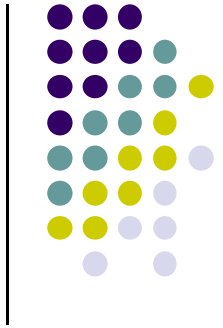
Adaptive Huffman Coding



- Example: {A, B, C, D, E}{1, 2, 2, 2, 10}
  - We use the number of occurrence as the weight
  - We will see what happens to the tree when we process repeated 'A'



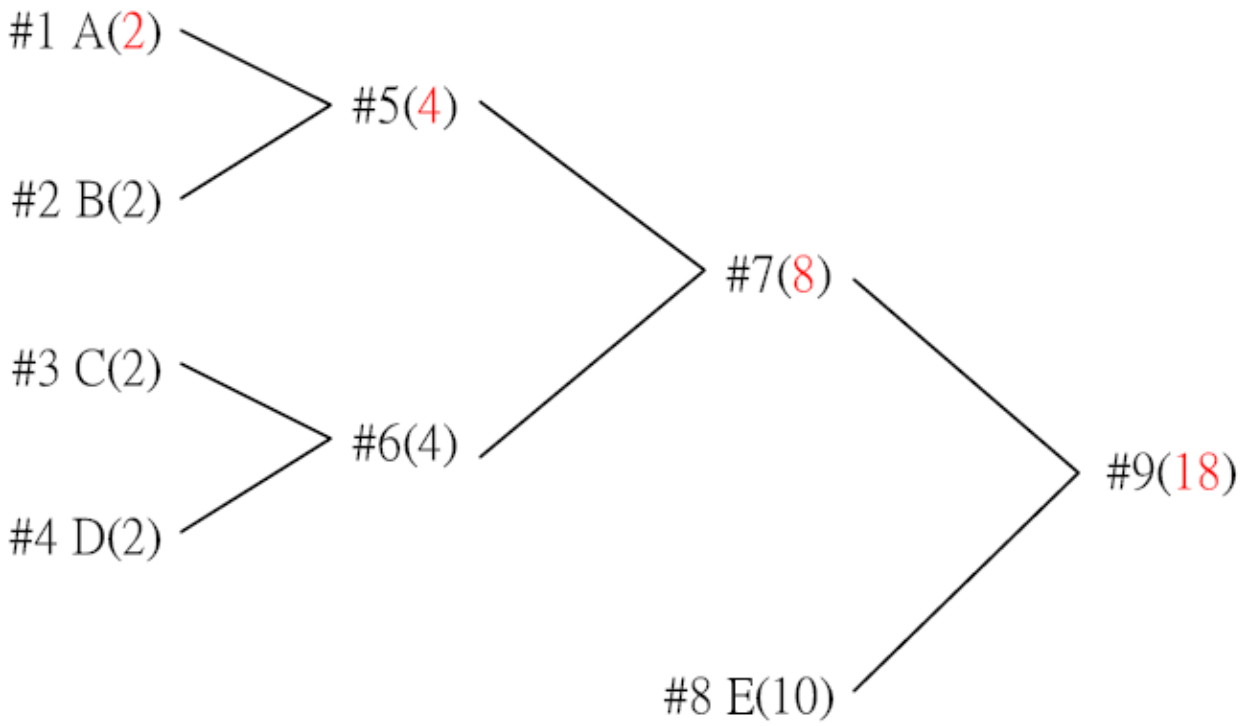
Adaptive Huffman Coding



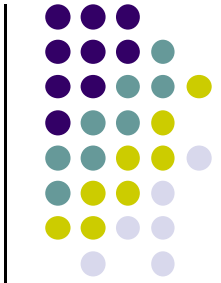
- Weight increasing process:
  - Increase the value of weight of a node will cause the change in all of this node's parent nodes.
- Switching process:
  - After each weight increasing, we need to check if the tree satisfies the sibling property or not. If against the sibling property, it needs to do the switching process.
    - Switch the node against the property to the most right and up node with the weight less than it.
- Basically, maintaining the sibling property during the update assures that we have a Huffman tree before and after the counts are updated.

Adaptive Huffman Coding

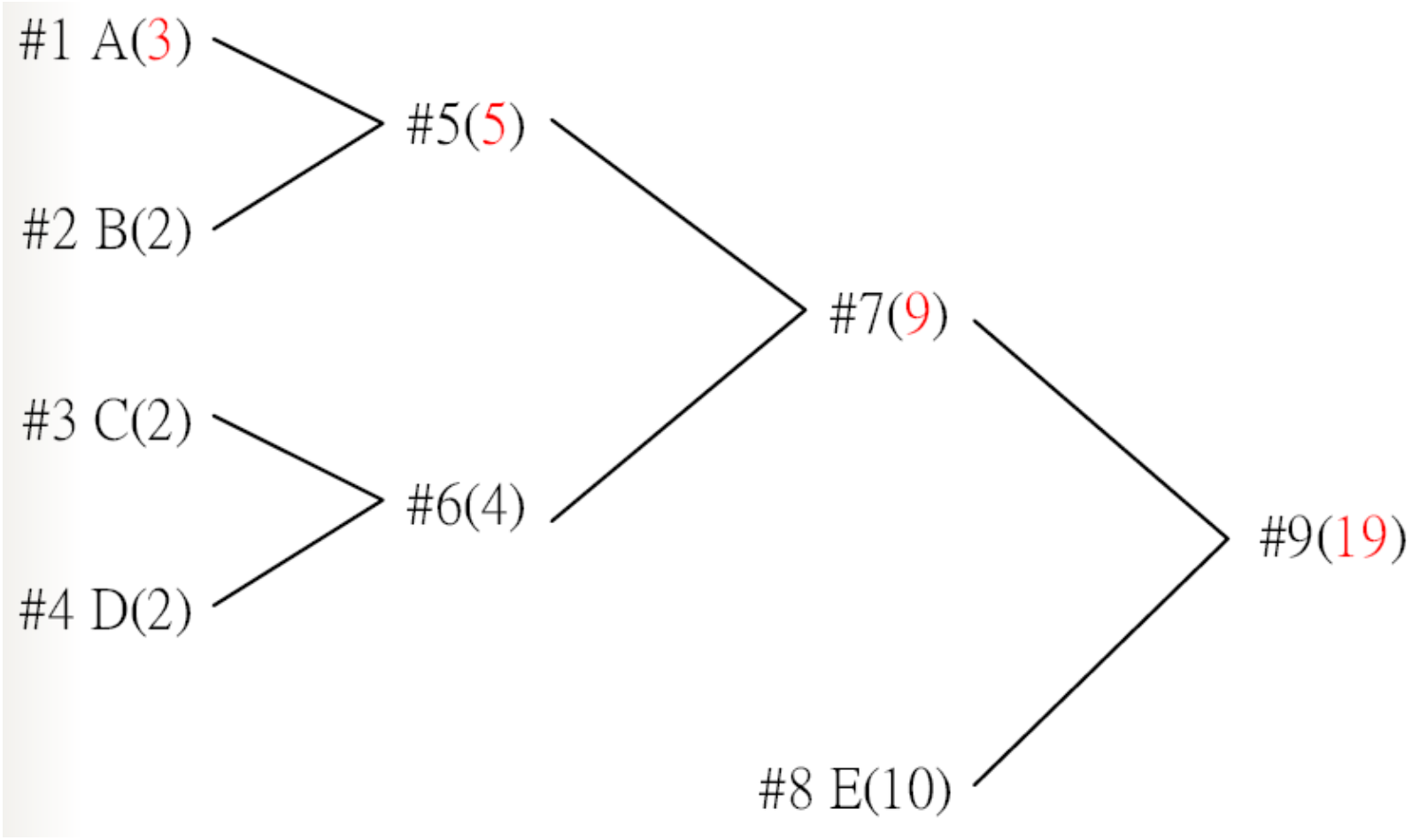
- Update the node ‘A’ and all of its parents. Huffman tree is maintained in each case



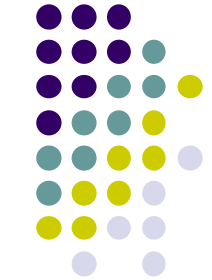
Adaptive Huffman Coding



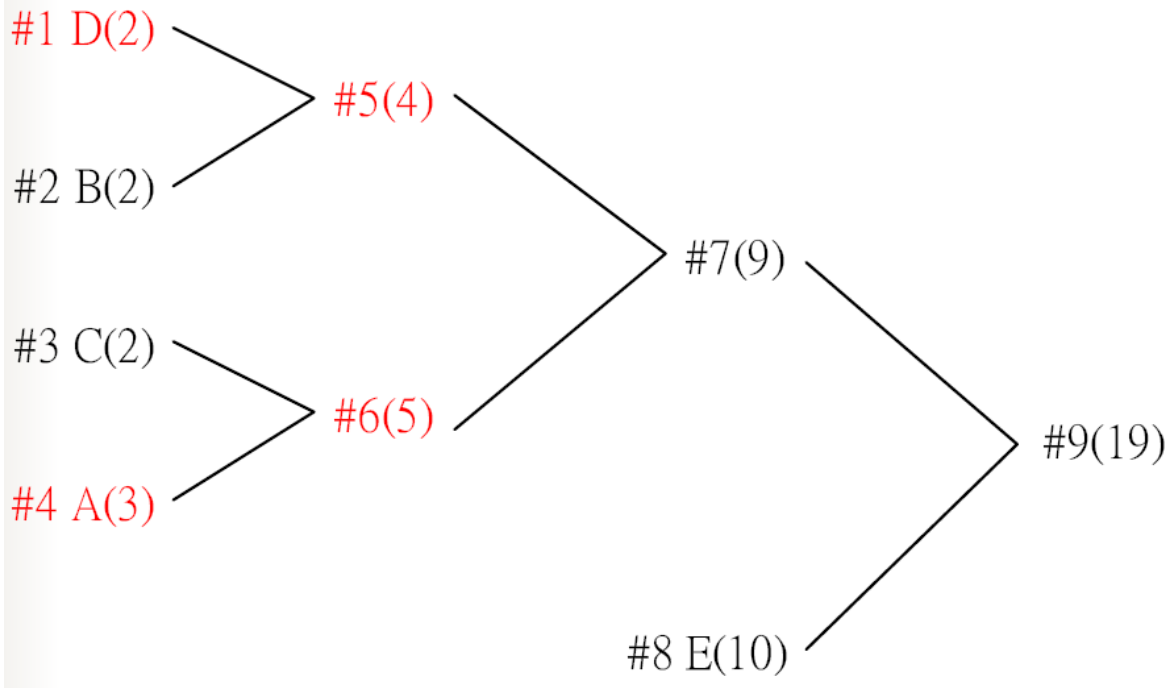
- The tree that does not obey our sibling property



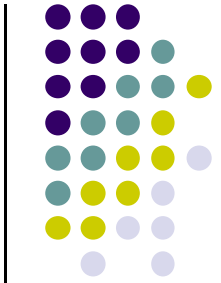
Adaptive Huffman Coding



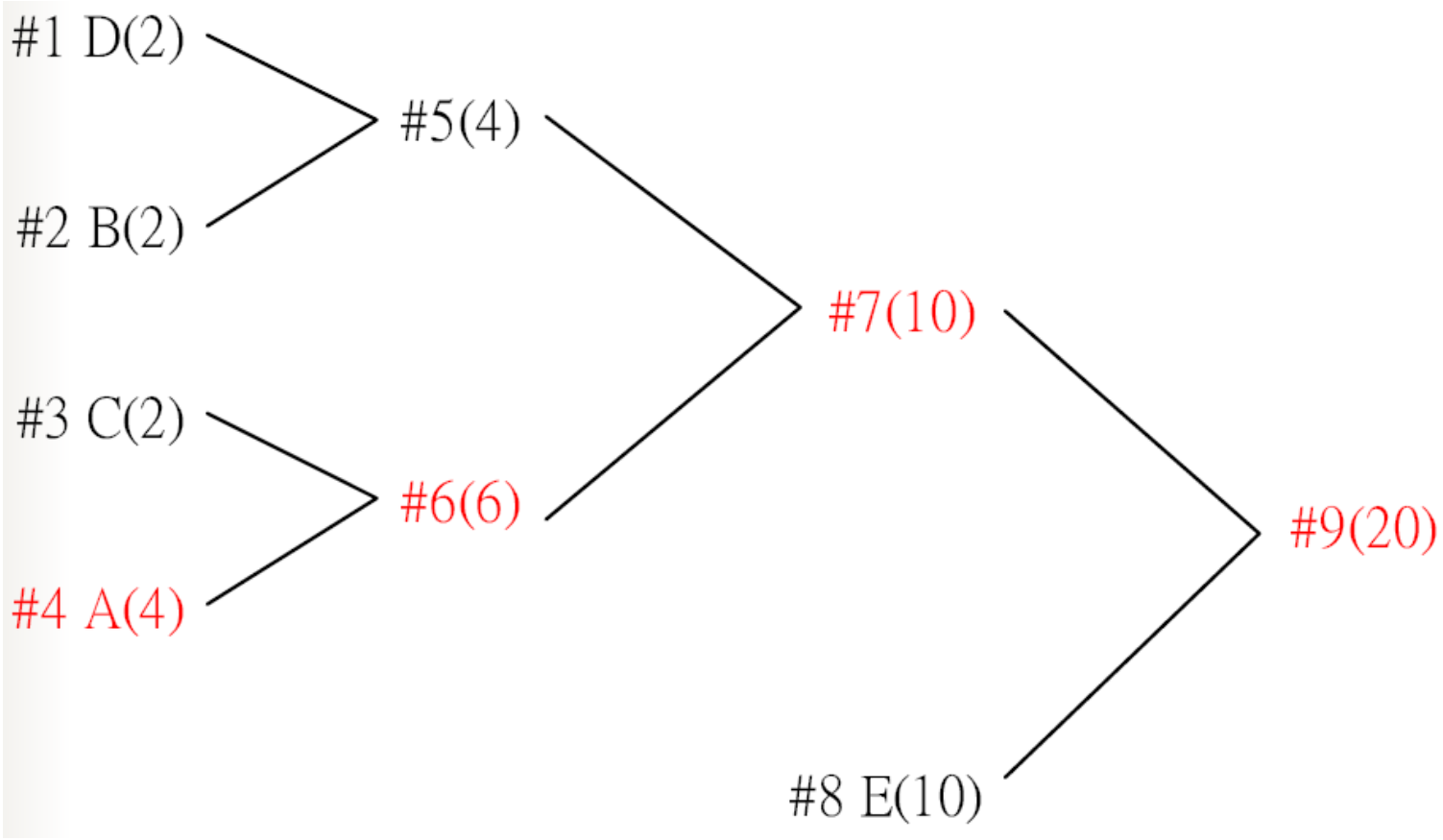
- D is swapped with A.
- The next node to be incremented will be the new parent of the incremented node. In this example, it is node #6.
- As each node is incremented, a check is performed for correct ordering. A swap is performed if necessary.



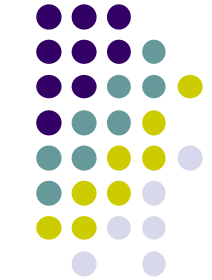
Adaptive Huffman Coding



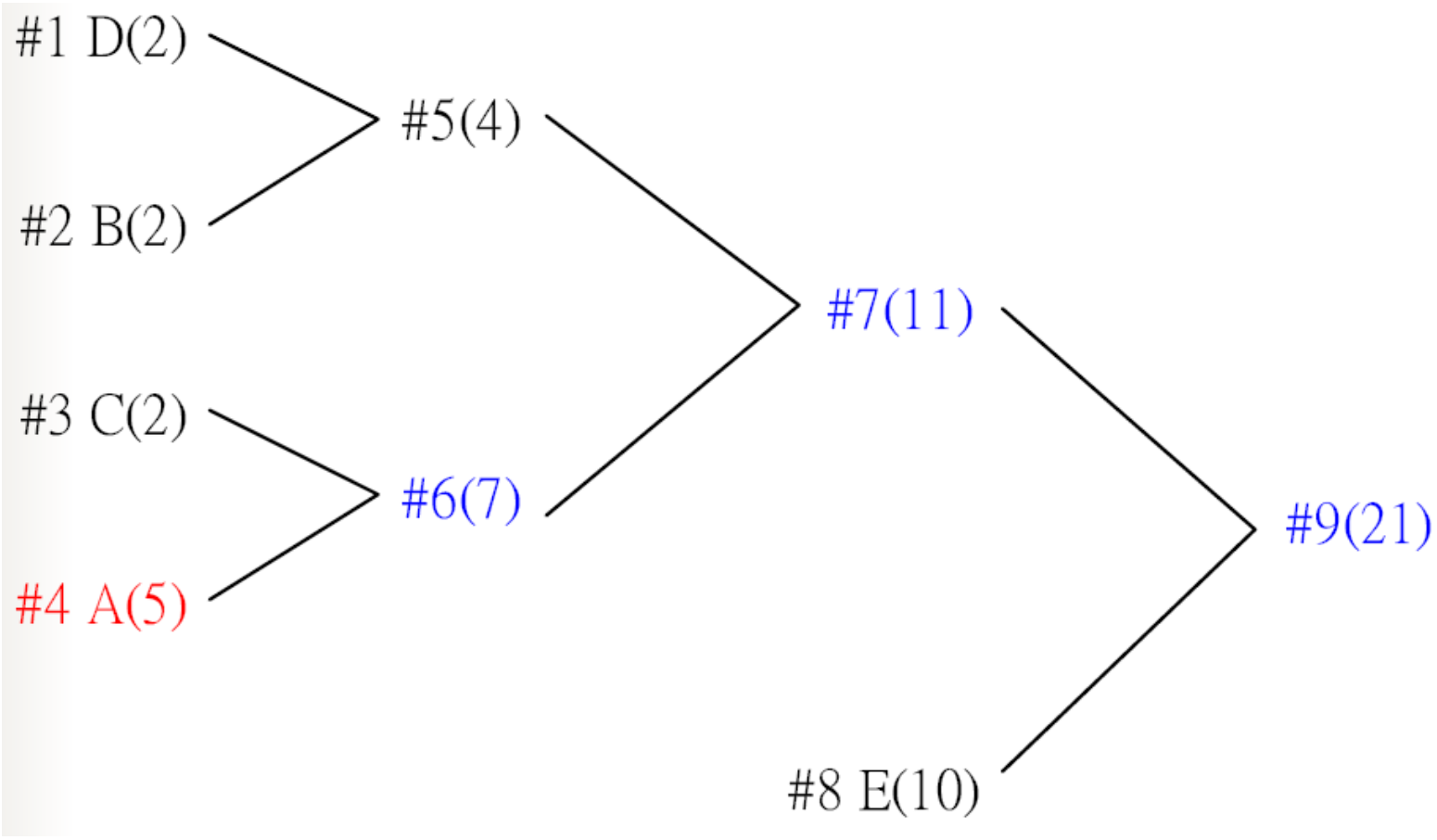
- ‘A’ comes again…



Adaptive Huffman Coding

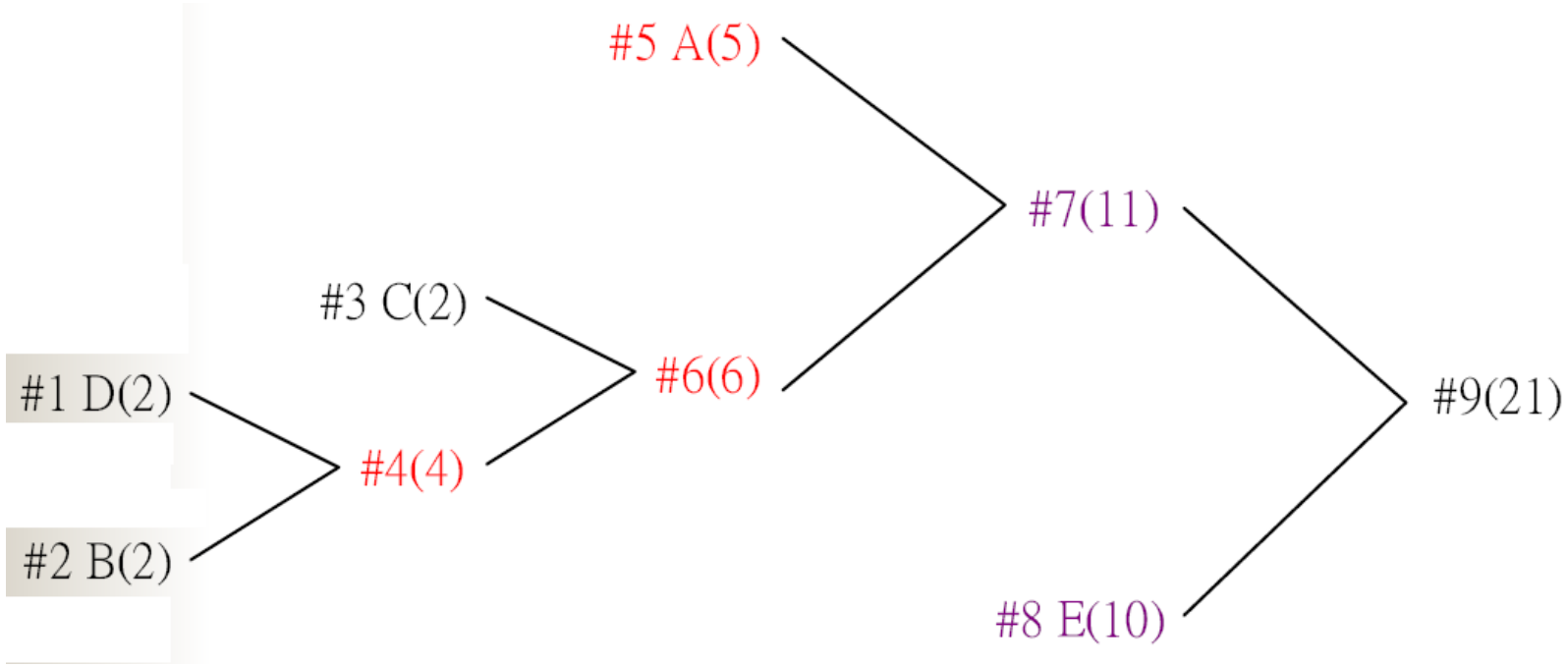


- ‘A’ comes again. We have to swap nodes this time.

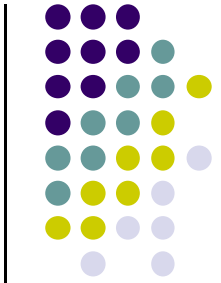


Adaptive Huffman Coding

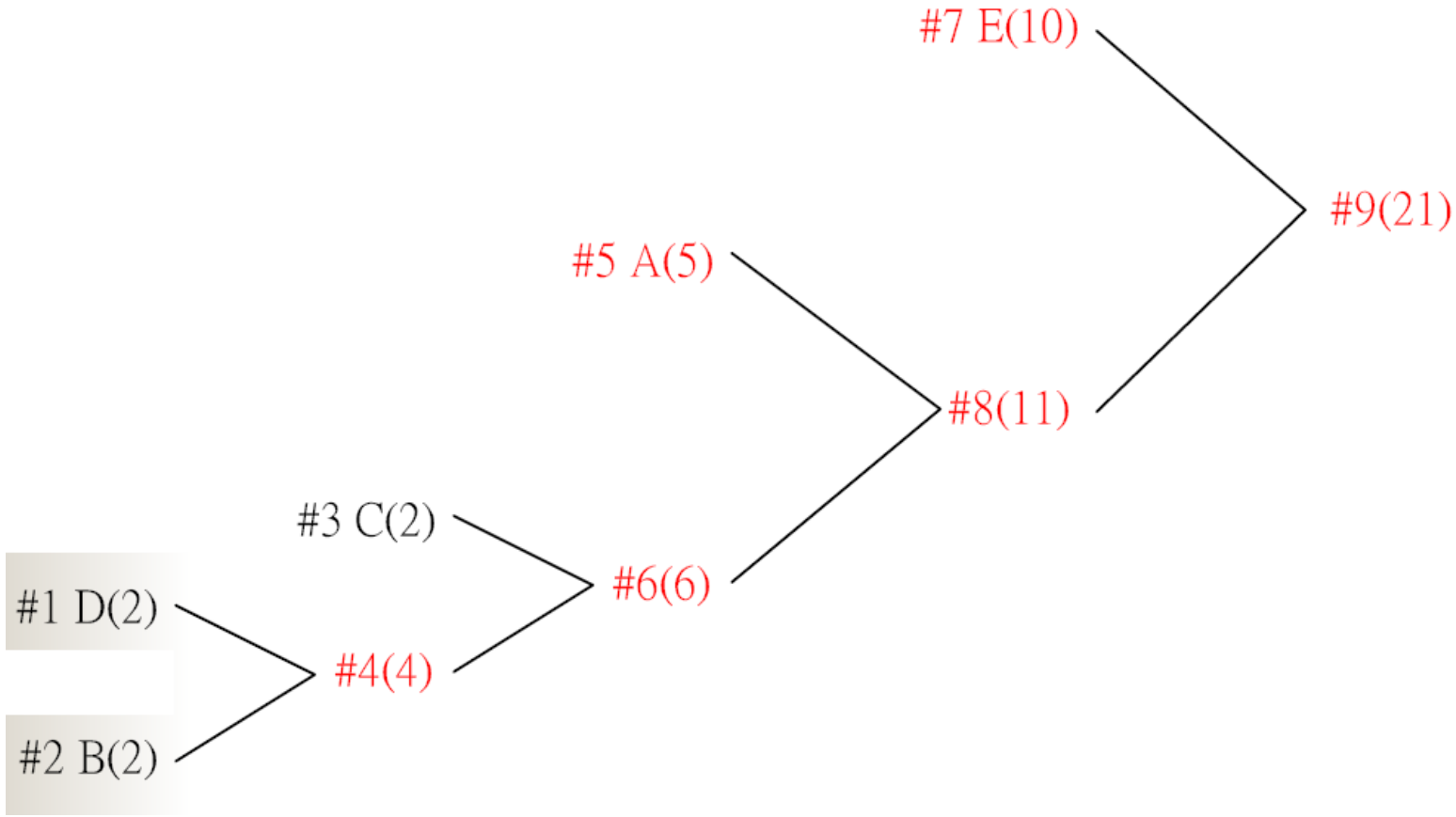
- Still not a Huffman tree when dealing with #7



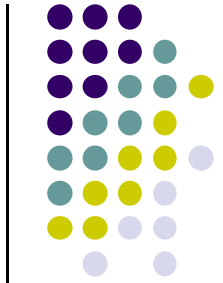
Adaptive Huffman Coding



- Huffman tree now

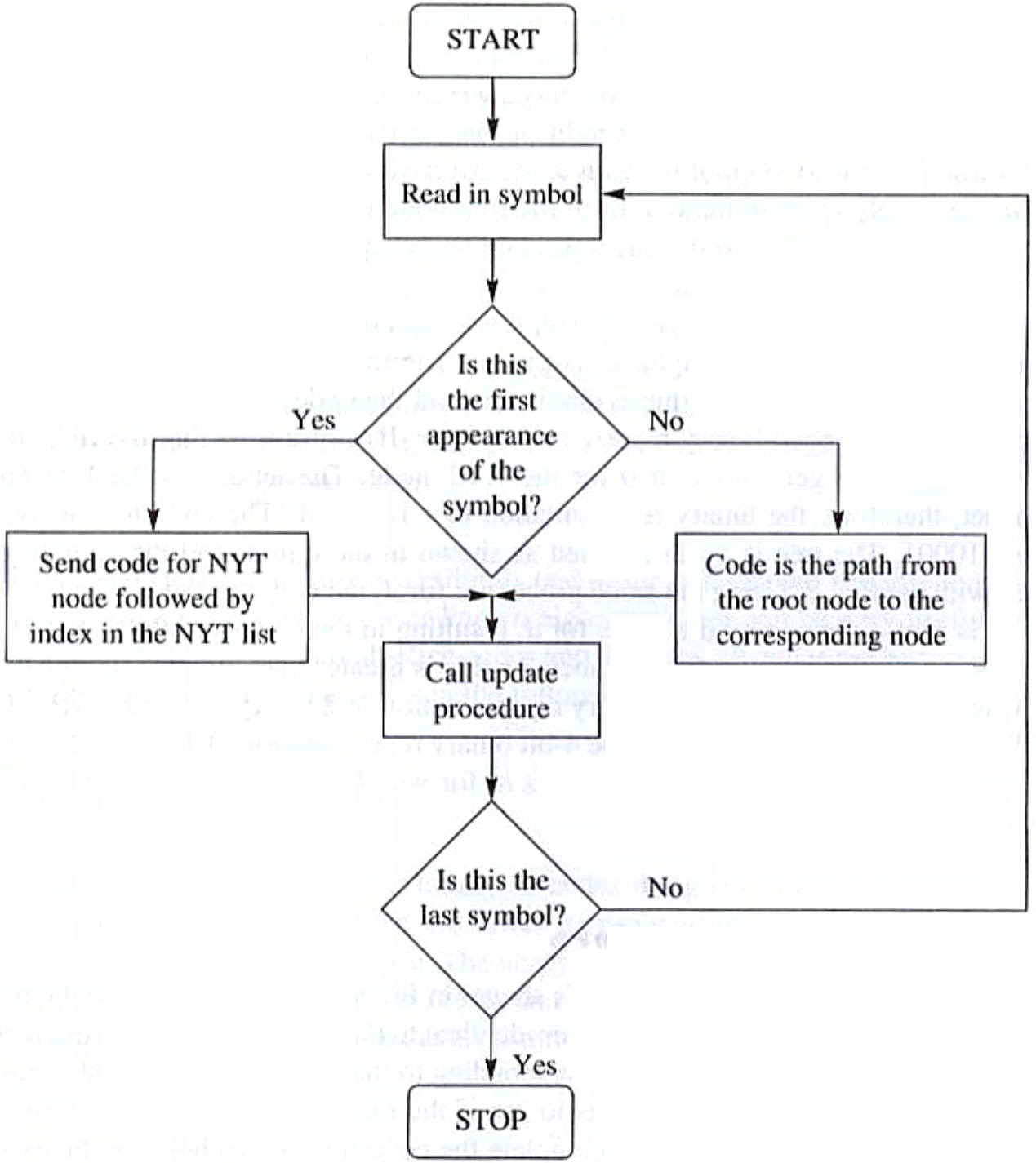


Adaptive Huffman Coding

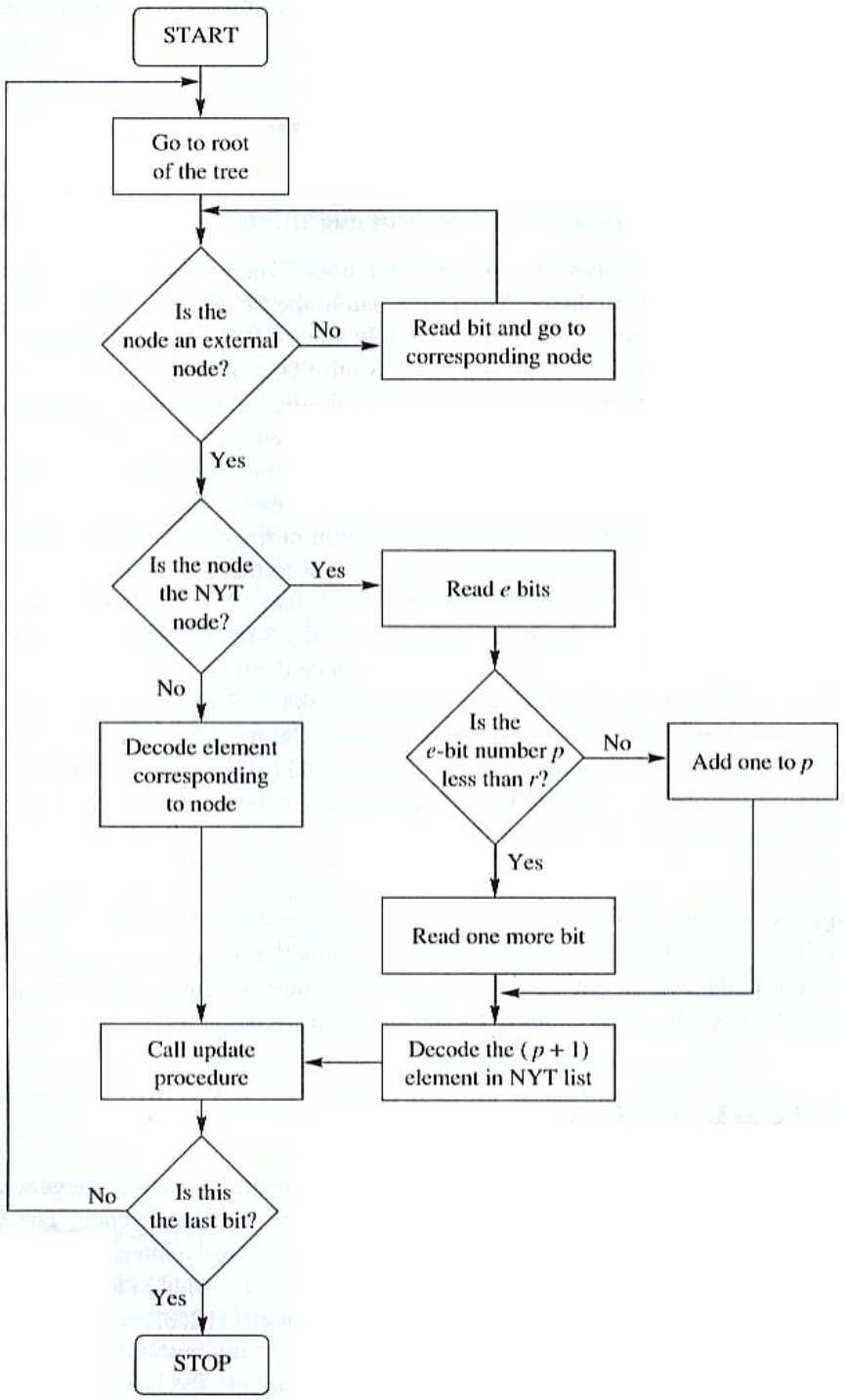


- Implementation:
  - The ordering is preserved by numbering the node.
  - Initial condition: Not Yet Transmitted (NYT)
  - The largest node number is given to the root.
  - The smallest node number is given to the NYT.
  - The nodes with the same weight are grouped as a "block".
  - When "switch" is needed, switch the targeted node with the node which has the largest node number in the block (and then increase the weight.)
- Encoder:
  - Read in symbol.
  - If first appear: send NYT + index of the symbol in the NYT list.
  - Else, code from root to leaf.
  - Update.
- Decoder
  - Read in bit.
  - If it's NYT: Fixed length decode.
  - Else: decode the symbol.
  - Update.

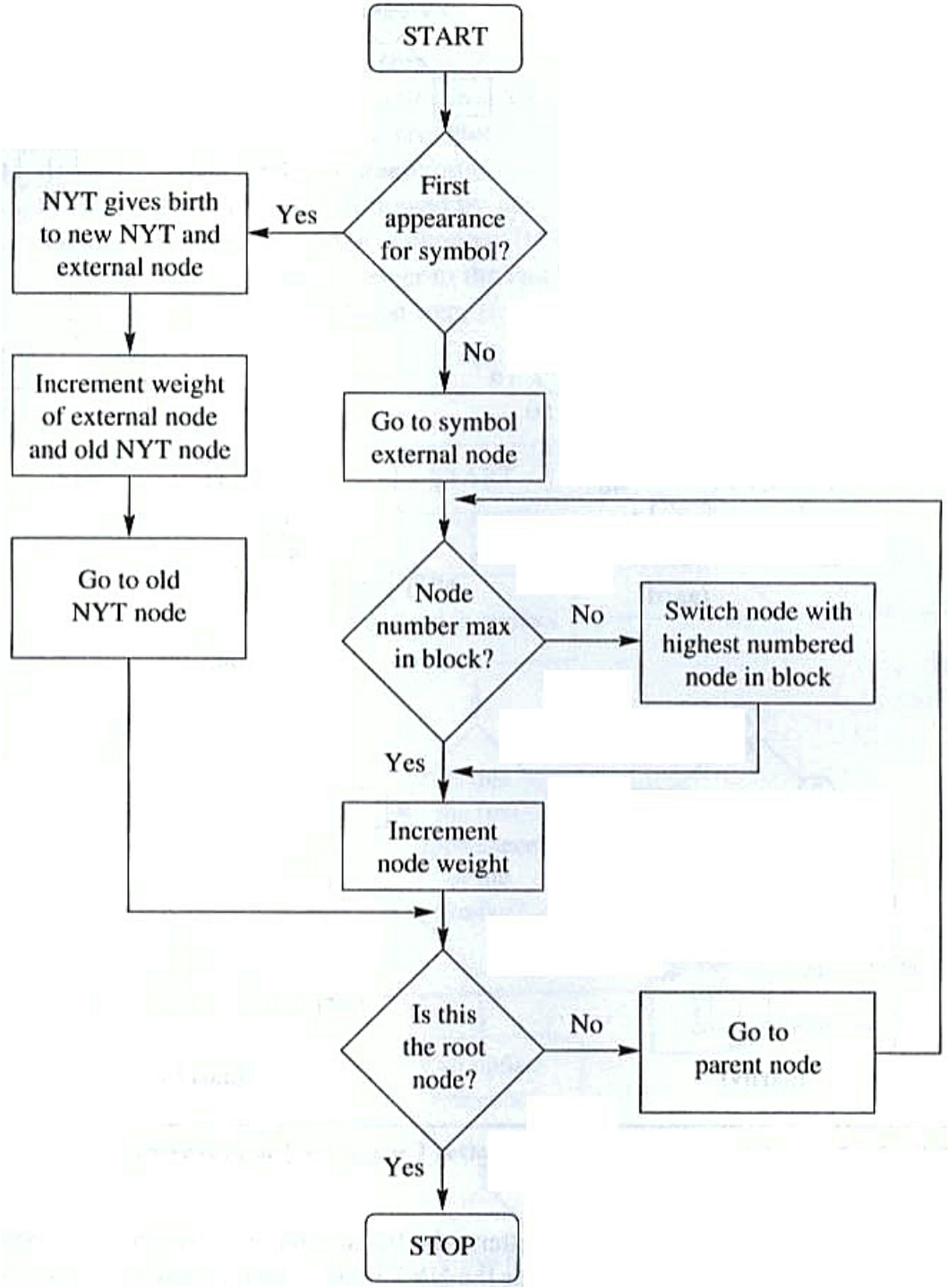
Encoding


START
Read in symbol
Is this the first appearance of the symbol?
Yes
No
Send code for NYT node followed by index in the NYT list
Code is the path from the root node to the corresponding node
Call update procedure
Is this the last symbol?
No
Yes
STOP

Decoding


START
Go to root of the tree
Is the node an external node?
No
Read bit and go to corresponding node
Yes
Is the node the NYT node?
Yes
Read e bits
No
Decode element corresponding to node
Is the e-bit number p less than r?
No
Add one to p
Yes
Read one more bit
Call update procedure
Decode the $(p+1)$ element in NYT list
No
Is this the last bit?
Yes
STOP

Update Procedure



Example

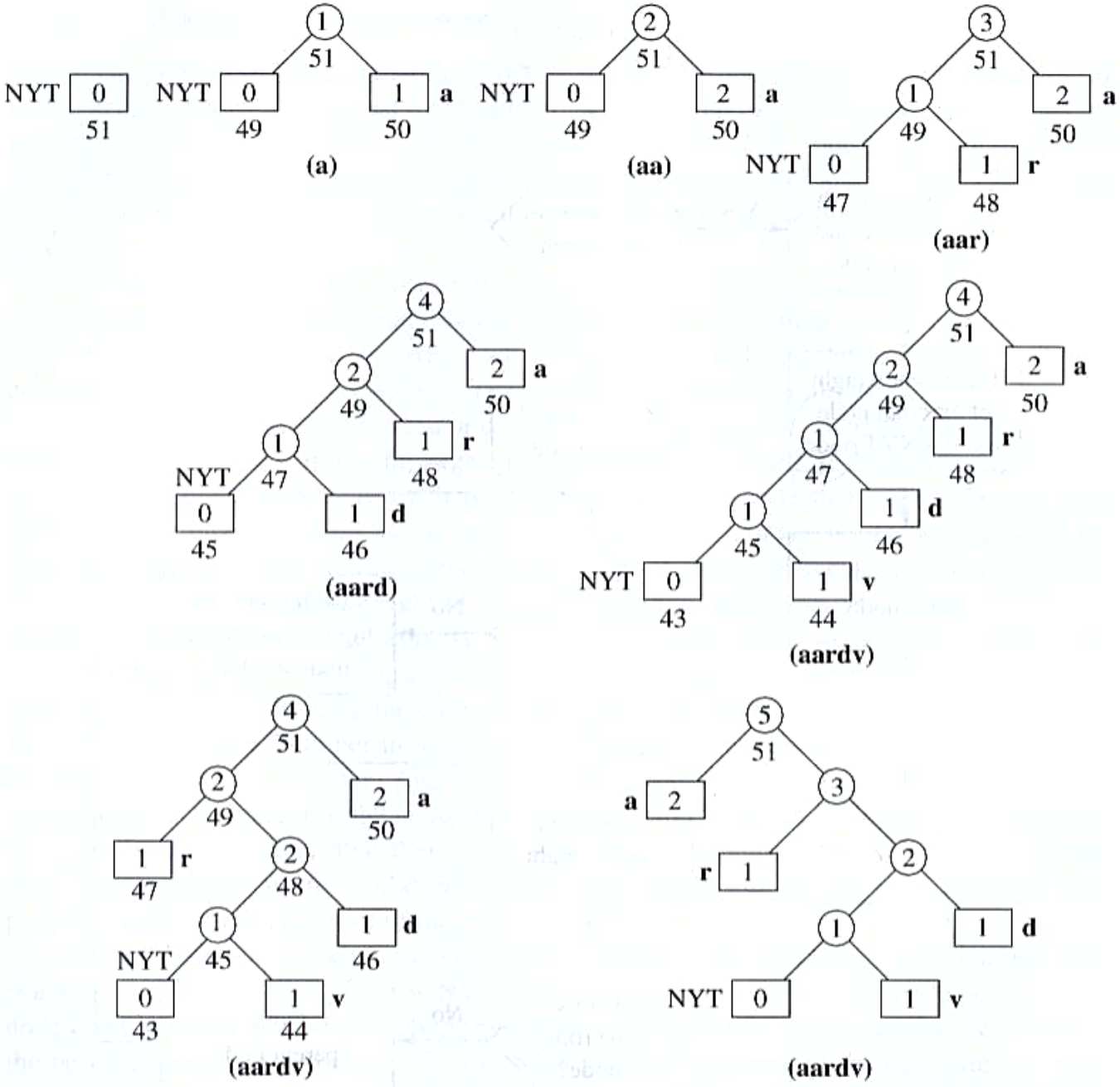


FIGURE 3.7 **Adaptive Huffman tree after [a a r d v] is processed.**

Run-Length Coding

- For sources that emit "runs" of identical symbols. Replace a sequence $\{x_n\}$ by a short sequence of symbol pair $\{a_k, r_k\}$, where a_k is the length of run and r_k is the symbol. Entropy coding (e.g., Huffman coding) can be applied on new symbol pairs $\{a_k, r_k\}$
  - Example:
    - 4 4 4 2 2 2 2 1 5 5… -> {3,4}{4,2}{1,1}{2,5}…
- Special case (when 0 appears very often)
  - {number_of_zero, followed_non_zero_symbol}
    - 0 0 0 3 0 0 0 0 2 0 0 5…->{3,3}{4,2}{2,5}…
  - For binary data with skewed probability
    - 0 0 0 0 1 0 0 0 1 0 1 1 0 0 0 0 1 0 1… -> 4 3 1 0 4 1…
- JPEG compression

run

Unary Code

n	Code	Alt. Code
1	0	1
2	10	01
3	110	001
4	1110	0001
5	11110	00001

- The unary code of the nonnegative integer n is defined as n-1 ones followed by a single zero (or alternatively, as n - 1 zeros followed by a single one). The length of the unary code for the integer n is thus n bits.
- The same as Huffman code for $P(n)=1/2^n$
- For geometric source with fast decay
- General form
  - Also known as start-step-stop codes.
  - (start, step, stop)
  - Codewords are created to code symbols used in the data, such that the *n*th codeword consists of *n* ones, followed by one 0, followed by all the combinations of *a* bits where *a* = start +*n* x step. If *a* = stop, then the single 0 preceding the *a* bits is dropped.

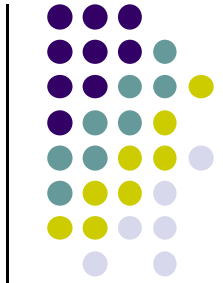
n	$a = 3 + n \cdot 2$	nth codeword	Number of codewords	Range of integers
0	3	$0xxx$	$2^3 = 8$	0–7
1	5	$10xxxxx$	$2^5 = 32$	8–39
2	7	$110xxxxxxx$	$2^7 = 128$	40–167
3	9	$111xxxxxxxxx$	$2^9 = 512$	168–679
		Total	680	

The General Unary Code (3,2,9).

n	$a = 2 + n \cdot 1$	nth codeword	Number of codewords	Range of integers
0	2	$0xx$	4	0–3
1	3	$10xxx$	8	4–11
2	4	$110xxxx$	16	12–27
3	5	$1110xxxxx$	32	28–59
	...	...		...
8	10	$\underbrace{11\ldots1}_{8}\underbrace{xx\ldots x}_{10}$	1024	1020–2043
		Total	2044	

The General Unary Code (2,1,10).

Golomb Code



- Idea: express x as x=mq+r
- Golomb code:
  - Encode q by unary code and encode r by a modified binary code and concatenate them
- Modified binary code
  - Choose a suitable m, calculate $q = \left\lfloor \frac{x}{m} \right\rfloor, \quad r = x - qm, \text{ and } c = \lceil \log_2 m \rceil,$
  - The first 2^c-m values of r are coded, as unsigned integers, in *c-1* bits each and the rest are coded in *c* bits each (ending with the biggest *c*-bit number, which consists of *c* 1's).
  - The case where m is a power of 2 ($m = 2^c$) is a special (common) case because all are coded with *c* bits – Rice code.
  - We know that x = r + qm; so once a Golomb code is decoded, the values of q and r can be used to easily reconstruct x.
- In practice, $m=2^c$ is often used so r can be encoded by constant length, $\log_2 m$.

Golomb Code

- m=3, r=0,1,2
  2^c=4 c=2
  2^c-m=4-3=1 1bit 0
  3-1=2 2bits 10 11

- m=5, r=0,1,2,3,4
  2^c=8 c=3
  2^c-m=8-5=3 2bits 00, 01, 10
  5-3=2 3bits 110, 111

m	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
c	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4
$2^c - m$	0	1	0	3	2	1	0	7	6	5	4	3	2	1	0

m \ x	0	1	2	3	4	5	6	7	8	9	10	11	12 ...
2	0|0	0|1	10|0	10|1	110|0	110|1	1110|0	1110|1	11110|0	11110|1	111110|0	111110|1	1111110|0
3	0|0	0|10	0|11	10|0	10|10	10|11	110|0	110|10	110|11	1110|0	1110|10	1110|11	11110|0
4	0|00	0|01	0|10	0|11	10|00	10|01	10|10	10|11	110|00	110|01	110|10	110|11	11110|00
5	0|00	0|01	0|10	0|110	0|111	10|00	10|01	10|10	10|110	10|111	110|00	110|01	110|10
6	0|00	0|01	0|100	0|101	0|110	0|111	10|00	10|01	10|100	10|101	10|110	10|111	110|00
7	0|00	0|010	0|011	0|100	0|101	0|110	0|111	10|00	10|010	10|011	10|100	10|101	10|110
8	0|000	0|001	0|010	0|011	0|100	0|101	0|110	0|111	10|000	10|001	10|010	10|011	10|100
9	0|000	0|001	0|010	0|011	0|100	0|101	0|110	0|1110	0|1111	10|000	10|001	10|010	10|011
10	0|000	0|001	0|010	0|011	0|100	0|101	0|1100	0|1101	0|1110	0|1111	10|000	10|001	10|010
11	0|000	0|001	0|010	0|011	0|100	0|1010	0|1011	0|1100	0|1101	0|1110	0|1111	10|000	10|001
12	0|000	0|001	0|010	0|011	0|1000	0|1001	0|1010	0|1011	0|1100	0|1101	0|1110	0|1111	10|000
13	0|000	0|001	0|010	0|0110	0|0111	0|1000	0|1001	0|1010	0|1011	0|1100	0|1101	0|1110	0|1111

Some Golomb Codes for $m = 2$ Through 13.

Adaptive Golomb Code

- To code binary data 0 and 1. Assume 0 appears more often. The number of 0 is L and the number of 1 is N.

$L = 0$; % initialize
$N = 0$;
$m = 1$; % or ask user for m
% main loop
for each run of r zeros do
  construct Golomb code for r using current m.
  write it on compressed stream.
  $L = L + r$: % update L, N, and m
  $N = N + 1$;
  $p = L/(L + N)$;
  $m = \lfloor -1/\log_2 p + 0.5 \rfloor$;

Exp-Golomb Code

Table 6.9 Exp-Golomb codewords

code_num	Codeword
0	1
1	010
2	011
3	00100
4	00101
5	00110
6	00111
7	0001000
8	0001001
...	...
	[M zeros][1][INFO]

- Encode

$$M = \text{floor}(\log_2[\text{code\_num} + 1])$$

$$\text{INFO} = \text{code\_num} + 1 - 2^M$$

- Decode

1. Read in M leading zeros followed by 1.
2. Read M-bit INFO field.
3. code_num = 2^M + INFO – 1

- Exp-Golomb code of order k
  - Take the number in binary except for the last *k* digits and add 1 to it (arithmetically). Write this down.
  - Count the bits written, subtract one, and write that number of starting zero bits preceding the previous bit string.
  - Write the last *k* bits in binary.

For k = 0 the code begins:
0 => 1 => 1
1 => 10 => 010
2 => 11 => 011
3 => 100 => 00100
4 => 101 => 00101
5 => 110 => 00110
6 => 111 => 00111
7 => 1000 => 0001000
8 => 1001 => 0001001

Tunstall Code

- All codewords are of equal length and each codeword represents a different number of letters
  - Huffman: F -> V, Tunstall: V -> F
- Example {A, B}
  AAA 00
  ABA 01
  AB 10
  B 11
  AAABAAB -> 0011?
- Requirements:
  - Be able to parse a source output sequence into symbol.
  - Maximize the average number of source symbols represented by each codeword.
- Tunstall code example
  - {A, B, C} P(A)=0.6, P(B)=0.3, P(C)=0.1

```
A0.6 B0.3 C0.1
B0.3 C0.1 AA0.36 AB 0.18 AC0.06
B0.3 C0.1 AB0.18 AC0.06 AAA0.216 AAB 0.108 AAC0.036
000  001  010    011    100      101       110
```

- Error resilience
  - Error in Tunstall code will not propagate