



Data Structures & Algorithms

Các thuật toán nén dữ liệu (Data Compression Algorithms)



Nguyễn Tri Tuấn
Khoa CNTT – ĐH.KHTN.Tp.HCM
Email: nttuan@fit.hcmus.edu.vn



Data Compression

Giới thiệu

Giải thuật nén RLE

Giải thuật nén Huffman



Giới thiệu

✦ Các thuật ngữ thường dùng:

- ◆ Data Compression
- ◆ Lossless Compression
- ◆ Lossy Compression
- ◆ Encoding
- ◆ Decoding
- ◆ Run / Run Length
- ◆ RLE, Huffman, LZW



Giới thiệu (tt)

✦ Mục đích của nén dữ liệu:

◆ Giảm kích thước dữ liệu:

- Khi lưu trữ
- Khi truyền dữ liệu

◆ Tăng tính bảo mật



Giới thiệu (tt)

✦ Có 2 hình thức nén:

◆ Nén bảo toàn thông tin (Lossless Compression):

- Không mất mát thông tin nguyên thủy
- Hiệu suất nén không cao: 10% - 60%
- Các giải thuật tiêu biểu: RLE, Arithmetic, Huffman, LZ77, LZ78,...

◆ Nén không bảo toàn thông tin (Lossy Compression):

- Thông tin nguyên thủy bị mất mát
- Hiệu suất nén cao 40% - 90%
- Các giải thuật tiêu biểu: JPEG, MP3, MP4,...

Giới thiệu (tt)

✦ Hiệu suất nén (%):

- ◆ Tỷ lệ % kích thước dữ liệu giảm được sau khi áp dụng thuật toán nén

- ◆ $D (\%) = (N - M) / N * 100$

D: Hiệu suất nén

N: kích thước data trước khi nén

M: kích thước data sau khi nén

✦ Hiệu suất nén tùy thuộc

- ◆ Phương pháp nén
- ◆ Đặc trưng của dữ liệu



Giới thiệu (tt)

✦ Nén tập tin:

- ◆ Dùng khi cần Backup, Restore,... dữ liệu
- ◆ Dùng các thuật toán nén bảo toàn thông tin
- ◆ Không quan tâm đến định dạng (format) của tập tin
- ◆ Các phần mềm: PKzip, WinZip, WinRar,...



Data Compression



Giới thiệu



Giải thuật nén RLE



Giải thuật nén Huffman



Giải thuật nén RLE

✦ RLE = Run Length Encoding: mã hoá theo độ dài lặp lại của dữ liệu

✦ Ý tưởng

✦ Dạng 1: RLE với file *.PCX

✦ Dạng 2: RLE với file *.BMP

✦ Nhận xét

Giải thuật nén RLE (tt)

✦ Ý tưởng:

Hình thức biểu diễn thông tin dư thừa đơn giản: “đường chạy” (run) – là dãy các ký tự lặp lại liên tiếp

- ◆ “đường chạy” được biểu diễn ngắn gọn: <Số lần lặp> <Ký tự>
- ◆ Khi độ dài đường chạy lớn → Tiết kiệm đáng kể

Ví dụ:

Data = AAAABBBBBBBBCCCCCCCCCDEE (# 25 bytes)

Data_{nén} = 4A8B10C1D2E (# 10 bytes)

Giải thuật nén RLE (tt)

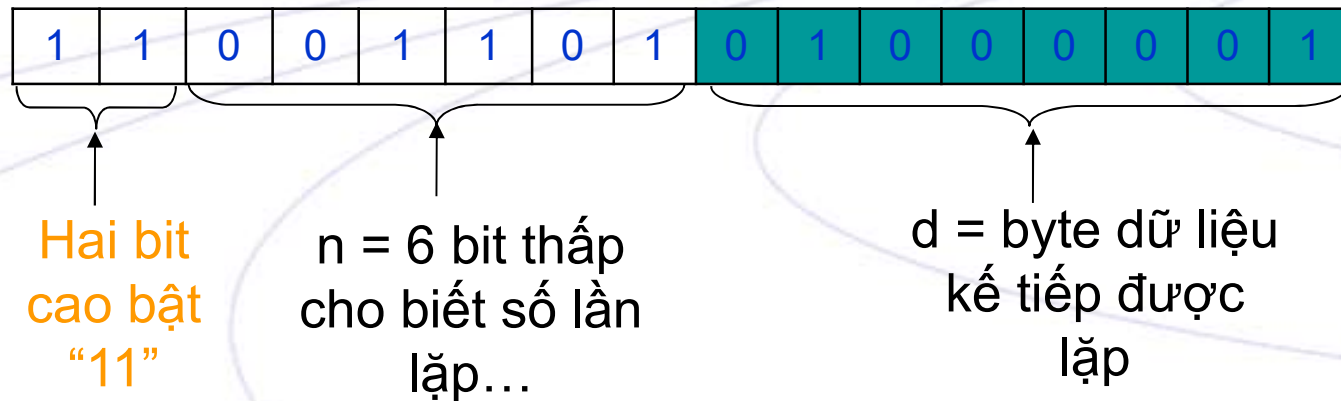
✦ Ý tưởng: (tt)

- ◆ Khi vận dụng thực tế, cần có biện pháp xử lý để tránh trường hợp “phản tác dụng” đối với các “run đặc biệt chỉ có 1 ký tự”

$X \text{ (# 1 bytes)} \rightarrow 1X \text{ (# 2 bytes)}$

Giải thuật nén RLE (tt)

✦ Dạng 1: RLE trong định dạng file *.PCX

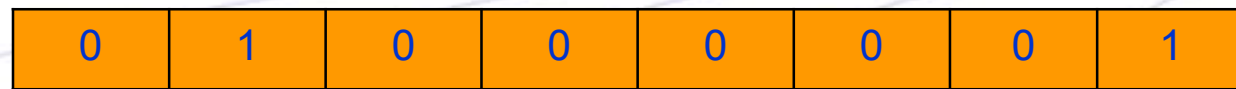


Trường hợp “run bình thường”:

AAAAAAAAAAAAA → 13 A (biểu diễn 2 bytes)
→ 0xCD 0x41

Giải thuật nén RLE (tt)

✦ RLE trong định dạng file *.PCX (tt)



Hai bit cao
không bật

Đây là byte dữ
liệu (số lần lặp=1)

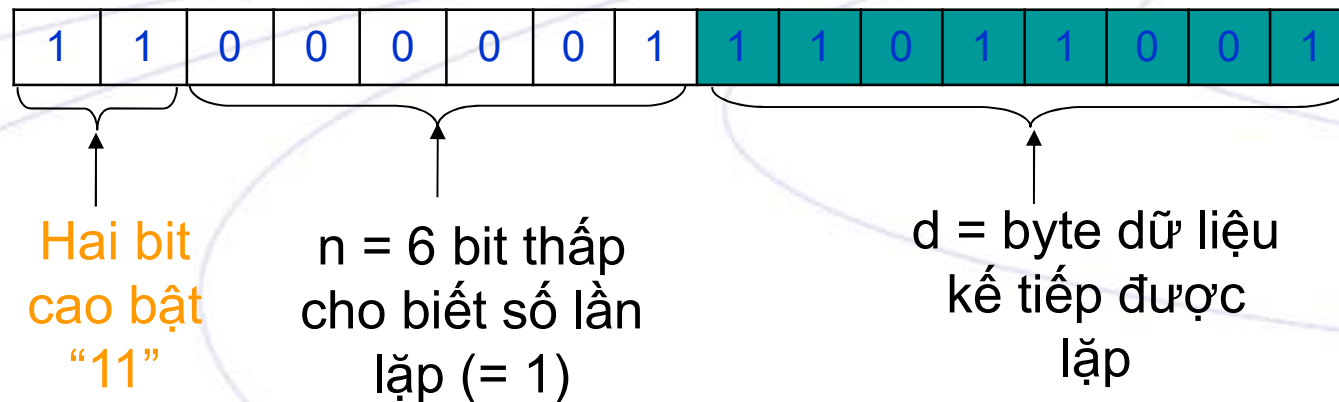
Trường hợp “run đặc biệt”:

A → A (biểu diễn 1 byte)

→ 0x41

Giải thuật nén RLE (tt)

✦ RLE trong định dạng file *.PCX (tt)



Trường hợp “run đặc biệt”:

$0xD9_{(217\ d)} \rightarrow 1\ 0xD9\ (\text{biểu diễn } 2\ \text{bytes})$
 $\rightarrow 0xC1\ 0xD9$

Giải thuật nén RLE (tt)

✦ RLE trong định dạng file *.PCX (tt)

◆ Ưu điểm:

- Cài đặt đơn giản
- Giảm 75% các trường hợp “phản tác dụng” của những run đặc biệt

◆ Khuyết điểm:

- Dùng 6 bit biểu diễn số lần lặp → chỉ thể hiện được chiều dài run max = 63 → Các đoạn lặp dài hơn sẽ phải chia nhỏ để mã hóa
- Không giải quyết được trường hợp “phản tác dụng” với run đặc biệt có mã ASCII $\geq 192_d$



Giải thuật nén RLE (tt)

✦ RLE trong định dạng file *.PCX (tt)

Vì sao dùng 2 bits làm cờ hiệu, mà không dùng 1 bit ?

Giải thuật nén RLE (tt)

```
#define MAX_RUNLENGTH 63

int PCXEncode_a_String(char *aString, int nLen, FILE *fEncode)
{
    unsigned char cThis, cLast;
    int i;
    int nTotal = 0;           // Tổng số byte sau khi mã hoá
    int nRunCount = 1;        // Chiều dài của 1 run

    cLast = *(aString);
    for (i=0; i<nLen; i++) {
        cThis = *(++aString);
        if (cThis == cLast) { // Tồn tại 1 run
            nRunCount++;
            if (nRunCount == MAX_RUNLENGTH) {
                nTotal +=
                    PCXEncode_a_Run(cLast, nRunCount, fEncode);
                nRunCount = 0;
            }
        }
    }
}
```

Continued...

Giải thuật nén RLE (tt)

```
else          // Hết 1 run, chuyển sang run kế tiếp
{
    if (nRunCount)
        nTotal +=
            PCXEncode_a_Run(cLast, nRunCount, fEncode) ;
    cLast = cThis;
    nRunCount = 1;
}
} // end for

if (nRunCount)          // Ghi run cuối cùng lên file
    nTotal += PCXEncode_a_Run(cLast, nRunCount, fEncode) ;

return (nTotal);

} // end function
```

Giải thuật nén RLE (tt)

```
int PCXEncode_a_Run(unsigned char c, int nRunCount,
                    FILE *fEncode)
{
    if (nRunCount) {
        if ((nRunCount == 1) && (c < 192))
        {
            putc(c, fEncode);
            return 1;
        }
        else
        {
            putc(0xC0 | nRunCount, fEncode);
            putc(c, fEncode);
            return 2;
        }
    }
}
```

Giải thuật nén RLE (tt)

```
int PCXDecode_a_File(FILE *fEncode, FILE *fDecode) {
    unsigned char c, n;

    while (!feof(fEncode))
    {
        c = (unsigned char) getc(fEncode);
        if (c == EOF) break;
        if ((c & 0xC0) == 0xC0)    // 2 bit cao bật
        {
            n = c & 0x3f;    // Lấy 6 bit thấp → số lần lặp...
            c = (unsigned char) getc(fEncode);
        }
        else n = 1;

        // Ghi dữ liệu đã giải mã lên file fDecode
        for (int i=0; i<n; i++)    putc(c, fDecode);
    }
    fclose(fDecode);
}
```



Giải thuật nén RLE (tt)

✦ Dạng 2: RLE trong định dạng file *.BMP

◆ File *.BMP

- Định dạng file chuẩn của Windows dùng để lưu ảnh bitmap
- Có khả năng lưu trữ ảnh B&W, 16 màu, 256 màu, 24bits màu
- Có sử dụng thuật toán nén RLE khi lưu trữ dữ liệu điểm ảnh

Giải thuật nén RLE (tt)

✦ RLE trong định dạng file *.BMP (tt)

AAA.....A

lặp 255 lần



0xFF' A' 0xFF' A'
0xFF' A' 0xFF' A'
0xC3' A'

Dữ liệu lưu lặp lại vì
số lần lặp chỉ sử
dụng có 6 bits

Giải thuật nén RLE (tt)

✦ RLE trong định dạng file *.BMP (tt)

Ý tưởng:

◆ Dữ liệu có 2 dạng

- Dạng 1: Run với độ dài > 1. VD. **AAAAAAAAAAAA**
- Dạng 2: Dãy các ký tự đơn lẻ. VD. **BCDEFG**

◆ Biểu diễn: phân biệt 2 dạng bằng cách dùng “mã nhận dạng” (**ESCAPE 0x00**)

- Dạng 1: <Số lần lặp> <Ký tự lặp>
VD. **0x0C 'A'**
- Dạng 2: <ESCAPE> <n> <Dãy ký tự>
VD. **0x00 0x06 'B' 'C' 'D' 'E' 'F' 'G'**

Giải thuật nén RLE (tt)

✦ RLE trong định dạng file *.BMP (tt)

AAA.....ABCDEFGG

lặp 255 lần



0xFF 'A' 0x00 0x06 "BCDEFG"



Giải thuật nén RLE (tt)

So sánh giữa PCX RLE và BMP RLE ?

Giải thuật nén RLE (tt)

```
int BMPDecode_a_File(FILE *fEncode, FILE *fDecode) {  
  
    unsigned char cMode, cData;  
    int i, n;  
  
    while (!feof(fEncode))  
    {  
        cMode = (unsigned char) getc(fEncode);  
        if (cMode==EOF) break;  
        if (cMode==0) {                // Dạng 2  
            n = (unsigned char) getc(fEncode);  
            for (i=0; i<n; i++) {  
                cData = (unsigned char) getc(fEncode);  
                putc(cData, fDecode);  
            }  
        }  
    }  
}
```

Continued...

Giải thuật nén RLE (tt)

```
else // Dạng 1
{
    n = cMode;           // Số lần lặp
    cData = (unsigned char) getc(fEncode);

    for (i=0; i<n; i++)
        putc(cData, fDecode);
}
// end while
}
// end
```



Giải thuật nén RLE (tt)

✦ Nhận xét / Ứng dụng:

- ◆ Dùng để nén các dữ liệu có nhiều đoạn lặp lại (run)
- ◆ Thích hợp cho dữ liệu ảnh → ứng dụng hẹp
- ◆ Chưa phải là một thuật toán nén có hiệu suất cao
- ◆ Đơn giản, dễ cài đặt



Data Compression

Giới thiệu

Giải thuật nén RLE

Giải thuật nén Huffman



Giải thuật nén Huffman

- ✦ Giới thiệu
- ✦ Huffman tĩnh (Static Huffman)
- ✦ Huffman động (Adaptive Huffman)

Giải thuật nén Huffman – Giới thiệu

✦ Hình thành

◆ Vấn đề:

- Một giải thuật nén bảo toàn thông tin;
- Không phụ thuộc vào tính chất của dữ liệu;
- Ứng dụng rộng rãi trên bất kỳ dữ liệu nào, với hiệu suất tốt

◆ Tư tưởng chính:

- Phương pháp cũ: dùng 1 dãy cố định (8 bits) để biểu diễn 1 ký tự
- Huffman:
 - Sử dụng vài bits để biểu diễn 1 ký tự (gọi là “mã bit” – bits code)
 - Độ dài “mã bit” cho các ký tự không giống nhau:
 - Ký tự xuất hiện nhiều lần → biểu diễn bằng mã ngắn;
 - Ký tự xuất hiện ít → biểu diễn bằng mã dài
 - Mã hóa bằng mã có độ dài thay đổi (Variable Length Encoding)

◆ David Huffman – 1952: tìm ra phương pháp xác định mã tối ưu trên dữ liệu tĩnh

Giải thuật nén Huffman – Giới thiệu (tt)

✦ Giả sử có dữ liệu như sau:

$f = \text{"ADDAABBCCBAAABBCCCBBBCDAADDEEAA"}$

✦ Biểu diễn bình thường (8 bits/ký tự):

$$\begin{aligned}\text{Sizeof}(f) &= 10 \cdot 8 + 8 \cdot 8 + 6 \cdot 8 + 5 \cdot 8 + 2 \cdot 8 \\ &= 248 \text{ bits}\end{aligned}$$

Ký tự	Số lần xuất hiện trong file f
A	10
B	8
C	6
D	5
E	2

Giải thuật nén Huffman – Giới thiệu (tt)

- ✦ Biểu diễn bằng mã bit có độ dài thay đổi (theo bảng):

$$\begin{aligned}\text{Sizeof}(f) &= 10*2 + 8*2 + 6*2 + 5*3 + 2*3 \\ &= 69 \text{ bits}\end{aligned}$$

Ký tự	Mã bit
A	11
B	10
C	00
D	011
E	010



Static Huffman

- ✦ Thuật toán nén
- ✦ Tạo cây Huffman
- ✦ Phát sinh bảng mã bit
- ✦ Lưu trữ thông tin dùng để giải nén
- ✦ Thuật toán giải nén



Static Huffman (tt)

★ Thuật toán nén:

- ◆ [b1] Duyệt file → Lập bảng thống kê số lần xuất hiện của mỗi loại ký tự
- ◆ [b2] Phát sinh cây Huffman dựa vào bảng thống kê
- ◆ [b3] Từ cây Huffman → phát sinh bảng mã bit cho các ký tự
- ◆ [b4] Duyệt file → Thay thế các ký tự bằng mã bit tương ứng
- ◆ [b5] Lưu lại thông tin của cây Huffman dùng để giải nén

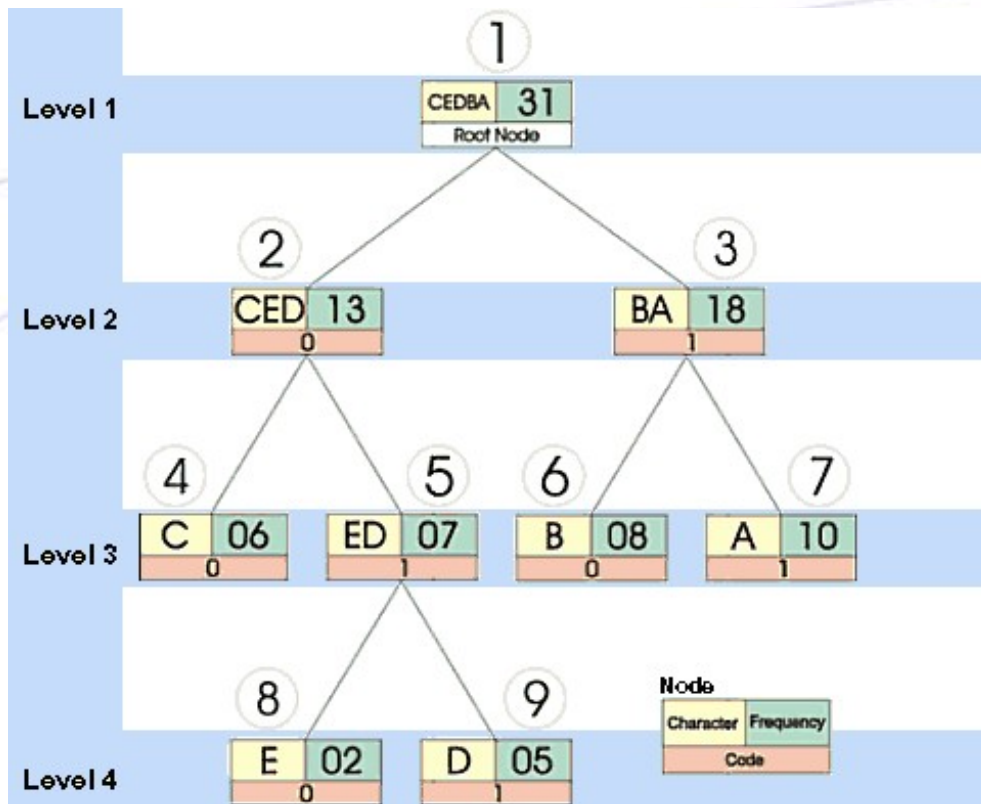
Static Huffman (tt)

$f = \text{"ADDAABBCCBAAABBBCCBBBCDAADDEEEAA"}$

[b1]

Ký tự	Số lần xuất hiện
A	10
B	8
C	6
D	5
E	2

[b2]



[b3]

Ký tự	Mã bit
A	11
B	10
C	00
D	011
E	010

[b4]

$f_{\text{nén}} = 110110111111010000101111111010000000$
 $101010000111111011011010010111$

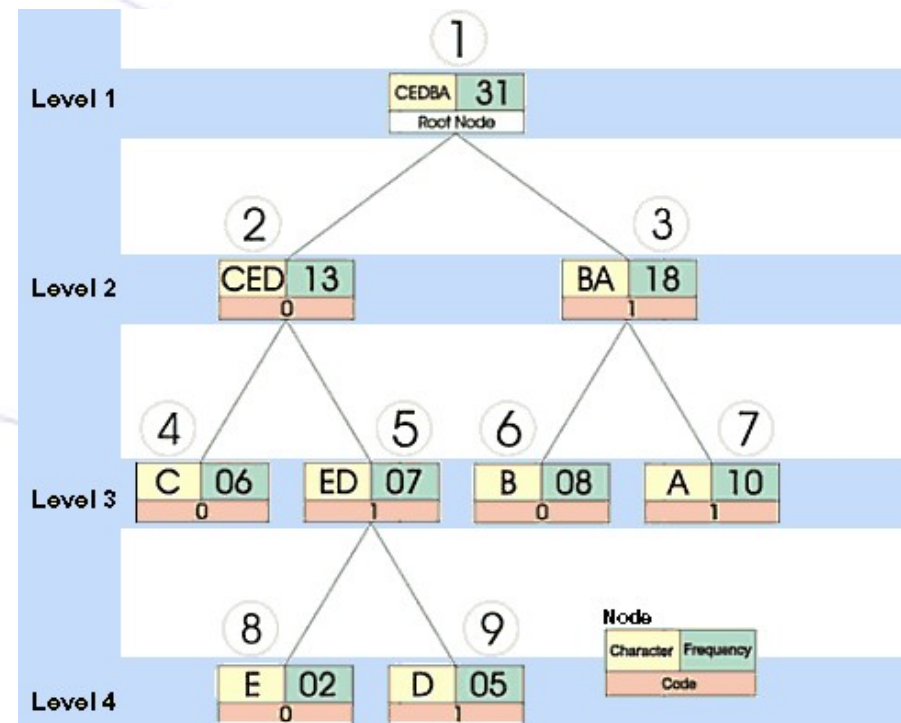
Static Huffman (tt)

✦ Tạo cây Huffman:

◆ Mô tả cây Huffman: mã Huffman được biểu diễn bằng 1 cây nhị phân

- Mỗi nút lá chứa 1 ký tự
- Nút cha sẽ chứa các ký tự của những nút con
- Mỗi nút được gán một trọng số:

- Nút lá có trọng số bằng số lần xuất hiện của ký tự trong file
- Nút cha có trọng số bằng tổng trọng số của các nút con



Static Huffman (tt)

★ Tạo cây Huffman: (tt)

◆ Tính chất cây Huffman:

- Nhánh trái tương ứng với mã hoá bit '0'; nhánh phải tương ứng với mã hoá bit '1'
- Các nút có tần số thấp nằm ở xa gốc → mã bit dài
- Các nút có tần số cao nằm ở gần gốc → mã bit ngắn
- Số nút của cây: $(2n-1)$

Static Huffman (tt)

```
// Cấu trúc dữ liệu lưu trữ cây Huffman
#define MAX_NODES 511 // 2*256 - 1

typedef struct {
    char c; // ký tự
    bool used; // đã sử dụng/chưa
    long nFreq; // trọng số
    int nLeft; // cây con trái
    int nRight; // cây con phải
} HUFFNode;

HUFFNode HuffTree[MAX_NODES];
```

Static Huffman (tt)

✦ Tạo cây Huffman: (tt)

◆ Thuật toán phát sinh cây:

[b1] Chọn trong bảng thống kê 2 phần tử x, y có trọng số thấp nhất
→ tạo thành nút cha z :

```
z.c = min(x.c + y.c) ;  
z.nFreq = x.nFreq + y.nFreq;  
z.nLeft = x (*)  
z.nRight = y (*)
```

[b2] Loại bỏ nút x và y khỏi bảng;

[b3] Thêm nút z vào bảng;

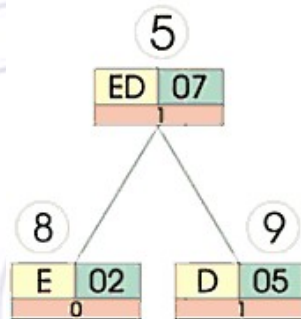
[b4] Lặp lại bước [b1] - [b3] cho đến khi chỉ còn lại 1 nút duy nhất trong bảng

(*) Qui ước:

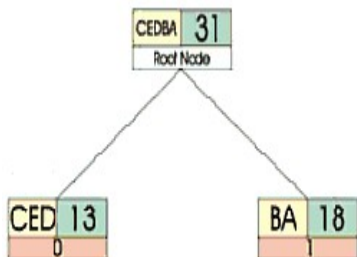
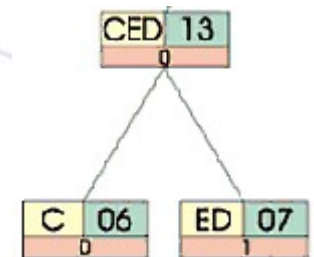
- nút có trọng số nhỏ nằm bên nhánh trái; nút có trọng số lớn nằm bên nhánh phải;
- nếu trọng số bằng nhau, nút có ký tự nhỏ nằm bên nhánh trái; nút có ký tự lớn nằm bên nhánh phải
- nếu có các node có trọng số bằng nhau → ưu tiên xử lý các node có ký tự ASCII nhỏ trước

Static Huffman (tt)

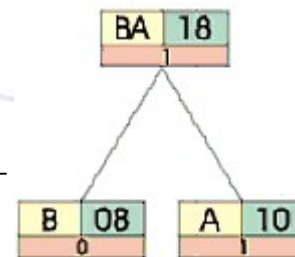
Ký tự	SLXH
A	10
B	8
C	6
D	5
E	2



Ký tự	SLXH
A	10
B	8
ED	7
C	6



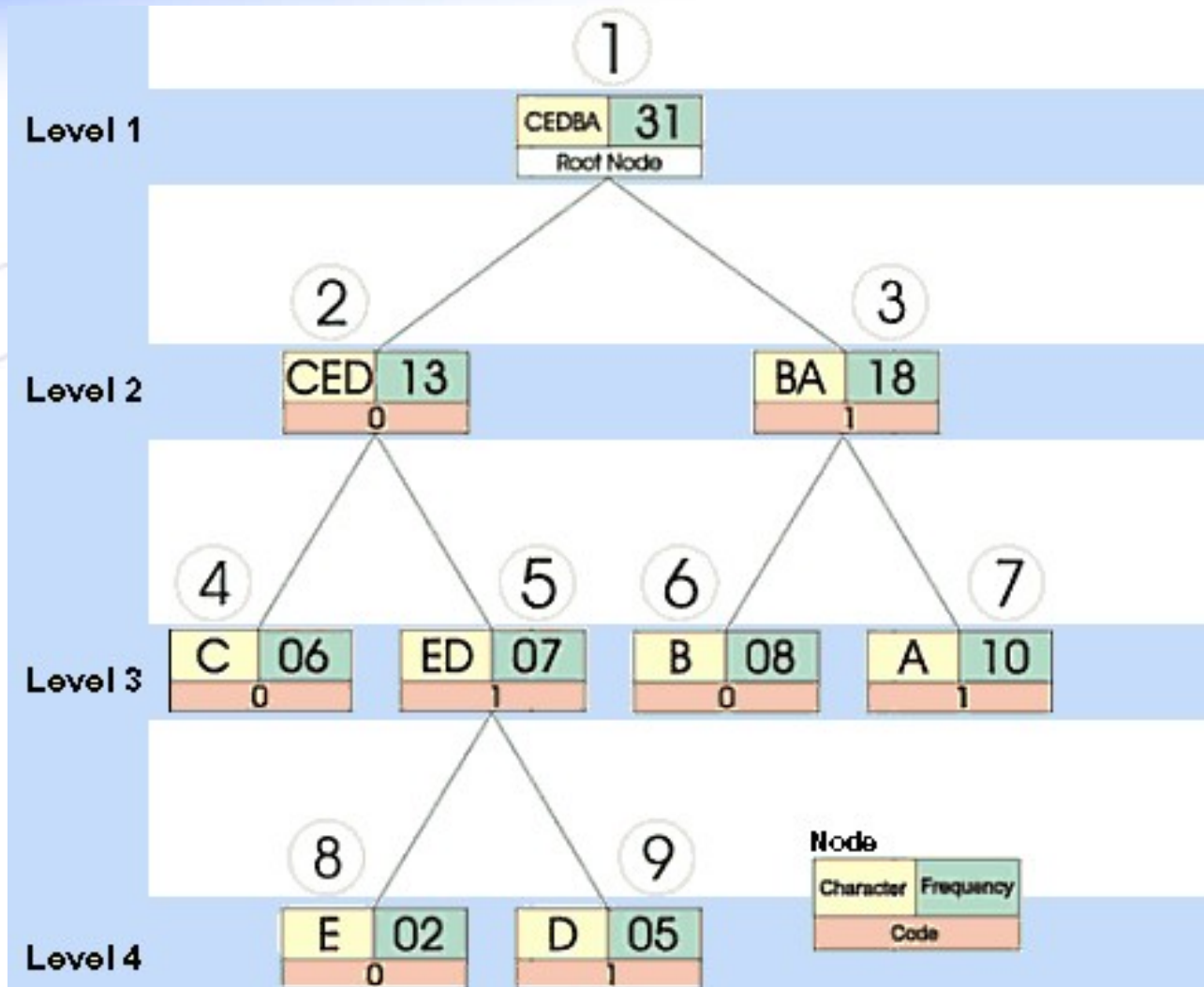
Ký tự	SLXH
BA	18
CED	13



Ký tự	SLXH
CED	13
A	10
B	8

Minh họa quá trình tạo cây

Static Huffman (tt)



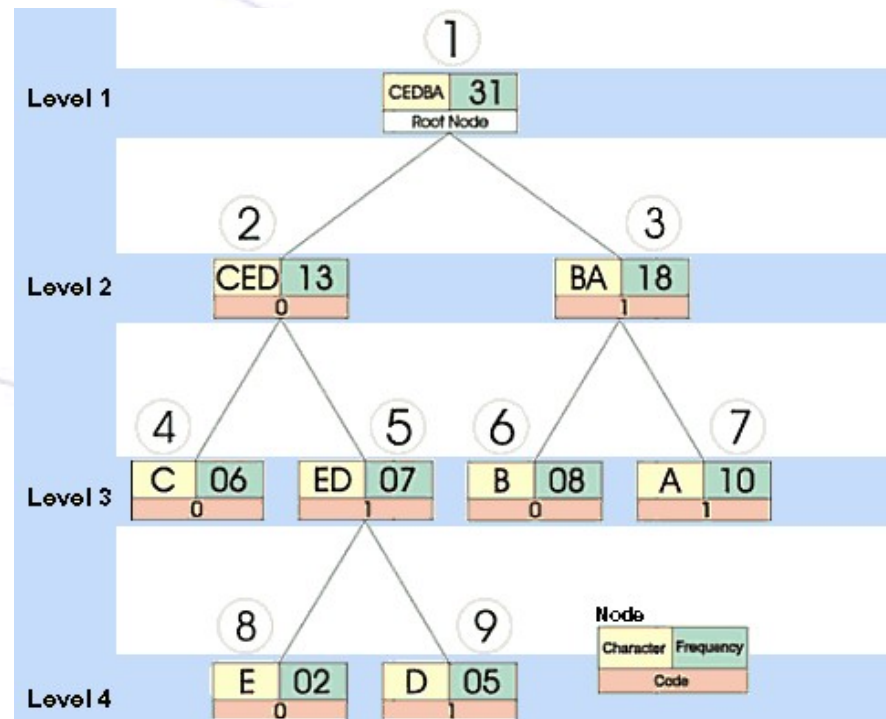
Cây Huffman sau khi tạo

Static Huffman (tt)

✦ Phát sinh mã bit cho các ký tự:

- ◆ Mã của mỗi ký tự được tạo bằng cách duyệt từ nút gốc đến nút lá chứa ký tự đó;
- ◆ Khi duyệt sang trái, tạo ra 1 bit 0;
- ◆ Khi duyệt sang phải, tạo ra 1 bit 1;

Ký tự	Mã
A	11
B	10
C	00
D	011
E	010



Static Huffman (tt)

✦ Lưu trữ thông tin dùng để giải nén:

P.Pháp 1: lưu bảng mã bit

Ký tự	Mã
A	11
B	10
C	00
D	011
E	010

P.Pháp 2: lưu số lần xuất hiện

Ký tự	Số lần xuất hiện
A	10
B	8
C	6
D	5
E	2

Static Huffman (tt)

✦ Thuật toán giải nén:

[b1] Xây dựng lại cây Huffman (từ thông tin được lưu)

[b2] Khởi tạo nút hiện hành **pCurr** = **pRoot**

[b3] Đọc 1 bit **b** từ file nén \mathbf{f}_n

[b4] Nếu (**b**==0) thì **pCurr** = **pCurr.nLeft**

ngược lại **pCurr** = **pCurr.nRight**

[b5] Nếu **pCurr** là nút lá thì:

- Xuất ký tự tại **pCurr** ra file

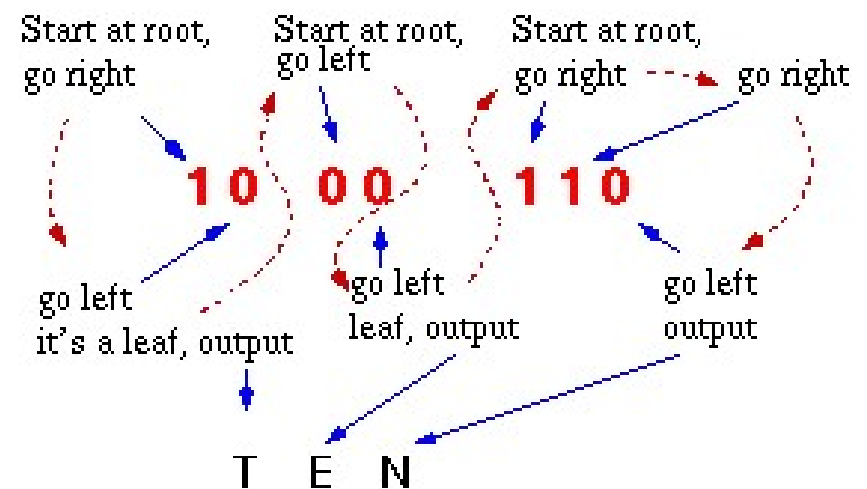
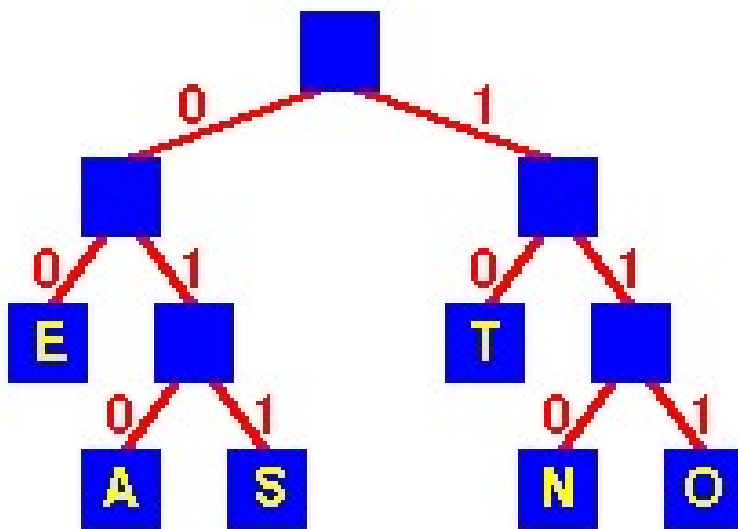
- Quay lại bước [b2]

ngược lại

- Quay lại bước [b3]

[b6] Thuật toán sẽ dừng khi hết file \mathbf{f}_n

Static Huffman (tt)



Cây Huffman và qui trình giải nén cho chuỗi được mã hoá “1000110”



Adaptive Huffman

- ✦ Giới thiệu
- ✦ Thuật toán tổng quát
- ✦ Cây Huffman động
- ✦ Thuật toán nén (Encoding)
- ✦ Thuật toán giải nén (Decoding)



Adaptive Huffman (tt)

✦ Giới thiệu:

◆ Hạn chế của Huffman tĩnh:

- Cần 2 lần duyệt file (quá trình nén) → chi phí cao
- Cần phải lưu trữ thông tin để giải nén → tăng kích thước dữ liệu nén
- Dữ liệu cần nén phải có sẵn → không nén được trên dữ liệu phát sinh theo thời gian thực



Adaptive Huffman (tt)

✦ Giới thiệu: (tt)

◆ Lịch sử hình thành:

- Được đề xuất bởi Faller (1973) và Gallager (1978)
- Knuth (1985) đưa ra một số cải tiến và hoàn chỉnh thuật toán
→ thuật toán còn có tên “thuật toán FGK”
- Vitter (1987): trình bày các cải tiến liên quan đến việc tối ưu cây Huffman



Adaptive Huffman (tt)

✦ Giới thiệu: (tt)

◆ Ưu điểm:

- Không cần tính trước số lần xuất hiện của các ký tự
- Quá trình nén: chỉ cần 1 lần duyệt file
- Không cần lưu thông tin phục vụ cho việc giải nén
- Nén “on-line”: trên dữ liệu phát sinh theo thời gian thực

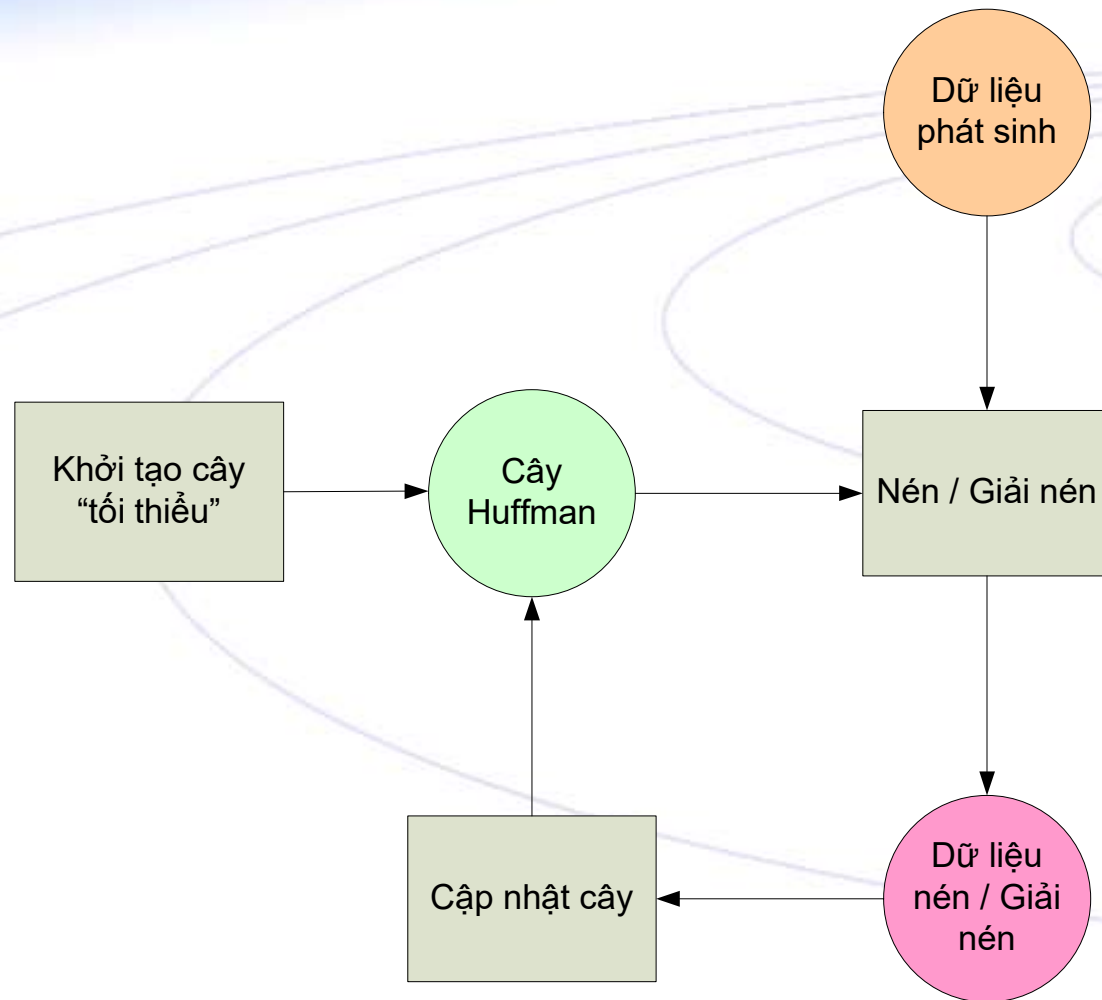


Adaptive Huffman (tt)

✦ Thuật toán tổng quát:

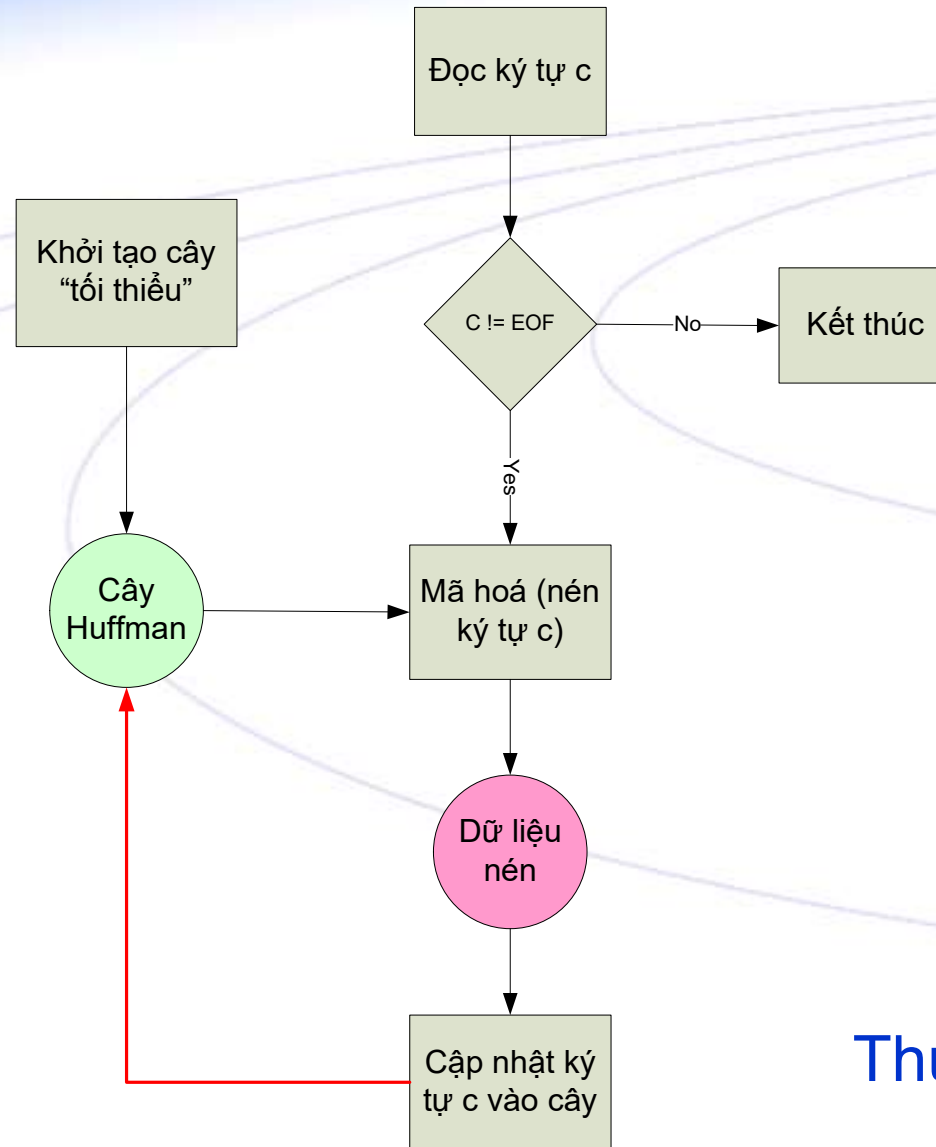
- ◆ Huffman tĩnh: cây Huffman được tạo thành từ bảng thống kê số lần xuất hiện của các ký tự
- ◆ Huffman động:
 - Nén “on-line” → không có trước bảng thống kê
 - Tạo cây như thế nào ?
 - Phương pháp: khởi tạo cây “tối thiểu” ban đầu; cây sẽ được “cập nhật dần dần” (~ thích nghi – Adaptive) dựa trên dữ liệu phát sinh trong quá trình nén/giải nén

Adaptive Huffman (tt)



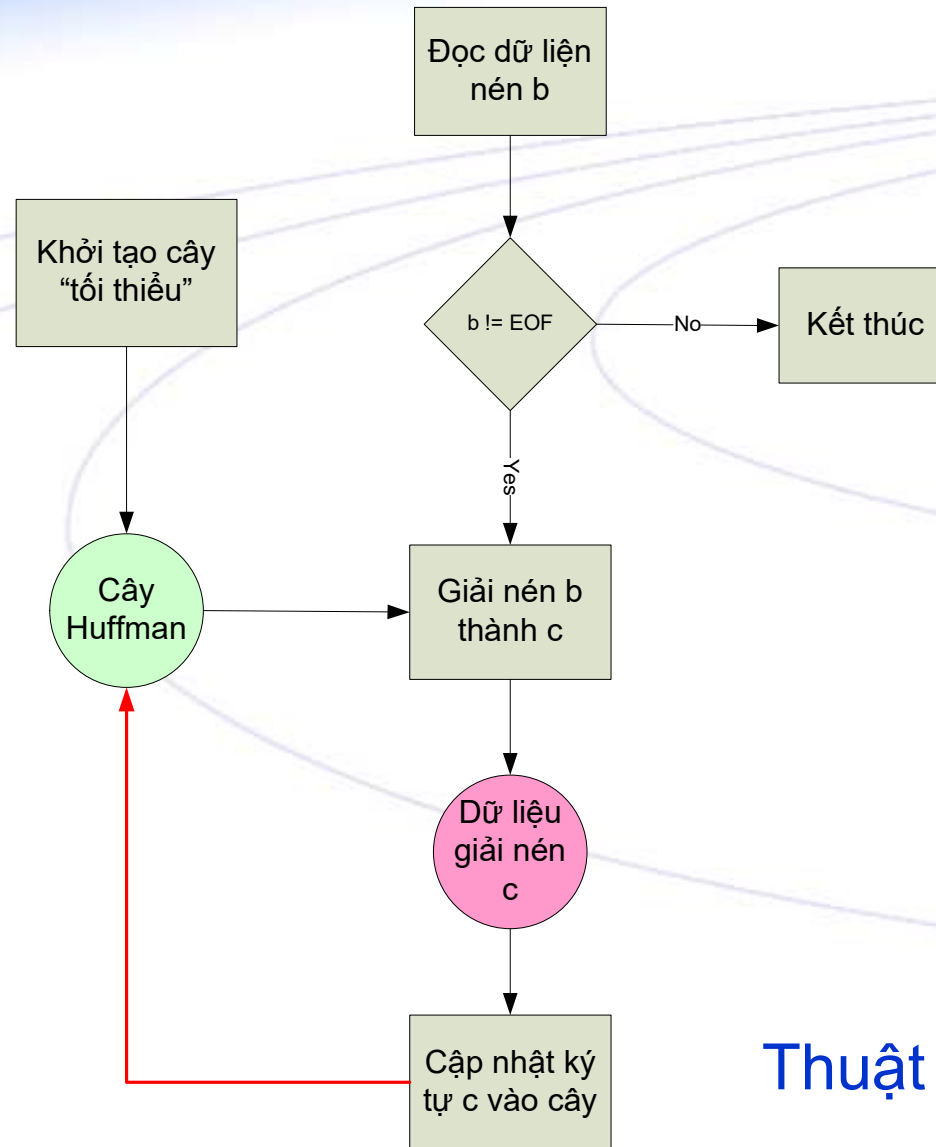
Sự phối hợp giữa việc dùng cây (cho thuật toán nén/giải nén) và cập nhật cây

Adaptive Huffman (tt)



Thuật toán nén

Adaptive Huffman (tt)



Thuật toán giải nén

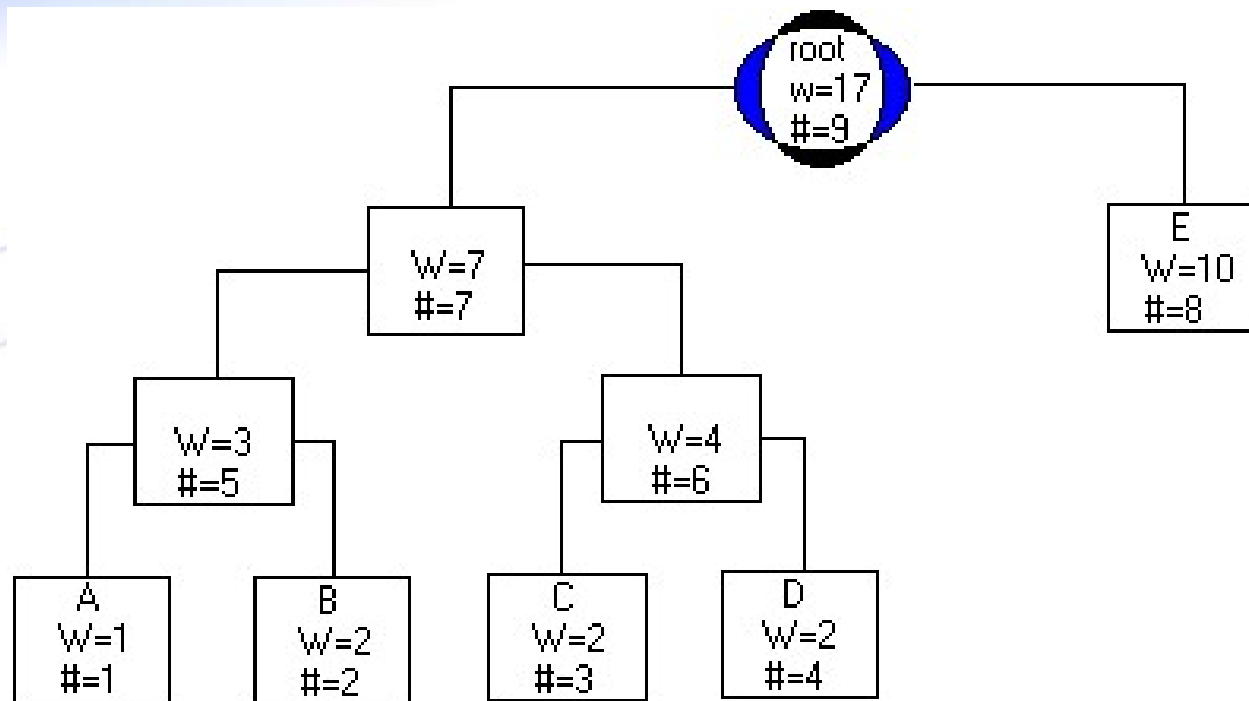
Adaptive Huffman (tt)

★ Cây Huffman (động):

Một cây nhị phân có n nút lá được gọi là cây Huffman nếu thỏa:

- ◆ Các nút lá có trọng số $w_i \geq 0, i \in [1..n]$
- ◆ Các nút nhánh có trọng số bằng tổng trọng số các nút con của nó
- ◆ Tính chất Anh/Em (Sibling Property):
 - Mỗi nút, ngoại trừ nút gốc, đều tồn tại 1 nút anh/em (có cùng nút cha)
 - Khi sắp xếp các nút trong cây theo thứ tự tăng dần của trọng số thì mỗi nút luôn ở kề với nút anh/em của nó

Adaptive Huffman (tt)



A binary tree is a Huffman tree if and only if it obeys the sibling property.
symbol weights: A=1, B=2, C=2, D=2, E=10

Thứ tự	#1	#2	#3	#4	#5	#6	#7	#8	#9
W_i	1	2	2	2	3	4	7	10	17
Giá trị	A	B	C	D				E	Root

Adaptive Huffman (tt)

✦ Cách thức tạo cây:

- ◆ Khởi tạo cây “tối thiểu”, chỉ có nút Escape (0-node)

Escape

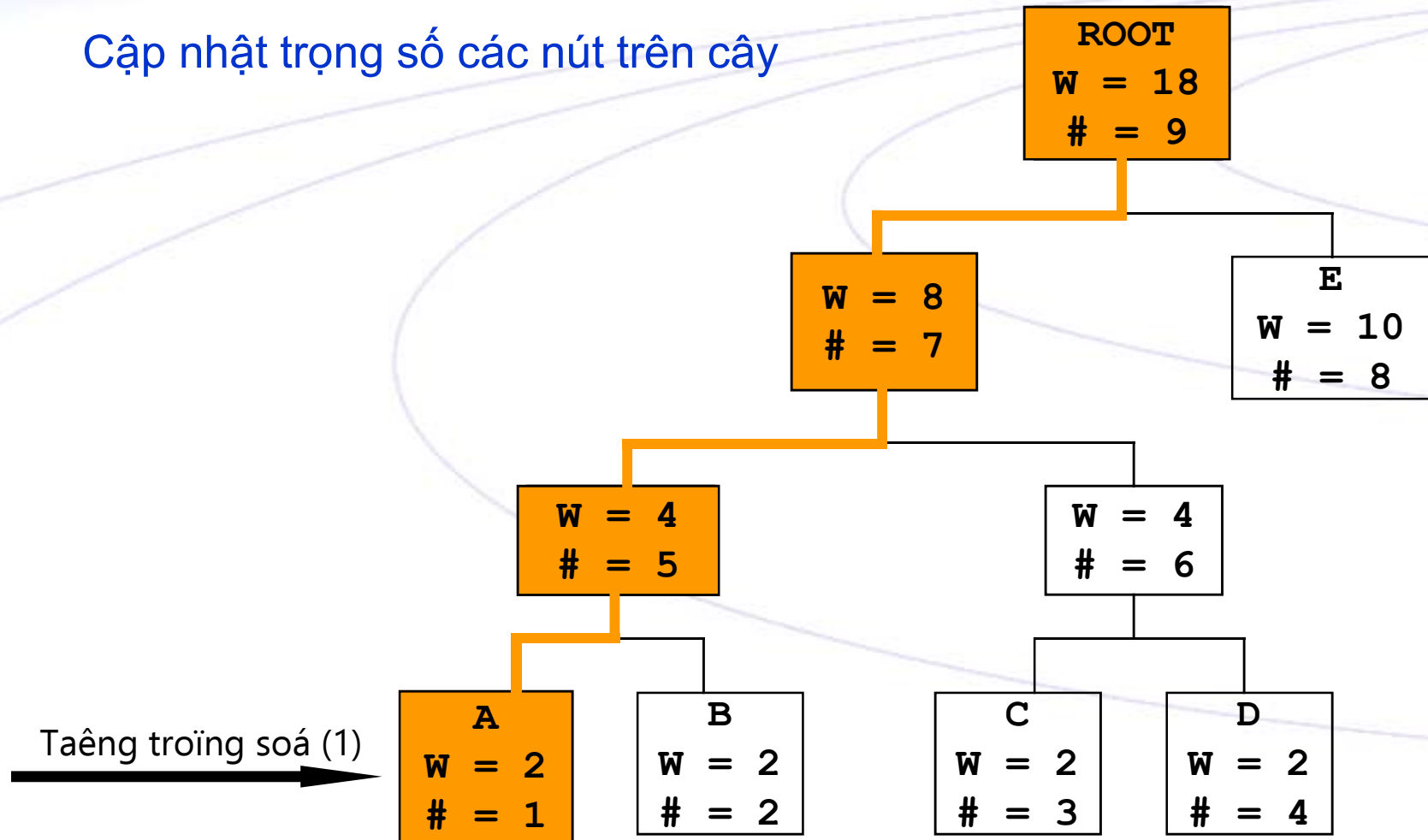
Cây “tối thiểu” chỉ có 1 nút Escape

◆ Cập nhật 1 ký tự c vào cây:

- Nếu c chưa có trong cây → thêm mới nút lá c
- Nếu c đã có trong cây → tăng trọng số nút c lên 1 (+1)
- Cập nhật trọng số của các nút liên quan trong cây

Adaptive Huffman (tt)

Cập nhật trọng số các nút trên cây





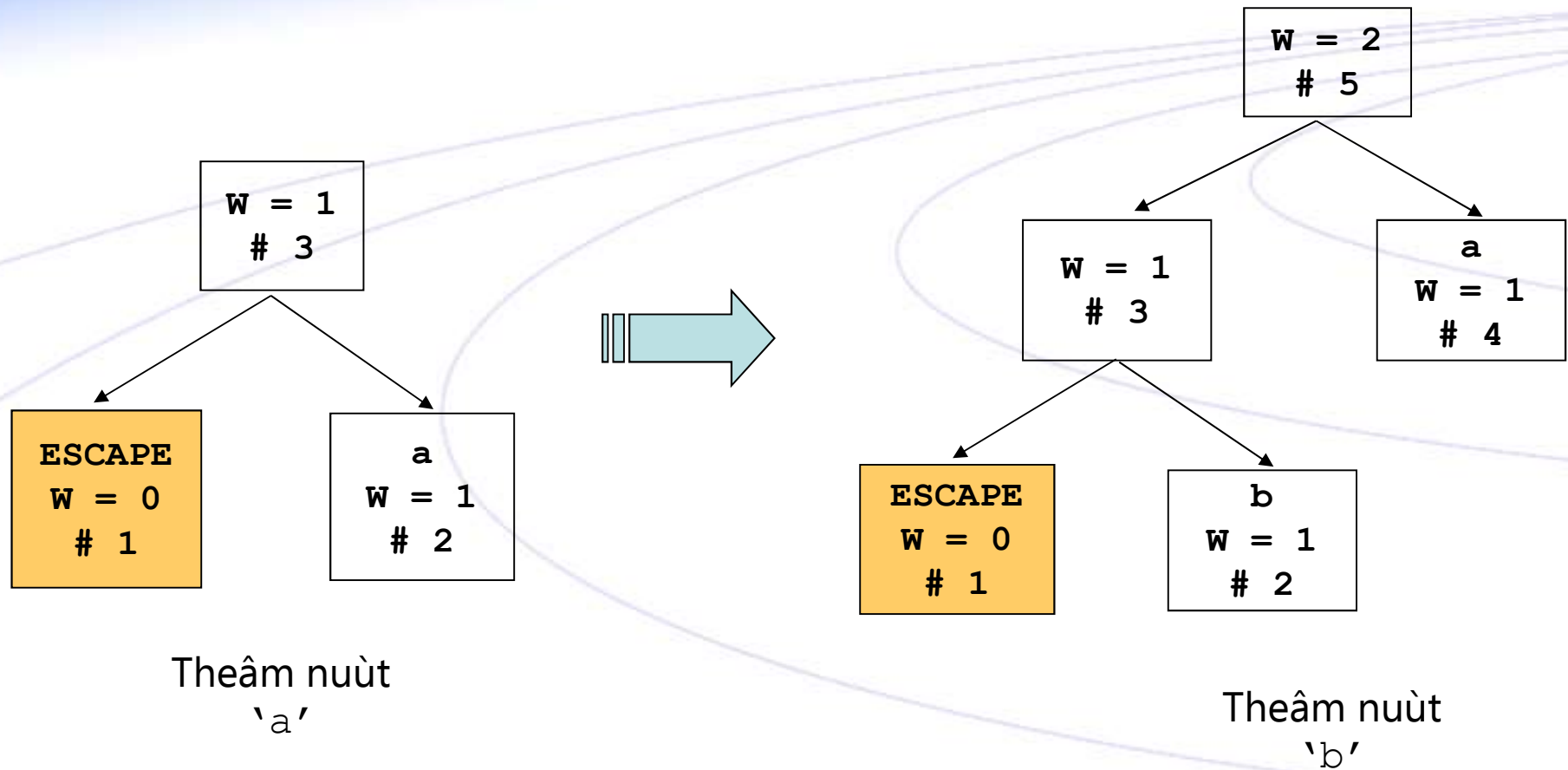
Adaptive Huffman (tt)

✦ Cách thức tạo cây: (tt)

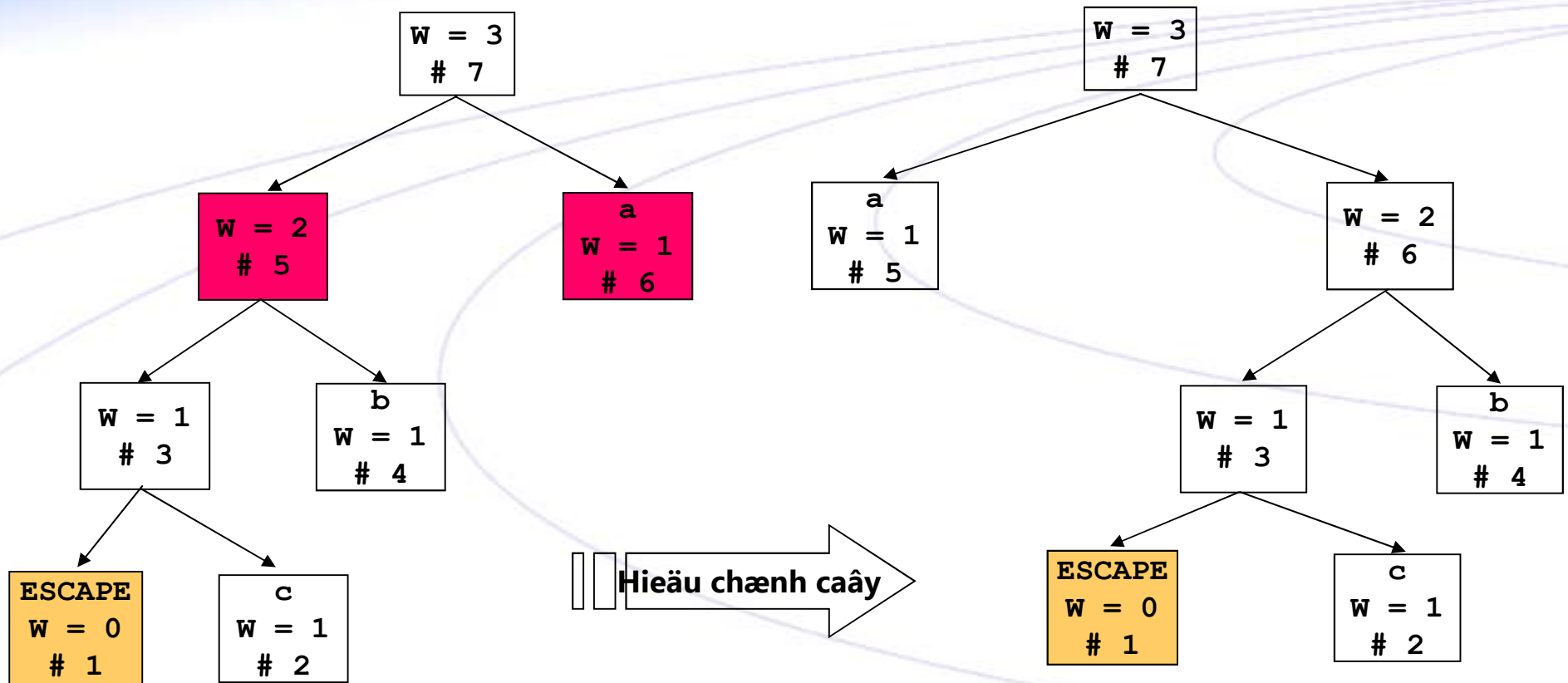
Thuật toán “Cập nhật trọng số”:

- ◆ Tăng trọng số của nút lá lên 1
- ◆ Đi từ nút lá \rightarrow nút gốc: tăng trọng số của các nút lên 1
- ◆ Kiểm tra tính chất anh/em và hiệu chỉnh lại cây (nếu vi phạm)

Adaptive Huffman (tt)



Adaptive Huffman (tt)



Hiệu chỉnh cây

Theâm nuòt
'c'



Adaptive Huffman (tt)

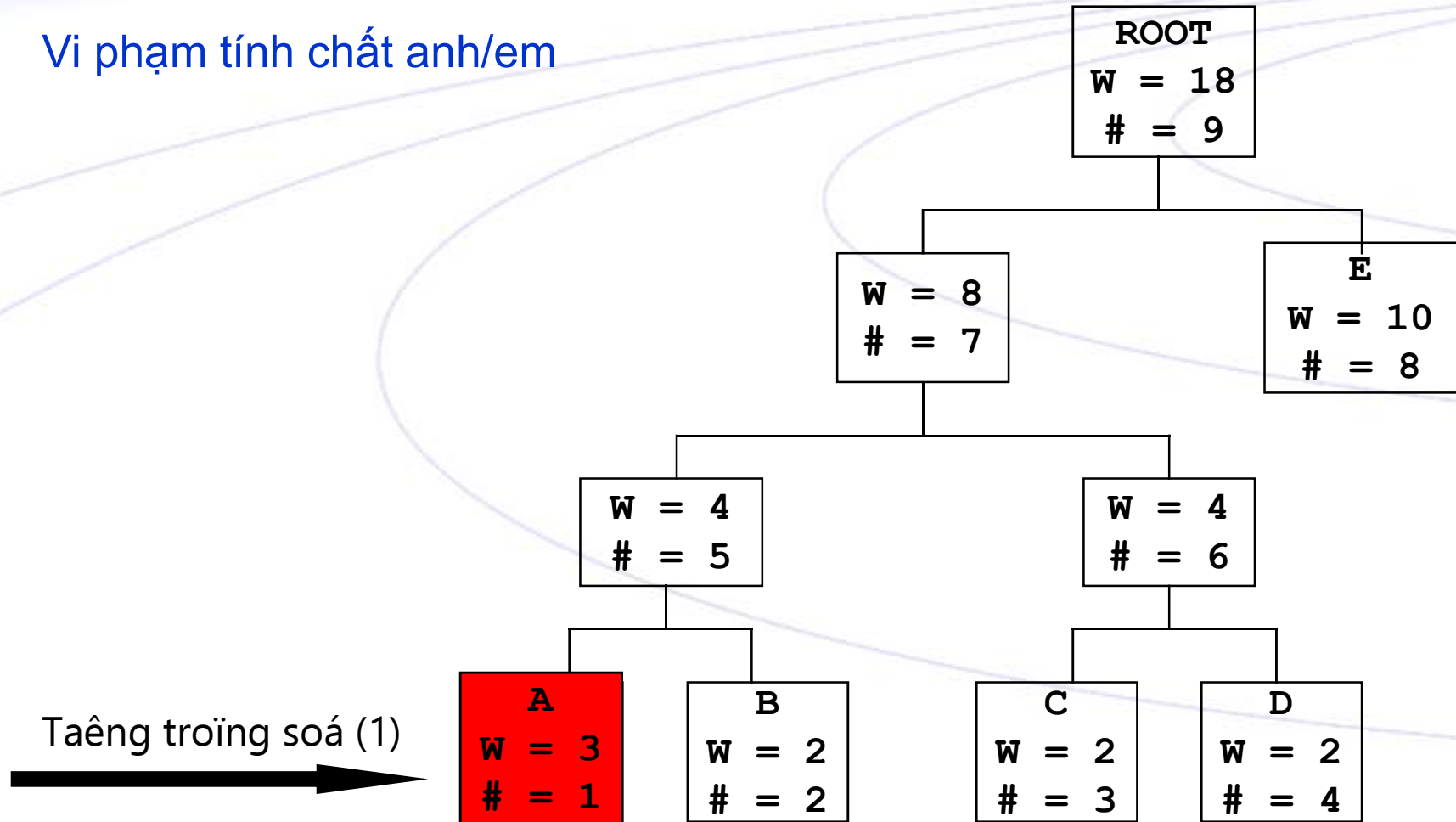
✦ Cách thức tạo cây: (tt)

Khi thêm 1 nút mới hoặc tăng trọng số:

- ◆ Vi phạm tính chất anh/em
- ◆ Tràn số

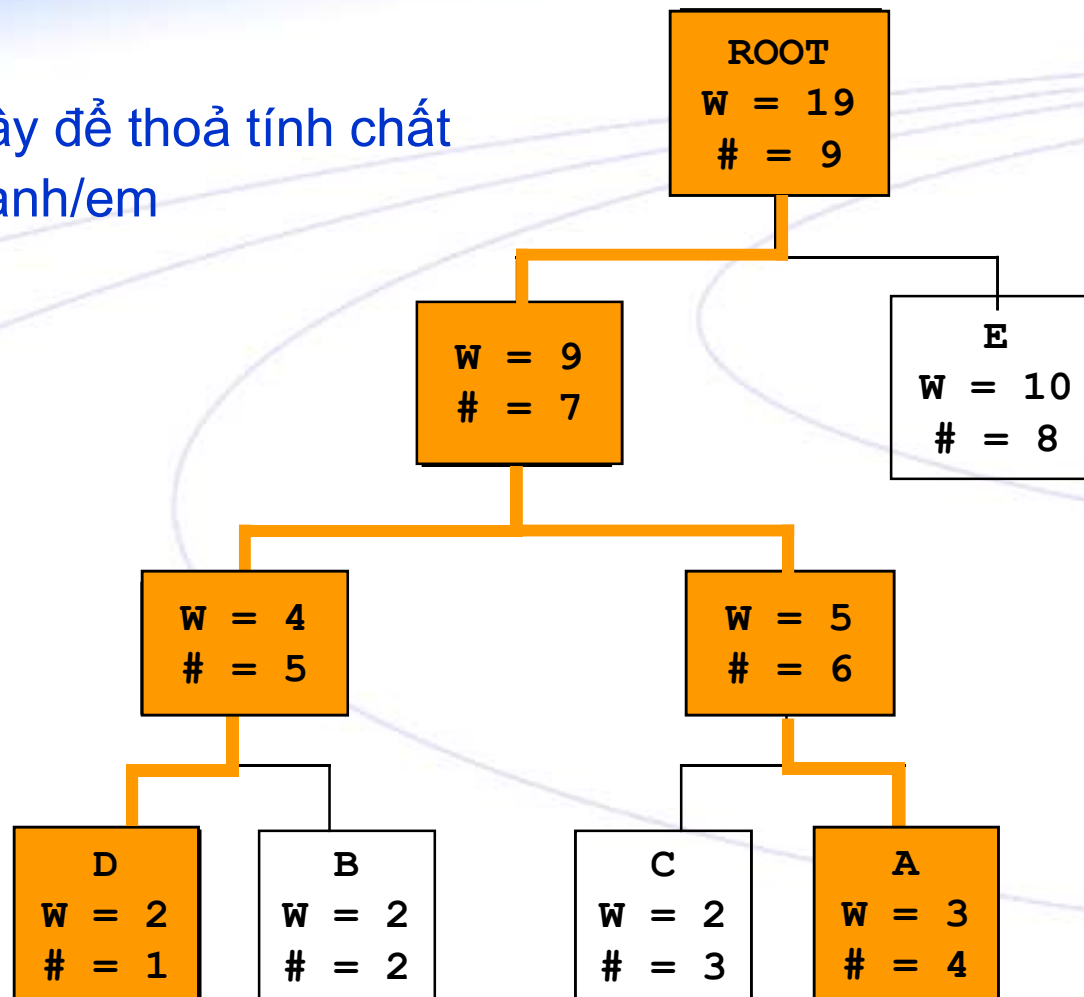
Adaptive Huffman (tt)

Vi phạm tính chất anh/em

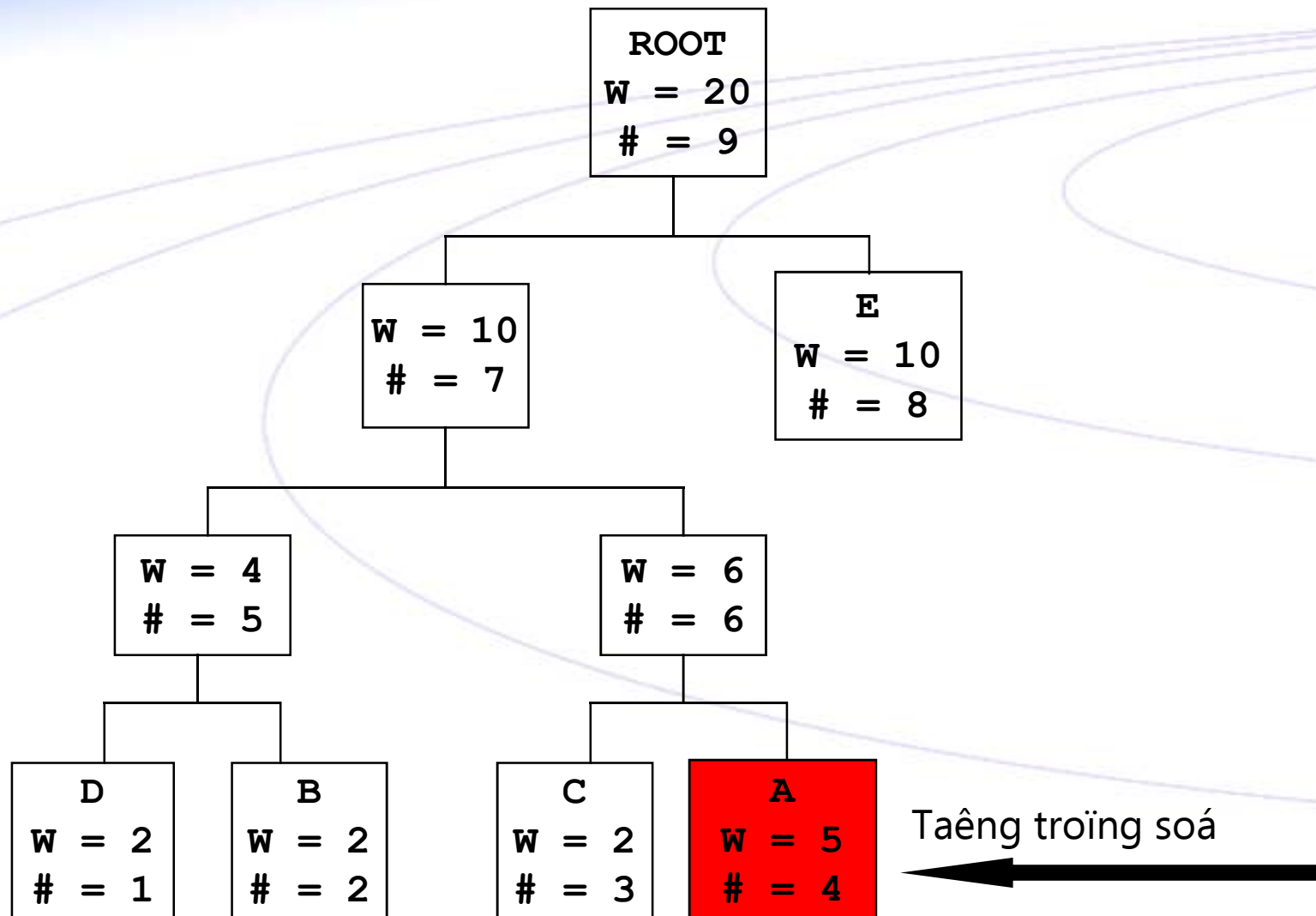


Adaptive Huffman (tt)

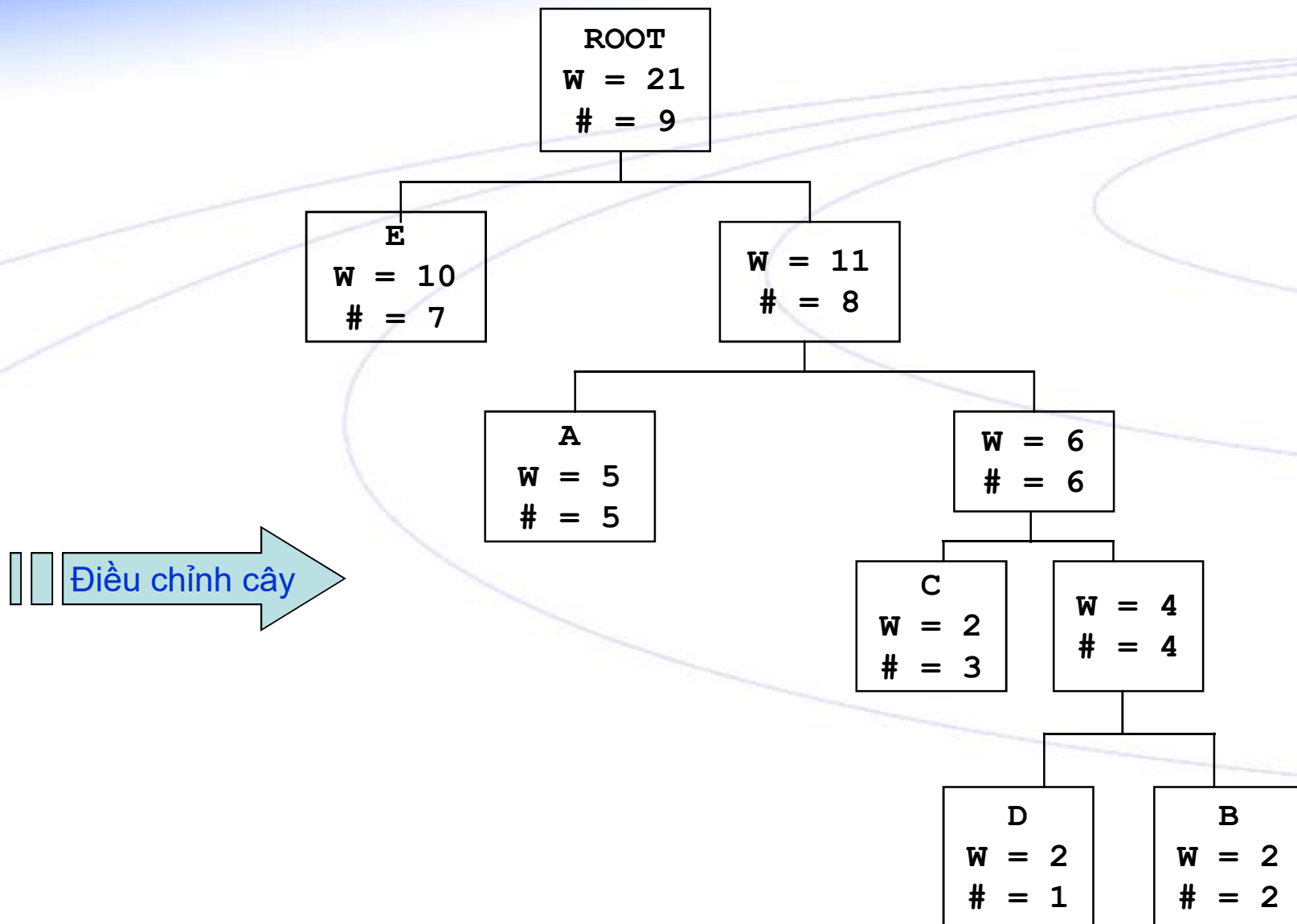
Điều chỉnh cây để thỏa tính chất
anh/em



Adaptive Huffman (tt)



Adaptive Huffman (tt)



Adaptive Huffman (tt)

✦ Cách thức tạo cây: (tt)

Thuật toán “Xác định nút vi phạm”:

- ◆ Gọi x là nút hiện hành
- ◆ So sánh x với các nút tiếp theo sau (từ trái \rightarrow phải, từ dưới \rightarrow trên)
- ◆ Nếu $\exists y$ sao cho: $y.Weight < x.Weight \rightarrow x$ là nút bị vi phạm

Adaptive Huffman (tt)

✦ Cách thức tạo cây: (tt)

Thuật toán “Điều chỉnh cây thỏa tính chất anh/em”:

- ◆ Gọi ***x*** là nút vi phạm
- ◆ Tìm nút ***y xa nhất***, phía sau ***x***, thoả:
$$y.Weight < x.Weight$$
- ◆ Hoán đổi nút ***x*** và nút ***y*** trên cây
- ◆ Cập nhật lại các nút cha tương ứng
- ◆ Lặp lại bước [1] cho đến khi không còn nút vi phạm

Adaptive Huffman (tt)

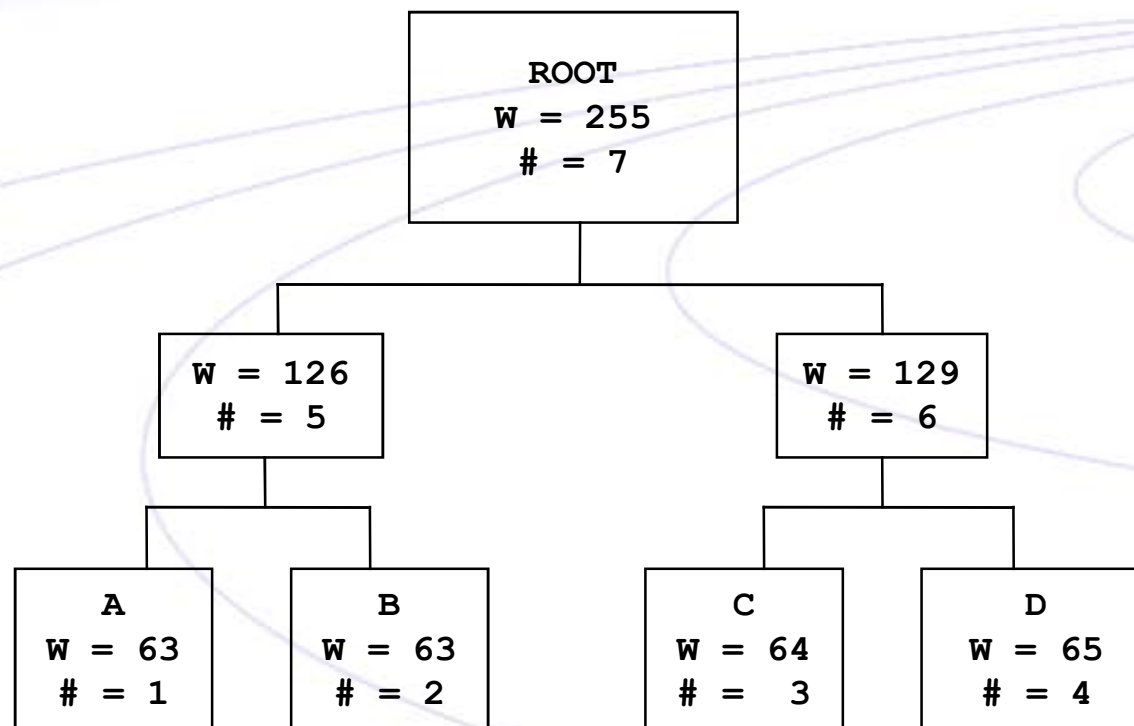
✦ Cách thức tạo cây: (tt)

Vấn đề “tràn số”

- ◆ Quá trình cập nhật cây → tăng trọng số của các nút
- ◆ Trọng số của nút gốc tăng rất nhanh...
→ giá trị trọng số vượt quá khả năng lưu trữ của kiểu dữ liệu

VD. `unsigned int Weight; // Giá trị max 65535`

Adaptive Huffman (tt)



Nút gốc sẽ bị tràn số khi ta tăng trọng số của bất kỳ nút nào

Adaptive Huffman (tt)

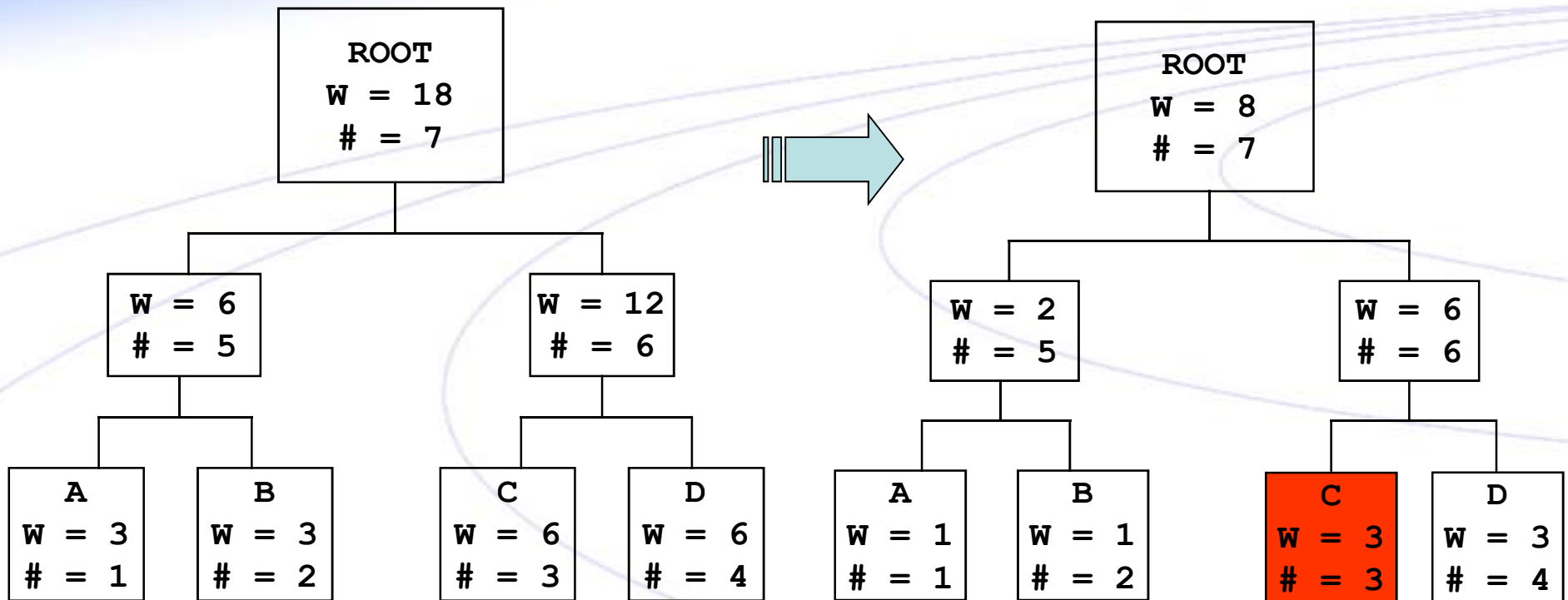
✦ Cách thức tạo cây: (tt)

Thuật toán “Xử lý trường hợp tràn số”:

- ◆ Khi cập nhật trọng số, kiểm tra trọng số của nút gốc
- ◆ Nếu trọng số của nút gốc > **MAX_VALUE**
 - Giảm trọng số các nút lá trong cây (chia cho 2)
 - Cập nhật trọng số các nút nhánh
 - Kiểm tra tính chất anh/em và điều chỉnh lại cây (*)

(*) do phép chia cho 2 làm mất phần dư của số nguyên

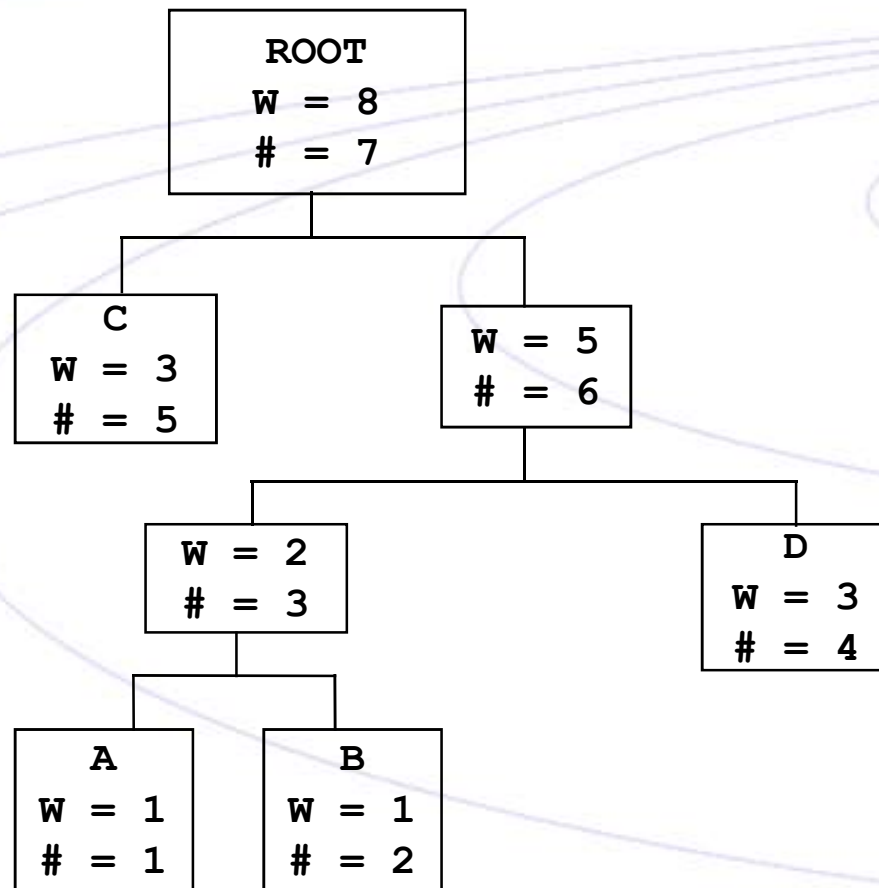
Adaptive Huffman (tt)



Cây bị tràn số

Cây sau khi chia trọng số các nút lá cho 2
và cập nhật lại trọng số các nút nhánh
→ vi phạm tính chất anh/em

Adaptive Huffman (tt)



Cây sau khi điều chỉnh

Adaptive Huffman (tt)

✦ Thuật toán nén (Encoding):

```
// inputfile: dữ liệu cần nén
// outputfile: dữ liệu đã nén
initialize_Tree(T); // khởi tạo cây "tối thiểu"
while(c != EOF) {
    c = getchar(inputfile); // đọc 1 byte dữ liệu
    encode(T, c, outputfile); // mã hoá (nén) c
    update_Tree(T, c); // cập nhật c vào cây
}
```

Adaptive Huffman (tt)

✦ Thuật toán nén (Encoding): (tt)

// Mã hoá ký tự **c** và ghi lên **outputfile**
encode(T, c, outputfile)

- ◆ Nếu **c** chưa có trong cây **T**

- Duyệt cây **T** tìm mã bit của **Escape**, và ghi lên file **outputfile**
- Ghi tiếp 8 bits mã ASCII của **c** lên file **outputfile**

- ◆ Nếu **c** đã có trong cây

- Duyệt cây **T** tìm mã bit của **c**, và ghi lên file **outputfile**

Adaptive Huffman (tt)

★ Thuật toán giải nén (Decoding)

```
// inputfile: dữ liệu ở dạng nén
// outputfile: dữ liệu giải nén
initialize_Tree(T); // khởi tạo cây "tối thiểu"

while((c = decode(T, inputfile)) != EOF) {
    putchar(c, outputfile); // ghi c lên outputfile
    update_Tree(T, c);      // cập nhật c vào cây
}
```

Adaptive Huffman (tt)

★ Thuật toán giải nén (Decoding): (tt)

```
// Giải mã 1 ký tự c từ inputfile  
decode(T, inputfile)
```

- Bắt đầu từ vị trí hiện tại trên **inputfile**
- Lấy từng bit **b**, duyệt trên cây (**b==0**: left; **b==1**: right)
 - ◆ Nếu đi đến 1 nút lá **x** → **return (x.char)**
 - ◆ Nếu đi đến nút **Escape**:
 - **c** = 8 bit tiếp theo từ **inputfile**
 - **return c**



Hỏi & Đáp