https://algs4.cs.princeton.edu

5.5 DATA COMPRESSION

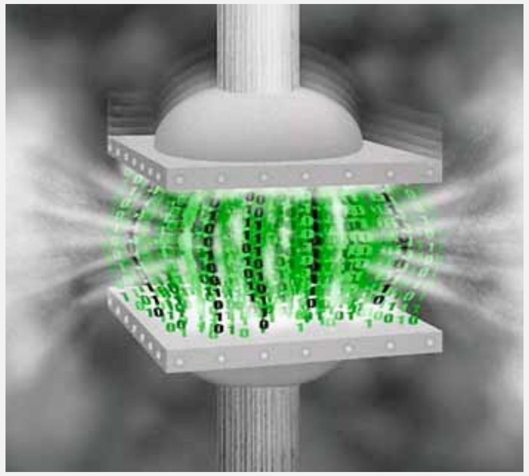
- *introduction*
- *run-length coding*
- *Huffman compression*
- *LZW compression*

https://algs4.cs.princeton.edu

5.5 Data Compression

- *introduction*
- *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, …

> " *Everyday, we create 2.5 quintillion bytes of data—so much that 90% of the data in the world today has been created in the last two years alone.* " *— IBM report on big data (2011)*

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

Applications

Generic file compression.

- Files: Gzip, bzip2, 7z.
- Archivers: PKZIP.
- File systems: NTFS, ZFS, HFS+, ReFS, GFS.

Multimedia.

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

Communication.

- ITU-T T4 Group 3 Fax.
- V.42bis modem.
- Skype, Google hangout.

Databases. Google, Facebook, NSA, ...

Lossless compression and expansion

Message. Bitstream 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. Compression ratio of about 25% can be achieved for natural language.

Compression via better data representation: genomic code

Genome. String over the alphabet `{ A, T, C, G }`.

Goal. Encode an n-character genome: `A T A G A T G C A T A G . . .`

Standard ASCII encoding.

- 8 bits per char.
- $8\ n$ bits.

char	hex	binary
'A'	41	01000001
'T'	54	01010100
'C'	43	01000011
'G'	47	01000111

Two-bit encoding.

- 2 bits per char.
- $2\ n$ bits (25% compression ratio).

char	binary
'A'	00
'T'	01
'C'	10
'G'	11

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

Reading and writing binary data

Binary standard input. Read bits from standard input.

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*

Binary standard output. Write bits to standard output

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*

Writing binary data

Date representation. Three different ways to represent 12/31/1999.

A character stream (StdOut)

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

```
00110001|00110010|00101111|00110111|00110001|00101111|00110001|00111001|00111001|00111001
   1         2        /        3        1        /        1        9        9        9
```

80 bits

Three ints (BinaryStdOut)

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

```
00000000|00000000|00000000|00001100|00000000|00000000|00000000|00011111|00000000|00000000|00000111|11001111
                  12                                  31                                 1999
```

96 bits

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);
```

```
11001111|10111110|01111000
 12   31     1999
```

21 bits (+ 3 bits for byte alignment at close)

Binary dumps

Q. How to examine the contents of a bitstream?

Standard character stream

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

Bitstream represented as 0 and 1 characters

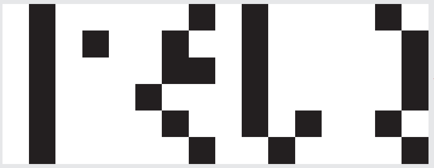
```
% java BinaryDump 16 < abra.txt
0100000101000010
0101001001000001
0100001101000001
0100010001000001
0100001001010010
0100000100100001
96 bits
```

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
```



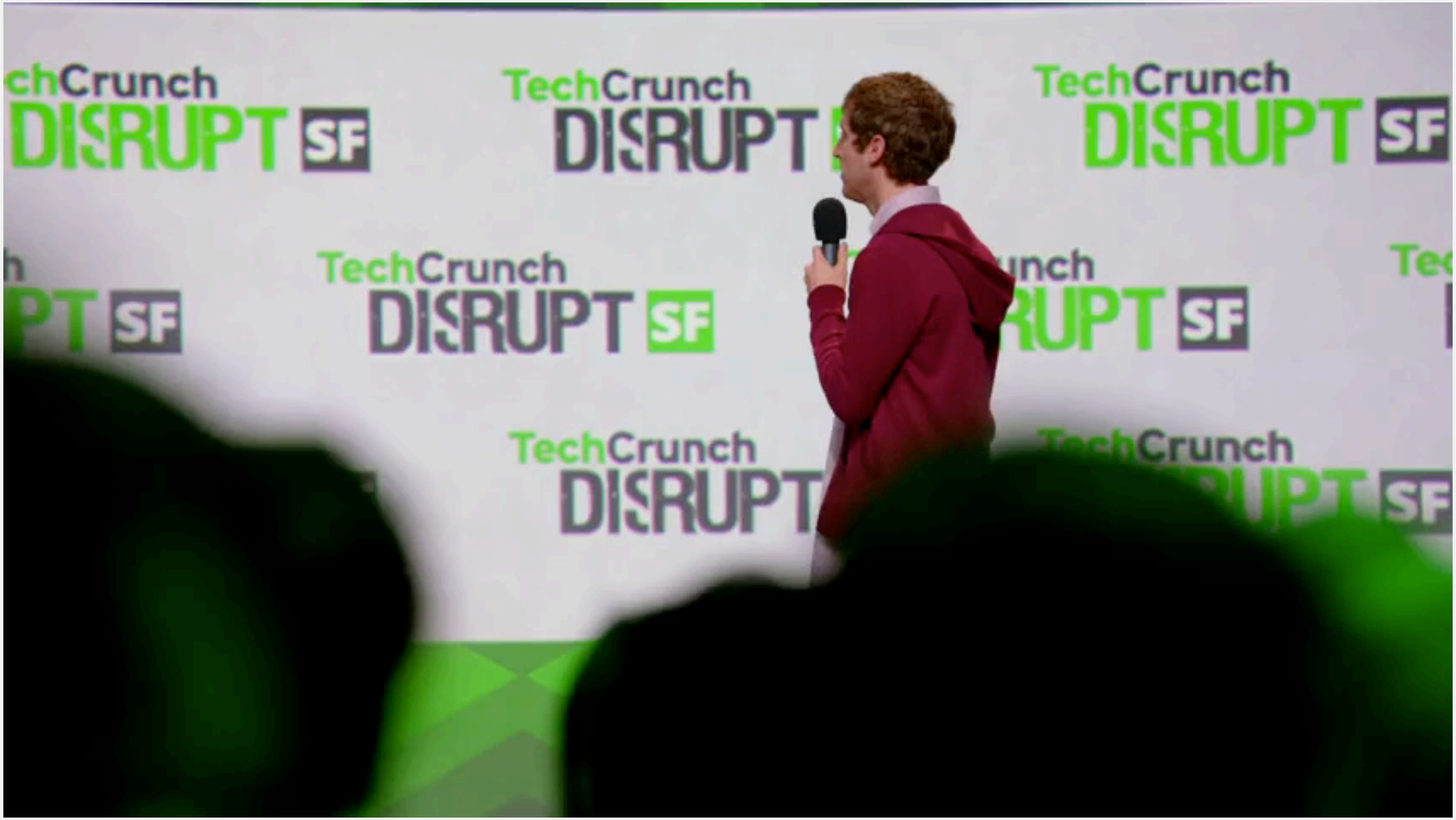

```
96 bits
```

	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?

Pied Piper. Claims 3.8:1 lossless compression of arbitrary data.



Universal data compression?

US Patent 5,533,051. Method which is capable of compressing all files.

US005533051A

United States Patent [19]
James

[11] **Patent Number: 5,533,051**
[45] **Date of Patent: Jul. 2, 1996**

[54] **METHOD FOR DATA COMPRESSION**

[75] Inventor: **David C. James**, Marco Island, Fla.

[73] Assignee: **The James Group**, Naples, Fla.

[21] Appl. No.: **30,741**

[22] Filed: **Mar. 12, 1993**

[51] **Int. Cl.**[6] **H04B 1/66**; H03M 7/40; H03M 7/30; H03M 7/34

[52] **U.S. Cl.** **375/240**; 341/67; 341/87; 341/51

[58] **Field of Search** 375/122; 341/67, 341/87, 51; 348/415, 409, 390, 384; 382/244; 364/715.02; 380/42, 49

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,694,813	9/1972	Loh et al.	340/172.5
4,369,463	1/1983	Anastassiou .	
4,491,934	1/1985	Heinz	364/900
4,545,032	10/1985	Mak	364/900
4,560,976	12/1985	Finn	341/51
4,597,057	6/1986	Snow	364/900
4,633,490	12/1986	Mitchell .	
4,652,856	2/1986	Mohiuddin .	
4,672,539	6/1987	Goertzel	364/300
4,725,884	2/1988	Gonzales .	
4,748,577	5/1988	Marchant	364/722
4,749,983	6/1988	Langdon .	
4,782,325	11/1988	Jeppsson	341/55
4,796,003	1/1989	Bentley	341/95
4,881,075	11/1989	Weng	341/87
4,905,297	2/1990	Langdon .	
4,906,991	3/1990	Fiala et al.	341/51
4,935,882	6/1990	Pennebaker .	
4,955,066	9/1990	Notenboom	382/56
4,973,961	11/1990	Chamzas .	
5,025,258	6/1991	Dutweiler .	
5,051,745	9/1991	Katz .	
5,325,091	6/1994	Kaplan et al.	341/51

OTHER PUBLICATIONS

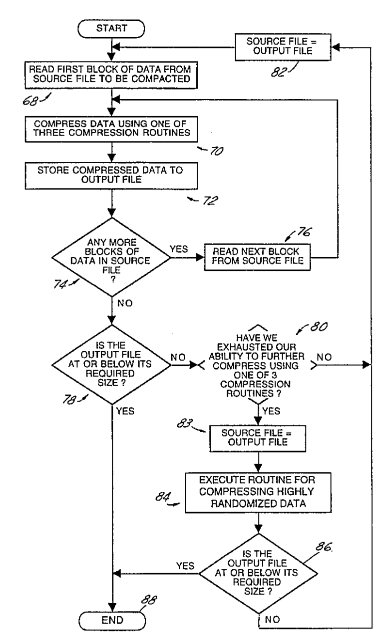
Oct. 1989 issue of Dr. Dob's Journal.

Primary Examiner—Scott A. Rogers
Assistant Examiner—Allan A. Esposo
Attorney, Agent, or Firm—Dykema Gossett

[57] **ABSTRACT**

Methods for compressing data including methods for compressing highly randomized data are disclosed. Nibble encode, distribution encode, and direct bit encode methods are disclosed for compressing data which is not highly randomized. A randomized data compression routine is also disclosed and is very effective for compressing data which is highly randomized. All of the compression methods disclosed operate on a bit level and accordingly are insensitive to the nature or origination of the data sought to be compressed. Accordingly, the methods of the present invention are universally applicable to any form of data regardless of its source of origination.

9 Claims, 31 Drawing Sheets



Universal data compression?

Proposition. No algorithm can compress every bitstring.

Proof. [by contradiction]

- Repeatedly compress the bitstring using the algorithm until it is 0 bits.

Alternative proof. [by counting]

- Suppose your algorithm that can compress all $1{,}000$-bit strings.
- 2^{1000} possible bitstrings with $1{,}000$ bits.
- Only $1 + 2 + 4 + \ldots + 2^{998} + 2^{999}$ can be encoded with ≤ 999 bits.

Corollary. If a compression algorithm shortens some bitstrings, it must expand other bitstrings.

Rdenudcany in Enlgsih lnagugae

Q. How much redundancy in the English language?

A. Quite a bit.

> " *... 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 denmtrasote. In a pubiltacion of New Scnieitst you could ramdinose all the letetrs, keipeng the first two and last two the same, and reibadailty would hadrly be aftcfeed. My ansaylis did not come to much beucase the thoery at the time was for shape and senqeuce retigcionon. Saberi's work sugsegts we may have some pofrweul palrlael prsooscers at work. The resaon for this is suerly that idnetiyfing coentnt by paarllel prseocsing speeds up regnicoiton. We only need the first and last two letetrs to spot chganes in meniang.* " — Graham Rawlinson

The gaol of data cmperisoson is to inetdify rdenudcany and epxloit it.

https://algs4.cs.princeton.edu

5.5 DATA COMPRESSION

- introduction
- run-length coding
- Huffman compression
- LZW compression

Run-length encoding (RLE)

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 1 ← 40 bits

(1 1 1 1 1 1 1: run of length 7)

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

1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 ← 16 bits (instead of 40)

(1 1 1 1 = 15; 0 1 1 1 = 7; 0 1 1 1 = 7; 1 0 1 1 = 11)

Q. How many bits to store the counts?

A. Typically 8 bits (but 4 on this slide for brevity).

Q. What if the input starts with a 1 rather than a 0?

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

A. Intersperse runs of length 0.

Run-length decoding: Java implementation

```
public class RunLength
{
   private static final int R   = 256;
   private static final int lgR = 8;

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

   public static void expand()
   {
      boolean bit = false;
      while (!BinaryStdIn.isEmpty())
      {
         int run = BinaryStdIn.readInt(lgR);
         for (int i = 0; i < run; i++)
            BinaryStdOut.write(bit);
         bit = !bit;
      }
      BinaryStdOut.close();
   }
}
```

- maximum run-length count
- number of bits per count
- initial runs are 0
- read 8-bit count from standard input
- write run of 0s or 1s to standard output
- flip bit (for next run)
- pad 0s for byte alignment

Data compression: quiz 1

What is the best compression ratio achievable from run-length encoding when using 8-bit counts?

A. 1 / 256

B. 1 / 16

C. 8 / 255

D. 1 / 8

E. 16 / 255

https://algs4.cs.princeton.edu

5.5 Data Compression

- introduction
- run-length coding
- **Huffman compression**
- LZW compression

Variable-length codes

Key idea. Use different number of bits to encode different characters.

Ex. Morse code: • • • — — — • • •

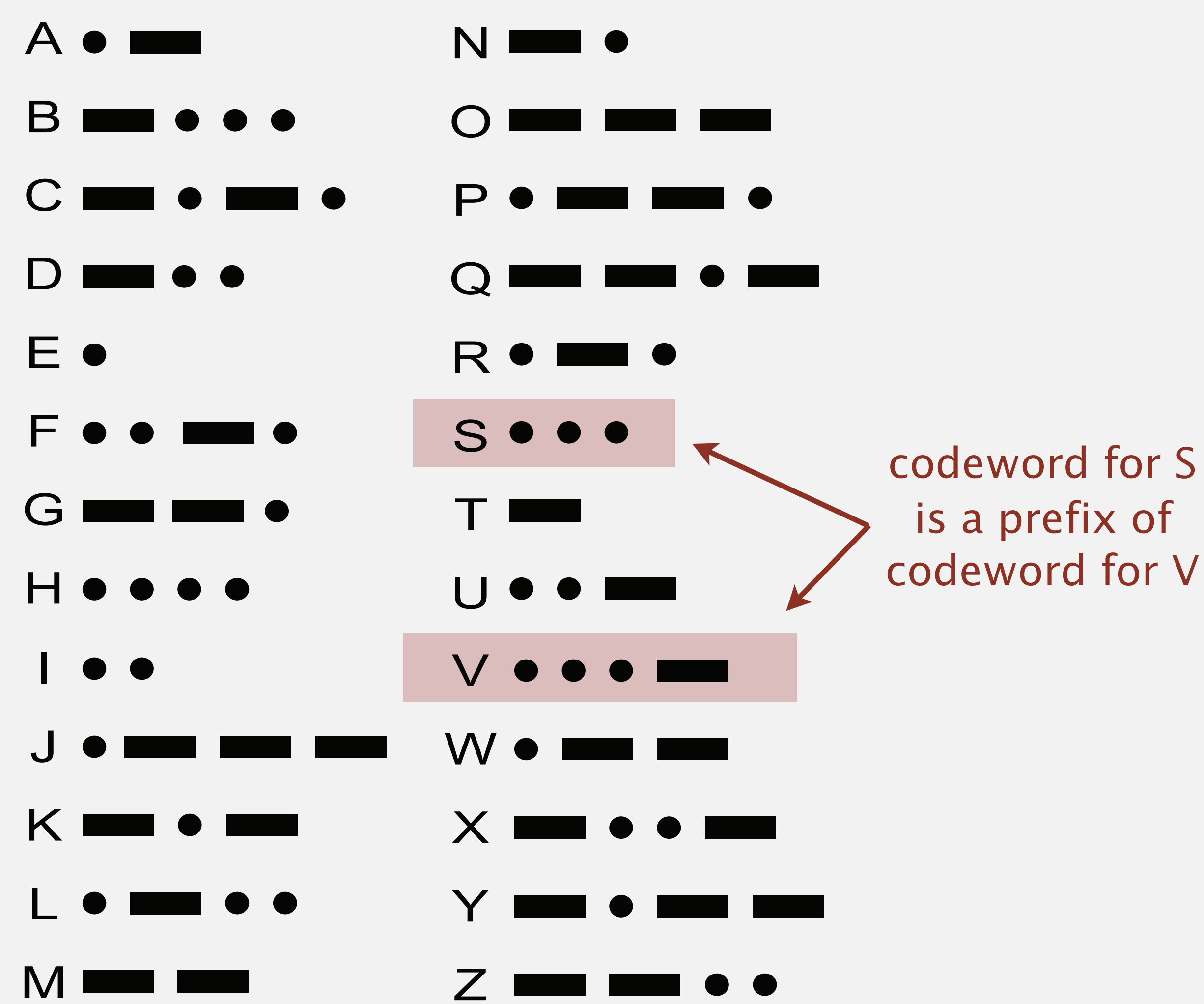
Issue. Ambiguity.

S O S ?

V Z E ?

E E J I E ?

E E W N I ?



In practice. Use a short gap to separate characters.

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" character 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*

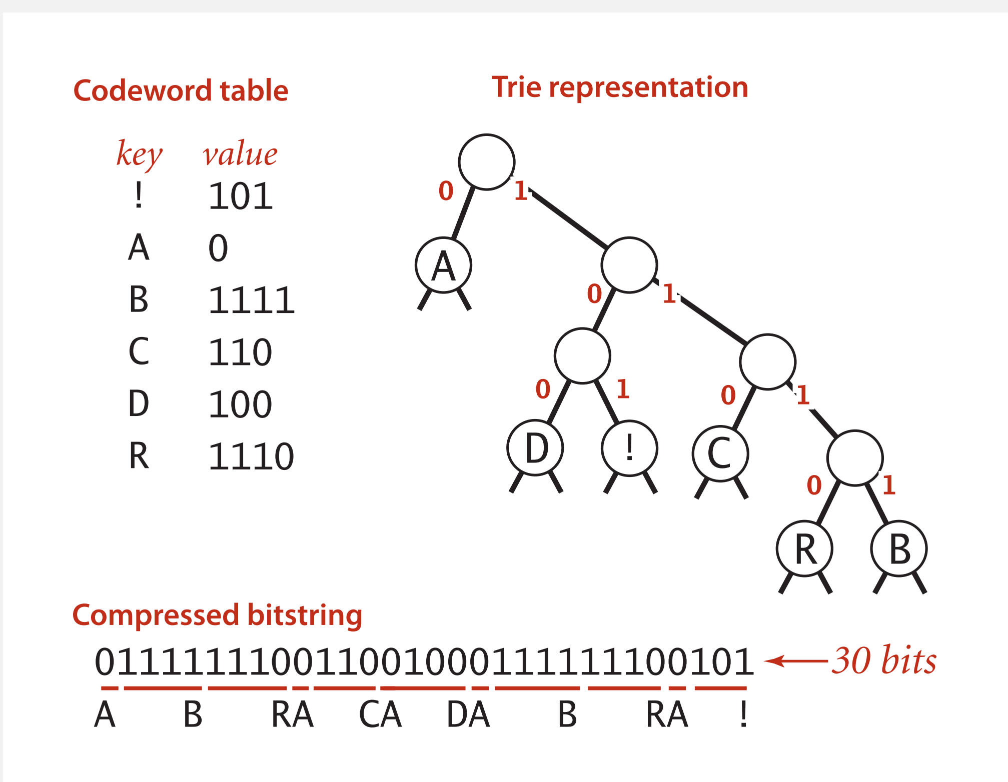
A B R A C A D A B R A !

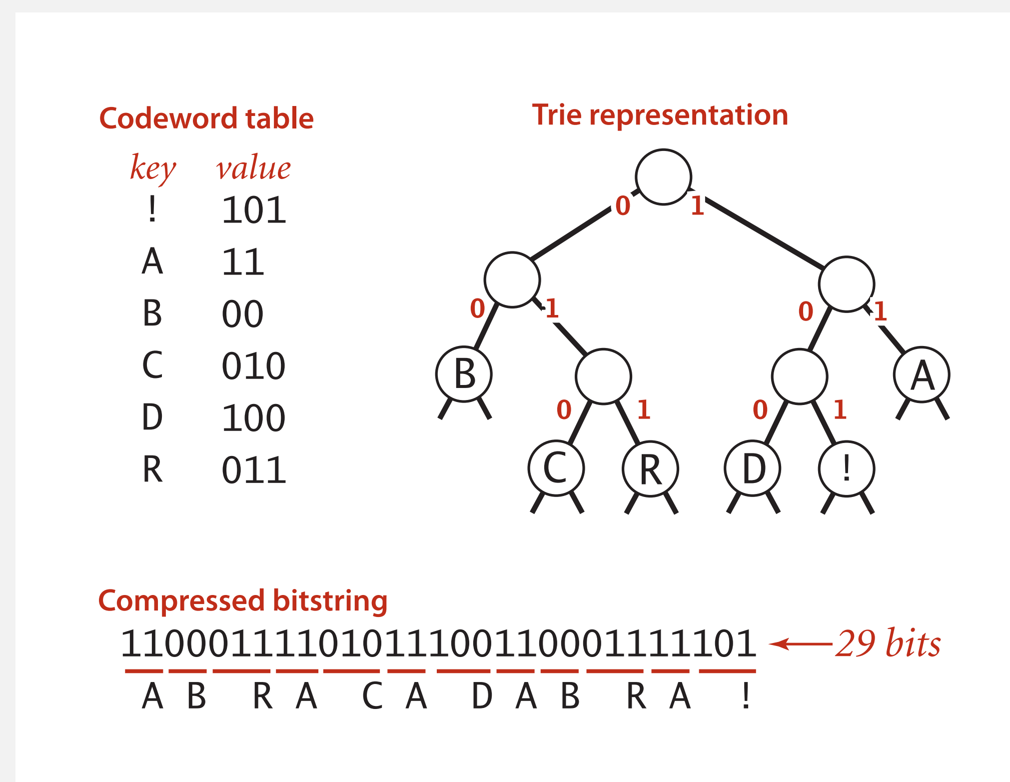
Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

A. A binary trie!

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



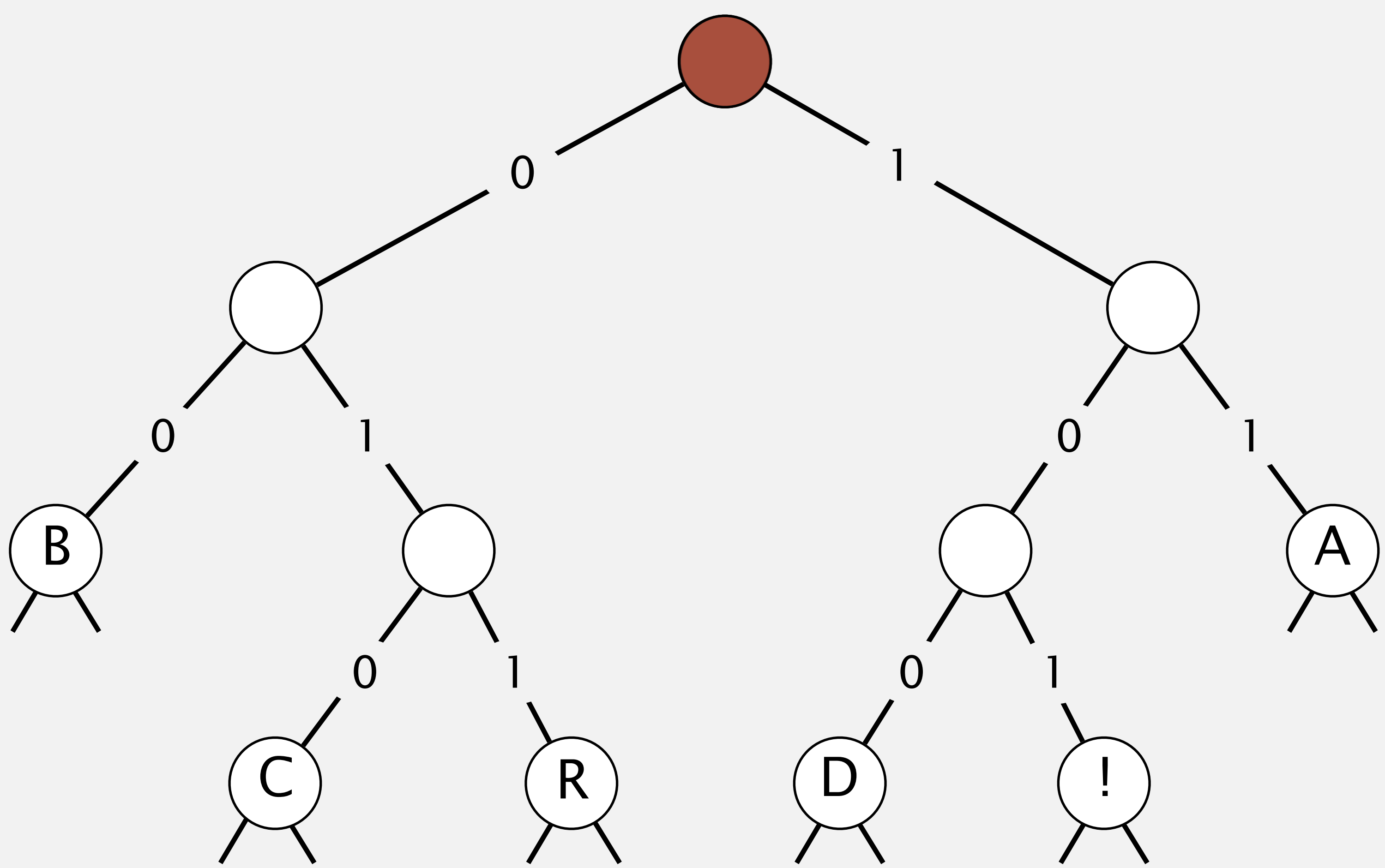


Prefix-free codes: expansion

Expansion.

- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, write character; return to root node; repeat.

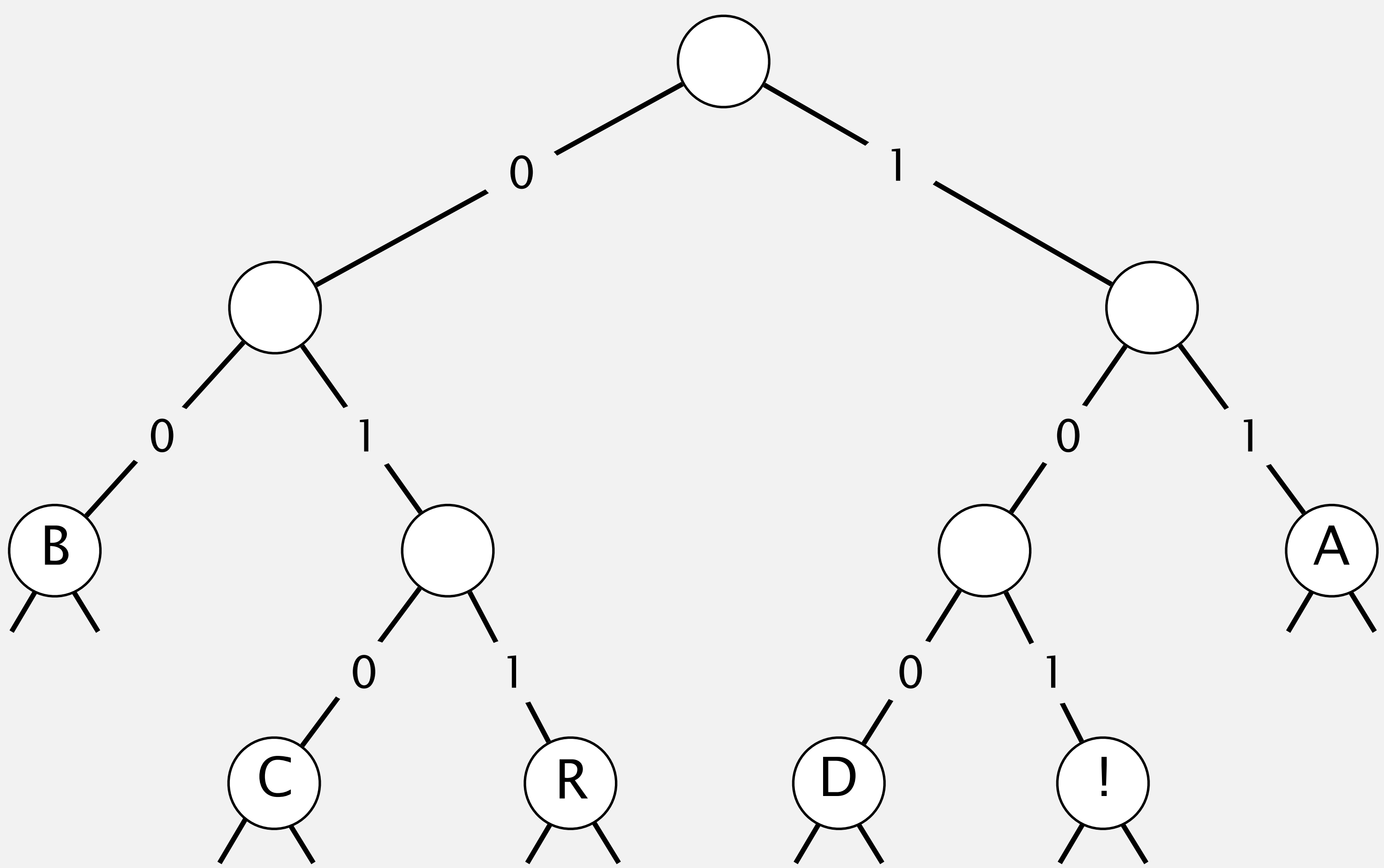
1 1	0 0	0 1 1	1 1	0 1 0	1 1	1 0 0	1 1	0 0	0 1 1	1 1	1 0 1
A	B	R	A	C	A	D	A	B	R	A	!



Prefix-free codes: compression

Compression.

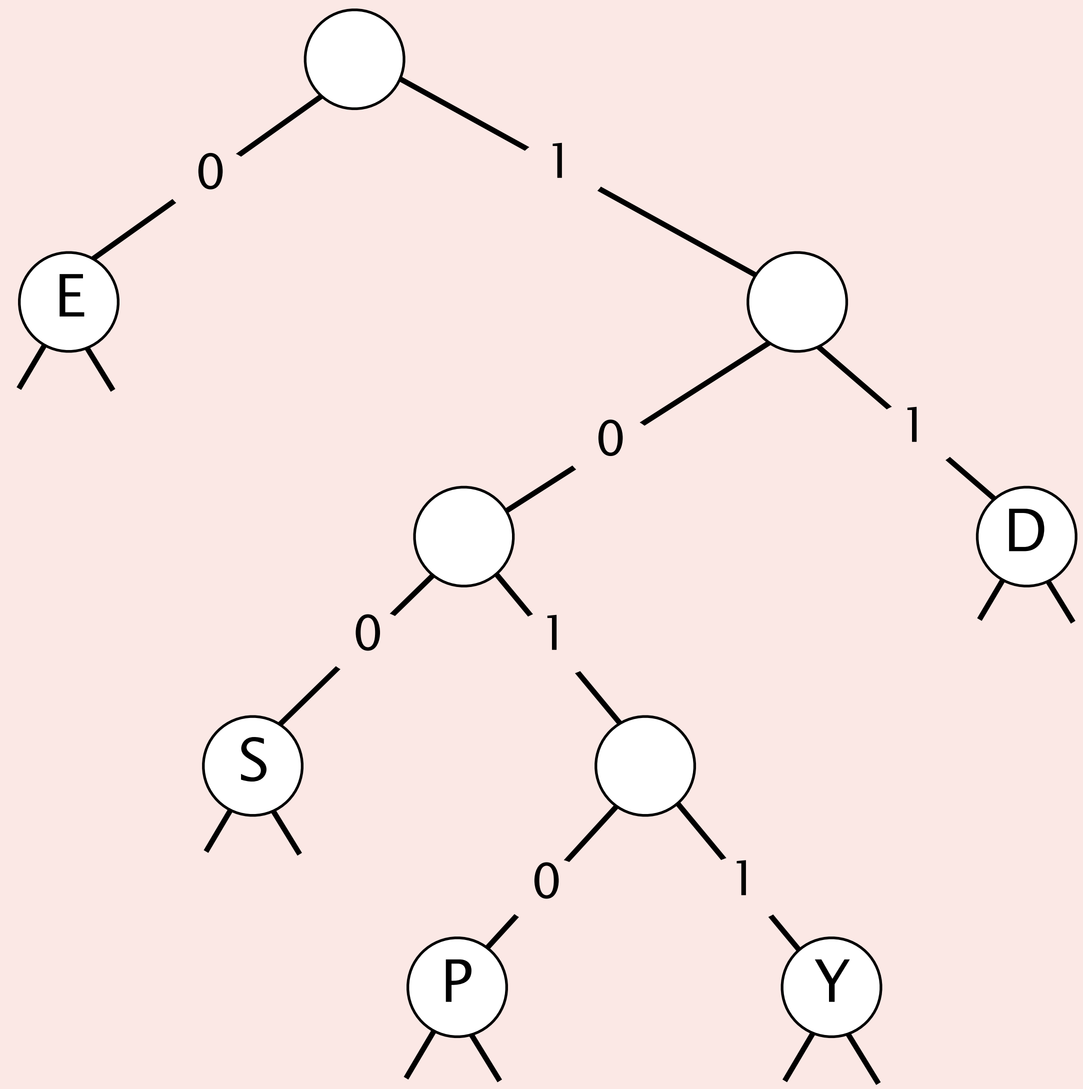
- Method 1: start at leaf; follow path up to the root; print bits in reverse.
- Method 2: create ST of key–value pairs.



Data compression: quiz 2

Consider the following trie representation of a prefix-free code. Expand the compressed bitstring 1 0 0 1 0 1 0 0 0 1 1 1 0 1 1 ?

- **A.** P E E D
- **B.** P E S D E Y
- **C.** S P E D
- **D.** S P E E D Y



Huffman coding overview

Static model. Use the same prefix-free code for all messages.

Dynamic model. Use a custom prefix-free code for each message.

Compression.

- Read message.
- Build best prefix-free code for message. How? [ahead]
- Write prefix-free code.
- Compress message using prefix-free code.

Expansion.

- Read prefix-free code.
- Read compressed message and expand using prefix-free code.

Huffman trie node data type

```
private static class Node implements Comparable<Node>
{
   private final char ch;    // used only for leaf nodes
   private final int freq;   // used only by compress()
   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();                          // read encoding trie
   int n = BinaryStdIn.readInt();                   // read number of chars

   for (int i = 0; i < n; i++)                      // for each encoded character i
   {
      Node x = root;                                // follow path from root to leaf
      while (!x.isLeaf())                           // to determine character
      {
         if (!BinaryStdIn.readBoolean())
            x = x.left;
         else
            x = x.right;
      }
      BinaryStdOut.write(x.ch, 8);                  // write character (8 bits)
   }
   BinaryStdOut.close();
}
```

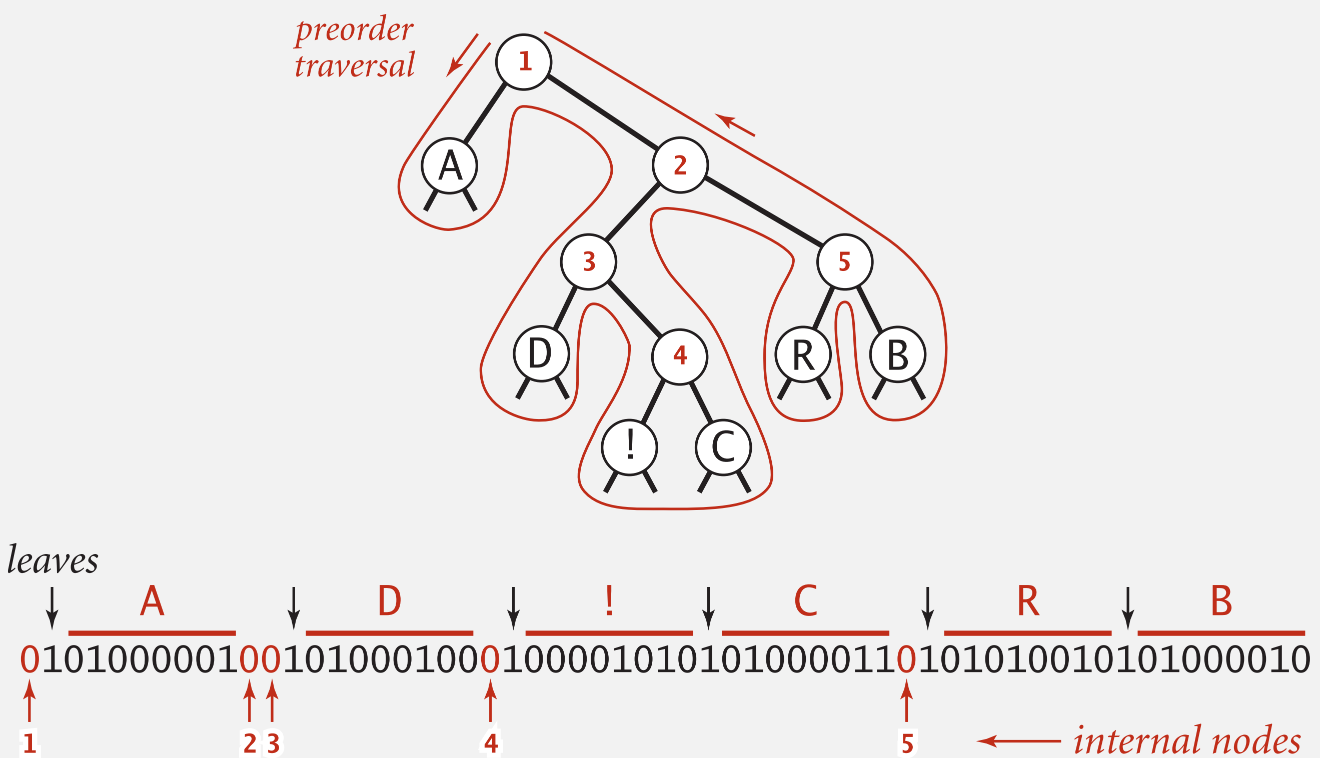
Running time. Linear in input size (number of bits).

Prefix-free codes: how to transmit

Q. How to write the trie?

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

0 for internal nodes
1 for leaf nodes



Using preorder traversal to encode a trie as a bitstream

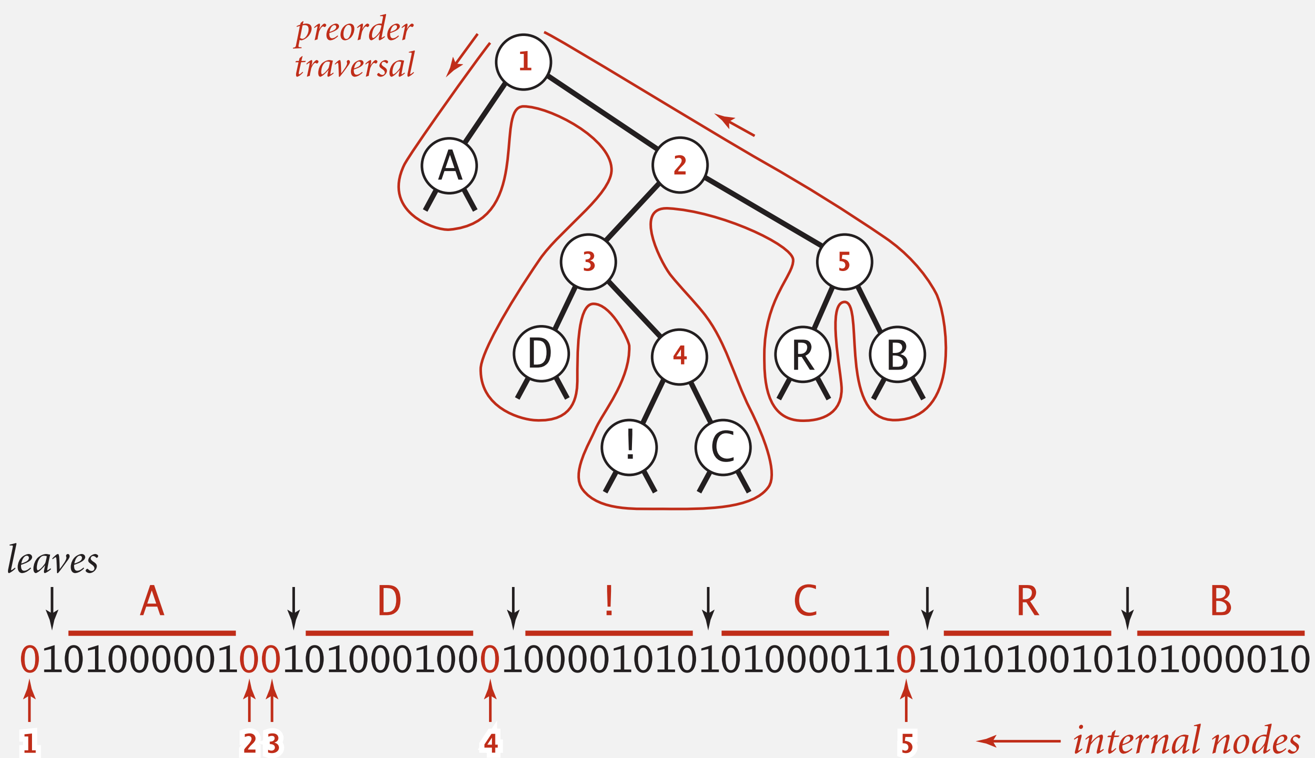
```
private static void writeTrie(Node x)
{
   if (x.isLeaf())
   {
      BinaryStdOut.write(true);
      BinaryStdOut.write(x.ch, 8);
      return;
   }
   BinaryStdOut.write(false);
   writeTrie(x.left);
   writeTrie(x.right);
}
```

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

Prefix-free codes: how to transmit

Q. How to read the trie?

A. Reconstruct from preorder traversal.



Using preorder traversal to encode a trie as a bitstream

```
private static Node readTrie()
{
   if (BinaryStdIn.readBoolean())
   {
      char c = BinaryStdIn.readChar(8);
      return new Node(c, 0, null, null);
   }
   Node x = readTrie();
   Node y = readTrie();
   return new Node('\0', 0, x, y);
}
```

arbitrary value
(value not used with internal nodes)

Exercise: find the best prefix-free code

Exercise 1 (warmup).

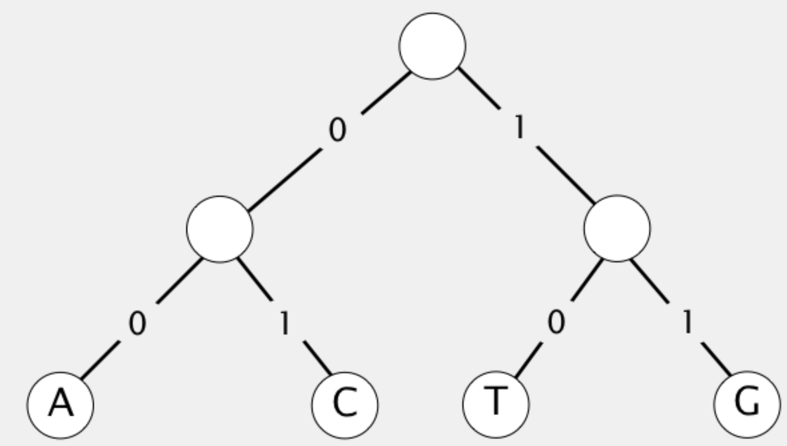
Alphabet: `{ A, T, C, G }`.

String may be a genome: `A T A G A T G C A T A G . . .`

Assume each character is equally likely.

Draw the trie for the best prefix-free code.

How many bits does it use per input symbol?



Exercise 2.

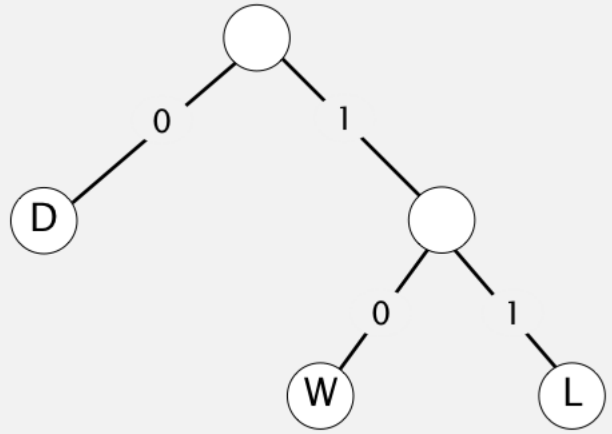
Alphabet: `{ W, L, D }`.

Example: results of games that can end in a Win/Loss/Draw for the home team.

Assume that the character frequencies are W: 25%; L: 25%; D: 50%.

Draw the trie for the best prefix-free code.

How many bits does it use per input symbol?



Huffman codes

Q. How to find best prefix-free code?

Huffman algorithm:

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

Applications:

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 for internal nodes

total frequency

two subtries

Huffman compression summary

Proposition. Huffman's algorithm produces an optimal prefix-free code.

Pf. See textbook.

no prefix-free code uses fewer bits

Two-pass implementation (for compression).

- Pass 1: tabulate character frequencies; build trie.
- Pass 2: encode file by traversing trie (or symbol table).

Running time (for compression). Using a binary heap $\Rightarrow$ $n + R \log R$.

Running time (for expansion). Using a binary trie $\Rightarrow$ n.

input size (n), alphabet size (R)

Q. Can we do better (in terms of compression ratio)? [stay tuned]

https://algs4.cs.princeton.edu

5.5 Data Compression

- *introduction*
- *run-length coding*
- *Huffman compression*
- ***LZW compression***

Statistical methods

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.

LZW compression demo

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

LZW compression for A B R A C A D A B R A B R A B R A

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
ABRA	8B

codeword table

- Input is 7-bit ASCII.
- ASCII value of 'A' is 65 (hex 41).
- Max ASCII value is 127 (hex 79).
- Codewords for single characters are the same as ASCII values.
- We use hex 80 as stop symbol.
- We start new codewords at hex 81.
- We use 8-bit codewords, so we have 127 more slots in table.

LZW expansion demo

value	41	42	52	41	43	41	44	81	83	82	88	41	80
output	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

key	value
⋮	⋮
41	A
42	B
43	C
44	D
⋮	⋮

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

key	value
87	DA
88	ABR
89	RAB
8A	BRA
8B	ABRA

codeword table

- Input is 7-bit ASCII.
- ASCII value of ‘A’ is 65 (hex 41).
- Max ASCII value is 127 (hex 79).
- Codewords for single characters are the same as ASCII values.
- We use hex 80 as stop symbol.
- We start new codewords at hex 81.
- We use 8-bit codewords, so we have 127 more slots in table.

Data compression: quiz 3

Which is the LZW compression for `ABABABA` ?

- **A.** 41 42 41 42 41 42 80
- **B.** 41 42 41 81 81 80
- **C.** 41 42 81 81 41 80
- **D.** 41 42 81 83 80

Data compression: quiz 3

Which is the LZW compression for A B A B A B A ?

input	A	B	A	B	A	B	A	
matches	A	B	A B		A B A			
value	41	42	81		83			80

key	value
⋮	⋮
A	41
B	42
C	43
D	44
⋮	⋮

key	value
AB	81
BA	82
ABA	83

Data compression: quiz 4

Which is a key data structure to implement LZW compression efficiently?

- **A.** array
- **B.** red–black BST
- **C.** hash table
- **D.** none of the above

Lempel–Ziv–Welch compression

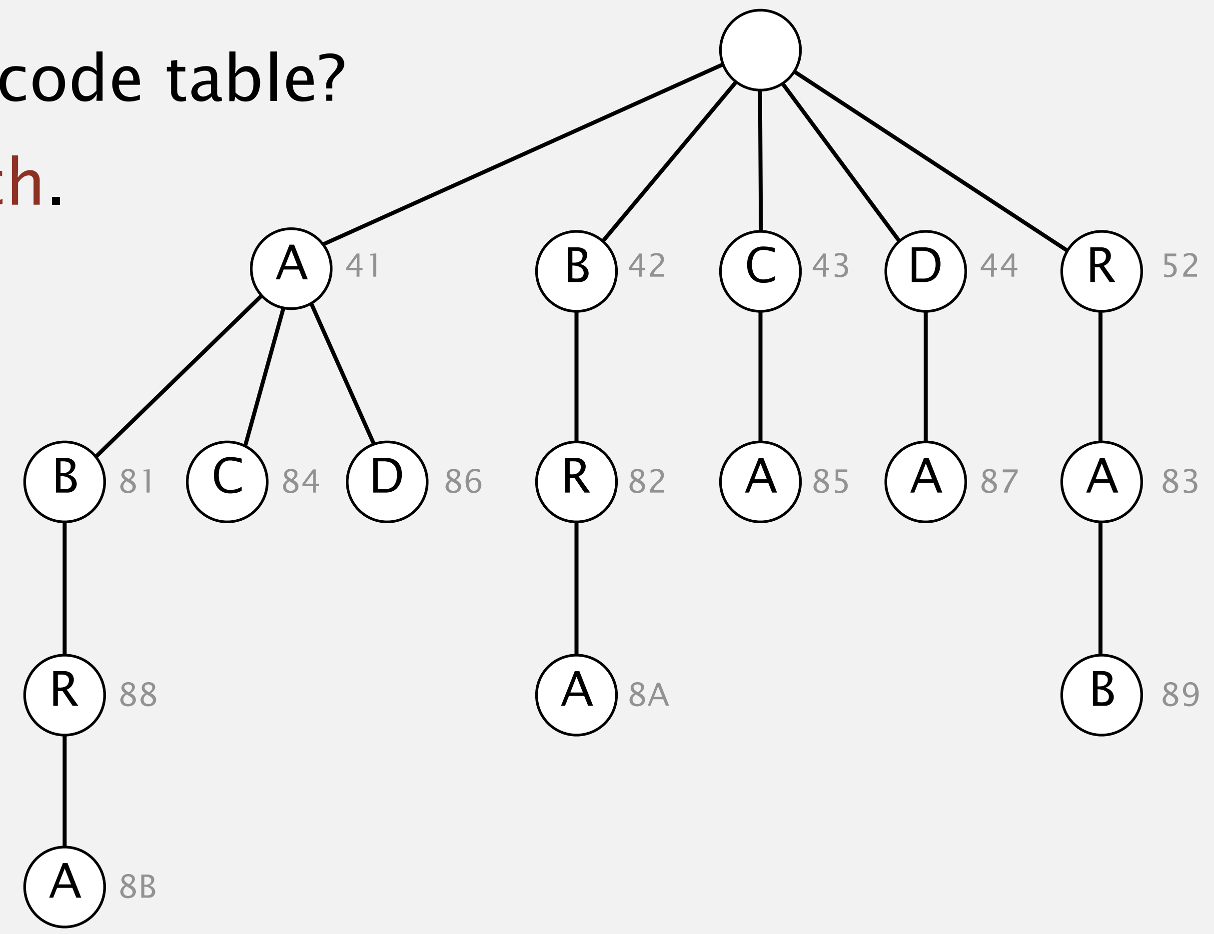
LZW compression.

- Create ST associating W-bit codewords with string keys.
- Initialize ST with codewords for single-character 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 character in the input.

longest prefix match

Q. How to represent LZW compression code table?

A. A trie to support longest prefix match.



LZW expansion

LZW expansion.

- Create ST associating string values with W-bit keys.
- Initialize ST to contain single-character values.
- Read a W-bit key.
- Find associated string value in ST and write it out.
- Update ST [key = size of table (ie. next unassigned integer);
  value = prev. string + first char of cur. string]

Q. How to represent LZW expansion code table?
A. An array of length 2^W.

Surprising fact.

- No need to transmit codeword table!
- It can be reconstructed on the fly, as shown above.

key	value
⋮	⋮
65	A
66	B
67	C
68	D
⋮	⋮
129	AB
130	BR
131	RA
132	AC
133	CA
134	AD
135	DA
136	ABR
137	RAB
138	BRA
139	ABRA
⋮	⋮

LZW tricky case: expansion

value	41	42	81	83	80
output	A	B	A B	A B A	

need to know code for 83 before it is in codeword table!

we can deduce that the code for 83 is ABx for some character x

now, we have deduced x!

LZW expansion for 41 42 81 83 80

key	value
⋮	⋮
41	A
42	B
43	C
44	D
⋮	⋮

key	value
81	AB
82	BA
83	ABA

codeword table

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

← next programming assignment

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/DCT, wavelets, fractals, ...

$$X_k = \sum_{i=0}^{n-1} x_i \cos\left[\frac{\pi}{n}\left(i + \frac{1}{2}\right) k\right]$$

Theoretical limits on compression. Shannon entropy: $H(X) = -\sum_{i}^{n} p(x_i) \lg p(x_i)$

Practical compression. Exploit extra knowledge whenever possible.