

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

DATA COMPRESSION

May. 12, 2016

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

DATA COMPRESSION

- ▶ **Run-length coding**
- ▶ **Huffman 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

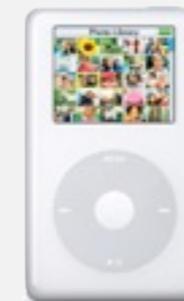
Generic file compression.

- Files: GZIP, BZIP, 7z.
- Archivers: PKZIP.
- File systems: NTFS, HFS+, ZFS.



Multimedia.

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



Communication.

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



Databases. Google, Facebook,



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)



Basic model for data compression

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

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

Food for thought

Data compression has been omnipresent since antiquity:

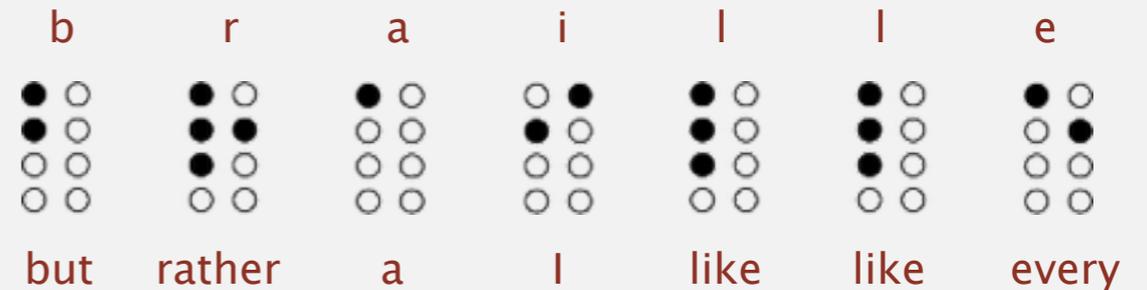
- Number systems.
- Natural languages.
- Mathematical notation.



$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

has played a central role in communications technology,

- Grade 2 Braille.
- Morse code.
- Telephone system.



and is part of modern life.

- MP3.
- MPEG.



Q. What role will it play in the future?

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 |

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

Amazing but true. Initial genomic databases in 1990s used ASCII.

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)]

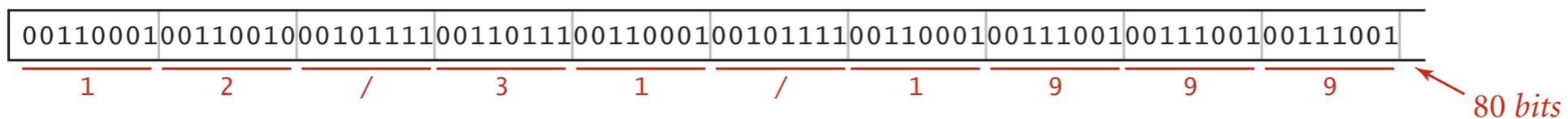
```
void close()          close the bitstream
```

Writing binary data

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

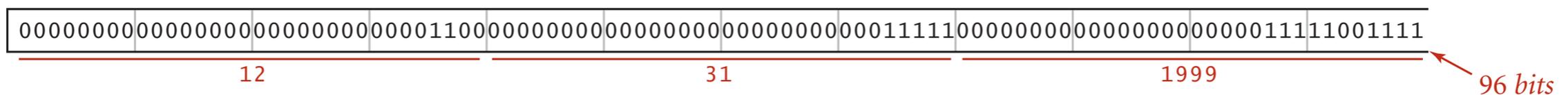
A character stream (StdOut)

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



Three ints (BinaryStdOut)

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



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);
```



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

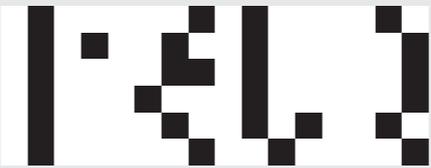
```
% java BinaryDump 16 < abra.txt  
0100000101000010  
0101001001000001  
0100001101000001  
0100010001000001  
0100001001010010  
0100000100100001  
96 bits
```

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
```



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

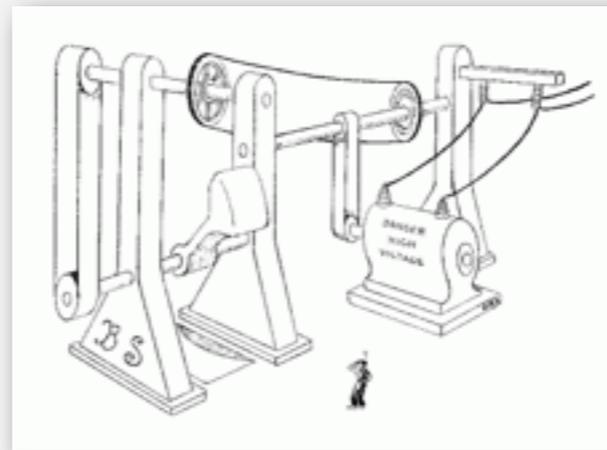
Universal data compression

US Patent 5,533,051 on "Methods for Data Compression", which is capable of compression **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

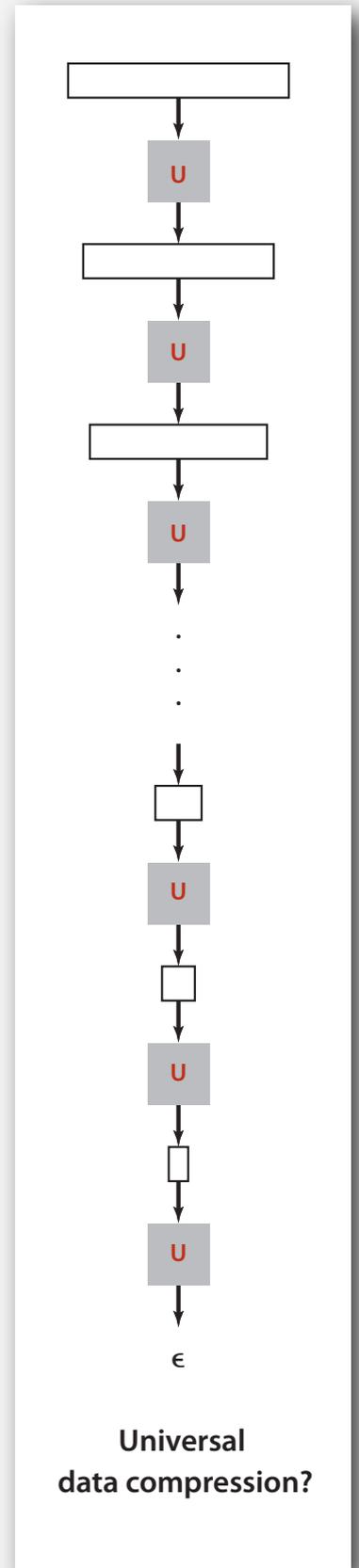
Proposition. No algorithm can compress every bitstring.

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 to 0 bits!

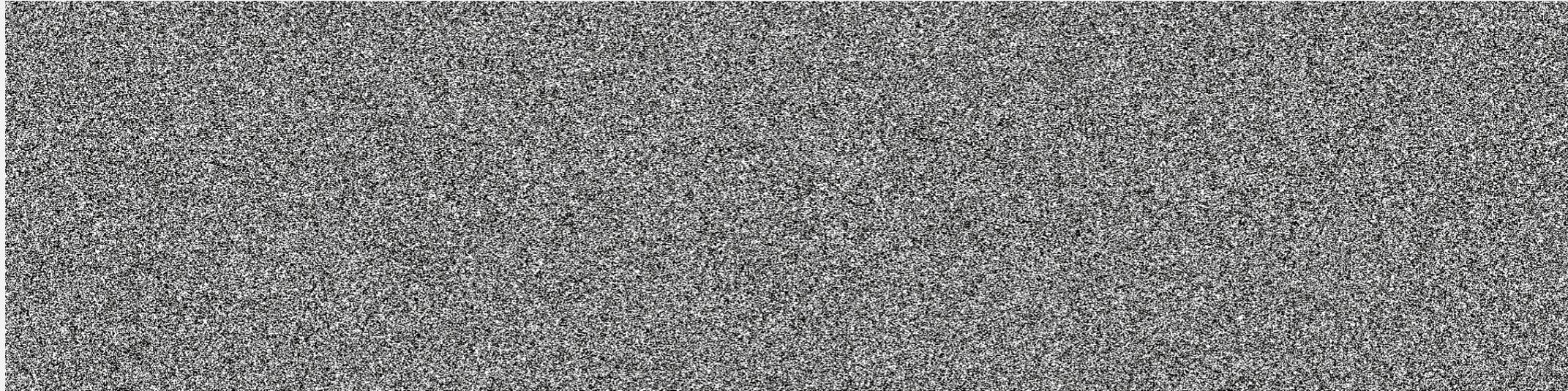
Pf 2. [by counting]

- Suppose your algorithm that can compress all 1,000-bit strings.
- 2^{1000} possible bitstrings with 1,000 bits.
- Only $1 + 2 + 4 + \dots + 2^{998} + 2^{999}$ can be encoded with ≤ 999 bits.
- Similarly, only 1 in 2^{499} bitstrings can be encoded with ≤ 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();
    }
}
```

Redundancy in English Language

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

Redundancy in Turkish Language

Q. How much redundancy is in the Turkish language?

“ Bir İngiliz Ünversitesinde yapılan araştırmaya göre, kelimelerin harflerinin hangi sıradaki yazıldığı önemli değilmiş. Önemli olan birinci ve sonuncu harflerinde olmasıymış. Ardaki harflerin sırası kırışıklıkta okunuyormuş. Çünkü kelimeleri harf harfinden bütün olarak okuyoruz” —*Anonymous*

A. Quite a bit

DATA COMPRESSION

- ▶ **Run-length coding**
- ▶ **Huffman compression**

Run-length encoding: Java implementation

```
public class RunLength
```

```
{
```

```
    private final static int R    = 256;
```

← maximum run-length count

```
    private final static int lgR = 8;
```

← number of bits per count

```
    public static void compress()
```

```
    { /* see textbook */ }
```

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

```
    {
```

```
        boolean bit = false;
```

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

```
        {
```

```
            int run = BinaryStdIn.readInt(lgR);
```

← read 8-bit count from standard input

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

```
                BinaryStdOut.write(bit);
```

← write 1 bit to standard output

```
            bit = !bit;
```

```
        }
```

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

← pad 0s for byte alignment

```
    }
```

```
}
```


Black and white bitmap compression: another approach

Fax machine (~1980).

- Slow scanner produces lines in sequential order.
- Compress to save time (reduce number of bits to send).

Electronic documents (~2000).

- High-resolution scanners produce huge files.
- Compress to save space (reduce number of bits to save).

Idea.

- use OCR to get back to ASCII (!)
- use Huffman on ASCII string (!)

Bottom line. Any extra information about file can yield dramatic gains.

DATA COMPRESSION

- ▶ Run-length coding
- ▶ **Huffman compression**

Variable-length codes

Use different number of bits to encode different chars.

Ex. Morse code: • • • — — — • • •

Issue. Ambiguity.

SOS ?

V7 ?

IAMIE ?

EEWNI ?

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

| <i>key</i> | <i>value</i> |
|------------|--------------|
| ! | 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

| <i>key</i> | <i>value</i> |
|------------|--------------|
| ! | 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

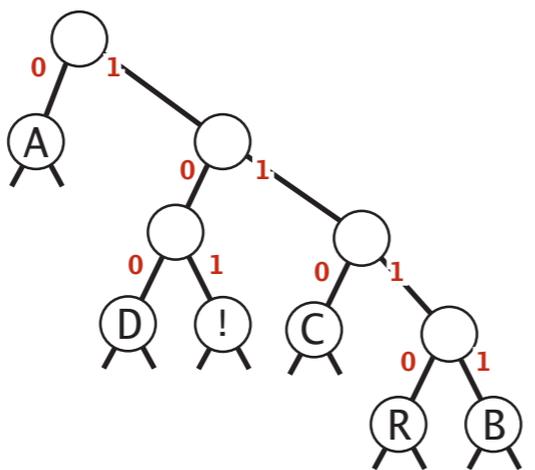
Compression.

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

Expansion.

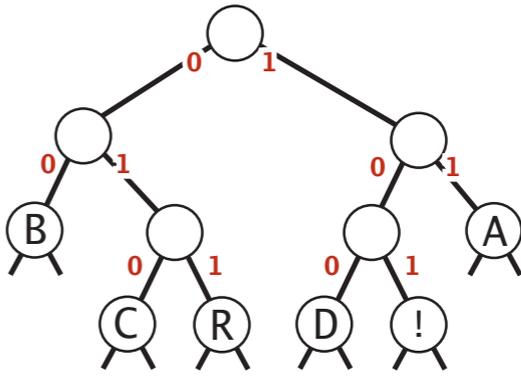
- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, print char and return to root.

| 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 !

| 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 !

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, 8);
```

```
    }
```

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

```
}
```

← read in encoding trie

← read in number of chars

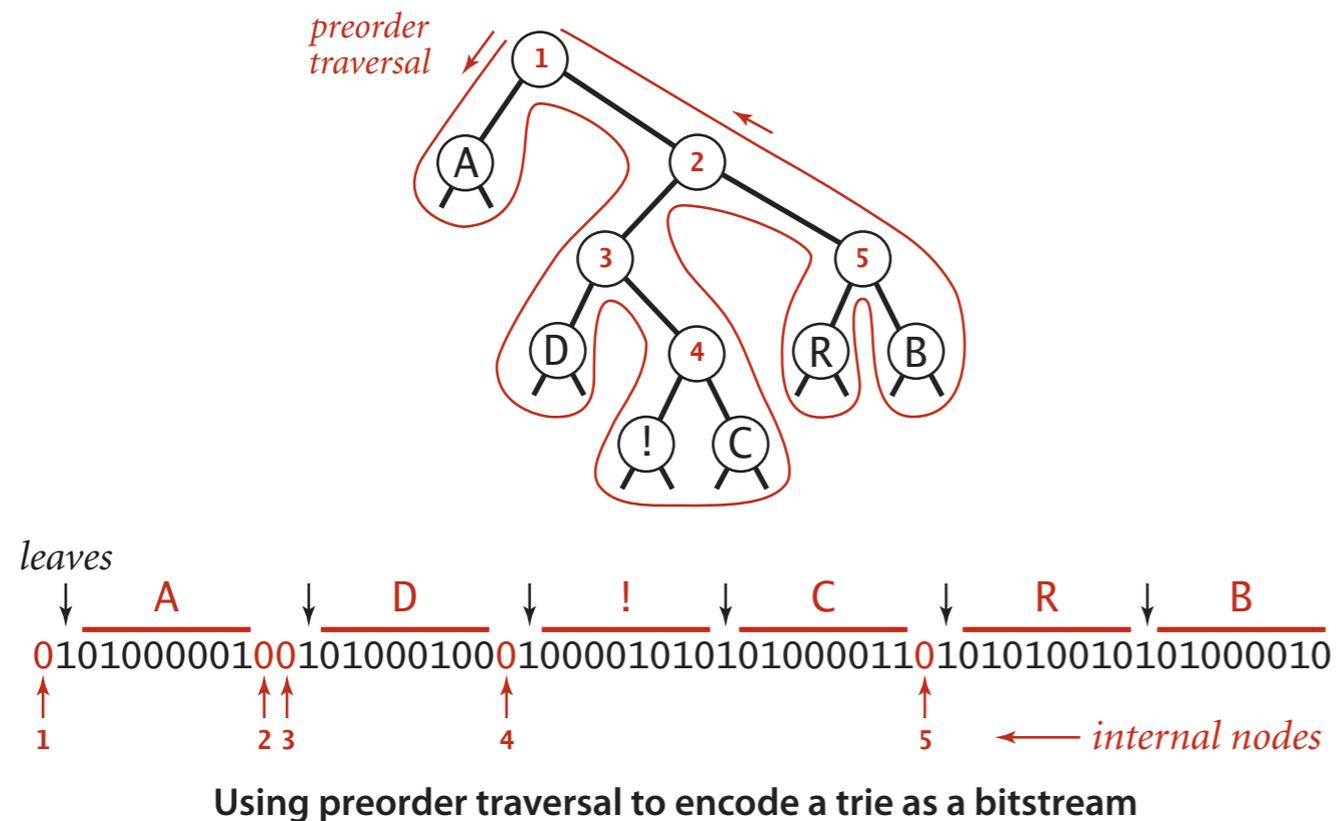
← expand codeword for i^{th} char

Running time. Linear in input size N .

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.



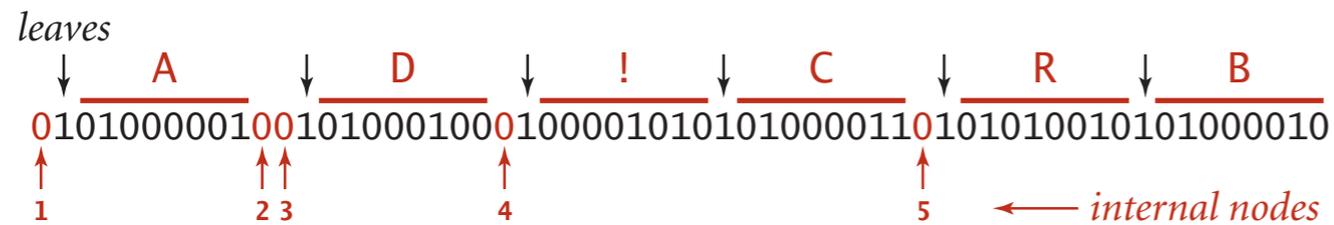
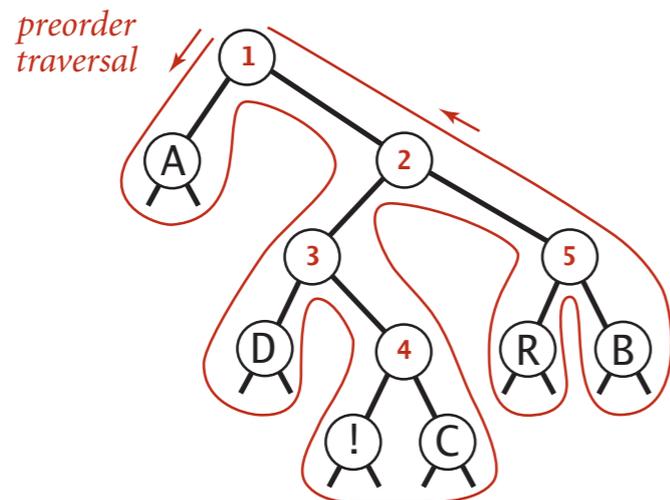
```
private static void writeTrie(Node x)
{
    if (x.isLeaf())
    {
        BinaryStdOut.write(true);
        BinaryStdOut.write(x.ch, 8);
        return;
    }
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```

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

Prefix-free codes: how to transmit

Q. How to read 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(8);
        return new Node(c, 0, null, null);
    }
    Node x = readTrie();
    Node y = readTrie();
    return new Node('\0', 0, x, y);
}
```

not used for internal nodes

Shannon-Fano codes

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 .

| char | freq | encoding |
|------|------|----------|
| A | 5 | 0... |
| C | 1 | 0... |

S_0 = codewords starting with 0

| char | freq | encoding |
|------|------|----------|
| B | 2 | 1... |
| D | 1 | 1... |
| R | 2 | 1... |
| ! | 1 | 1... |

S_1 = codewords starting with 1

Problem 1. How to divide up symbols?

Problem 2. Not optimal!

Huffman algorithm

- Count frequency for each character in input.

| <u>char</u> | <u>freq</u> | <u>encoding</u> |
|-------------|-------------|-----------------|
| A | 5 | |
| B | 2 | |
| C | 1 | |
| D | 1 | |
| R | 2 | |
| ! | 1 | |

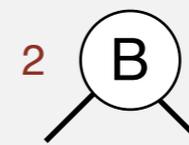
input

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

Huffman algorithm

- Start with one node corresponding to each character with weight equal to frequency.

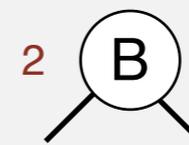
| <u>char</u> | <u>freq</u> | <u>encoding</u> |
|-------------|-------------|-----------------|
| A | 5 | |
| B | 2 | |
| C | 1 | |
| D | 1 | |
| R | 2 | |
| ! | 1 | |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

| <u>char</u> | <u>freq</u> | <u>encoding</u> |
|-------------|-------------|-----------------|
| A | 5 | |
| B | 2 | |
| C | 1 | |
| D | 1 | |
| R | 2 | |
| ! | 1 | |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

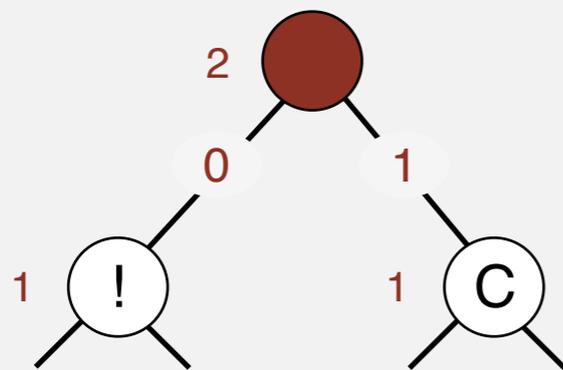
| <u>char</u> | <u>freq</u> | <u>encoding</u> |
|-------------|-------------|-----------------|
| A | 5 | |
| B | 2 | |
| C | 1 | |
| D | 1 | |
| R | 2 | |
| ! | 1 | |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

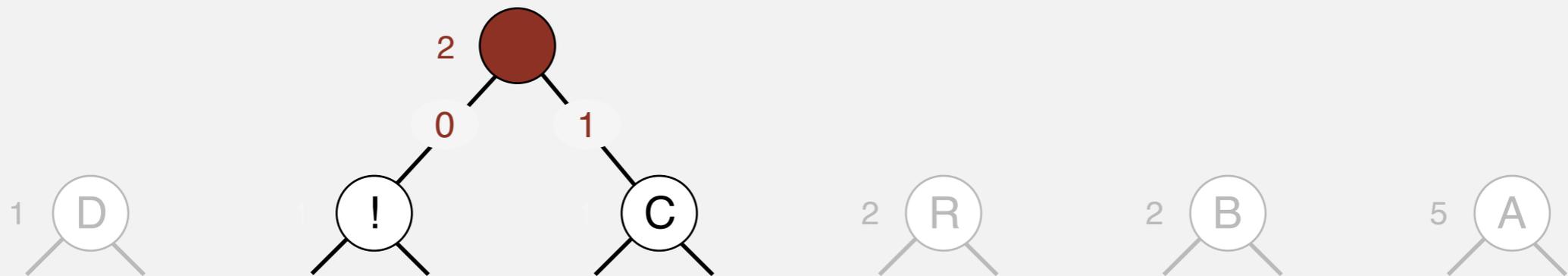
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | |
| C | 1 | 1 |
| D | 1 | |
| R | 2 | |
| ! | 1 | 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

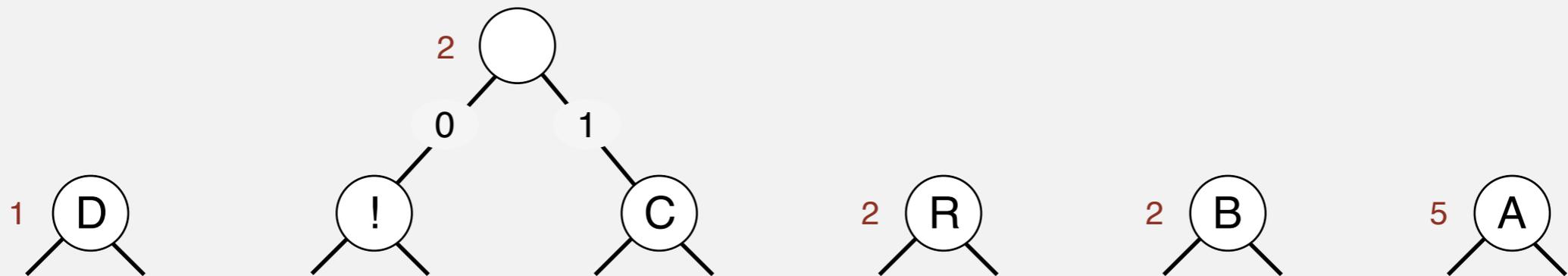
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | |
| C | 1 | 1 |
| D | 1 | |
| R | 2 | |
| ! | 1 | 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

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



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

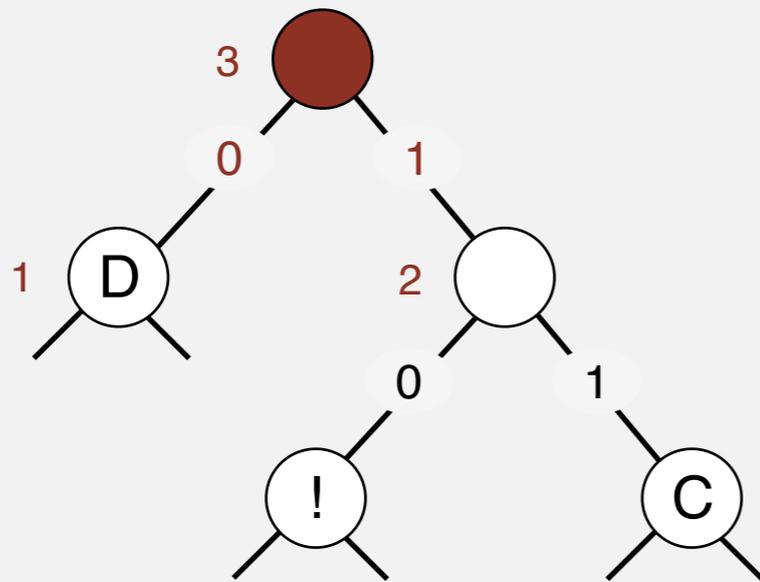
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | |
| C | 1 | 1 |
| D | 1 | |
| R | 2 | |
| ! | 1 | 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

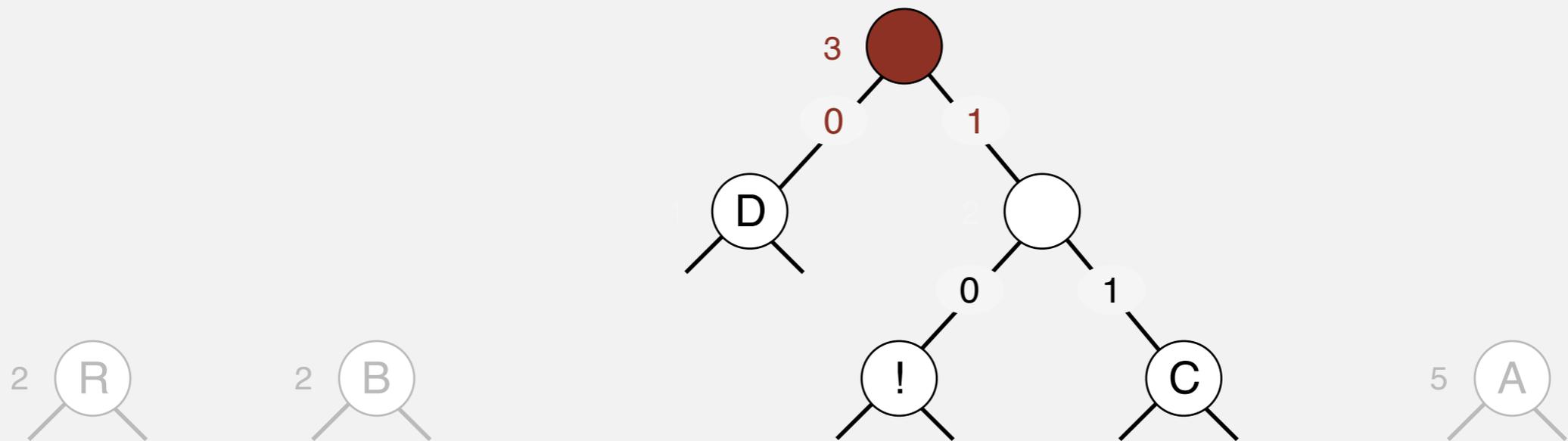
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | |
| C | 1 | 1 1 |
| D | 1 | 0 |
| R | 2 | |
| ! | 1 | 1 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

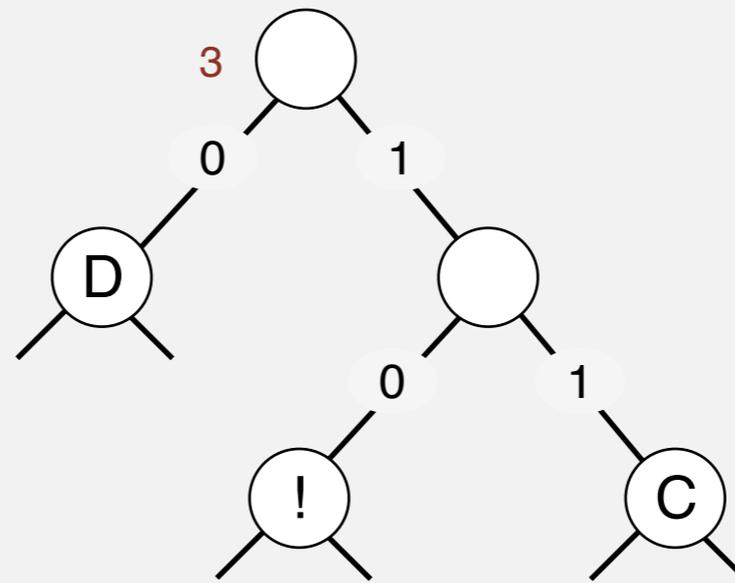
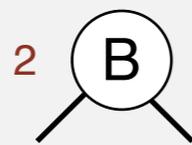
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | |
| C | 1 | 1 1 |
| D | 1 | 0 |
| R | 2 | |
| ! | 1 | 1 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

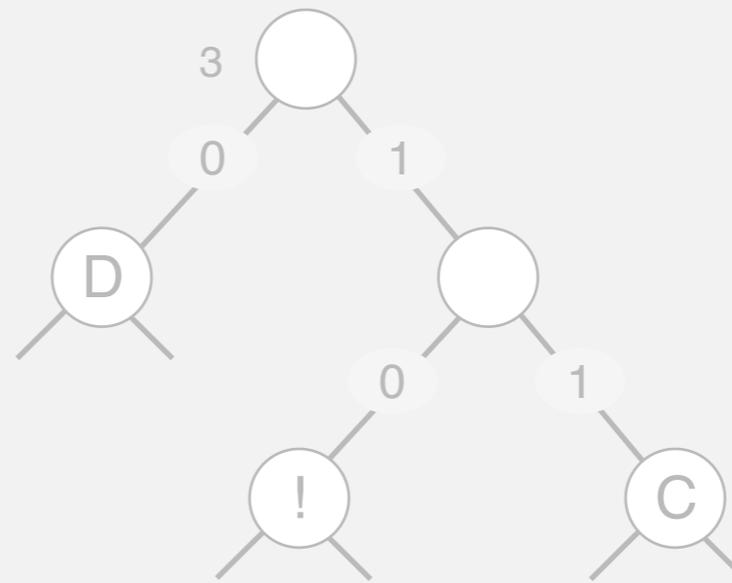
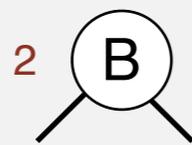
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | |
| C | 1 | 1 1 |
| D | 1 | 0 |
| R | 2 | |
| ! | 1 | 1 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

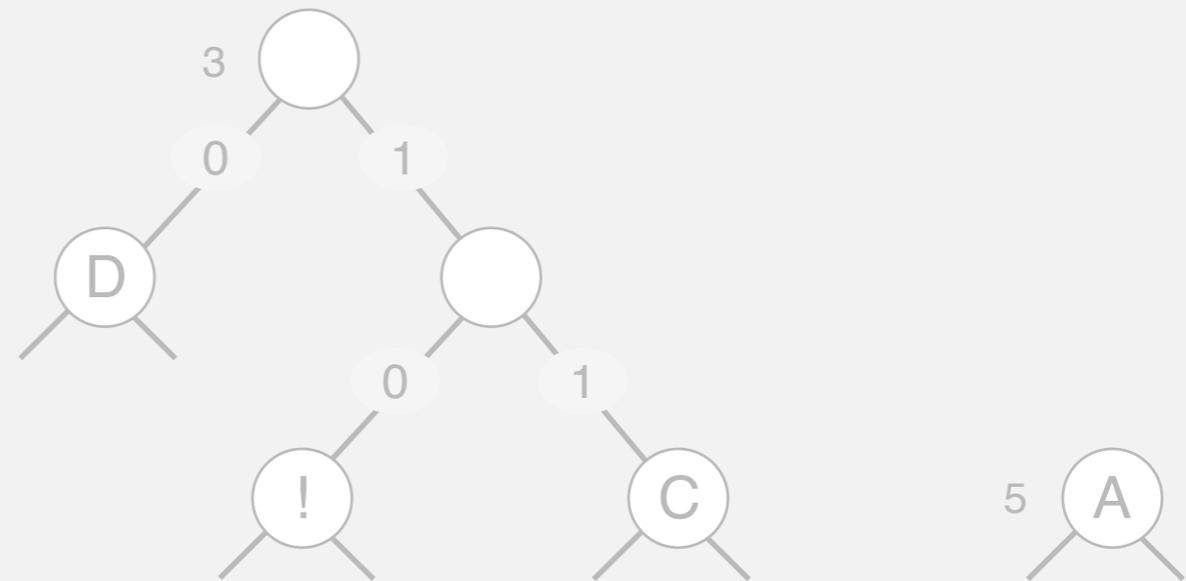
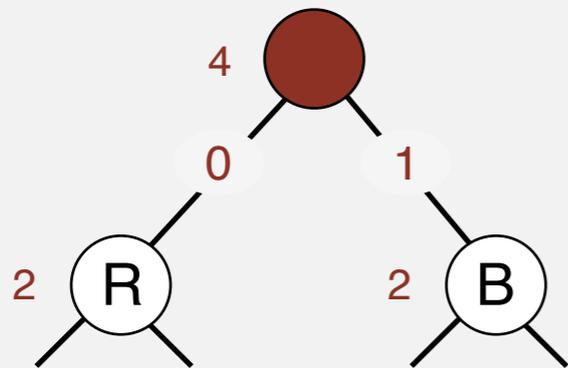
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | |
| C | 1 | 1 1 |
| D | 1 | 0 |
| R | 2 | |
| ! | 1 | 1 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

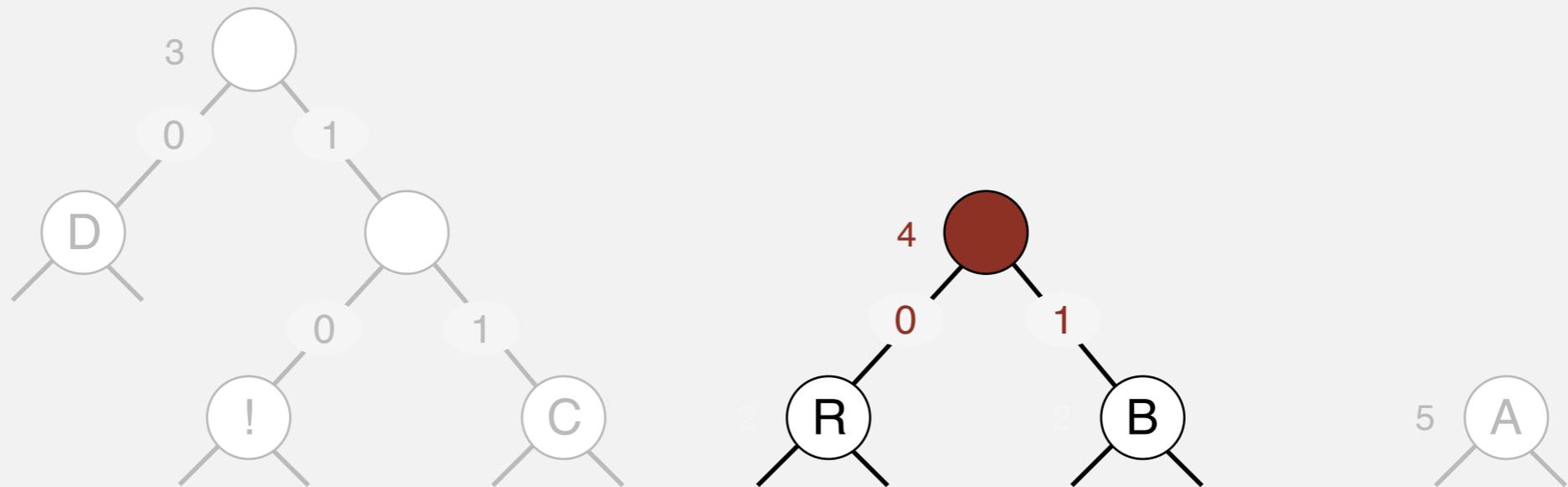
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | 1 |
| C | 1 | 1 1 |
| D | 1 | 0 |
| R | 2 | 0 |
| ! | 1 | 1 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

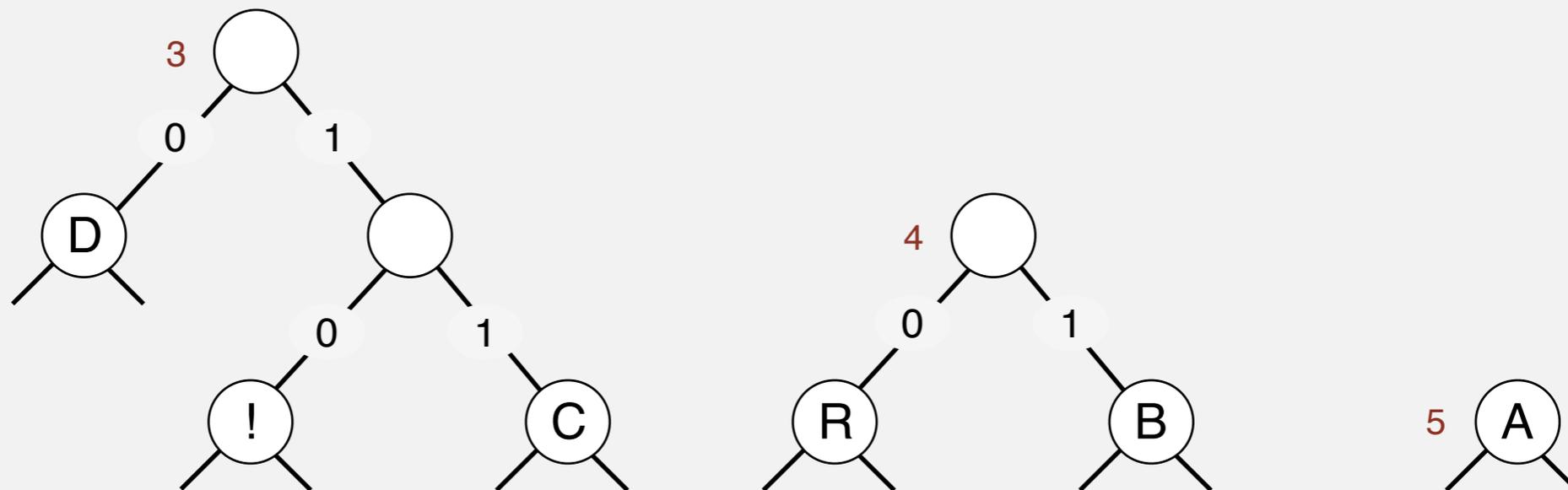
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | 1 |
| C | 1 | 1 1 |
| D | 1 | 0 |
| R | 2 | 0 |
| ! | 1 | 1 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

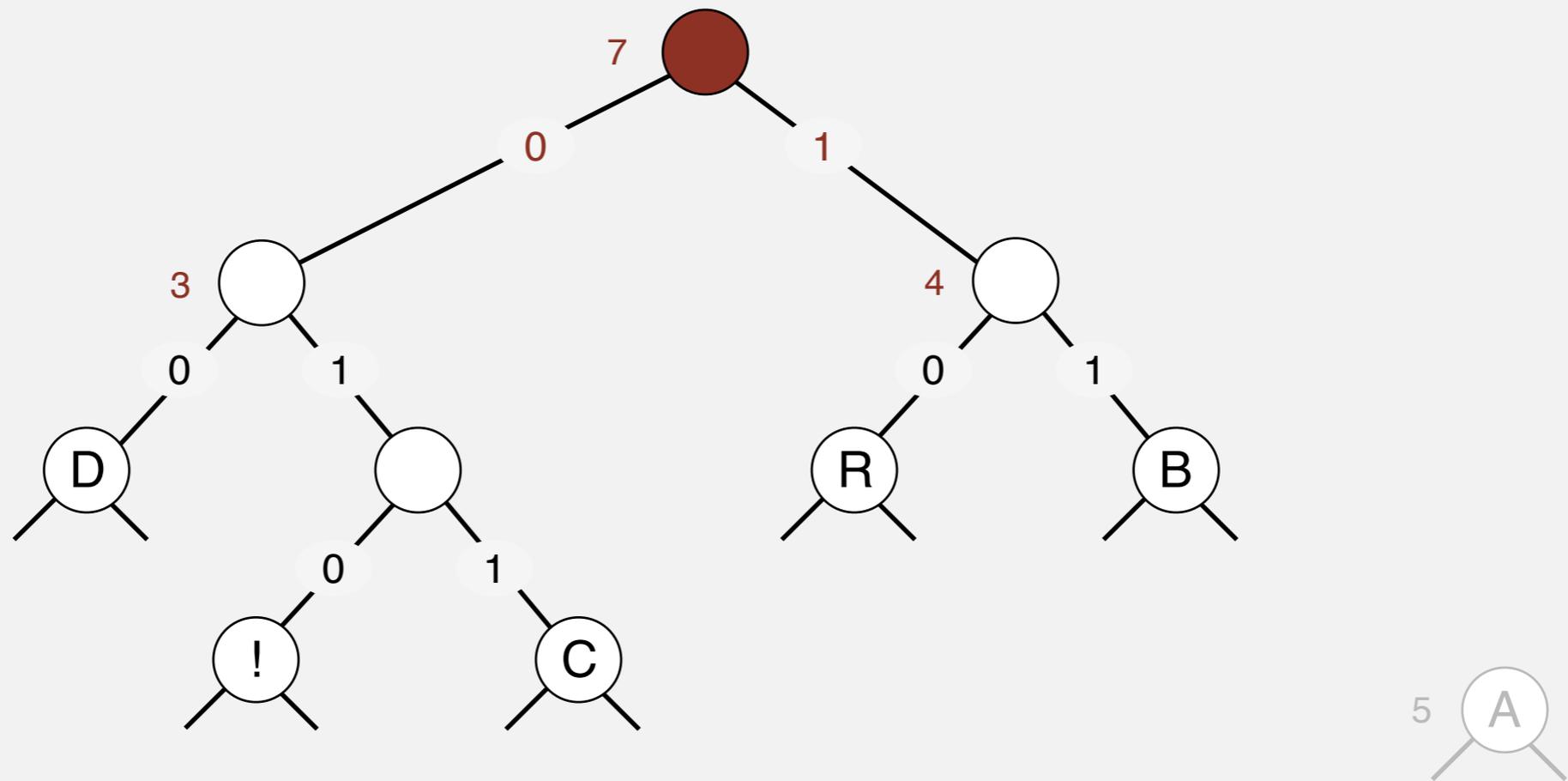
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | 1 |
| C | 1 | 1 1 |
| D | 1 | 0 |
| R | 2 | 0 |
| ! | 1 | 1 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

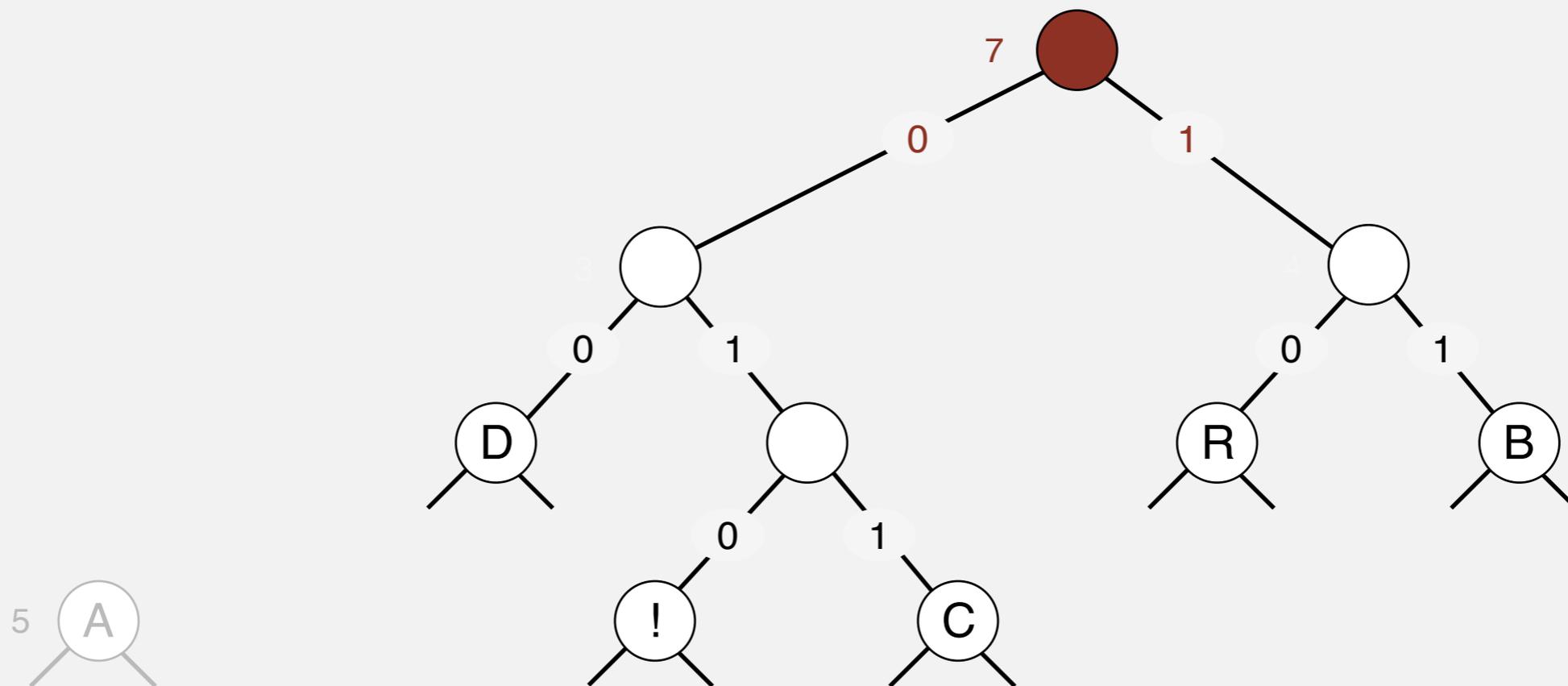
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | 1 1 |
| C | 1 | 0 1 1 |
| D | 1 | 0 0 |
| R | 2 | 1 0 |
| ! | 1 | 0 1 0 |



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

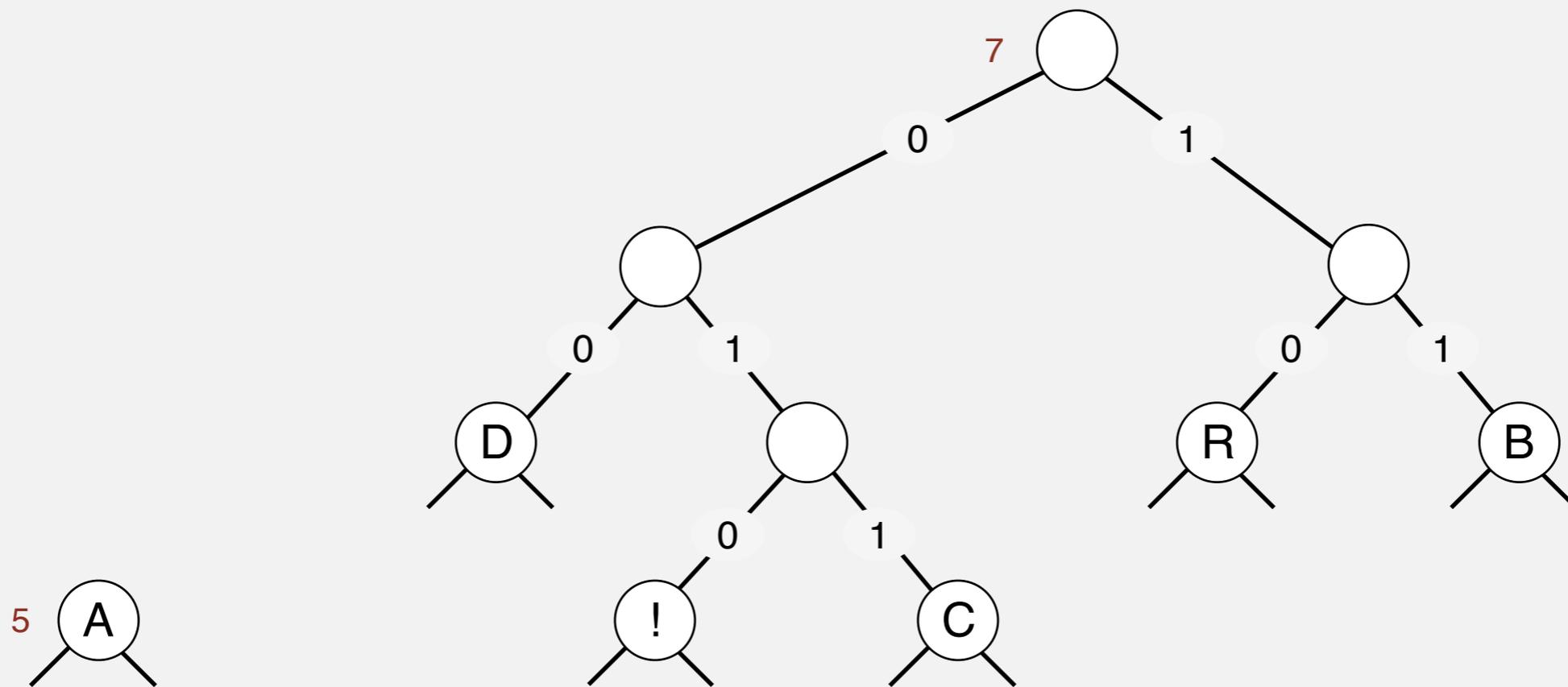
| char | freq | encoding |
|------|------|----------|
| A | 5 | |
| B | 2 | 1 1 |
| C | 1 | 0 1 1 |
| D | 1 | 0 0 |
| R | 2 | 1 0 |
| ! | 1 | 0 1 0 |



Huffman algorithm

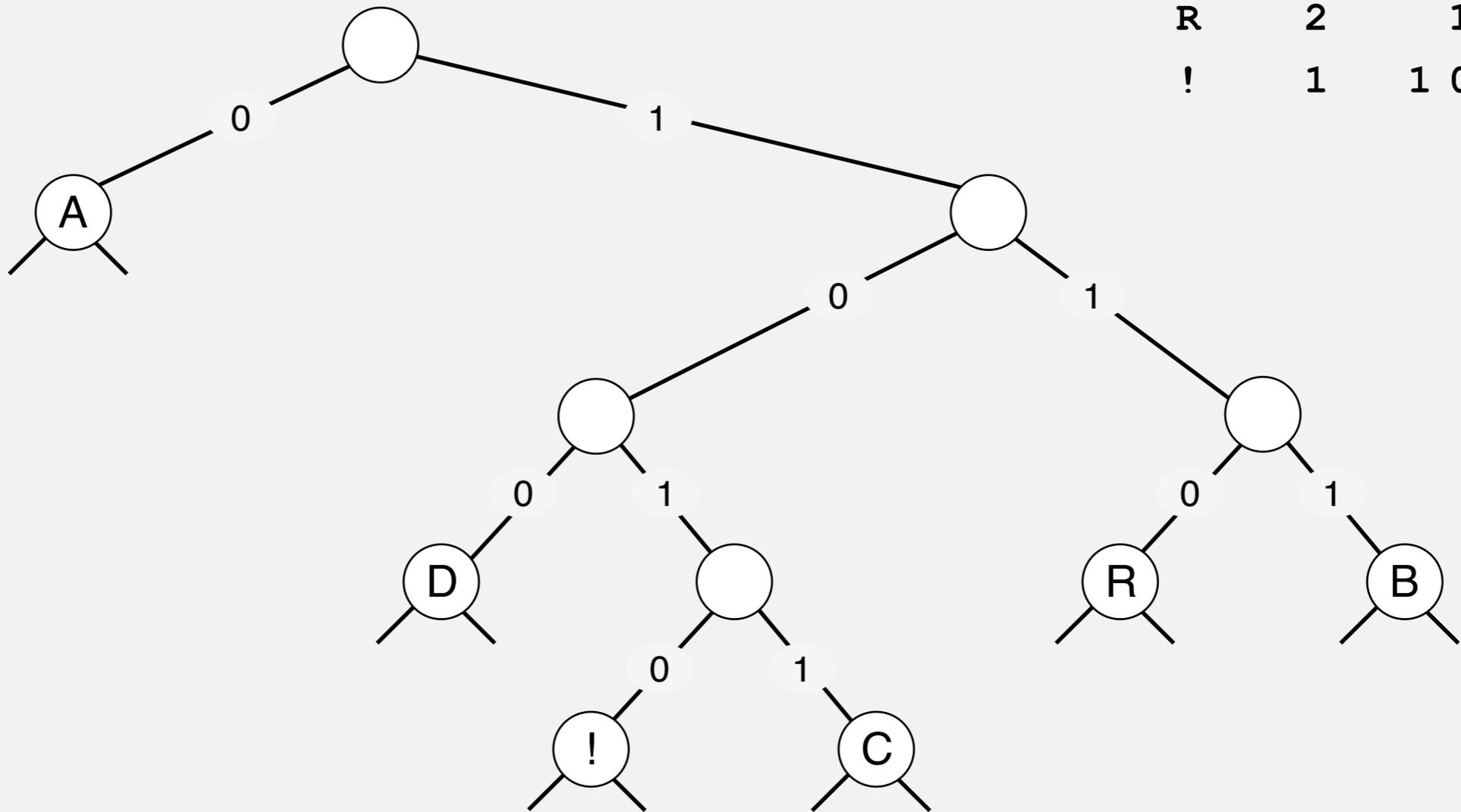
- Select two tries with min weight.
- Merge into single trie with cumulative weight.

| 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 algorithm

| 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 codes

Q. How to find best prefix-free code?

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:



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));
```

← initialize PQ with
singleton tries

```
    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);
```

```
    }
```

← merge two
smallest tries

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

↑
not used for
internal nodes

↑
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 $\Rightarrow N + R \log R$.

↑
input
size

↑
alphabet
size

Q. Can we do better? [stay tuned]