Chapter 11

Data Compression

Objectives

How to make a file smaller to use a disk more efficiently?

How to make a content smaller for decrease the bandwidth of a network?

How do applications for compress-decompress files work?

Discuss the following topics:

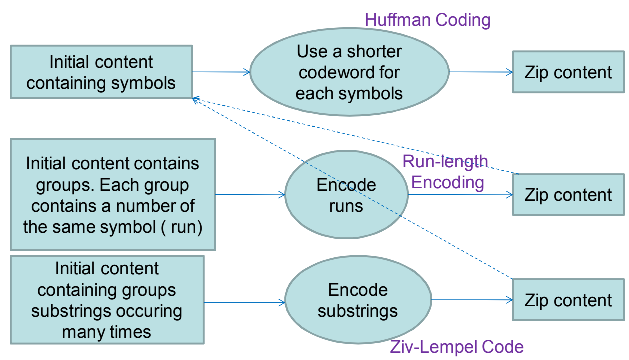
- Conditions for Data Compression
- Huffman Coding
- Run-Length Encoding
- Ziv-Lempel Code
- Case Study: Huffman Method with Run-Length Encoding

Introduction

- P(m): Probability of occurrence for the codeword m
- L(m)= -log(P(m)): minimum length of the codeword m
- The information content of the set *M,* called the **entropy** of the source *M,* is defined by:

  $L_{ave} = P(m_1)L(m_1) + \cdot \cdot \cdot + P(m_n)L(m_n)$
- Cloud E. Shannon, 1948: : L_{ave} is the best possible average length of a codeword when source symbols and the probabilities of their use are known.
- To compare the efficiency of different data compression methods when applied to the same data: Use the **compression rate** $\dfrac{\text{length(input) – length (output)}}{\text{length(input)}}$

Introduction



Conditions for Data Compression

1- **Each codeword corresponds to exectly one symbol**.

2- Decoding should not require any lookahead→ Use prefix code. **Each codeword can not be the prefix of another codeword.**

3- The length of the codeword for a given symbol m_j **should not exceed** the length of the codeword of a less probable symbol m_i; that is if $P(m_i) \le P(m_j)$ then $L(m_i) \ge L(m_j)$

4- In an optimal encoding system, there should not be any unused short codewords either as stand-alone encodings or as prefixes for longer codeword because this would mean that longer codewords were created unnecessarily

Giải thuật nén Huffman

- Nén tĩnh (Static Huffman)
- Nén động (Adaptive Huffman)

Huffman Coding

- The construction of an optimal code was developed by David Huffman, who utilized a tree structure in this construction: a binary tree for a binary code
- To assess the compression efficiency of the Huffman algorithm, a definition of the **weighted path length** is used

Nén tĩnh (Static Huffman)

Trong bản mã ASCII, mỗi ký tự được biểu diễn bằng chuỗi 8 bit.

Ý tưởng

- Giảm số bit để biểu diễn 1 ký tự
- Dùng chuỗi bit ngắn hơn để biểu diễn ký tự xuất hiện nhiều
- Sử dụng mã tiền tố để phân cách các ký tự

Ký tự	Mã bit
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101

Ký tự	Mã bit
A	000
B	001
C	010
D	011
E	100

Ký tự	Tần suất
A	9
B	15
C	10
D	6
E	7

Ký tự	Mã bit
A	000
B	1
C	01
D	011
E	100

Ký tự	Mã tiền tố
A	00
B	11
C	01
D	100
E	101

Cây Huffman

- Là cây nhị phân, mỗi nút chứa ký tự và trọng số (tần suất của ký tự đó).
- Mỗi ký tự được biểu diễn bằng 1 nút lá (tính tiền tố).
- Nút cha có tổng ký tự, tổng trọng số của 2 nút con.
- Các nút có trọng số, ký tự tăng dần từ trái sang phải.
- Các nút có trọng số lớn nằm gần nút gốc.
  Các nút có trọng số nhỏ nằm xa nút gốc hơn.

Mã Huffman

Là chuỗi nhị phân được sinh ra dựa trên cây Huffman.

Mã Huffman của ký tự là đường dẫn từ nút gốc đến nút lá đó.

- Sang trái ta được bit 0
- Sang phải ta được bit 1

Có độ dài biến đổi (tối ưu bảng mã).

- Các ký tự có tần suất lớn có độ dài ngắn.
- Các ký tự có tần suất nhỏ có độ dài dài hơn.

Thuật toán nén tĩnh (Static Huffman)

- **B1**: Duyệt file, lập bảng thống kê tuần suất xuất hiện của mỗi ký tự. — B1
- **B2**: Xây dựng cây Huffman dựa vào bảng thống kê. — B2
- **B3**: Sinh mã Huffman cho mỗi ký tự dựa vào cây Huffman. — B3
- **B4**: Duyệt file, thay toàn bộ ký tự bằng mã Huffman tương ứng. — B4
- **B5**: Lưu lại cây Huffman (bảng mã) dùng cho việc giải nén. Xuất file đã nén. — B5

Huffman Coding (continued)

`Huffman()`
    `for` *each symbol create a tree with a single root node and order all trees according to the probability of symbol occurrence*;
    `while` *more than one tree is left*
        take the two trees t_1, t_2 *with the lowest probabilities* p_1, p_2 ($p_1 \leq p_2$)
          and create a tree with t_1 *and* t_2 *as its children and with the probability in the new root equal to* $p_1 + p_2$;
    associate 0 with each left branch and 1 with each right branch;
    create a unique codeword for each symbol by traversing the tree from the root to the leaf containing the probability corresponding to this symbol and by putting all encountered 0s and 1s together;

1. Simple Example

Chuỗi ký tự cần nén

F = "ABABBCBBDEEEABABBAEEDDCCABBBCDEEDCBCCCCDBBBCAAA"

N = 47

Bảng tần suất xuất hiện

Ký tự	Tần suất
A	9
B	15
C	10
D	6
E	7

1. Simple Example

Xây dựng cây Huffman

Thuật toán tham lam

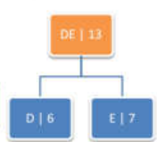
B1: Tạo N cây, mỗi cây chỉ có một nút gốc, mỗi nút gốc chỉ chứa một kí tự và trọng số (tần suất của ký tự đó). (N = số ký tự)

B2: Lặp lại thao tác sau cho đến khi chỉ còn 1 cây duy nhất:
+ Ghép 2 cây con có trọng số gốc nhỏ nhất thành 1 nút cha, có tổng ký tự, tổng trọng số trọng số của 2 nút con.
+ Xóa các cây đã duyệt.
+ Điều chỉnh lại cây nếu vi phạm tính chất.

Cont.

Xây dựng cây Huffman

Ký tự	Tần suất
B	15
C	10
A	9
E	7
D	6

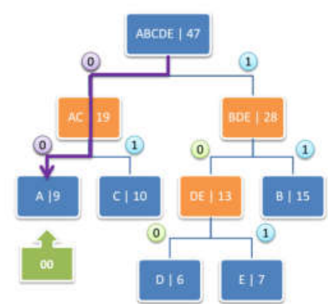


Ký tự	Tần suất
B	15
DE	13
C	10
A	9

Ký tự	Tần suất
AC	19
B	15
DE	13

Ký tự	Tần suất
BDE	28
AC	19

Ký tự	Tần suất
ABCDE	47



Bảng mã Huffman

Ký tự	Mã Huffman
A	00
B	11
C	01
D	100
E	101

Cont. → Result

Ký tự	Mã Huffman
A	00
B	11
C	01
D	100
E	101

Chuỗi ký tự cần nén

F = "ABABBCBBDEEEABABBAEEDDCCABBBCDEEDCBCCCCDBBBCAAA"

Chuỗi đã được nén

F_{Output} =
"0011001111011111110010110110100110011110010110110010001
01001111110110010110110001110101010110011111101000000"

Tiết kiệm: 8*47 - (2*9 + 2*15 + 2*10 + 3*6 + 3*7) = 376 - 107 = 269 bit
Tỷ lệ nén: (1 - 107/376)*100 = 72.54 %

Thuật toán giải nén

- **B1**: Xây dựng lại cây Huffman từ thông tin giải mã đã lưu.
- **B2:** Duyệt file, đọc lần lượt từng bit trong file nén và duyệt cây.
- **B3**: Xuất ký tự tương ứng khi duyệt hết nút lá.
- **B4**: Thực hiện B2, B3 cho đến khi duyệt hết file.
- **B5**: Xuất file đã giải nén.

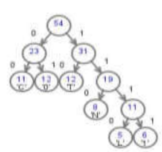
Bài tập cơ bản

Bài tập: Nén chuỗi sau bằng giải thuật nén tĩnh – Static Huffman

F =
"CNTT10110CLCCNNTTT10000CCCCLLLLCCCTTTT11000
NTNNN000TNT"

N = 54

Ký tự	Tần suất
0	12
1	6
C	11
L	5
N	8
T	12



Ký tự	Mã Huffman
0	01
1	1111
C	00
L	1110
N	110
T	10

Kết quả

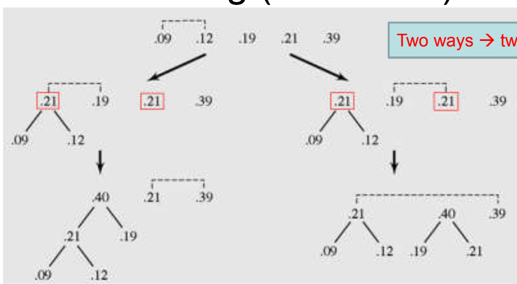
F_{Output} =
"0011010101111011111111110100111000001101101010101111010101010000000011
1011101110111000000001010101011111111010101110101101101100101011011010"

Ưu - Nhược điểm

Ưu điểm
- Hệ số nén tương đối cao.
- Phương pháp thực hiện tương đối đơn giản.
- Đòi hỏi ít bộ nhớ.

Nhược điểm
- Mất 2 lần duyệt file khi nén.
- Phải lưu trữ thông tin giải mã vào file nén.
- Phải xây dựng lại cây Huffman khi giải nén.

Advanced Example: Huffman Coding (continued)



Two ways → two result trees

Figure 11-1 Two Huffman trees created for five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09

Huffman Coding (continued)

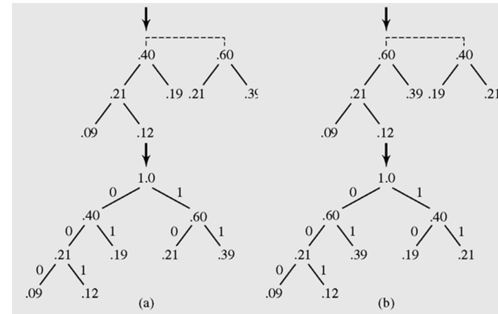


Figure 11-1 Two Huffman trees created for five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09 (continued)

Huffman Coding (continued)

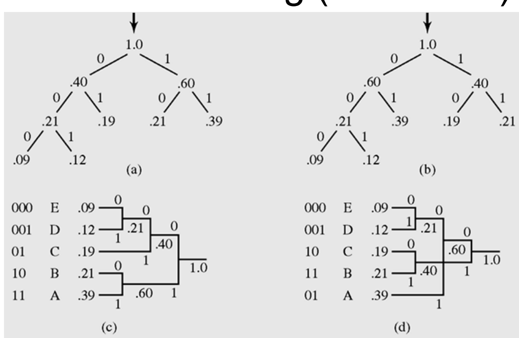


Figure 11-1 Two Huffman trees created for five letters A, B, C, D, and E with probabilities .39, .21, .19, .12, and .09 (continued)

Huffman Coding (continued)

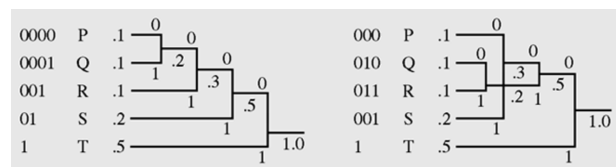


Figure 11-2 Two Huffman trees generated for letters P, Q, R, S, and T with probabilities .1, .1, .1, .2, and .5

If the selection a way to proceed grouping trees is not good, the result codewords may be longer.

Huffman Coding (continued)

```
createHuffmanTree(prob)
    declare the probabilities p1, p2, and the Huffman tree Htree;
    if only two probabilities are left in prob
            return a tree with p1, p2 in the leaves and p1 + p2 in the root;
    else remove the two smallest probabilities from prob and assign them to p1 and p2;
            insert p1 + p2 to prob;
            Htree = createHuffmanTree(prob);
            in Htree make the leaf with p1 + p2 the parent of two leaves with p1 and p2;
            return Htree;
```

Group two smallest probabilites in the prob to one then insert this new probability to the prob at suitable position. Proceed similarly until the prob contains only one element.

Huffman Coding (continued)

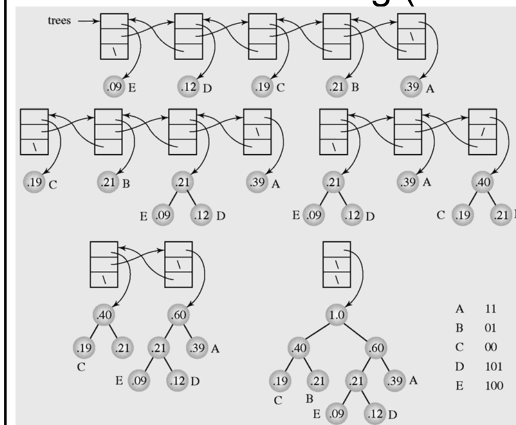


Figure 11-3 Using a doubly linked list to create the Huffman tree for the letters from Figure 11-1

Huffman Coding (continued)

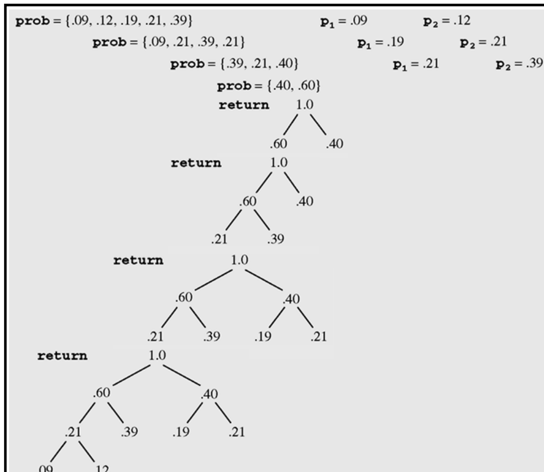


Figure 11-4 Top-down construction of a Huffman tree using recursive implementation

Huffman Coding (continued)

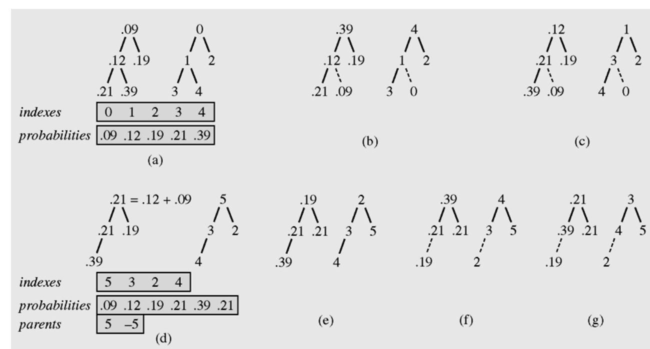


Figure 11-5 Huffman algorithm implemented with a heap

Huffman Coding (continued)

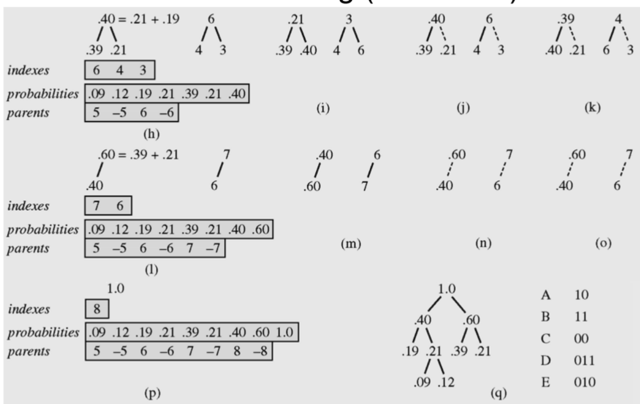
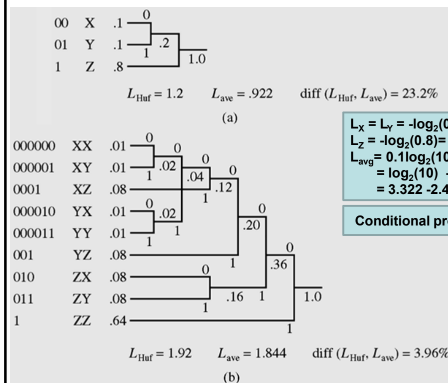


Figure 11-5 Huffman algorithm implemented with a heap.

(*)Bài tập: Huffman Coding (continued)



L_{Huf} = 2*0.1 + 2*0.1+ 1*0.8= 1.2

$L_X = L_Y = -\log_2(0.1) = -\log_2(1/10) = \log_2(10)$
$L_Z = -\log_2(0.8) = -\log_2(8/10) = \log_2(10) - \log_2(8)$
$L_{avg} = 0.1\log_2(10) + 0.1\log_2(10) + 0.8(\log_2(10) - \log_2(8))$
$= \log_2(10) - 0.8\log_2(8) = 3.322 - 0.8*3$
$= 3.322 - 2.4 = 0.922$

Conditional probability: $P_{XY} = P_x.P_y$

Figure 11-6 Improving the average length of the codeword by applying the Huffman algorithm to (b) pairs of letters instead of (a) single letters

Nén động (Adaptive Huffman)

Adaptive Huffman Coding

- An adaptive Huffman encoding technique was devised first by Robert G. Gallager and then improved by Donald Knuth.
- The algorithm is based on the **sibling property**: node for a symbol with higher occurrency must be nearer the root of the tree → code length is shorter → compresion rate is higher.
- In adaptive Huffman coding, the Huffman tree includes a counter for each symbol, and the counter is updated every time a corresponding input symbol is being coded.

Adaptive Huffman Coding (continued)

- Adaptive Huffman coding surpasses (vượt trội) simple Huffman coding in two respects:
  - It requires only one pass through the input
  - It adds only a codeword of a symbol to the output
- Both versions are relatively fast and can be applied to any kind of file, not only to text files
- They can compress object or executable files

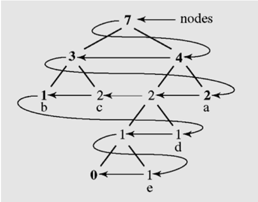
Cây Huffman

Tính chất anh em:
Trọng số của nút bên trái phải nhỏ hơn nút bên phải, nhỏ hơn nút cha

Nút NYT (not yet transmitted) có trọng số luôn = 0, dùng để nhận biết ký tự đã xuất hiện trong cây hay chưa.

Trọng số nút cha bằng tổng trọng số 2 nút con.

Adaptive Huffman Coding (continued)



Right to left breadth-first search this tree:
➔ 6 blocks: $block_7$, $block_4$, $block_3$, $block_2$, $block_1$, $block_0$. The first node in each block is shown in boldface **(sibling property)**

-Value (frequency) in a node is the sum of two subnodes
-This technique is used to generate a Huffman codinh tree by Robert G. Gallager anDonald Knuth.

Figure 11-7 Doubly linked list nodes formed by breadth-first right-to-left tree traversal

Thuật toán nén động (Adaptive Huffman)

B1: Duyệt tuần tự từng ký tự có trong file nhập.

TH1: Nếu ký tự chưa tồn tại:
+ Chuỗi bit: đường dẫn đến NYT + Mã bit của ký tự.
+ Chèn nút mới (Ký tự | trọng số = 1) vào NYT. Đánh lại số thứ tự.

TH2: Nếu ký tự đã tồn tại:
+ Chuỗi bit: đường dẫn đến ký tự đó.
+ Tăng trọng số của ký tự đó. (+1)

B2: + Tăng trọng số của các nút cha. (+1)
+ Nếu vi phạm tính anh em → điều chỉnh cho đến khi hết vi phạm.

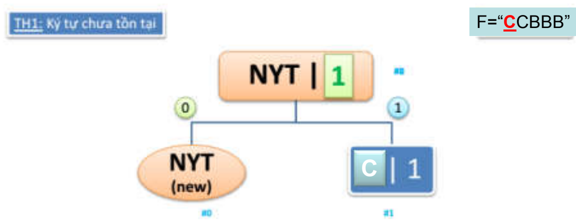
B3: Lưu chuỗi bit vào file xuất. Lặp lại B1, B2 đến khi duyệt hết file.

Thuật toán điều chỉnh

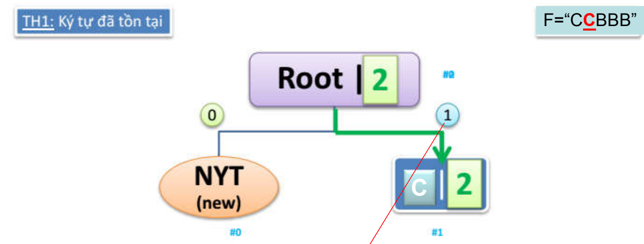
+ Nếu trọng số nút hiện hành > nút lân cận từ phải sang trái, từ dưới lên trên → Vi phạm.

+ Tìm nút xa nhất có trọng số cao nhất < trọng số nút vi phạm → Hoán đổi vị trí.

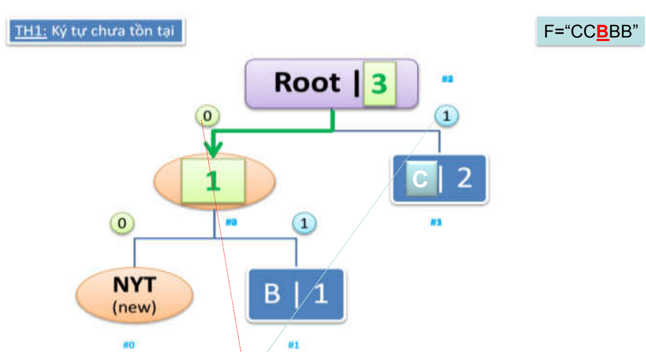
Simple Example



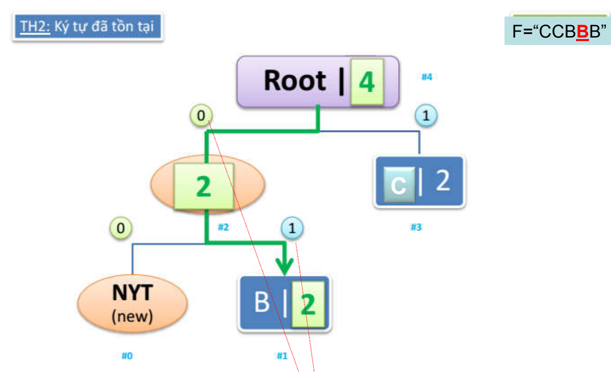
F_{Output} = 01000011



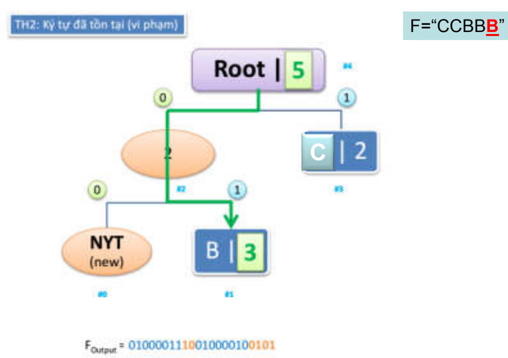
F_{Output} = 010000111



F_{Output} = 010000111001000010



F_{Output} = 01000011100100001001



Advanced Example: Adaptive Huffman Coding (continued)

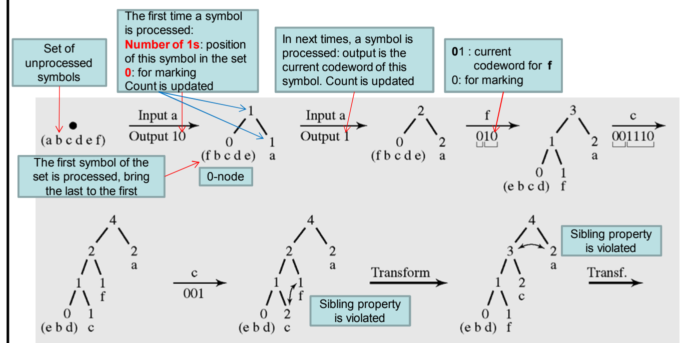


Figure 11-8 Transmitting the message "aafcccbd" using an adaptive Huffman algorithm

Adaptive Huffman Coding (continued)

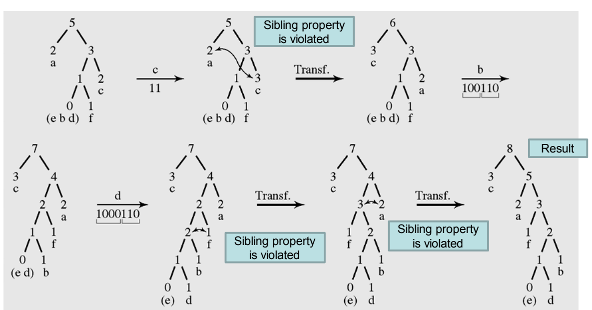


Figure 11-8 Transmitting the message "aafcccbd" using an adaptive Huffman algorithm (continued)

Run-Length Encoding

- A **run** is defined as a sequence of identical characters: "aaabbaccc" → 4 runs
- Specify a run: (n-number of occurrence, ch - character)
  - → "aaabbaccc" → 3a2b1a3c
  - →"111223" → "312213" → ???
  - → Another way is needed → an ASCII code represents the count. String of 43 consecutice letter "c" → "+c" , ASCII code of '+' is 43
- How to differentiate an abbreviated form or a literal character → A character for marker (cm) is needed → (cm, ch, n)
- In case of the count character is use → Use a marker such as: %% represents the character '%' (C language use this form).

Run-Length Encoding

➔ Run-length encoding is efficient only for text files in which only the blank character has a tendency to be repeated. **Null suppression compresses only runs of blanks** and eliminates the need to identify the character being compressed.

Which content is usually compressed using run-lengh method?

- Fax images (black and white bitmaps)
- Data in fixed-length columns of relational databases (blanks are appended automatically).

Drawbacks: If the content contains many run of 1 or 2 → the destination can be larger than source.

- Ex: **AAAAABBB → 2 runs, ABABABAB → 8 runs**

It can be used in a combination with the Huffman method. Case study of the chapter.

Ziv-Lempel Code

- The "pure form" of Huffman encoder: We have to know the frequecies of symbols (previous knowledge of the source characteristics) before codewords are assigned.
- In the adptive coding, the frequencies are updated during data transmission → **A universal coding scheme** → the source characteristics can not be known in advance.
- The Ziv-Lempel code is an example of a universal data compression code

Ziv-Lempel Code: LZ77 Coding

- Input symbols are move to a buffer in succession.
- Buffer include two equal parts, l1 positions(left haft) holds the l1 most recently encoded symbols from the input, l2 positions (right half) contains l2 symbol to be about encoded.
- In each iteration, a matching between two halfs to find out a substring matching a prefix of right part → output <match position, length of a match, the first nonmatching symbol>

1.Ý tưởng

Thuật toán LZW được phát triển theo nguyên lý tạo ra 1 dãy mã:

-Mã từ 0 đến 255 miêu tả 1 dãy ký tự thay thế cho ký tự 8-bit tương ứng.

-Mã từ 256 đến 4095 được tạo bên trong 1 từ điển cho trường hợp lặp chuỗi trong dữ liệu.

-Mỗi bước trong khi nén,byte nhập vào được tập hợp lại thành 1 chuỗi cho đến khi ký tự tiếp theo sẽ tạo thành 1 chuỗi chưa tồn tại trong từ điển,và 1 mã mới cho chuỗi được tạo sẽ được thêm vào từ điển,và mã ấy sẽ được xuất ra file output.

5.Ví dụ

- Chuỗi sau sẽ được mã hóa : "TOBEORNOTTOBEORTOBEORNOT#"

Symbol	Binary	Decimal
#	00000	0
A	00001	1
B	00010	2
C	00011	3
D	00100	4
E	00101	5
F	00110	6
G	00111	7
H	01000	8
I	01001	9
J	01010	10
K	01011	11

Symbol	Binary	Decimal
L	01100	12
M	01101	13
N	01110	14
O	01111	15
P	10000	16
Q	10001	17
R	10010	18
S	10011	19
T	10100	20
U	10101	21
V	10110	22

Symbol	Binary	Decimal
W	10111	23
X	11000	24
Y	11001	25
Z	11010	26

"TOBEORNOTTOBEORTOBEORNOT#"

Current Sequence	Next Char	Output	Extended Dictionary	Comment
Null	T			
T	O	20:10100	27:TO	
O	B	15:01111	28:OB	
B	E	2:00010	29:BE	
E	O	5:00101	30:EO	
O	R	15:01111	31:OR	
R	N	18:10010	32:RN	Tăng lên 6bits
N	O	14:001110	33:NO	
O	T	15:001111	34:OT	
T	T	20:010100	35:TT	
TO	B	27:011011	36:TOB	
BE	O	29:011101	37:BEO	
OR	T	31:011111	38:ORT	
TOB	E	36:100100	39:TOBE	
EO	R	30:011110	40:EOR	
RN	O	32:100000	41:RNO	
OT	#	34:100010		#stop
		0:000000		Stop code

Kết quả

- Dung lượng chưa mã hóa:
  - 25 ký tự * 5bits/ký tự = 125 bits.

  Mã hóa:
  - (6 mã * 5bits/mã) + (11 mã * 6bits/mã) = 96 bits

  Vậy sử dụng LZW đã tiết kiệm 29 bits trên tổng số 125 bits,giảm 22%.

	Intput	Output Sequence	Từ Điển đầy đủ	Bộ dự trữ	comments
6 mã 5 bits	20:10100	T		27:T?	
	15:01111	O	27:TO	28:O	
	2:00010	B	28:OB	29:B?	
	5:00101	E	29:BE	30:E?	
	15:01111	O	30:EO	31:O?	
	18:10010	R	31:OR	32:R?	
11 mã 6 bits	14:001110	N	32:RN	33:N?	
	15:001111	O	33:NO	34:O?	
	20:010100	T	34:OT	35:T?	
	27:011011	TO	35:TT	36:TO?	
	29:011101	BE	36:TOB	37:BE?	
	31:011111	OR	37:BEO	38:OR?	
	36:100100	TOB	38:ORT	39:TOB?	
	30:011110	EO	39:TOBE	40:EO?	
	32:100000	RN	40:EOR	41:RN?	
	34:100010	OT	41:RNO	42:OT?	
	0:000000	#			

6.Ứng dụng

- Nén LZW đã trở thành phương thức nén dữ liệu phổ biến trên máy tính.Một file text English có thể được nén thông qua LZW để giảm ½ dung lượng gốc.
- LZW đã được sử dụng trong phần mềm nén mã nguồn mỡ,nó đã trở thành 1 phần không thể thiếu trong HDH UNIX CIRCA 1986.
- LZW đã trở nên phổ biến khi nó được sữ dụng làm 1 phẩn của file GIF năm 1987.Nó cũng có thể được sử dụng trong TIFF và PDF file.

Bài Tập

- Hãy mã hóa chuỗi sau cho ra file output(trên giấy) và tính xem nén được bao nhiêu % dung lượng gốc:
- "TSYUYIROSUYITSONNTTEO#"

Advanced Example: Ziv-Lempel Code (continued)

Buffer includes two equal parts : L1, left haft, L2,right haft.. In thos ex. Buffer length=8

The first symbol in the input ois 'a', Load left haft of the buffer with "a"

Longest prefix matching

Initialize

Copy 4 symbols from the input into the right haft of the buffer

Remove 3 first symbols from the input, and 2 matched symbols buffer

Input	Buffer	Code Transmitted
aababacbaacbaadaa . . .	aaaa	a
aababacbaacbaadaa . . .	aaaaaaba	22b
abacbaacbaadaaa . . .	aaababac	23c
baacbaadaaa . . .	abacbaac	12a
cbaadaaa . . .	cbaacbaa	03a
daaa . . .	cbaadaaa	30d
aaa . . .	. . .	

Match
0123
aaaa,
aaba →(2, 2, b)
Shift out 3 symbols at positions 0,1, 2
Shift in **bac**

Match
0123
aa**aba**
abac→(2,3,c)

Figure 11-9 Encoding the string "aababacbaacbaadaaa . . ." with LZ77

A recent ly processed symbol is always to be left because it can be the beginning of new substring

Ziv-Lempel Code (continued)

Encoder		Table		
Input	Output	Index (Codeword)	Full String	Abbreviated String
		1	a	a
		2	b	b
		3	c	c
a		4	d	d
a	1	5	aa	1a
b	1	6	ab	1b
ab	2	7	ba	2a
a	6	8	aba	6a
c	1	9	ac	1c
ba	3	10	cb	3b
ac	7	11	baa	7a
baa	9	12	acb	9b
d	11	13	baad	11d
aa	4	14	da	4a
a	5	15	aaa	5a
	...			

Figure 11-10 LZW applied to the string "aababacbaacbaadaaa . . . ."

Case Study: Huffman Method with Run-Length Encoding

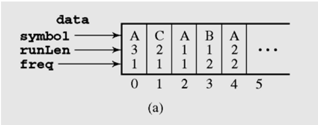


Figure 11-11 (a) Contents of the array `data` after the message AAABAACCAABA has been processed

Case Study: Huffman Method with Run-Length Encoding (continued)

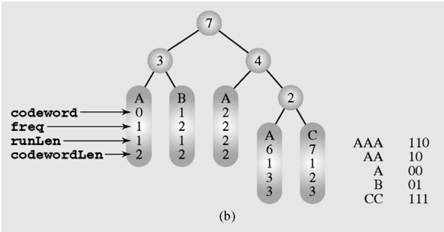


Figure 11-11 (b) Huffman tree generated from these data (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```java
//*************************  HuffmanCoding.java  *********************

import java.io.*;

class HuffmanNode {
    public byte symbol;
    public int codeword;
    public int freq;
    public int runLen;
```

Figure 11-12 Implementation of Huffman method with run-length encoding

Case Study: Huffman Method with Run-Length Encoding (continued)

```java
    public int codewordLen;
    public HuffmanNode left = null, right = null;
    public HuffmanNode() {
    }
    public HuffmanNode(byte s, int f, int r) {
        this(s,f,r,null,null);
    }
    public HuffmanNode(byte s, int f, int r, HuffmanNode lt, HuffmanNode rt) {
        symbol = s; freq = f; runLen = r; left = lt; right = rt;
    }
}

class ListNode {
    public HuffmanNode tree;
    public ListNode next = null, prev = null;
    public ListNode() {
    }
    public ListNode(ListNode p, ListNode n) {
        prev = p; next = n;
    }
}
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
class DataRec implements Comparable {
    public byte symbol;
    public int runLen;
    public int freq;
    public DataRec() {
    }
    public DataRec(byte s, int r) {
        symbol = s; runLen = r; freq = 1;
    }
    public boolean equals(Object el) {
        return symbol == ((DataRec)el).symbol && runLen == ((DataRec)el).runLen;
    }
    public int compareTo(Object el) {
        return freq - ((DataRec)el).freq;
    }
}

class HuffmanCoding {
    public HuffmanCoding() {
    }
    private final int ASCII = 256,
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
              intBytes = 4, // bytes per int;
              bits = 8;     // bits per byte;
    private HuffmanNode HuffmanTree;
    private HuffmanNode[] chars = new HuffmanNode[ASCII + 1];
    private java.util.ArrayList data = new java.util.ArrayList();
    private long charCnt;

    private void error(String s) {
        System.err.println(s); System.exit(-1);
    }
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
    private void garnerData(RandomAccessFile fIn) throws IOException {
        int ch, ch2, runLen, i;
        for (ch = fIn.read(); ch != -1; ch = ch2) {
            for (runLen = 1, ch2 = fIn.read(); ch2 != -1 && ch2 == ch; runLen++)
                ch2 = fIn.read();
            DataRec r = new DataRec((byte)ch,runLen);
            if ((i = data.indexOf(r)) == -1)
                 data.add(r);
            else ((DataRec)data.get(i)).freq++;
        }
        java.util.Collections.sort(data);
    }
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
    private void outputFrequencies(RandomAccessFile fIn, RandomAccessFile fOut)
                           throws IOException {
        fOut.writeInt(data.size());
        fOut.writeLong(fIn.getFilePointer());
        for (int j = 0; j < data.size(); j++) {
            DataRec r = (DataRec)data.get(j);
            fOut.write(r.symbol);
            fOut.writeInt(r.runLen);
            fOut.writeInt(r.freq);
        }
    }

    private void inputFrequencies(RandomAccessFile fIn) throws IOException {
        int dataIndex = fIn.readInt();
        charCnt = fIn.readLong();
        data.ensureCapacity(dataIndex);
        for (int j = 0; j < dataIndex; j++) {
            DataRec r = new DataRec();
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
            r.symbol = (byte) fIn.read();
            r.runLen = fIn.readInt();
            r.freq = fIn.readInt();
            data.add(r);
        }
    }

    private void createHuffmanTree() {
        ListNode p, newNode, head, tail;
        head = tail = new ListNode();           // initialize list pointers;
        DataRec r = (DataRec)data.get(0);
        head.tree = new HuffmanNode(r.symbol,r.freq,r.runLen);
        for (int i = 1; i < data.size(); i++) { // create the rest of the list;
            tail.next = new ListNode(tail,null);
            tail = tail.next;
            r = (DataRec)data.get(i);
            tail.tree = new HuffmanNode(r.symbol,r.freq,r.runLen);
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
        }
        while (head != tail) {                  // create one Huffman tree;
            int newFreq = head.tree.freq + head.next.tree.freq; // two lowest
                                                                // frequencies
            for (p = tail; p != null && p.tree.freq > newFreq; p = p.prev);
            newNode = new ListNode(p,p.next);
            p.next = newNode;
            if (p == tail)
                 tail = newNode;
            else newNode.next.prev = newNode;
            newNode.tree =
                 new HuffmanNode((byte)0,newFreq,0,head.tree,head.next.tree);
            head = head.next.next;
            head.prev = null;
        }
        HuffmanTree = head.tree;
    }
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
    private void createCodewords(HuffmanNode p, int codeword, int lvl) {
        if (p.left == null && p.right == null) {     // if p is a leaf,
             p.codeword    = codeword;               // store codeword
             p.codewordLen = lvl;                    // and its length,
        }
        else {                                       // otherwise add 0
             createCodewords(p.left,  codeword<<1,   lvl+1);// for left branch
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
             createCodewords(p.right,(codeword<<1)+1,lvl+1);// and 1 for right;
        }
    }

    private void transformTreeToArrayOfLists(HuffmanNode p) {
        if (p.left == null && p.right == null) {     // if p is a leaf,
             p.right = chars[p.symbol+128];          // include it in
             chars[p.symbol+128] = p;                // a list associated
        }                                            // with symbol found in p;
        else {                                       // add 128 to change the
             transformTreeToArrayOfLists(p.left);    // range of bytes from
             transformTreeToArrayOfLists(p.right);   // [-128, 127] to
        }                                            // [0, 255];
    }
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
private void encode(RandomAccessFile fIn, RandomAccessFile fOut) throws
IOException {
    int packCnt = 0, hold, maxPack = 4 * bits, pack = 0;
    int ch, ch2, bitsLeft, runLen;
    HuffmanNode p;
    for (ch = fIn.read(); ch != -1; ) {
        for (runLen = 1, ch2 = fIn.read();  ch2 != -1 && ch2 == ch; runLen++)
            ch2 = fIn.read();
        for (p = chars[(byte)ch+128]; p != null && runLen != p.runLen;
                                   p = p.right)
            ;
        if (p == null)
             error("A problem in transmitCode()");
        if (p.codewordLen < maxPack - packCnt) { // if enough room in
             pack = (pack << p.codewordLen) | p.codeword; // pack to store
             packCnt += p.codewordLen;            // new codeword, shift its
        }                                         // content to the left
                                                  // and attach new codeword;
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
        else {                                   // otherwise move
            bitsLeft = maxPack - packCnt;        // pack's content to
            pack <<= bitsLeft;                   // the left by the
            if (bitsLeft != p.codewordLen) {     // number of left
                hold = p.codeword;               // spaces and if new
                hold >>>= p.codewordLen - bitsLeft;// codeword is longer
                pack |= hold;                    // than room left, transfer
            }                                    // only as many bits as
                                                 // can fit in pack;
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
            else pack |= p.codeword;             // if new codeword
                                                 // exactly fits in
                                                 // pack, transfer it;
            fOut.writeInt(pack);                 // output pack as
                                                 // four bytes;
            if (bitsLeft != p.codewordLen) {     // transfer
                pack = p.codeword;               // unprocessed bits
                packCnt = maxPack - (p.codewordLen - bitsLeft);// of new
                packCnt = p.codewordLen - bitsLeft;// codeword to pack;
            }
            else packCnt = 0;
        }
        ch = ch2;
    }
    if (packCnt != 0) {
        pack <<= maxPack - packCnt; // transfer leftover codewords
        fOut.writeInt(pack);        // and some 0's;
    }
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
}
public void compressFile(String inFileName, RandomAccessFile fIn) throws
IOException {
    String outFileName = new String(inFileName+".z");
    RandomAccessFile fOut = new RandomAccessFile(outFileName,"rw");
    Date start = new Date();
    garnerData(fIn);
    outputFrequencies(fIn,fOut);
    createHuffmanTree();
    createCodewords(HuffmanTree,0,0);
    for (int i = 0; i <= ASCII; i++)
        chars[i] = null;
    transformTreeToArrayOfLists(HuffmanTree);
    fIn.seek(0);
    encode(fIn,fOut);
}
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
private void decode(RandomAccessFile fIn, RandomAccessFile fOut) throws
IOException {
    int chars, j, ch, bitCnt = 1, mask = 1;
    mask <<= bits - 1;  // change 00000001 to 100000000
    for (chars = 0, ch = fIn.read(); ch != -1 && chars < charCnt; ) {
        for (HuffmanNode p = HuffmanTree; ; ) {
            if (p.left == null && p.right == null) {
                for (j = 0; j < p.runLen; j++)
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
                    fOut.write(p.symbol);
                chars += p.runLen;
                break;
            }
            else if ((ch & mask) == 0)
                 p = p.left;
            else p = p.right;
            if (bitCnt++ == bits) {  // read next character from FIn
                 ch = fIn.read();    // if all bits in ch are checked;
                 bitCnt = 1;
            }                        // otherwise move all bits in ch
            else ch <<= 1;           // to the left by one position;
        }
    }
}
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
    public void decompressFile(String inFileName, RandomAccessFile fIn)
                    throws IOException {
        String outFileName = new String(inFileName+".dec");
        RandomAccessFile fOut = new RandomAccessFile(outFileName,"rw");
        Date start = new Date();
        inputFrequencies(fIn);
        createHuffmanTree();
        createCodewords(HuffmanTree,0,0);
        for (int i = 0; i <= ASCII; i++)
            chars[i] = null;
        decode(fIn,fOut);
    }
}
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
//************************  HuffmanEncoder.java  ********************

import java.io.*;

public class HuffmanEncoder {
    static public void main (String args[]) {
        String fileName = "";
        HuffmanCoding Htree = new HuffmanCoding();
        RandomAccessFile fIn;
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader buffer = new BufferedReader(isr);
        try {
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
            if (args.length == 0) {
                System.out.print("Enter a file name: ");
                fileName = buffer.readLine();
                fIn = new RandomAccessFile(fileName,"r");
            }
            else {
                fIn = new RandomAccessFile(args[0],"r");
                fileName = args[0];
            }
            Htree.compressFile(fileName,fIn);
            fIn.close();
        } catch(IOException io) {
            System.err.println("Cannot open " + fileName);
        }
    }
}
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
//************************  HuffmanDecoder.java  ***********************

import java.io.*;

public class HuffmanDecoder {
    static public void main (String args[]) {
        String fileName = "";
        HuffmanCoding Htree = new HuffmanCoding();
        RandomAccessFile fIn;
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader buffer = new BufferedReader(isr);
        try {
            if (args.length == 0) {
                System.out.print("Enter a file name: ");
                fileName = buffer.readLine();
                fIn = new RandomAccessFile(fileName,"r");
            }
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Case Study: Huffman Method with Run-Length Encoding (continued)

```
            else {
                fIn = new RandomAccessFile(args[0],"r");
                fileName = args[0];
            }
            Htree.decompressFile(fileName,fIn);
            fIn.close();
        } catch(IOException io) {
            System.err.println("Cannot open " + fileName);
        }
    }
}
```

Figure 11-12 Implementation of Huffman method with run-length encoding (continued)

Summary

- To compare the efficiency of different data compression methods when applied to the same data, the same measure is used; this measure is the compression rate
- The construction of an optimal code was developed by David Huffman, who utilized a tree structure in this construction: a binary tree for a binary code

Summary (continued)

- In adaptive Huffman coding, the Huffman tree includes a counter for each symbol, and the counter is updated every time a corresponding input symbol is being coded
- A run is defined as a sequence of identical characters
- Run-length encoding is useful when applied to files that are almost guaranteed to have many runs of at least four characters, such as relational databases

Summary (continued)

- Null suppression compresses only runs of blanks and eliminates the need to identify the character being compressed
- The Ziv-Lempel code is an example of a universal data compression code
- Pair: Encodinging method- Appropriate Decoding method.