

## 5.5 Data Compression



- ▶ basics
- ▶ run-length coding
- ▶ Huffman compression
- ▶ LZW compression

# Data compression

Compression reduces the size of a file:

- To save **space** when storing it.
- To save **time** when transmitting it.
- Most files have lots of redundancy.

Who needs compression?

- Moore's law: # transistors on a chip doubles every 18-24 months.
- Parkinson's law: data expands to fill space available.
- Text, images, sound, video, ...

*“ All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year. Not all bits have equal value. ” — Carl Sagan*

Basic concepts ancient (1950s), best technology recently developed.

# Applications

## Generic file compression.

- Files: GZIP, BZIP, BOA.
- Archivers: PKZIP.
- File systems: NTFS.

## Multimedia.

- Images: GIF, JPEG.
- Sound: MP3.
- Video: MPEG, DivX™, HDTV.

## Communication.

- ITU-T T4 Group 3 Fax.
- V.42bis modem.

## Databases. Google.



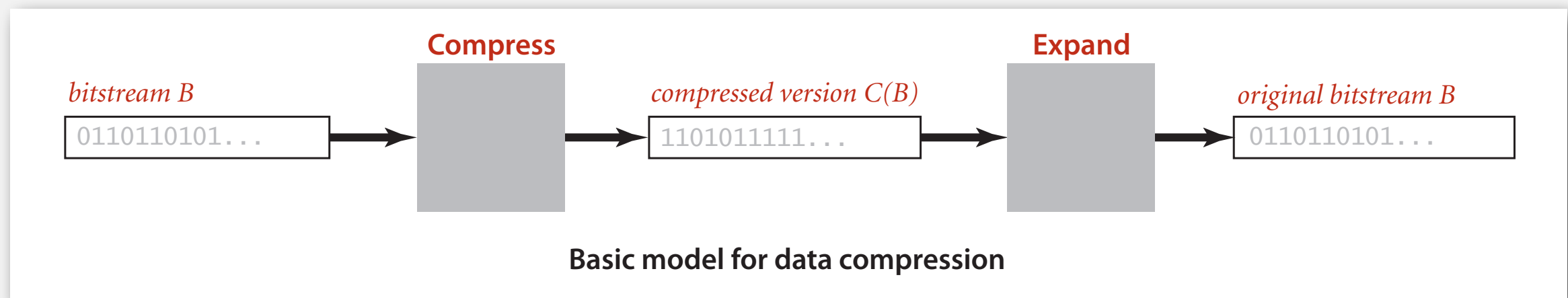
# Lossless compression and expansion

**Message.** Binary data  $B$  we want to compress.

**Compress.** Generates a "compressed" representation  $C(B)$ .

**Expand.** Reconstructs original bitstream  $B$ .

uses fewer bits (you hope)



**Compression ratio.** Bits in  $C(B)$  / bits in  $B$ .

**Ex.** 50-75% or better compression ratio for natural language.

Data compression has been omnipresent since antiquity:

- Number systems.
- Natural languages.
- Mathematical notation.

has played a central role in communications technology,

- Braille.
- Morse code.
- Telephone system.

and is part of modern life.

- MP3.
- MPEG.

Q. What role will it play in the future?

- ▶ **basics**
- ▶ run-length coding
- ▶ Huffman compression
- ▶ LZW compression

# Data representation: genomic code

**Genome.** String over the alphabet  $\{ A, C, T, G \}$ .

**Goal.** Encode an  $N$ -character genome: `ATAGATGCATAG...`

**Standard ASCII encoding.**

- 8 bits per char.
- $8N$  bits.

char	hex	binary
A	41	01000001
C	43	01000011
T	54	01010100
G	47	01000111

**Two-bit encoding.**

- 2 bits per char.
- $2N$  bits.

char	binary
A	00
C	01
T	10
G	11

**Amazing but true.** Initial genomic databases in 1990s did not use such a code!

**Fixed-length code.**  $k$ -bit code supports alphabet of size  $2^k$ .

# Reading and writing binary data

Binary standard input and standard output. Libraries to read and write bits from standard input and to standard output.

```
public class BinaryStdIn
```

---

```
boolean readBoolean()      read 1 bit of data and return as a boolean value
```

```
char readChar()           read 8 bits of data and return as a char value
```

```
char readChar(int r)      read r bits of data and return as a char value
```

*[similar methods for byte (8 bits); short (16 bits); int (32 bits); long and double (64 bits)]*

```
boolean isEmpty()         is the bitstream empty?
```

```
void close()             close the bitstream
```

---

```
public class BinaryStdOut
```

---

```
void write(boolean b)     write the specified bit
```

```
void write(char c)       write the specified 8-bit char
```

```
void write(char c, int r) write the r least significant bits of the specified char
```

*[similar methods for byte (8 bits); short (16 bits); int (32 bits); long and double (64 bits)]*

```
void close()             close the bitstream
```

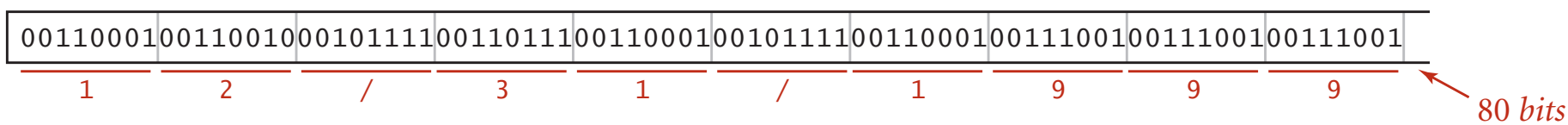
---



Date representation. Different ways to represent 12/31/1999.

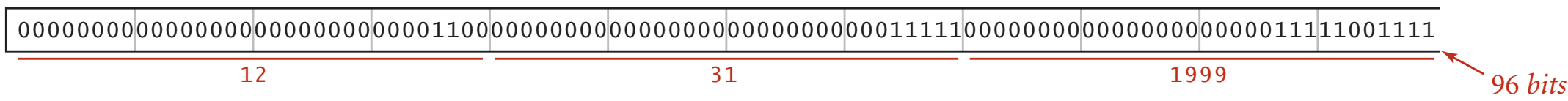
A character stream (StdOut)

```
StdOut.print(month + "/" + day + "/" + year);
```



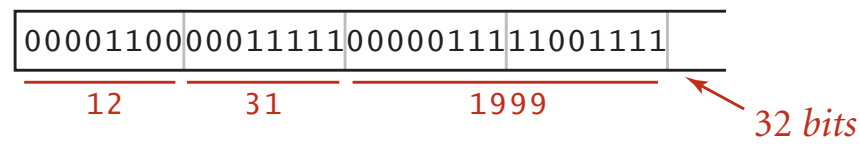
Three ints (BinaryStdOut)

```
BinaryStdOut.write(month);  
BinaryStdOut.write(day);  
BinaryStdOut.write(year);
```



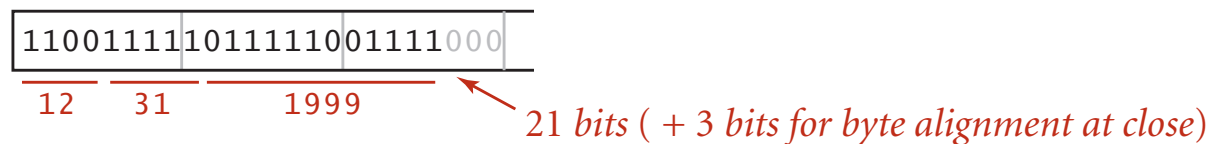
Two chars and a short (BinaryStdOut)

```
BinaryStdOut.write((char) month);  
BinaryStdOut.write((char) day);  
BinaryStdOut.write((short) year);
```



A 4-bit field, a 5-bit field, and a 12-bit field (BinaryStdOut)

```
BinaryStdOut.write(month, 4);  
BinaryStdOut.write(day, 5);  
BinaryStdOut.write(year, 12);
```



Four ways to put a date onto standard output

Q. How to examine the contents of a bitstream?

Standard character stream

```
% more abra.txt
ABRACADABRA!
```

Bitstream represented as 0 and 1 characters

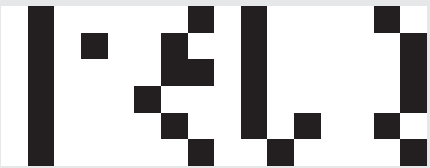
```
% java BinaryDump 16 < abra.txt
0100000101000010
0101001001000001
0100001101000001
0100010001000001
0100001001010010
0100000100100001
96 bits
```

Bitstream represented with hex digits

```
% java HexDump 4 < abra.txt
41 42 52 41
43 41 44 41
42 52 41 21
12 bytes
```

Bitstream represented as pixels in a Picture

```
% java PictureDump 16 6 < abra.txt
```



16-by-6 pixel window, magnified

96 bits

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	“	#	\$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Hexadecimal to ASCII conversion table

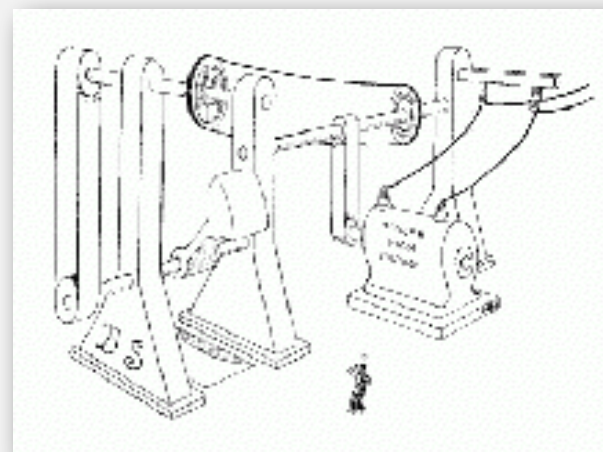
## Universal data compression

US Patent 5,533,051 on "Methods for Data Compression", which is capable of compressing **all** files.

Slashdot reports of the Zero Space Tuner™ and BinaryAccelerator™.

*“ ZeoSync has announced a breakthrough in data compression that allows for 100:1 lossless compression of **random** data. If this is true, our bandwidth problems just got a lot smaller.... ”*

Physical analog. Perpetual motion machines.



Gravity engine by Bob Schadewald

# Universal data compression

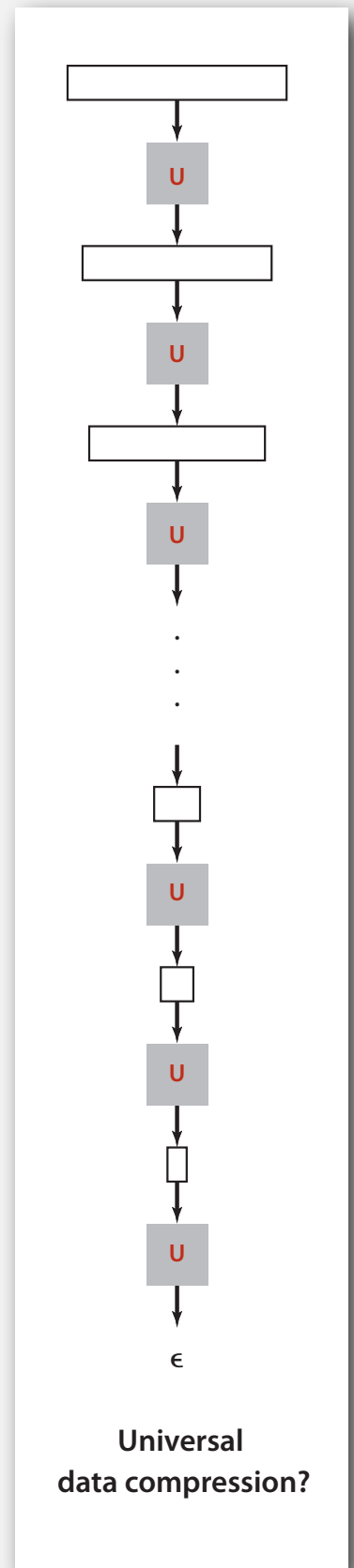
**Proposition.** No algorithm can compress every bitstring.

**Pf 1.** [by contradiction]

- Suppose you have a universal data compression algorithm  $U$  that can compress every bitstream.
- Given bitstring  $B_0$ , compress it to get smaller bitstring  $B_1$ .
- Compress  $B_1$  to get a smaller bitstring  $B_2$ .
- Continue until reaching bitstring of size 0.
- Implication: all bitstrings can be compressed with 0 bits!

**Pf 2.** [by counting]

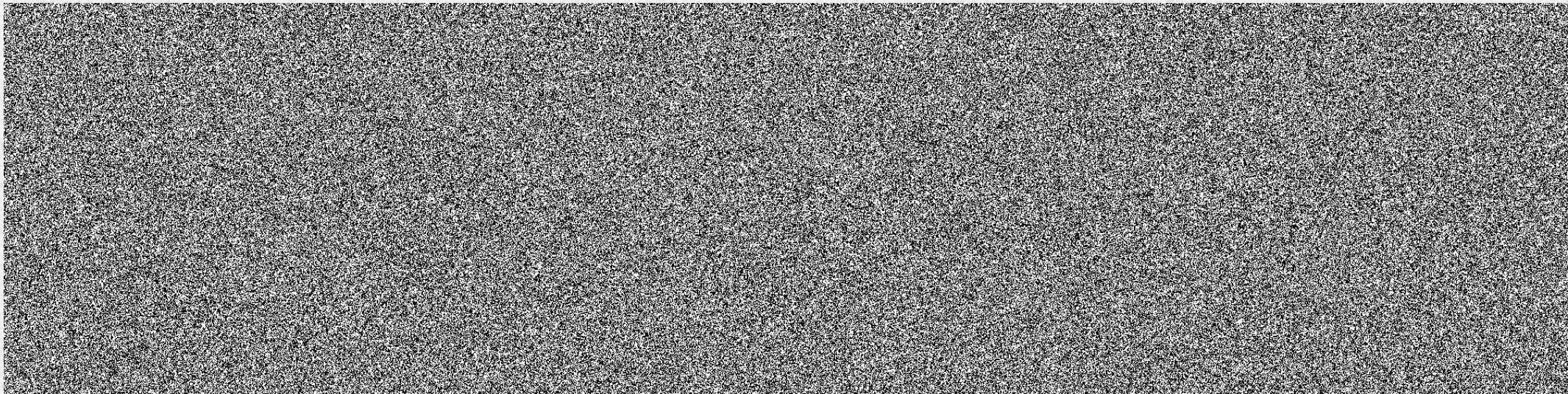
- Suppose your algorithm that can compress all 1000-bit strings.
- $2^{1000}$  possible bitstrings with 1000 bits.
- Only  $1 + 2 + 4 + \dots + 2^{998} + 2^{999}$  can be encoded with  $\leq 999$  bits.
- Similarly, only 1 in  $2^{499}$  bitstrings can be encoded with  $\leq 500$  bits!





# Undecidability

```
% java RandomBits | java PictureDump 2000 500
```



1000000 bits

A difficult file to compress: one million (pseudo-) random bits

```
public class RandomBits
{
    public static void main(String[] args)
    {
        int x = 11111;
        for (int i = 0; i < 1000000; i++)
        {
            x = x * 314159 + 218281;
            BinaryStdOut.write(x > 0);
        }
        BinaryStdOut.close();
    }
}
```

Q. How much redundancy is in the English language?

*“ ... randomising letters in the middle of words [has] little or no effect on the ability of skilled readers to understand the text. This is easy to demonstrate. In a publication of New Scientist you could randomise all the letters, keeping the first two and last two the same, and readability would hardly be affected. My analysis did not come to much because the theory at the time was for shape and sentence recognition. Saberi's work suggests we may have some powerful parallel processors at work. The reason for this is surely that identifying content by parallel processing speeds up recognition. We only need the first and last two letters to spot changes in meaning. ” — Graham Rawlinson*

A. Quite a bit.

- ▶ basics
- ▶ **run-length coding**
- ▶ Huffman compression
- ▶ LZW compression

# Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

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

Representation. Use 4-bit counts to represent alternating runs of 0s and 1s:  
15 0s, then 7 1s, then 7 0s, then 11 1s.



Q. How many bits to store the counts?

A. We'll use 8.

Q. What to do when run length exceeds max count?

A. If longer than 255, intersperse runs of length 0.

Applications. JPEG, ITU-T T4 Group 3 Fax, ...



## Run-length encoding: Java implementation

```
public class RunLength
{
    private final static int R = 256;
```

```
    public static void compress()
    { /* see textbook */ }
```

```
    public static void expand()
```

```
{
```

```
    boolean b = false;
```

```
    while (!BinaryStdIn.isEmpty())
```

```
{
```

```
        char run = BinaryStdIn.readChar();
```

← read 8-bit count from standard input

```
        for (int i = 0; i < run; i++)
```

```
            BinaryStdOut.write(b);
```

← write 1 bit to standard output

```
            b = !b;
```

```
        }
```

```
        BinaryStdOut.close();
```

```
    }
```

```
}
```

## An application: compress a bitmap

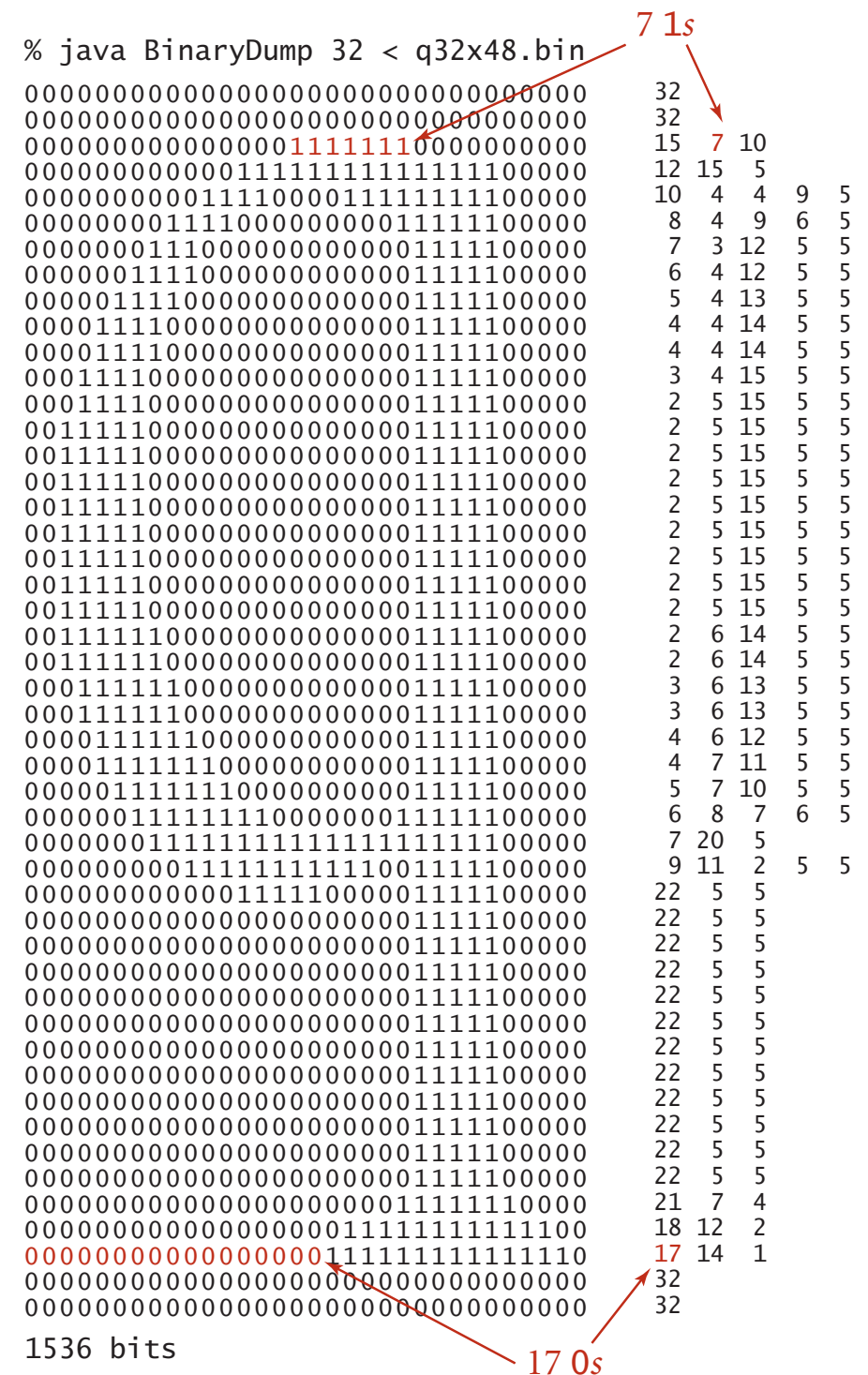
Typical black-and-white-scanned image.

- 300 pixels/inch.
- 8.5-by-11 inches.
- $300 \times 8.5 \times 300 \times 11 = 8.415$  million bits.

Observation. Bits are mostly white.

Typical amount of text on a page.

40 lines  $\times$  75 chars per line = 3,000 chars.



### A typical bitmap, with run lengths for each row

- ▶ basics
- ▶ run-length coding
- ▶ **Huffman compression**
- ▶ LZW compression

# Variable-length codes

Use different number of bits to encode different chars.

Ex. Morse code: • • • \_ \_ \_ • • •

Issue. Ambiguity.

SOS ?

IAMIE ?

EEWNI ?

V7 ?

In practice. Use a medium gap to separate codewords.

Letters		Numbers	
A	• —	1	• — — — —
B	— • • •	2	• • — — —
C	— • — •	3	• • • — —
D	— • •	4	• • • • —
E	•	5	• • • • •
F	• • — •	6	— • • • •
G	— — •	7	— — • • •
H	• • • •	8	— — — • •
I	• •	9	— — — — •
J	• — — —	0	— — — — —
K	— • —		
L	• — • •		
M	— —		
N	— •		
O	— — —		
P	• — — •		
Q	— — • —		
R	• — •		
S	• • •		
T	—		
U	• • —		
V	• • • —		
W	• — —		
X	— • • —		
Y	— • — —		
Z	— — • •		

codeword for S is a prefix  
of codeword for V

# Variable-length codes

Q. How do we avoid ambiguity?

A. Ensure that no codeword is a **prefix** of another.

Ex 1. Fixed-length code.

Ex 2. Append special stop char to each codeword.

Ex 3. General prefix-free code.

## Codeword table

key	value
!	101
A	0
B	1111
C	110
D	100
R	1110

## Compressed bitstring

011111110011001000111111100101 ← 30 bits  
A B RA CA DA B RA !

## Codeword table

key	value
!	101
A	11
B	00
C	010
D	100
R	011

## Compressed bitstring

11000111101011100110001111101 ← 29 bits  
A B R A C A D A B R A !

# Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

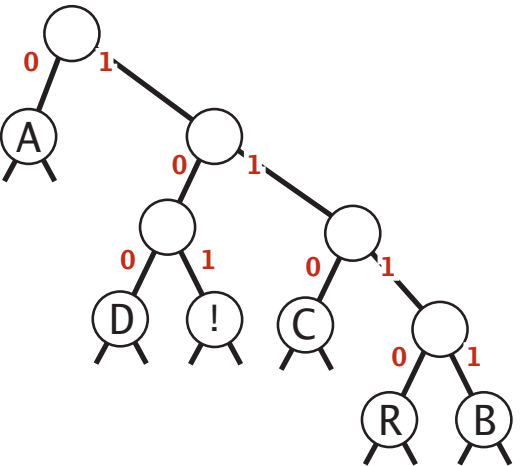
A. A binary trie!

- Chars in leaves.
- Codeword is path from root to leaf.

Codeword table

key	value
!	101
A	0
B	1111
C	110
D	100
R	1110

Trie representation



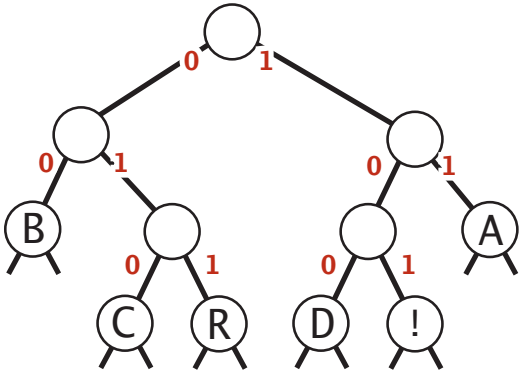
Compressed bitstring

011111110011001000111111100101 ← 30 bits  
A B RA CA DA B RA !

Codeword table

key	value
!	101
A	11
B	00
C	010
D	100
R	011

Trie representation



Compressed bitstring

11000111101011100110001111101 ← 29 bits  
A B R A C A D A B R A !

# Prefix-free codes: compression and expansion

## Compression.

- Method 1: start at leaf; follow path up to the root; print bits in reverse.
- Method 2: create ST of key-value pairs.

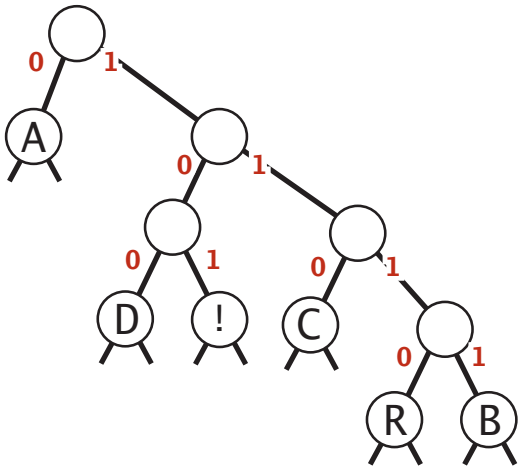
## Expansion.

- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, print char and return to root.

Codeword table

key	value
!	101
A	0
B	1111
C	110
D	100
R	1110

Trie representation



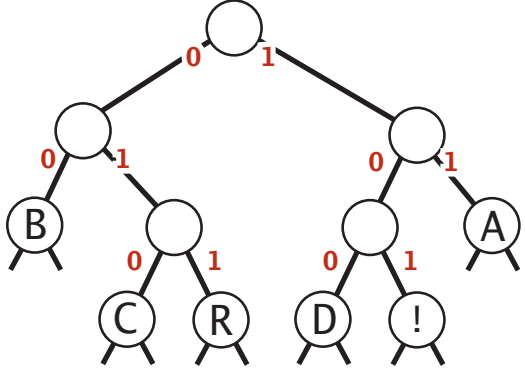
Compressed bitstring

011111110011001000111111100101 ← 30 bits  
A B RA CA DA B RA !

Codeword table

key	value
!	101
A	11
B	00
C	010
D	100
R	011

Trie representation



Compressed bitstring

11000111101011100110001111101 ← 29 bits  
A B R A C A D A B R A !

# Huffman trie node data type

```
private static class Node implements Comparable<Node>
{
    private char ch;    // Unused for internal nodes.
    private int freq;   // Unused for expand.
    private final Node left, right;
```

```
    public Node(char ch, int freq, Node left, Node right)
    {
        this.ch      = ch;
        this.freq    = freq;
        this.left    = left;
        this.right   = right;
    }
```

```
    public boolean isLeaf()
    { return left == null && right == null; }
```

```
    public int compareTo(Node that)
    { return this.freq - that.freq; }
```

```
}
```

← initializing constructor

← is Node a leaf?

← compare Nodes by frequency  
(stay tuned)



## Prefix-free codes: expansion

```
public void expand()
{
```

```
    Node root = readTrie();
```

```
    int N = BinaryStdIn.readInt();
```

```
    for (int i = 0; i < N; i++)
    {
```

```
        Node x = root;
```

```
        while (!x.isLeaf())
```

```
        {
```

```
            if (!BinaryStdIn.readBoolean())
```

```
                x = x.left;
```

```
            else
```

```
                x = x.right;
```

```
        }
```

```
        BinaryStdOut.write(x.ch);
```

```
    }
```

```
    BinaryStdOut.close();
```

```
}
```

← read in encoding trie

← read in number of chars

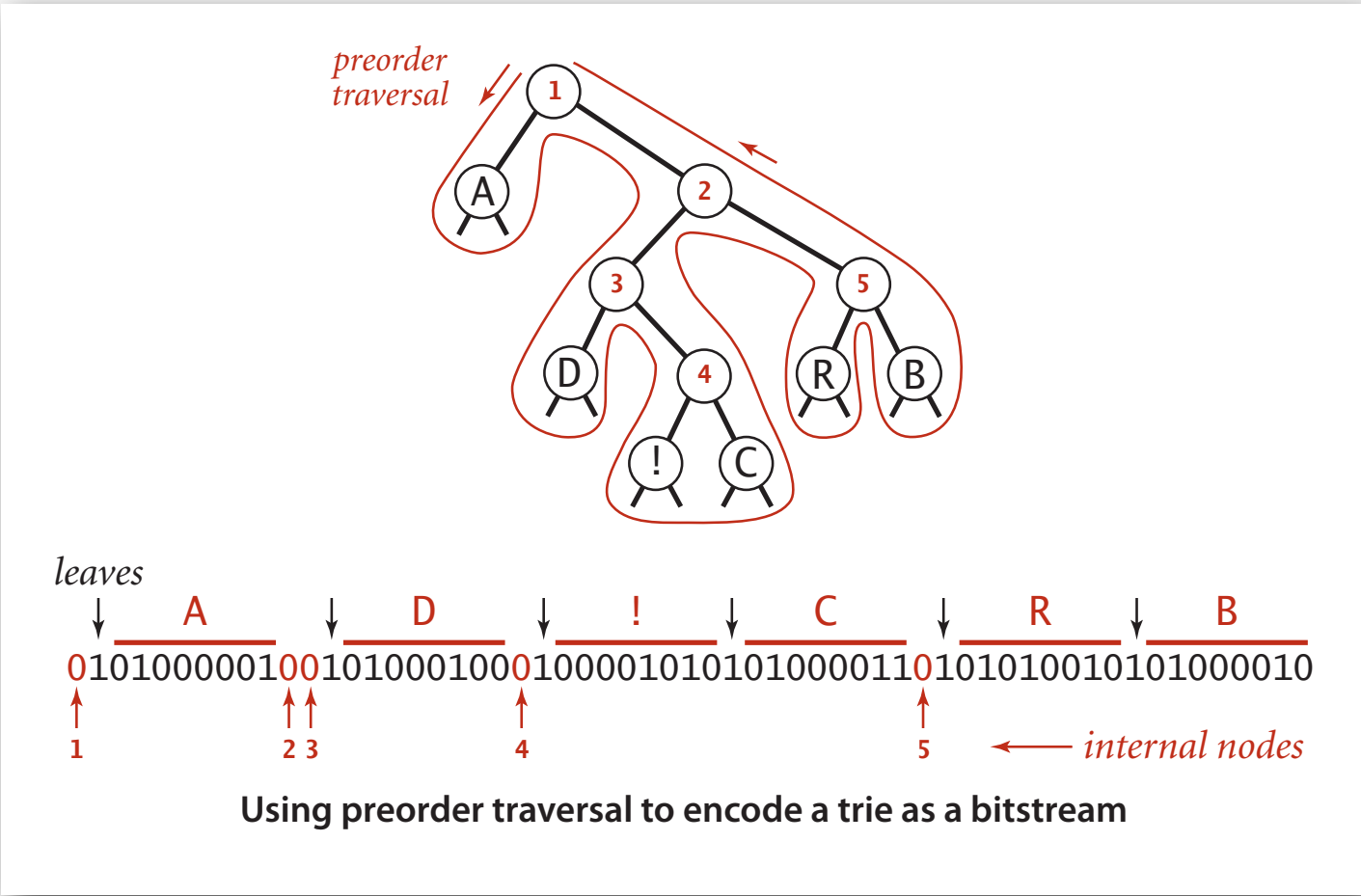
← expand codeword for i<sup>th</sup> char

**Running time.** Linear in input size (constant amount of work per bit read).

# Prefix-free codes: how to transmit

Q. How to write the trie?

A. Write preorder traversal of trie; mark leaf and internal nodes with a bit.

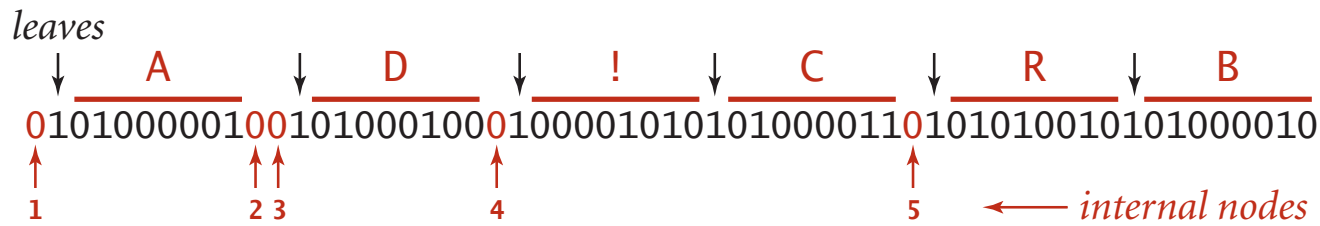
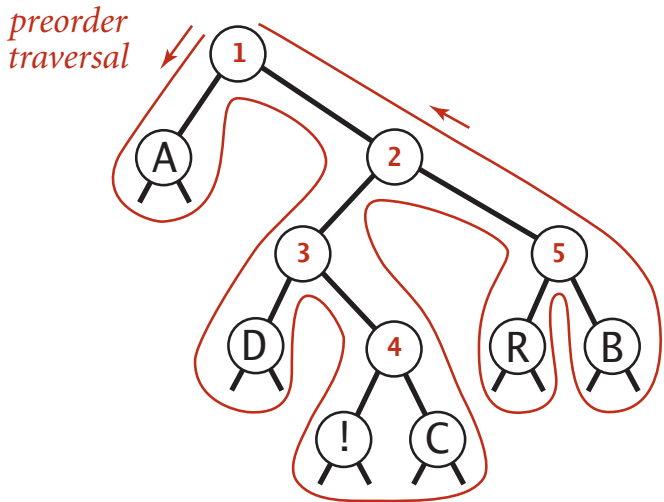


Note. If message is long, overhead of transmitting trie is small.

# Prefix-free codes: how to transmit

Q. How to write the trie?

A. Write preorder traversal of trie; mark leaf and internal nodes with a bit.



Using preorder traversal to encode a trie as a bitstream

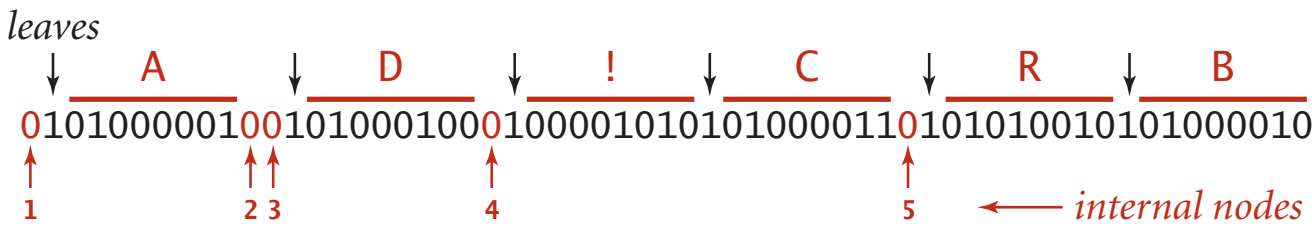
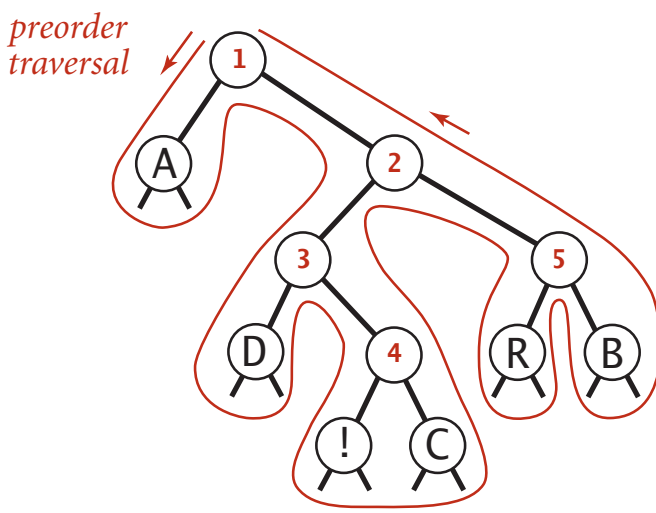
```
private static void writeTrie(Node x)
{
    if (x.isLeaf())
    {
        BinaryStdOut.write(true);
        BinaryStdOut.write(x.ch);
        return;
    }
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```

Note. If message is long, overhead of transmitting trie is small.

# Prefix-free codes: how to transmit

Q. How to read in the trie?

A. Reconstruct from preorder traversal of trie.



Using preorder traversal to encode a trie as a bitstream

```
private static Node readTrie()
{
    if (BinaryStdIn.readBoolean())
    {
        char c = BinaryStdIn.readChar();
        return new Node(c, 0, null, null);
    }
    Node x = readTrie();
    Node y = readTrie();
    return new Node('\0', 0, x, y);
}
```

not used

Q. How to find best prefix-free code?

Shannon-Fano algorithm:

- Partition symbols  $S$  into two subsets  $S_0$  and  $S_1$  of (roughly) equal frequency.
- Codewords for symbols in  $S_0$  start with 0; for symbols in  $S_1$  start with 1.
- Recur in  $S_0$  and  $S_1$ .

Problem 1. How to divide up symbols?

Problem 2. Not optimal!

# Huffman codes

Q. How to find best prefix-free code?



David Huffman

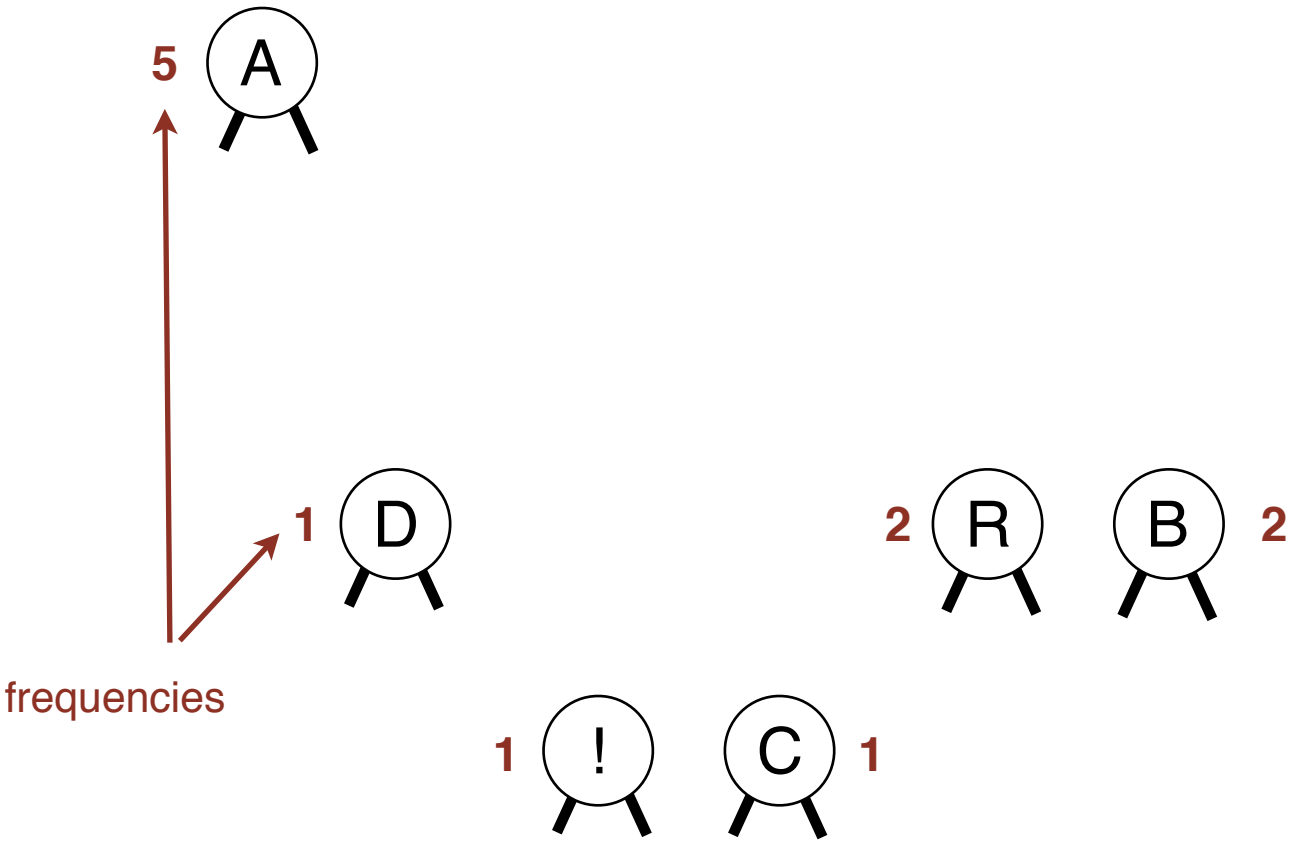
## Huffman algorithm:

- Count frequency  $\text{freq}[i]$  for each char  $i$  in input.
- Start with one node corresponding to each char  $i$  (with weight  $\text{freq}[i]$ ).
- Repeat until single trie formed:
  - select two tries with min weight  $\text{freq}[i]$  and  $\text{freq}[j]$
  - merge into single trie with weight  $\text{freq}[i] + \text{freq}[j]$

Applications. JPEG, MP3, MPEG, PKZIP, GZIP, PDF, ...

# Constructing a Huffman encoding trie

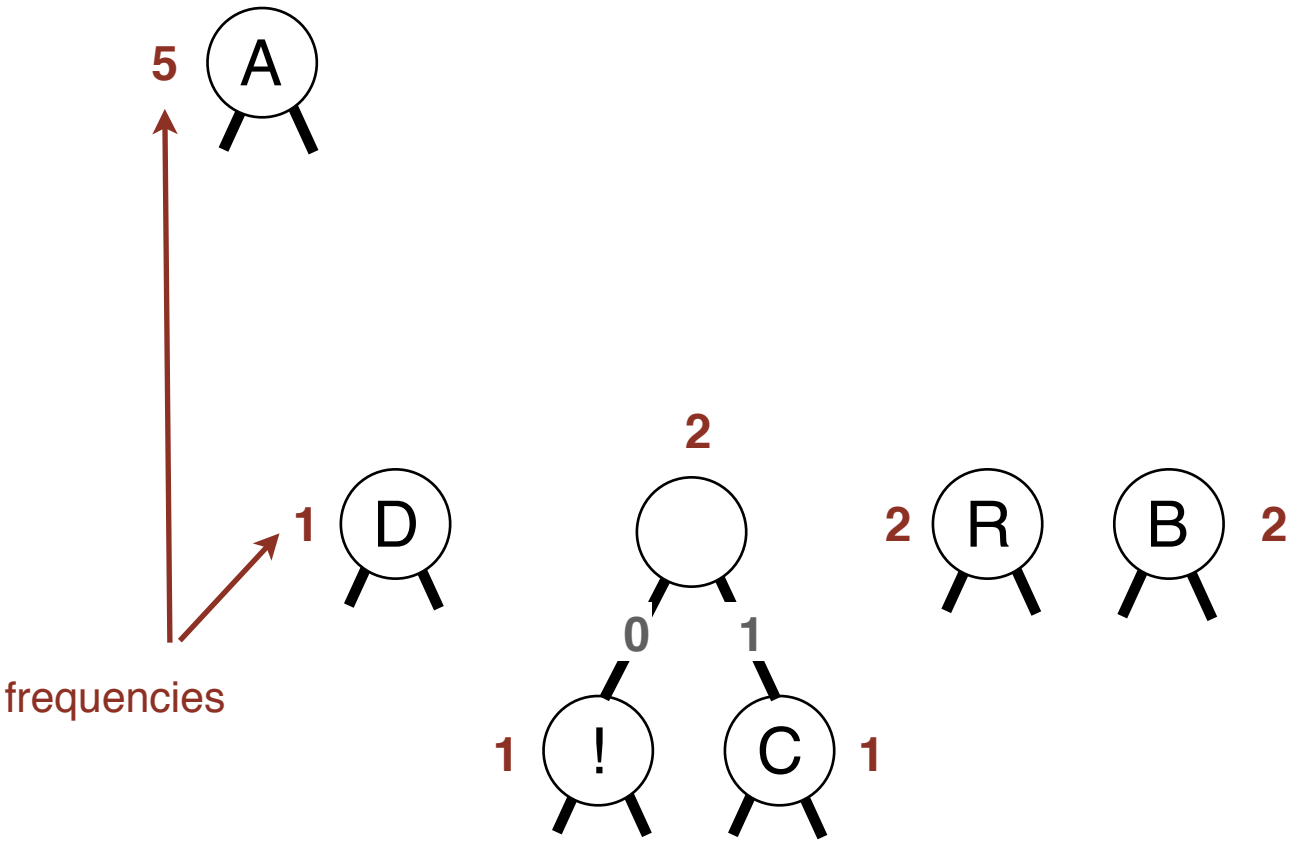
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman code construction for A B R A C A D A B R A !

# Constructing a Huffman encoding trie

char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0

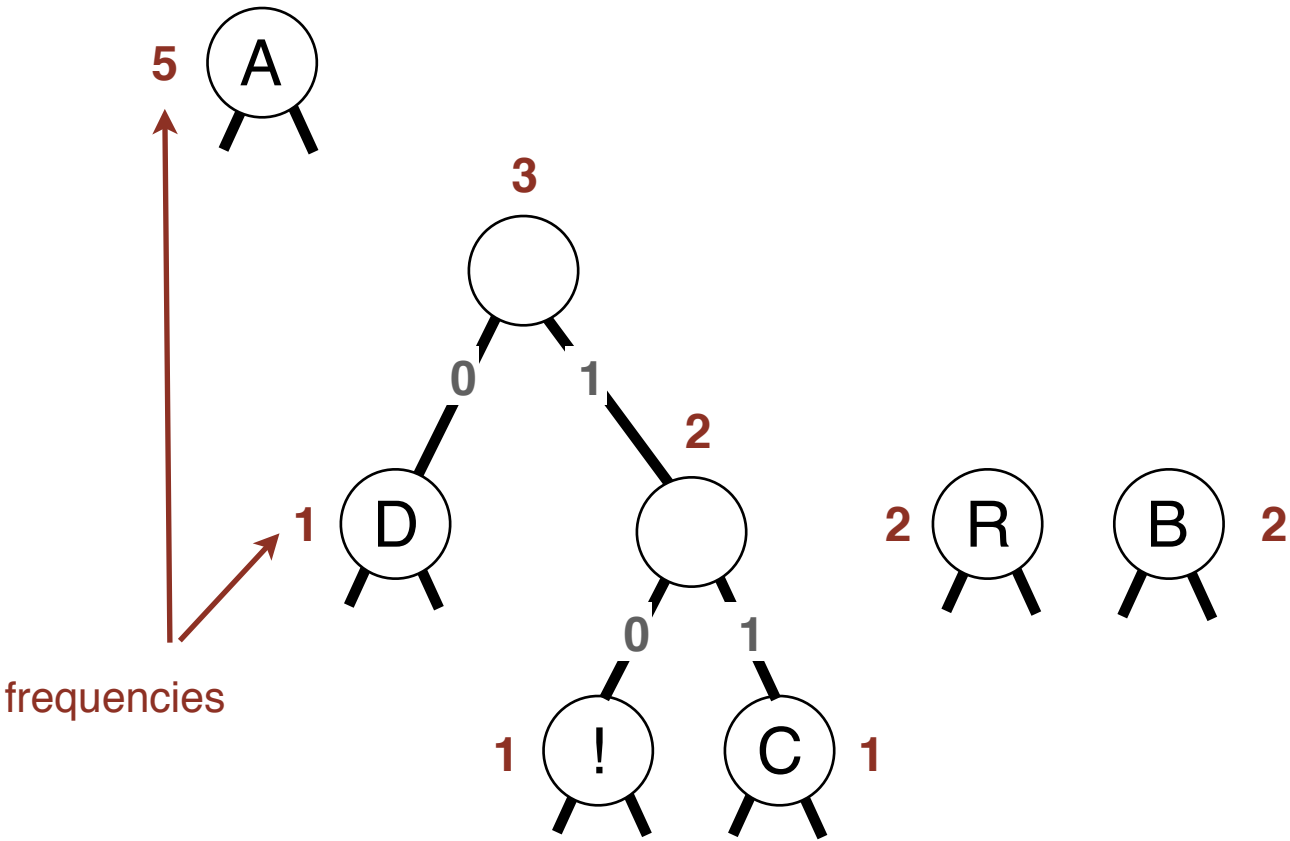


Huffman code construction for A B R A C A D A B R A !



# Constructing a Huffman encoding trie

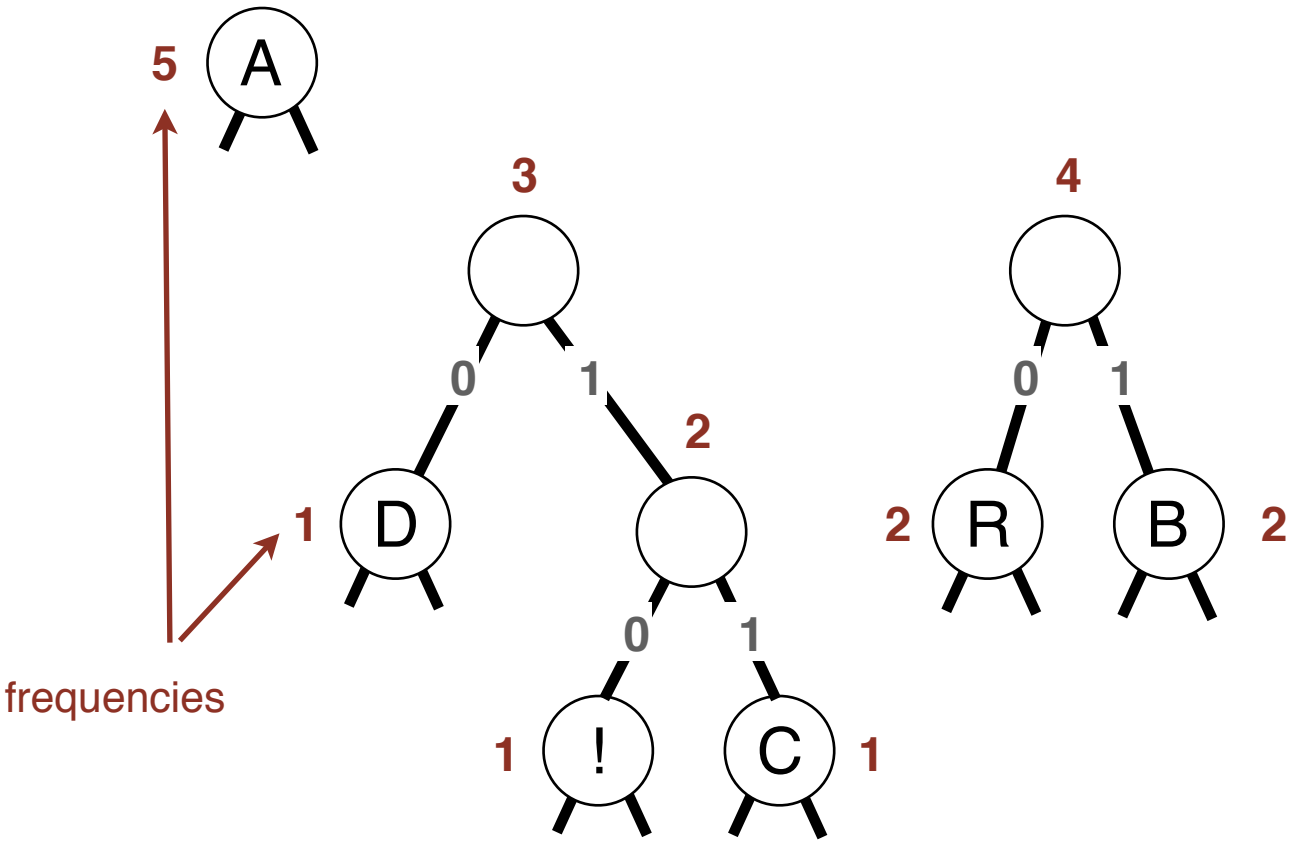
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman code construction for A B R A C A D A B R A !

# Constructing a Huffman encoding trie

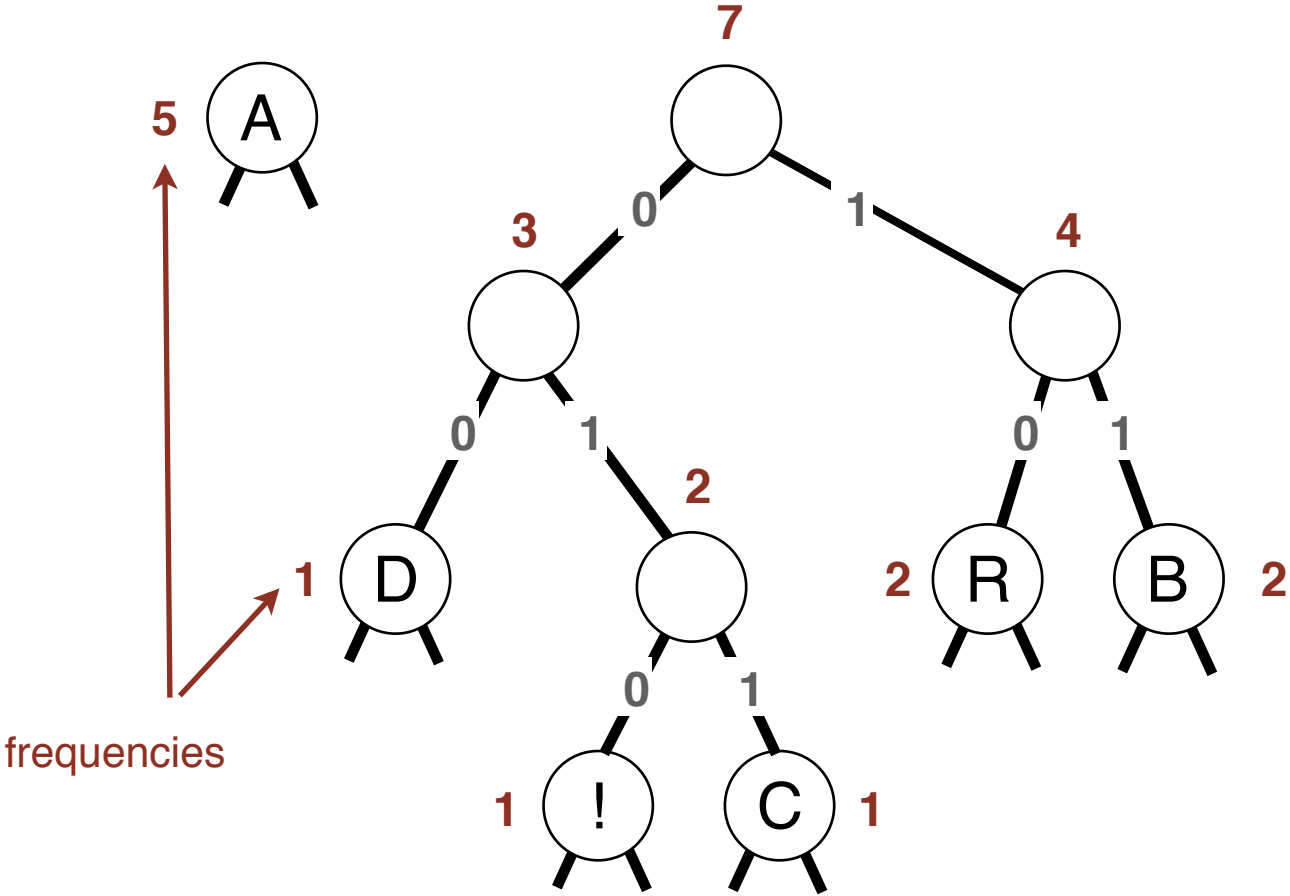
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman code construction for A B R A C A D A B R A !

# Constructing a Huffman encoding trie

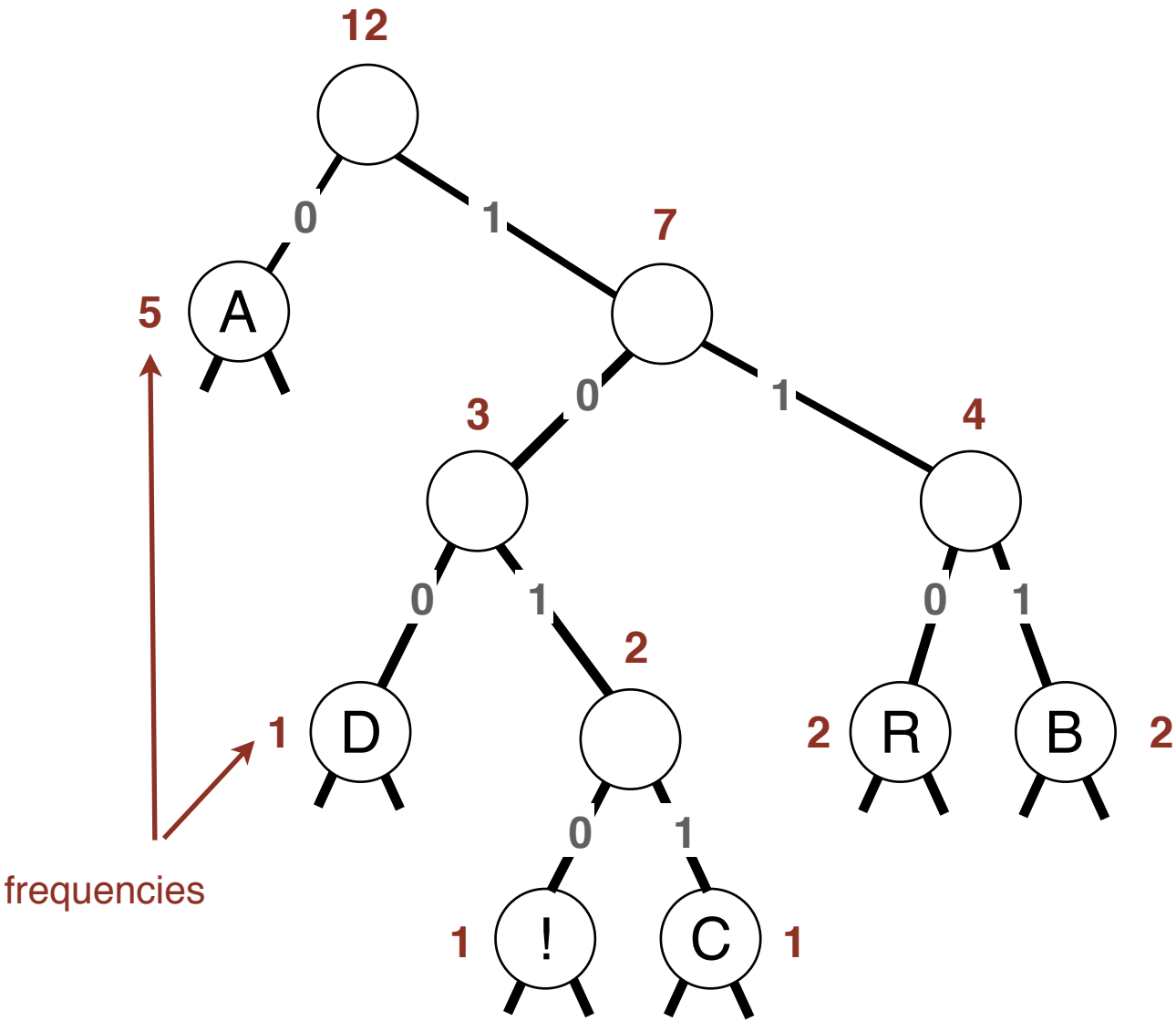
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman code construction for A B R A C A D A B R A !

# Constructing a Huffman encoding trie

char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



Huffman code construction for A B R A C A D A B R A !

# Constructing a Huffman encoding trie: Java implementation

```
private static Node buildTrie(int[] freq)
{
```

```
    MinPQ<Node> pq = new MinPQ<Node>();
```

```
    for (char i = 0; i < R; i++)
```

```
        if (freq[i] > 0)
```

```
            pq.insert(new Node(i, freq[i], null, null));
```

```
    while (pq.size() > 1)
```

```
    {
```

```
        Node x = pq.delMin();
```

```
        Node y = pq.delMin();
```

```
        Node parent = new Node('\0', x.freq + y.freq, x, y);
```

```
        pq.insert(parent);
```

```
    }
```

```
    return pq.delMin();
```

```
}
```

← initialize PQ with  
singleton tries

← merge two  
smallest tries

↑  
not used

↑  
total frequency

↑ ↑  
two subtrees

# Huffman encoding summary

**Proposition.** [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code.

**Pf.** See textbook.

no prefix-free code uses fewer bits

**Implementation.**

- Pass 1: tabulate char frequencies and build trie.
- Pass 2: encode file by traversing trie or lookup table.

**Running time.** Using a binary heap  $\Rightarrow O(N + R \log R)$ .

input  
size

alphabet  
size

**Q.** Can we do better? [stay tuned]

- ▶ basics
- ▶ run-length coding
- ▶ Huffman compression
- ▶ **LZW compression**



**Abraham Lempel**



**Jacob Ziv**

**Static model.** Same model for all texts.

- Fast.
- Not optimal: different texts have different statistical properties.
- Ex: ASCII, Morse code.

**Dynamic model.** Generate model based on text.

- Preliminary pass needed to generate model.
- Must transmit the model.
- Ex: Huffman code.

**Adaptive model.** Progressively learn and update model as you read text.

- More accurate modeling produces better compression.
- Decoding must start from beginning.
- Ex: LZW.



# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>																	
<i>value</i>																	

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
-----	-------

key	value
-----	-------

# Lempel-Ziv-Welch compression example

input

A

B

R

A

C

A

D

A

B

R

A

B

R

A

B

R

A

matches

A

value

41

## LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
-----	-------

key	value
-----	-------

# Lempel-Ziv-Welch compression example

input

A

B

R

A

C

A

D

A

B

R

A

B

R

A

B

R

A

matches

A

value

41

## LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B															
<i>value</i>	41	42															

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B															
<i>value</i>	41	42															

## LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R														
<i>value</i>	41	42	52														

## LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R														
<i>value</i>	41	42	52														

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A													
<i>value</i>	41	42	52	41													

## LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83

key	value
-----	-------



# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A													
<i>value</i>	41	42	52	41													

## LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83
AC	84

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C												
<i>value</i>	41	42	52	41	43												

## LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83
AC	84

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C												
<i>value</i>	41	42	52	41	43												

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83
AC	84
CA	85

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C	A											
<i>value</i>	41	42	52	41	43	41											

## LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83
AC	84
CA	85

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C	A											
<i>value</i>	41	42	52	41	43	41											

## LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C	A	D										
<i>value</i>	41	42	52	41	43	41	44										

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
-----	-------

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C	A	D										
<i>value</i>	41	42	52	41	43	41	44										

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
DA	87

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C	A	D	A	B								
<i>value</i>	41	42	52	41	43	41	44	81									

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
DA	87



# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C	A	D	A	B								
<i>value</i>	41	42	52	41	43	41	44	81									

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
DA	87
ABR	88

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C	A	D	A	B		R	A					
<i>value</i>	41	42	52	41	43	41	44	81			83						

## LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
DA	87
ABR	88

codeword table

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C	A	D	A	B		R	A					
<i>value</i>	41	42	52	41	43	41	44	81			83						

LZW compression for ABRACADABRABRABRA

key	value	key	value	key	value
...		AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84		
D	44	CA	85		
...		AD	86		

codeword table

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	<b>A</b>	<b>B</b>	<b>R</b>	<b>A</b>
<i>matches</i>	A	B	R	A	C	A	D	A	B		R	A		B	R		
<i>value</i>	<b>41</b>	<b>42</b>	<b>52</b>	<b>41</b>	<b>43</b>	<b>41</b>	<b>44</b>	<b>81</b>			<b>83</b>			<b>82</b>			

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
DA	87
ABR	88
RAB	89

codeword table

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	<b>A</b>	<b>B</b>	<b>R</b>	<b>A</b>
<i>matches</i>	A	B	R	A	C	A	D	A	B		R	A		B	R		
<i>value</i>	<b>41</b>	<b>42</b>	<b>52</b>	<b>41</b>	<b>43</b>	<b>41</b>	<b>44</b>	<b>81</b>			<b>83</b>			<b>82</b>			

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
DA	87
ABR	88
RAB	89
BRA	8A

codeword table

# Lempel-Ziv-Welch compression example

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	<b>A</b>		
matches	A	B	R	A	C	A	D	A	B		R	A		B	R		A	B	R
value	41	42	52	41	43	41	44	81			83			82			88		

LZW compression for ABRACADABRABRABRA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86

key	value
DA	87
ABR	88
RAB	89
BRA	8A

codeword table

# Lempel-Ziv-Welch compression example

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	<b>A</b>		
matches	A	B	R	A	C	A	D	A	B		R	A		B	R		A	B	R
value	41	42	52	41	43	41	44	81			83			82			88		

LZW compression for ABRACADABRABRABRA

key	value	key	value	key	value
...		AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84	BRA	8A
D	44	CA	85	ABRA	8B
...		AD	86		

codeword table

# Lempel-Ziv-Welch compression example

<i>input</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>matches</i>	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A
<i>value</i>	41	42	52	41	43	41	44	81		83		82		88			41

LZW compression for ABRACADABRABRABRA

key	value	key	value	key	value
...		AB	81	DA	87
A	41	BR	82	ABR	88
B	42	RA	83	RAB	89
C	43	AC	84	BRA	8A
D	44	CA	85	ABRA	8B
...		AD	86		

codeword table



# Lempel-Ziv-Welch compression

## LZW compression.

- Create ST associating  $W$ -bit codewords with string keys.
- Initialize ST with codewords for single-char keys.
- Find longest string  $s$  in ST that is a prefix of unscanned part of input.
- Write the  $W$ -bit codeword associated with  $s$ .
- Add  $s + c$  to ST, where  $c$  is next char in the input.

input	A	B	R	A	C	A	D	A	B	R	A	B	R	A	B	R	A	EOF
matches	A	B	R	A	C	A	D	A B		R A		B R		A B R		R	A	↓
output	41	42	52	41	43	41	44	81		83		82		88			41	80

input substring

AB 81

BR 82

RA 83

AC 84

CA 85

AD 86

DA 87

ABR 88

RAB 89

BRA 8A

ABRA 8B

LZW codeword

lookahead character

codeword table

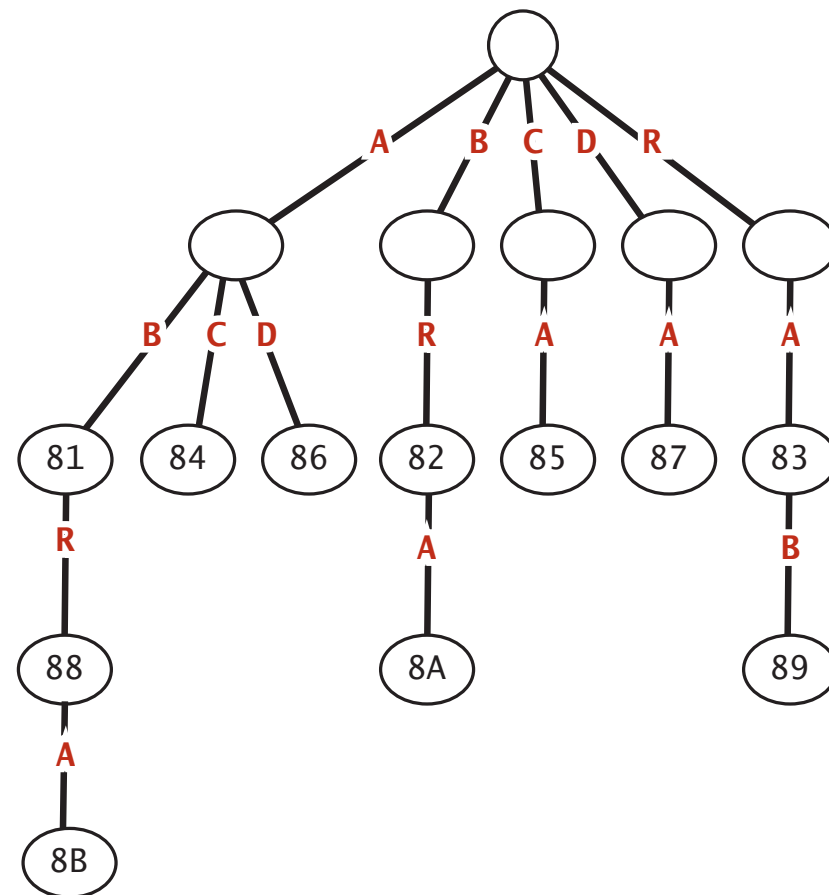
key	value
AB	81
BR	82
RA	83
AC	84
CA	85
AD	86
DA	87
ABR	88
RAB	89
BRA	8A
ABRA	8B

LZW compression for ABRACADABRABRABRA

# Representation of LZW code table

Q. How to represent LZW code table?

A. A trie: supports efficient longest prefix match.



Trie representation of LZW code table

Remark. Every prefix of a key in encoding table is also in encoding table.

# Lempel-Ziv-Welch expansion example

value

41425241434144818382884180

output

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
-------	-----

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41425241434144818382884180

output

A

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
-------	-----

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41

42

52

41

43

41

44

81

83

82

88

41

80

output

A

B

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
-------	-----

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41

42

52

41

43

41

44

81

83

82

88

41

80

output

A

B

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41

42

52

41

43

41

44

81

83

82

88

41

80

output

A

B

R

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41

42

52

41

43

41

44

81

83

82

88

41

80

output

A

B

R

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB
82	BR

value	key
-------	-----



# Lempel-Ziv-Welch expansion example

value

41425241434144818382884180

output

A B R A

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB
82	BR

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41425241434144818382884180

output

A B R A

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB
82	BR
83	RA

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41

42

52

41

43

41

44

81

83

82

88

41

80

output

A

B

R

A

C

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB
82	BR
83	RA

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41

42

52

41

43

41

44

81

83

82

88

41

80

output

A

B

R

A

C

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB
82	BR
83	RA
84	AC

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41425241434144818382884180

output

A B R A C A

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB
82	BR
83	RA
84	AC

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41425241434144818382884180

output

A B R A C A

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB
82	BR
83	RA
84	AC
85	CA

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

value

41425241434144818382884180

output

A B R A C A D

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB
82	BR
83	RA
84	AC
85	CA

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D						

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
81	AB
82	BR
83	RA
84	AC
85	CA
86	AD

value	key
-------	-----

codeword table



# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D	A B					

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

codeword table

value	key
81	AB
82	BR
83	RA
84	AC
85	CA
86	AD

value	key
-------	-----

# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D	A B					

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
81	AB
82	BR
83	RA
84	AC
85	CA
86	AD

value	key
87	DA

codeword table

# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D	A B	R A				

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key	value	key	value	key
...	...	81	AB	87	DA
41	A	82	BR		
42	B	83	RA		
43	C	84	AC		
44	D	85	CA		
...	...	86	AD		

codeword table

# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D	A B	R A				

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key	value	key	value	key
...	...	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA		
43	C	84	AC		
44	D	85	CA		
...	...	86	AD		

codeword table

# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D	A B	R A	B R			

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key	value	key	value	key
...	...	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA		
43	C	84	AC		
44	D	85	CA		
...	...	86	AD		

codeword table

# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D	A B	R A	B R			

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
81	AB
82	BR
83	RA
84	AC
85	CA
86	AD

value	key
87	DA
88	ABR
89	RAB

codeword table

# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D	A B	R A	B R	A B R		

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key	value	key	value	key
...	...	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA	89	RAB
43	C	84	AC		
44	D	85	CA		
...	...	86	AD		

codeword table

# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D	A B	R A	B R	A B R		

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key	value	key	value	key
...	...	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA	89	RAB
43	C	84	AC	8A	BRA
44	D	85	CA		
...	...	86	AD		

codeword table



# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D	A B	R A	B R	A B R	A	

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

value	key	value	key	value	key
...	...	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA	89	RAB
43	C	84	AC	8A	BRA
44	D	85	CA		
...	...	86	AD		

codeword table

# Lempel-Ziv-Welch expansion example

<i>value</i>	41	42	52	41	43	41	44	81	83	82	88	41	80
<i>output</i>	A	B	R	A	C	A	D	A B	R A	B R	A B R	A	

LZW expansion for 41 42 52 41 43 41 44 81 83 82 88 41 80

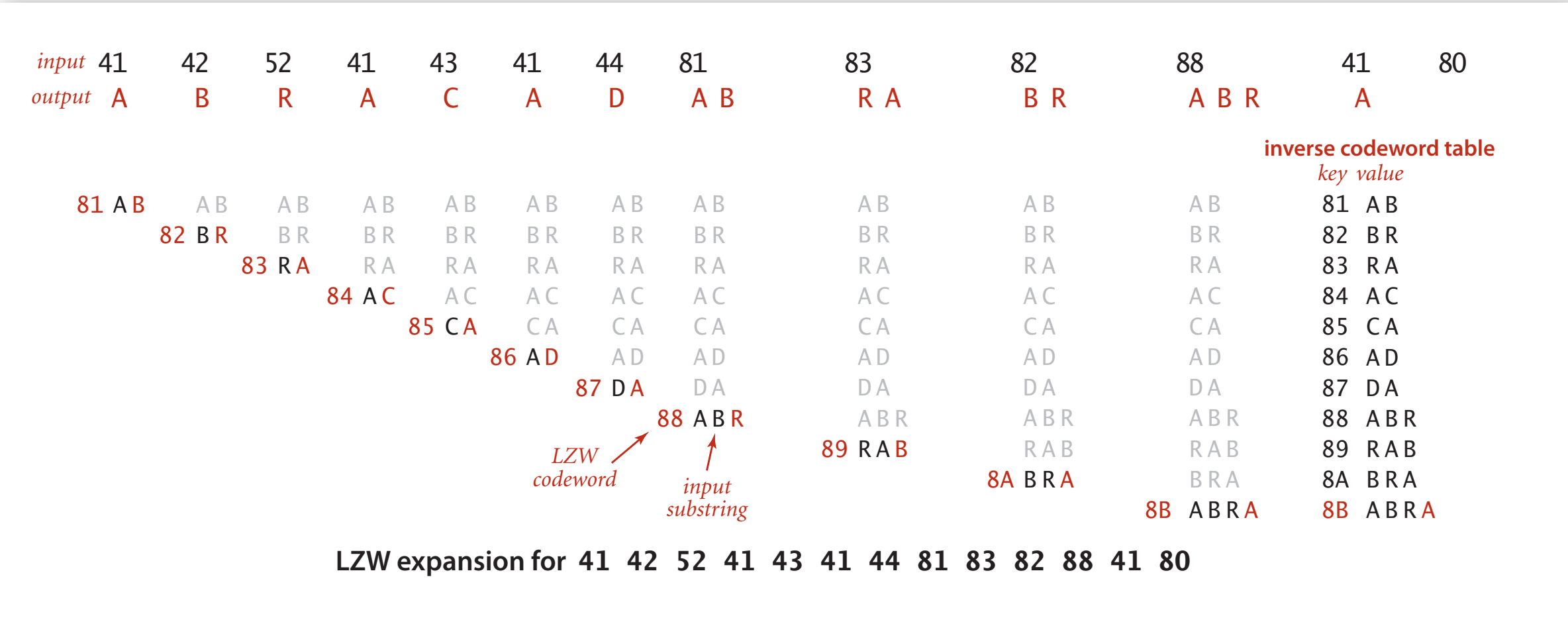
value	key	value	key	value	key
...	...	81	AB	87	DA
41	A	82	BR	88	ABR
42	B	83	RA	89	RAB
43	C	84	AC	8A	BRA
44	D	85	CA	8B	ABRA
...	...	86	AD		

codeword table

# LZW expansion

## LZW expansion.

- Create ST associating string values with  $W$ -bit keys.
- Initialize ST to contain single-char values.
- Read a  $W$ -bit key.
- Find associated string value in ST and write it out.
- Update ST.



# LZW example: tricky situation

input

A

B

A

B

A

B

A

matches

value

LZW compression for ABABABA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
-----	-------

# LZW example: tricky situation

*input*    **A**    **B**    **A**    **B**    **A**    **B**    **A**

*matches* **A**

*value*    **41**

LZW compression for ABABABA

key	value
...	
A	41
B	42
C	43
D	44
...	

codeword table

key	value
-----	-------

# LZW example: tricky situation

*input*    **A**    **B**    **A**    **B**    **A**    **B**    **A**

*matches* **A**

*value*    **41**

LZW compression for ABABABA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81

codeword table

# LZW example: tricky situation

input

A

B

A

B

A

B

A

matches

A

B

value

41

42

LZW compression for ABABABA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81

codeword table

# LZW example: tricky situation

input

A

B

A

B

A

B

A

matches

A

B

value

41

42

LZW compression for ABABABA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81
BA	82

codeword table



# LZW example: tricky situation

*input*

A

B

A

B

A

B

A

*matches*

A

B

A B

*value*

41

42

81

LZW compression for ABABABA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81
BA	82

codeword table

# LZW example: tricky situation

<i>input</i>	A	B	A	B	<b>A</b>	<b>B</b>	<b>A</b>
<i>matches</i>	A	B	A B				
<i>value</i>	<b>41</b>	<b>42</b>	<b>81</b>				

LZW compression for ABABABA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81
BA	82
ABA	83

codeword table

# LZW example: tricky situation

<i>input</i>	A	B	A	B	A	B	A
<i>matches</i>	A	B	A B		A B A		
<i>value</i>	41	42	81		83		

LZW compression for ABABABA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81
BA	82
ABA	83

codeword table

# LZW example: tricky situation

<i>input</i>	A	B	A	B	A	B	A
<i>matches</i>	A	B	A B		A B A		
<i>value</i>	41	42	81		83		80

LZW compression for ABABABA

key	value
...	
A	41
B	42
C	43
D	44
...	

key	value
AB	81
BA	82
ABA	83

codeword table

# LZW example: tricky situation

<i>value</i>	41	42	81	83	80
<i>output</i>					

LZW expansion for 41 42 81 83 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
-------	-----

codeword table

# LZW example: tricky situation

value

41

42

81

83

80

output

A

LZW expansion for 41 42 81 83 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
-------	-----

codeword table

# LZW example: tricky situation

value

41

42

81

83

80

output

A

B

LZW expansion for 41 42 81 83 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
-------	-----

codeword table

# LZW example: tricky situation

value

41

42

81

83

80

output

A

B

LZW expansion for 41 42 81 83 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
81	AB

codeword table



# LZW example: tricky situation

value

41

42

81

83

80

output

A

B

A

B

LZW expansion for 41 42 81 83 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
81	AB

codeword table

# LZW example: tricky situation

value

41

42

81

83

80

output

A

B

A

B

LZW expansion for 41 42 81 83 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
81	AB
82	BA

codeword table

# LZW example: tricky situation

<i>value</i>	41	42	81	83	80
<i>output</i>	A	B	A B		

need to know which  
key has value 83  
before it is in ST!

LZW expansion for 41 42 81 83 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
81	AB
82	BA

codeword table

# LZW example: tricky situation

<i>value</i>	41	42	81	83	80
<i>output</i>	A	B	A B	A B A	

need to know which  
key has value 83  
before it is in ST!

LZW expansion for 41 42 81 83 80

value	key
...	...
41	A
42	B
43	C
44	D
...	...

value	key
81	AB
82	BA
83	ABA

codeword table

# LZW in the real world

Lempel-Ziv and friends.

- LZ77.
- LZ78.
- LZW.
- Deflate = LZ77 variant + Huffman.

PNG: LZ77.


7zip, gzip, jar, pdf, java.util.zip: deflate.

Unix compress: LZW.

Pkzip: LZW + Shannon-Fano.

GIF, TIFF, V.42bis modem: LZW.

Google: zlib which is based on deflate.

 never expands a file

# Lossless data compression benchmarks

year	scheme	bits / char
1967	ASCII	7
1950	Huffman	4.7
1977	LZ77	3.94
1984	LZMW	3.32
1987	LZH	3.3
1987	move-to-front	3.24
1987	LZB	3.18
1987	gzip	2.71
1988	PPMC	2.48
1994	SAKDC	2.47
1994	PPM	2.34
1995	Burrows-Wheeler	2.29
1997	BOA	1.99
1999	RK	1.89

data compression using Calgary corpus

# Data compression summary

## Lossless compression.

- Represent fixed-length symbols with variable-length codes. [Huffman]
- Represent variable-length symbols with fixed-length codes. [LZW]

## Lossy compression. [not covered in this course]

- JPEG, MPEG, MP3, ...
- FFT, wavelets, fractals, ...

## Theoretical limits on compression. Shannon entropy.

## Practical compression. Use extra knowledge whenever possible.