University of Science, VNU-HCM
Faculty of Information Technology

Data Structure and Algorithm

Data Compression
RLE, Huffman

Lecturer: Lê Ngọc Thành
Email: lnthanh@fit.hcmus.edu.vn

HCM City

Outlines

- **Data Compression**
  - Lossless compression
  - Lossy compression
- RLE Algorithm
- Huffman Algorithm

Data Compression

- Data compression:
  - Converts the original data to a smaller representation.
  - Data can be used directly after compression or uncompression must be performed to convert to its original form.
- Why do we need to compress data?
  - Minimize storage costs
  - Speed up data transmission
  - Increase security
  - Serve backup data

Data compression types

- There are two data compression types:
  - Compression to preserve information (Lossless compression):
    - No loss of original information
    - Compression efficiency is not high: 10% - 60%
    - Typical algorithms: RLE, Arithmetic, Huffman, LZ77, LZ78,…
  - Compression does not preserve information (Lossy compression):
    - The original information is lost
    - High compression efficiency: 40% - 90%
    - Typical algorithms: JPEG, MP3, MP4, ...

Compression efficiency

- Compression efficiency (%):
  - The percentage of data size is reduced after applying the compression algorithm

$$D\ (\%) = (N - M) / N * 100$$

where D: compression efficiency

N: size of data before compression

M: the size of the data after compression

- Compression efficiency depends on:
  - Compression method
  - Characteristics of the data

Outlines

- Data Compression
  - Lossless compression
  - Lossy compression
- **RLE Algorithm**
  - PCX
  - BMP
- Huffman Algorithm

RLE Algorithm

- RLE = Run Length Encoding: encoding according to the repeated of data
- Run-length: is a sequence of consecutive repeating characters.
  - The "run-length" is represented:
    <Number of iterations> <Character>
  - When the running length is large → data size is reduced significantly.

Example

Data = AAAABBBBBBBBCCCCCCCCCCDEE (# 25 bytes)

Compressed data = 4A8B10C1D2E (# 10 bytes)

RLE Algorithm

- In fact, there is a possibility of ‘side effect’:
  - Data: ABCDEFGH(8 bytes)
  - Compressed data: 1A1B1C1D1E1F1G1H (16 bytes)

- Need to make appropriate modifications.

Outlines

- Data Compression
  - Lossless compression
  - Lossy compression
- **RLE Algorithm**
  - PCX
  - BMP
- Huffman Algorithm

PCX image data

- PCX image data are compressed using run-length encoding (RLE) with fixed 'side effect':
  - Byte specifies quantity (more than 1): 2 bits 6,7 is enabled.
- Example:
  - A string of 5 characters A, 0x41, (AAAAA) is encoded

1	1	0	0	0	1	0	1	0	1	0	0	0	0	0	1
0xC5								0x41							
197_{10}								65_{10}							

PCX

- Fixes ‘side effect’:
  - Byte specifies quantity: bit 6,7.
    - Maximum number of repetitions: **63**
    - Maximum data value: **191** (0-191)
  - How about data with a number of iterations of 1?
    - Data has a value below 192?
    - Data has a value from 192?

PCX

- For number of iterations of 1
  - Data has a value below 192?
    - Not affected
    - E.g.: compress 2 characters **0x41 0x43**

0	1	0	0	0	0	0	1	0	1	0	0	0	0	1	1

  - Data has a value from 192?
    - Affected (confused with quantity information).
    - Uses 2 bytes: < Quantity = 1> <Data>
    - E.g.: compress character **0xDB** (219_{10})

1	1	0	0	0	0	0	1	1	1	0	1	1	0	1	1

PCX

- Advantage:
  - Simple
  - Reduces “side effect"
- Disadvantages:
  - Using 6 bits to represent the number of iterations only represents a maximum length of 63.
  - Long run will have to repeat again.
  - Cannot solve “side effect" with ASCII code ≥ 192

PCX

```
#define MAX_RUNLENGTH 63
int PCXEncode_a_String(char *aString, int nLen, FILE *fEncode)
{
   unsigned char cThis, cLast;
   int nTotal = 0; // Total number of bytes after compression
   int nRunCount = 1; // Length of 1 run
   cLast = *(aString);
   for (int i=0; i<nLen; i++) {
        cThis = *(++aString);
        if (cThis == cLast) { // Exists 1 run
               nRunCount++;
               if (nRunCount == MAX_RUNLENGTH) {
               nTotal += PCXEncode_a_Run(cLast,nRunCount,fEncode);
               nRunCount = 0;
               }
        }
```

PCX

```
        else // End of 1 run, move to the next run
        {
                if (nRunCount)
                  nTotal += PCXEncode_a_Run(cLast,nRunCount,fEncode);
                cLast = cThis;
                nRunCount = 1;
        }
    } // end for
    if (nRunCount) // Record last run to file
        nTotal += PCXEncode_a_Run(cLast, nRunCount, fEncode);
    return (nTotal);
}
```

PCX

```
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;
   }
}
```

Outlines

- Data Compression
  - Lossless compression
  - Lossy compression
- **RLE Algorithm**
  - PCX
  - BMP
- Huffman Algorithm

BMP

- Limitations of RLE on PCX :
  - Compress 255 characters A?

AAA...AAA...AAA

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

(Because 255 = 4 x 63 + 3)

BMP

- Idea:
  - Handling separately for each run case (repeat and non-repeat).
    - **AAAAA**BCDEF
  - Markers are used

BMP

- Normal run (with repeat):

  <Number of repeats> <Character>

Compressed Data	Uncompressed Data
0x01 0x00	0x00
0x0A 0xFF	0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

BMP

- Trường hợp là ký tự riêng lẻ:

<Marker> <Number of non-repeat characters> <Character>

  – Marker: **0x00**
  – Used in the case of a run of 3 or more non-repeat characters.
  – E.g:

Compressed Data	Uncompressed Data
00 03 01 02 03	01 02 03
00 04 0x41 0x42 0x43 0x44	0x41 0x42 0x43 0x44

BMP

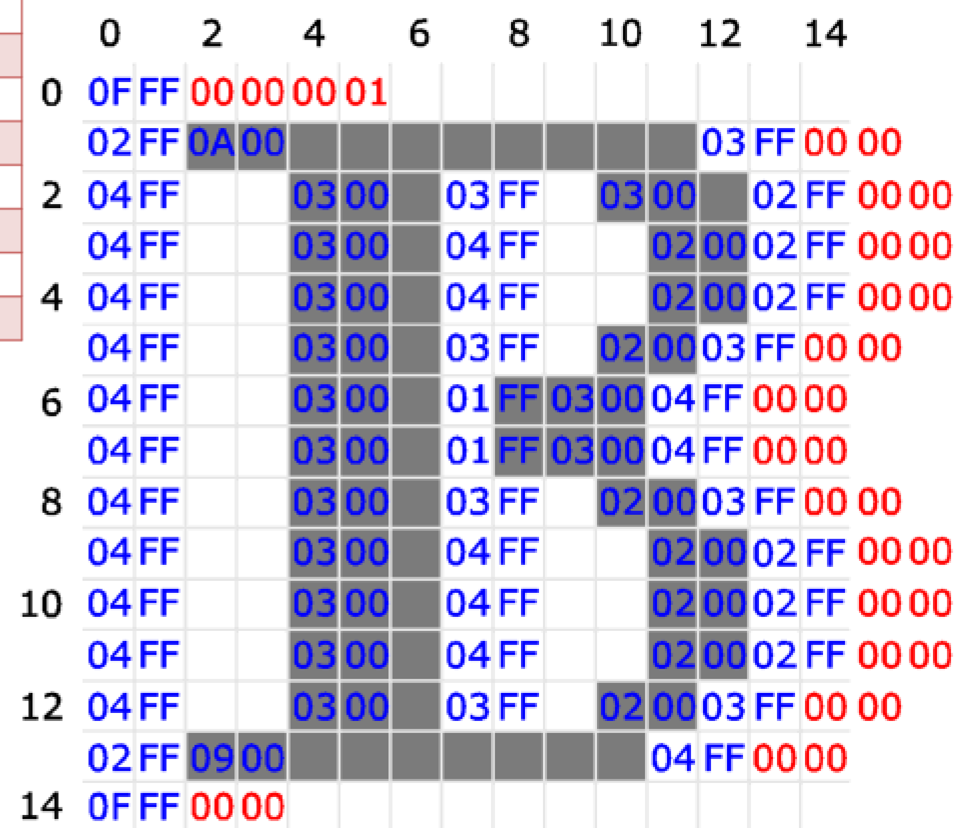
- Other case:
  - **0x00** 0x00: kết thúc dòng
  - **0x00** 0x01: kết thúc tập tin
  - **0x00** 0x02 <DeltaX> <DeltaY>: the jump (DeltaX, DeltaY) from the current position. The next data is applied at the new location.

Example

00	0F FF 00 00
01	02 FF 09 00 04 FF 00 00
02	04 FF 03 00 03 FF 02 00 03 FF 00 00
03	04 FF 03 00 04 FF 02 00 02 FF 00 00
04	04 FF 03 00 04 FF 02 00 02 FF 00 00
05	04 FF 03 00 04 FF 02 00 02 FF 00 00
06	04 FF 03 00 03 FF 02 00 03 FF 00 00
07	04 FF 03 00 01 FF 03 00 04 FF 00 00
08	04 FF 03 00 01 FF 03 00 04 FF 00 00
09	04 FF 03 00 03 FF 02 00 03 FF 00 00
10	04 FF 03 00 04 FF 02 00 02 FF 00 00
11	04 FF 03 00 04 FF 02 00 02 FF 00 00
12	04 FF 03 00 03 FF 03 00 02 FF 00 00
13	02 FF 0A 00 03 FF 00 00
14	0F FF 00 00 00 01

Example

00	0F FF 00 00
01	02 FF 09 00 04 FF 00 00
02	04 FF 03 00 03 FF 02 00 03 FF 00 00
03	04 FF 03 00 04 FF 02 00 02 FF 00 00
04	04 FF 03 00 04 FF 02 00 02 FF 00 00
05	04 FF 03 00 04 FF 02 00 02 FF 00 00
06	04 FF 03 00 03 FF 02 00 03 FF 00 00
07	04 FF 03 00 01 FF 03 00 04 FF 00 00
08	04 FF 03 00 01 FF 03 00 04 FF 00 00
09	04 FF 03 00 03 FF 02 00 03 FF 00 00
10	04 FF 03 00 04 FF 02 00 02 FF 00 00
11	04 FF 03 00 04 FF 02 00 02 FF 00 00
12	04 FF 03 00 03 FF 03 00 02 FF 00 00
13	02 FF 0A 00 03 FF 00 00
14	0F FF 00 00 00 01



Outlines

- Data Compression
  - Lossless compression
  - Lossy compression
- RLE Algorithm
- **Huffman Algorithm**
  - Static Huffman
  - Adaptive Huffman

Huffman

- Huffman
  - The compression method to preserve information
  - Does not depend on the type of the data
- Idea:
  - Using bits to represent characters (called "bit code")
  - The bit code length for each character can vary (variable length encoding):
    - Characters appear many times → short code
    - Characters appear a few times → long code

Huffman

- Data:

f = “ADDAABBCCBAAABBCCCBBBCDAADDEEAA”

- Normal representation (8 bits / character):

Sizeof (f) = 10 * 8 + 8 * 8 + 6 * 8 + 5 * 8 + 2 * 8 = 248 bits

Character	Frequency
A	10
B	8
C	6
D	5
E	2

Huffman

- Represented by variable length encoding:

Sizeof(f) = 10*2 + 8*2 + 6*2 + 5*3 + 2*3 = 69 bits

Character	Code
A	11
B	10
C	00
D	011
E	010

f = “ADDAABBCCBAAABBCCCBBBCDAADDEEAA”

Static Huffman vs Adaptive Huffman

- Huffman is divided into two types:
  - Static Huffman:
    - Data must be available to generate the bit code. Cannot compress when data is coming (online).
    - Need to save information (bit / frequency code) to serve the uncompression process.
    - Need to read the file 2 times to compress.
  - Adaptive Huffman:
    - Data don’t need available for compression, data can be generated in real time.
    - No need to save information for uncompression.
    - Only read the file once for compression because there is no need to pre-calculate the frequency of characters.

Outlines

- Data Compression
  - Lossless compression
  - Lossy compression
- RLE Algorithm
- **Huffman Algorithm**
  - Static Huffman
  - Adaptive Huffman

Static Huffman

- Compression algorithm:
  - S1: Read file → Make statistics table of occurrences of each character
  - S2: Huffman tree generation based on statistics
  - S3: From Huffman tree → generate bit encoding for characters
  - S4: Read file again → replace characters with corresponding encoded bits.
  - S5: Save the information of the Huffman tree to be uncompressed

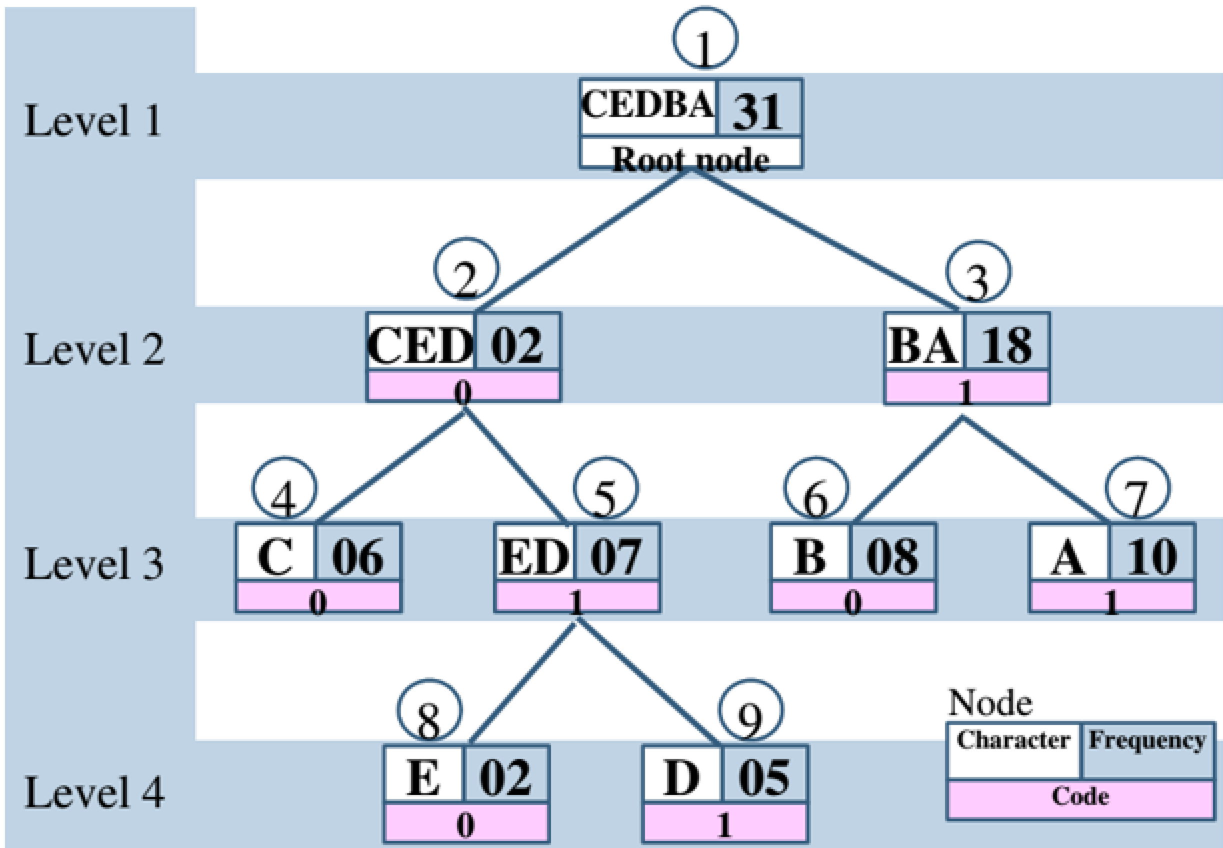
Example

f = "ADDAABBCCBAAABBCCCBBBCDAADDEEAA"

[S1]

Char	Frequency
A	10
B	8
C	6
D	5
E	2

[S2]



[S3]

Char	Bit Code
A	11
B	10
C	00
D	011
E	010

[S4]

f = "11011011111110100000101111111010000000
1010100001111110110110100101111"

Huffman Tree

- The Huffman tree is a binary tree
  - Each leaf node contains 1 character
  - The parent node will contain the characters of the child nodes
  - Each node is assigned a weight
    - The leaf node is weighted by the number of occurrences of the character in the file
    - The parent node has weights equal to the total weights of the children

Huffman Tree

```
#define MAX_NODES 511 // 2*256 - 1
typedef struct {
     char c;         //character
     long nFreq;     // weight
     int nLeft;      // left subtree
     int nRight;     // right subtree
} HUFFNode;
HUFFNode HuffTree[MAX_NODES];
```

Generate Huffman Tree

- Tree generation algorithm:
  - S1: Select in the statistics table 2 elements x, y with the lowest weight → forming a parent node z:

    z.c = x.c + y.c;

    z.nFreq = x.nFreq + y.nFreq;

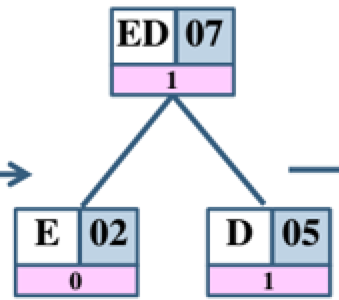
    z.nLeft = x (*)

    z.nRight = y (*)
  - S2: Remove x and y from the table;
  - S3: Adds the z to the table;
  - S4: Repeat steps S1 to S3 until there is only 1 value left in the table.

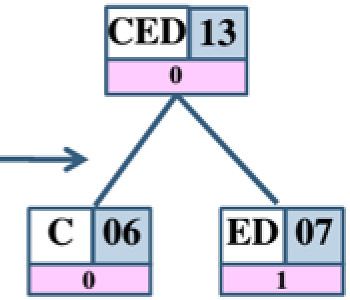
(*) Rules:
- The smaller weight node is on the left branch; The larger weight node is on the right branch
- If the weights are equal, the node with the smaller string is on the left branch, the node with the larger string is on the right

Example

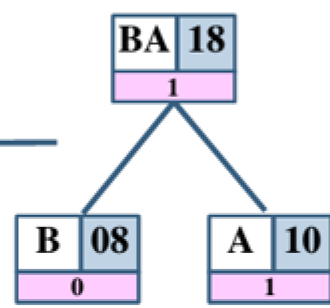
Char	Frequency
A	10
B	8
C	6
D	5
E	2



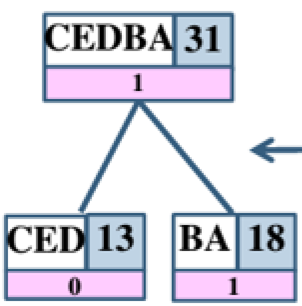
Char	Frequency
A	10
B	8
ED	7
C	6



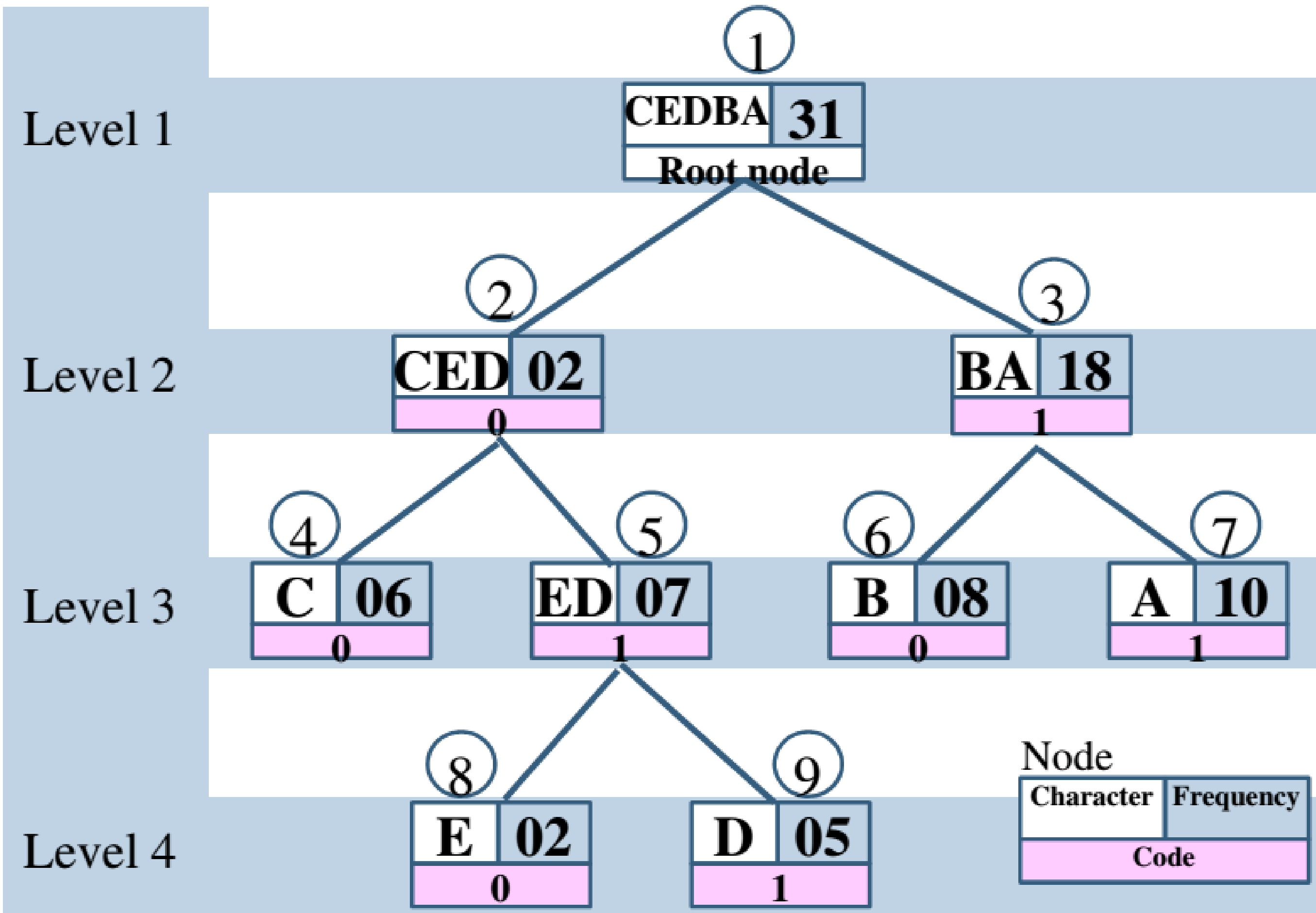
Char	Frequency
CED	13
A	10
B	8



Char	Frequency
BA	18
CED	13



Example



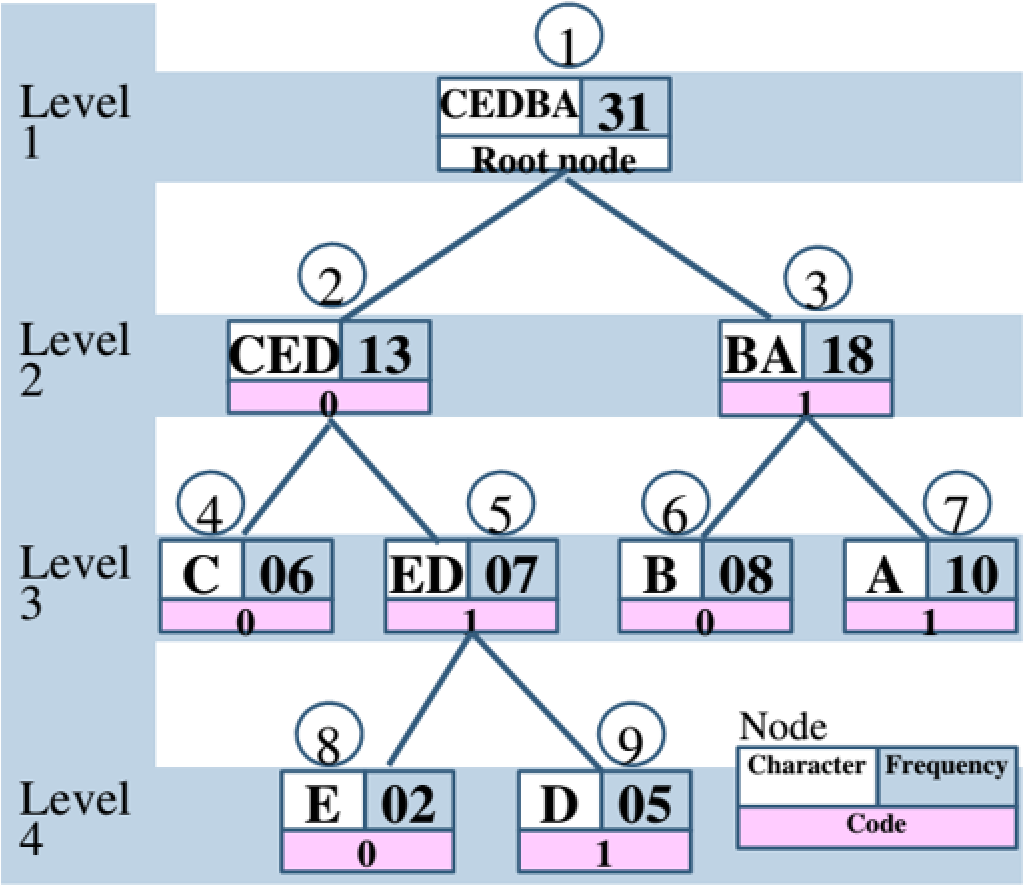
Level 1
Level 2
Level 3
Level 4
1
CEDBA 31
Root node
2
CED 02
0
3
BA 18
1
4
C 06
0
5
ED 07
1
6
B 08
0
7
A 10
1
8
E 02
0
9
D 05
1
Node
Character
Frequency
Code

Generate Bit Code

- Generate bit codes for the characters:
  - The code for each character is created by traversing the root node to the leaf node containing that character;
    - When traversing to the left produces bit 0;
    - When traversing to the right produces bit 1;

Example

Char	Frequency
A	11
B	10
C	00
D	011
E	010



Save information to uncompress

Method 1:

Char	Frequency
A	11
B	10
C	00
D	011
E	010

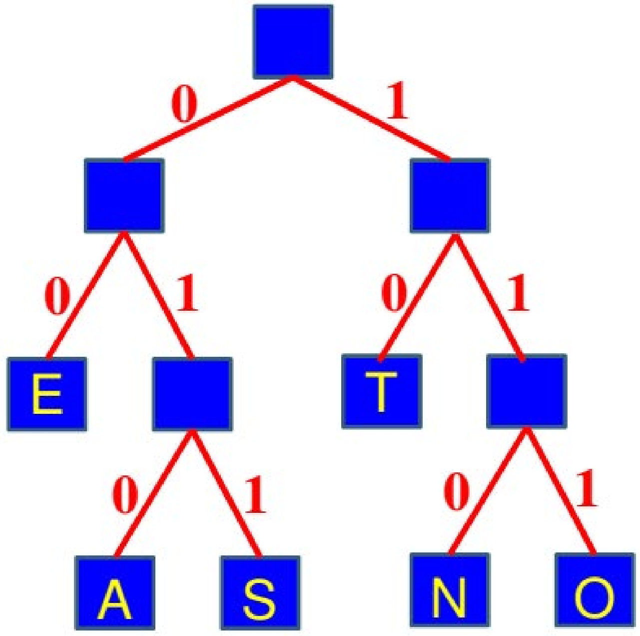
Method 2:

Char	Bit Code
A	10
B	8
C	6
D	5
E	2

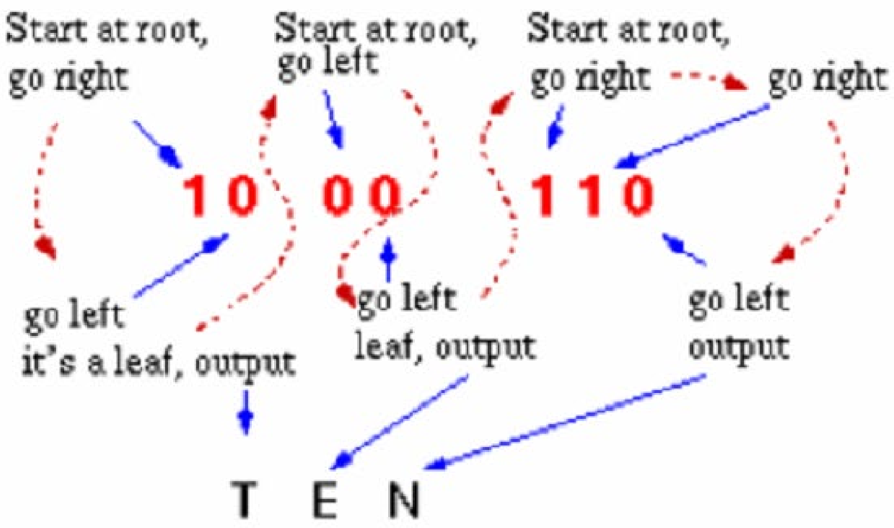
Uncompress

- Decompression algorithm:
  - S1: Rebuilding the Huffman tree (from saved info)
  - S2: Initialize current node pCurr = pRoot
  - S3: Read 1 bit b from the compressed f_n
  - S4: If (b == 0) then pCurr = pCurr.nLeft otherwise pCurr = pCurr.nRight
  - S5:
    - If pCurr is a leaf node:
      - Export characters at pCurr to file
      - Go back to step S2
    - Otherwise
      - Go back to step S3
  - S6: Stop if go to end of f_n file

Example



0
1
0
1
0
1
E
T
0
1
0
1
A
S
N
O



Start at root,
go right
Start at root,
go left
Start at root,
go right
go right
10
00
110
go left
it's a leaf, output
go left
leaf, output
go left
output
T
E
N

Outlines

- Data Compression
  - Lossless compression
  - Lossy compression
- RLE Algorithm
- **Huffman Algorithm**
  - Static Huffman
  - Adaptive Huffman

Adaptive Huffman

- The compression/uncompression of Adaptive Huffman will be done at the same time as the tree update process.
- Compress with Adaptive Huffman:
  - Tree initialization
  - Read input characters
  - Compress character and update tree
- Uncompress with Adaptive Huffman:
  - Tree initialization
  - Read characters from compressed data
  - Uncompress and update tree

Adaptive Huffman Tree

- A binary tree with n leaf nodes is called a Huffman tree if it satisfies:
  - Leaf nodes with weight $W_i \geq 0, \; i \in [1..n]$
  - Inside nodes have weights equal to the total weights of their children.
  - Sibling Property:
    - Each node, except for the root node, has only one sibling.
    - When arranging nodes in the tree in ascending order of weight, each node is always adjacent to its sibling node.

Adaptive Huffman Tree

- To easy in the creation of trees, we have some conventions:
  - Nodes will be assigned a descending order number.
  - Newly added nodes always have a smaller number than the existing nodes in the tree.
- In the process of creating a tree, sibling property must be preserved
  - nodes with a large number must have a weight >= nodes with small weights
  - If not, need adjustment

Adaptive Huffman Tree

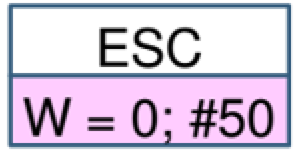
- Initialize the tree:
  - Creates a "minimal" tree, only the Escape node (0-node) or the NYT (Not yet transmitted) node: a childless, characterless node.
- Insert a letter c into the tree:
  - If c is not in the tree, add a new leaf node
  - If c is already in the tree, increase the weight of node c by 1
  - Update the weights of the relevant nodes in the tree

Add new leaf node

- When adding a leaf button to the letter c:
  - Create 2 nodes:
    - a node for the c character (W = 1), and
    - a new Escape node (W = 0)
  - Two new nodes are added as children of the current Escape node
  - Check for sibling property
  - Update weights of the relevant nodes

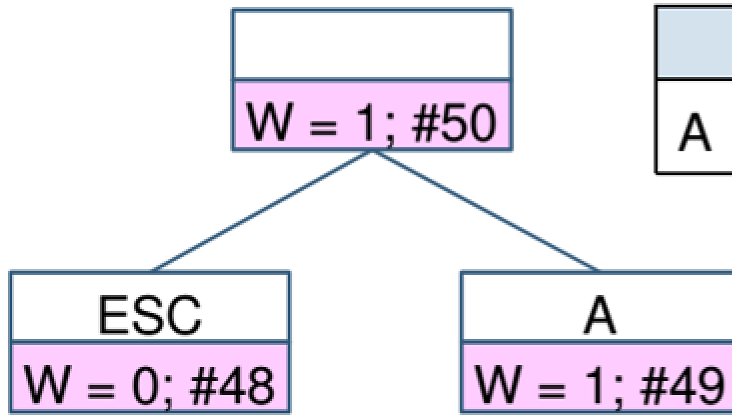
Compress

- Initialize tree:



Compressed Data

- Add ‘A’:
  - Compress ‘A’
  - Update tree

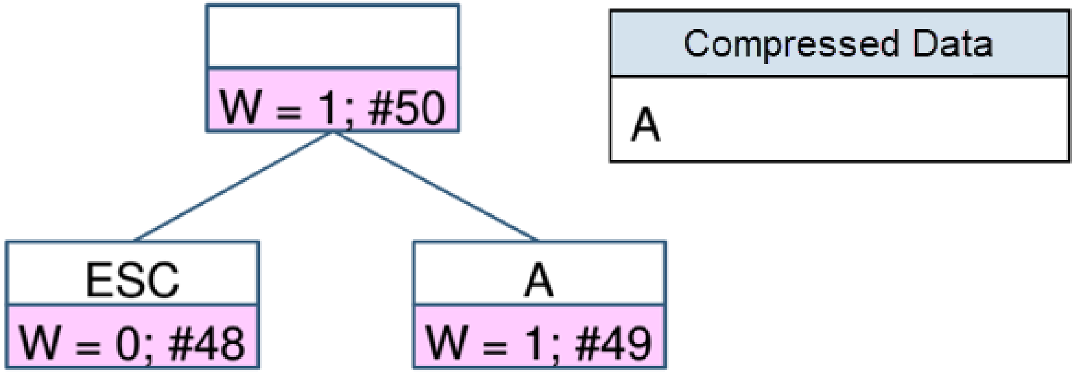


Compressed Data
A

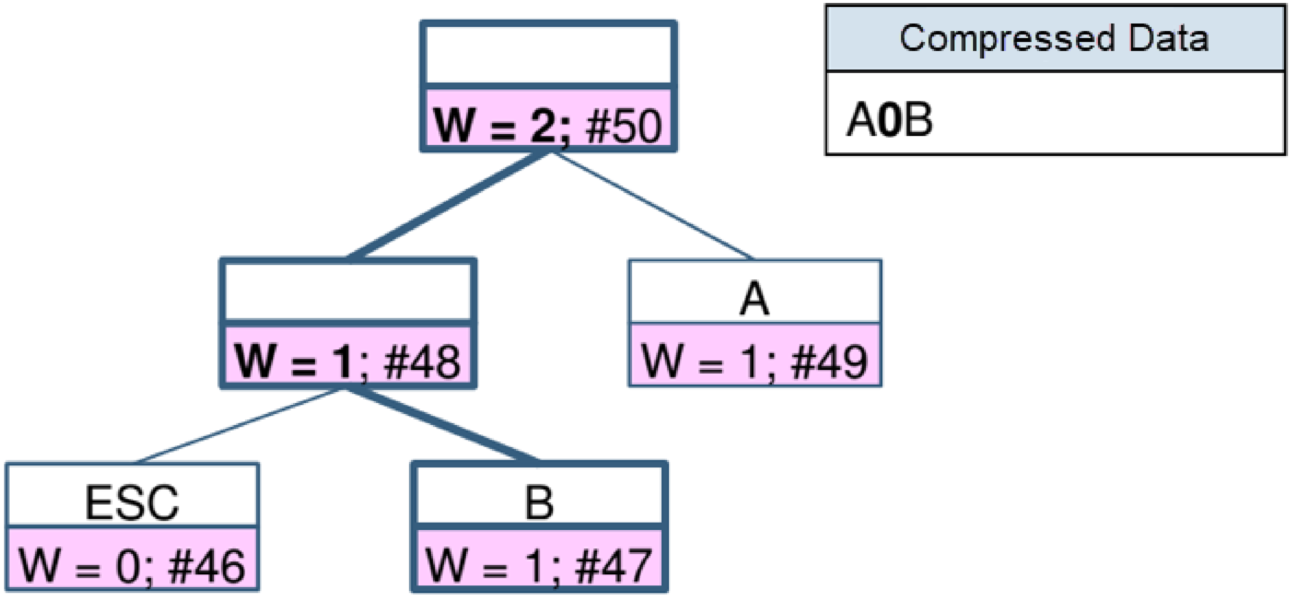
Update weight

- Start from inserted or updated node:
  - Go back to the root node and increase weight of nodes by 1.
  - Check the sibling property of node
    - Adjust if sibling property violation

Example



- Add ‘B’:
  - Compress ‘B’
  - Update tree



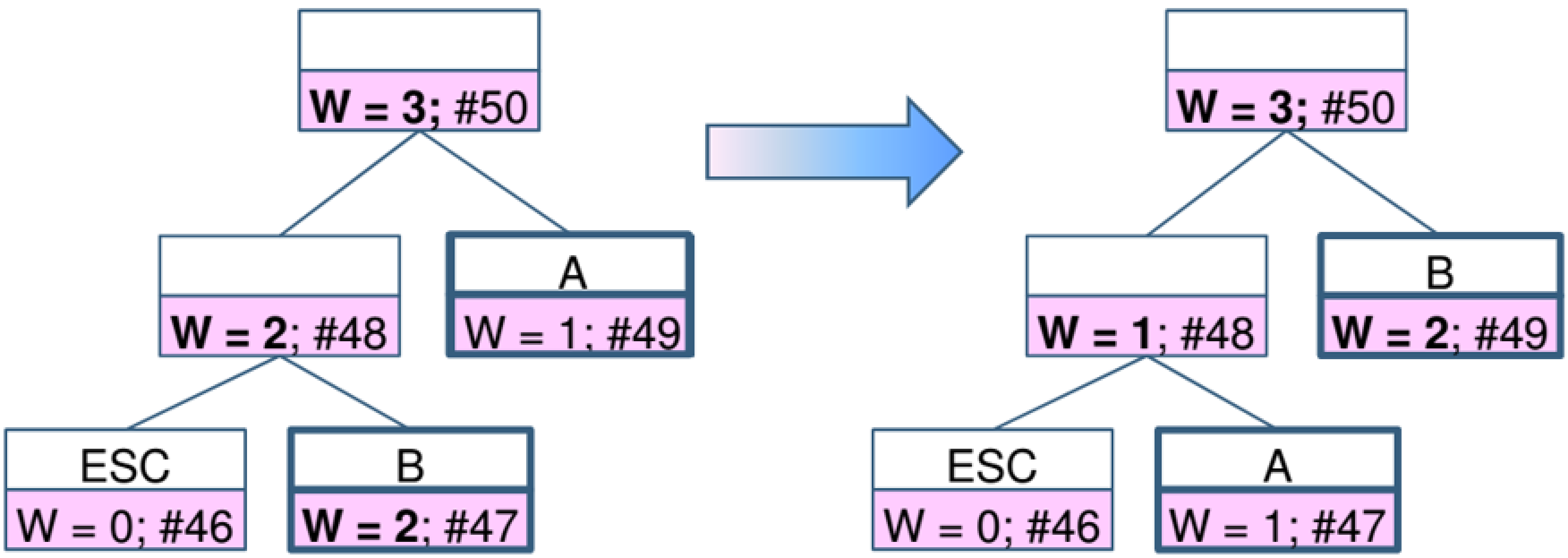
Check sibling property

- The sibling property is violated when:
  - There exists a node X with weight = W + 1 whose ordinal number is less than a node Y with weight = W.
- Adjust:
  - Swap data in node X with node which has largest order and weigh = W.
  - Update weights of relevant nodes
  - Check propagation to other nodes.

Example

- Add ‘B’:

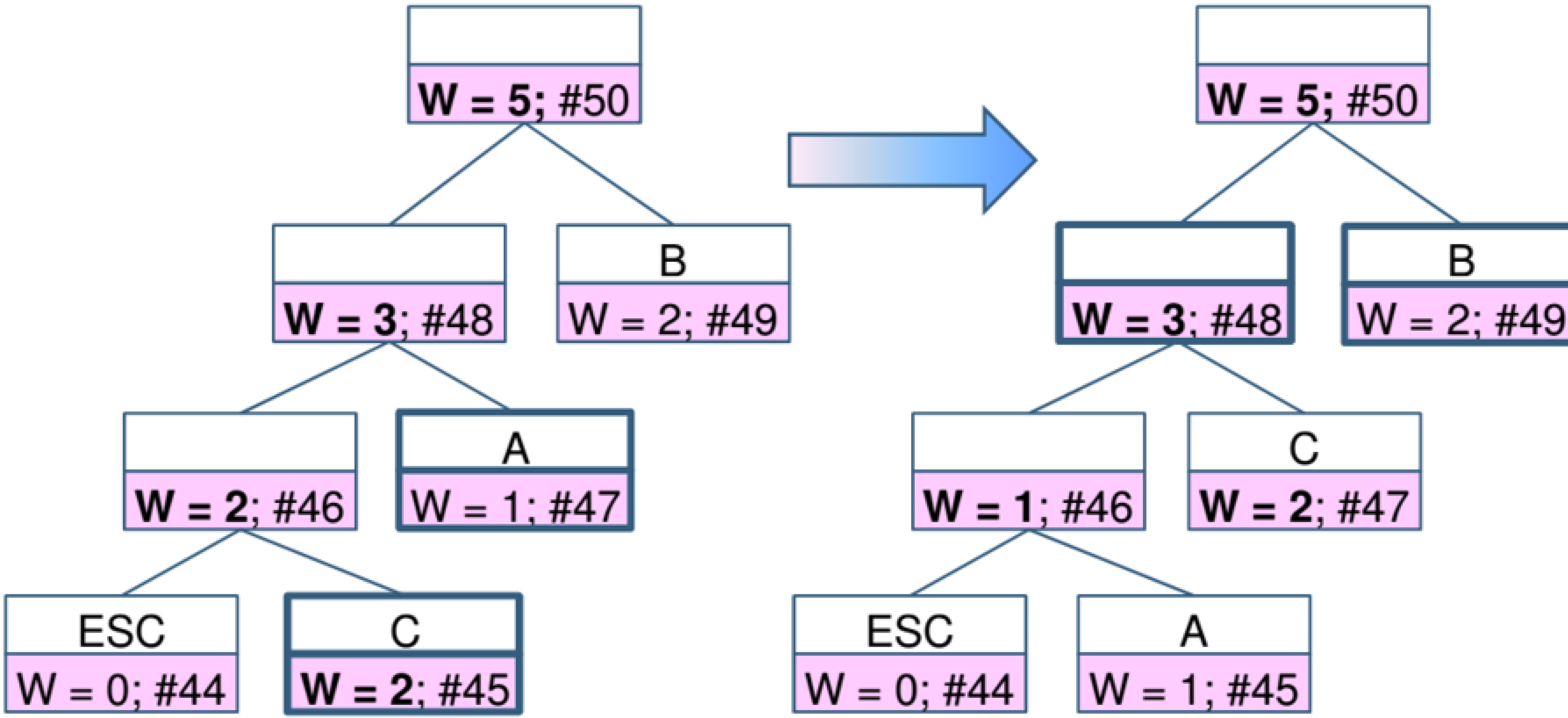
Compressed Data
A0B01



Example

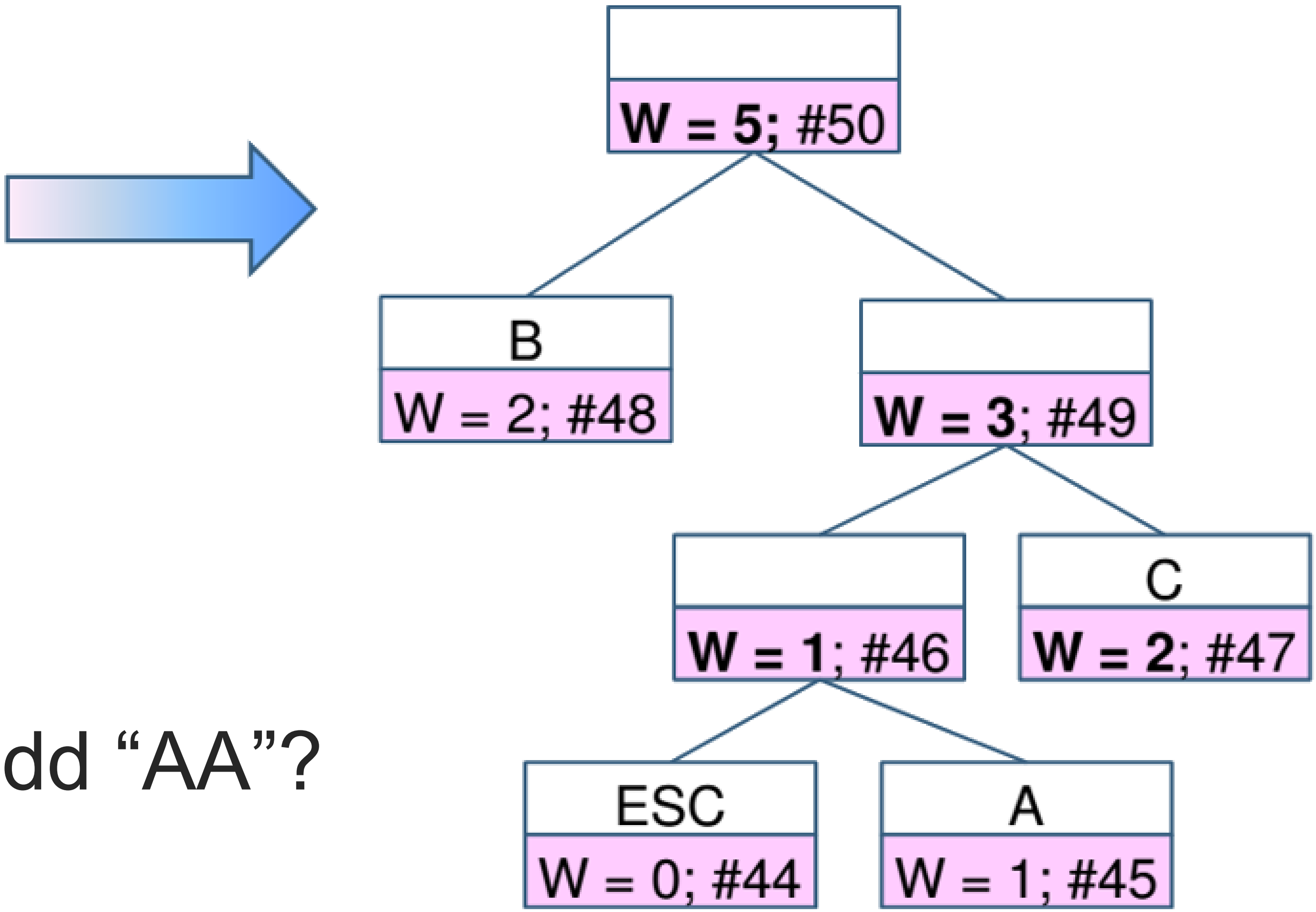
- Add “CC”:

Compressed Data
A0B0100C001



Example

- Update tree



- How to add “AA”?

Uncompress

- Tree initialization
- Read bits from compressed data one by one:
  - Traverse the Huffman tree to extract characters
    - If the character is Escape, read next uncompressed character (8-bit).
    - If it is normal character, output the corresponding character
  - Update the Huffman tree with the existing character (similar to compression).

Exercise

- Use adaptive Huffman to compress the following data:

    aabcdad

- Uncompress after that.

The End.