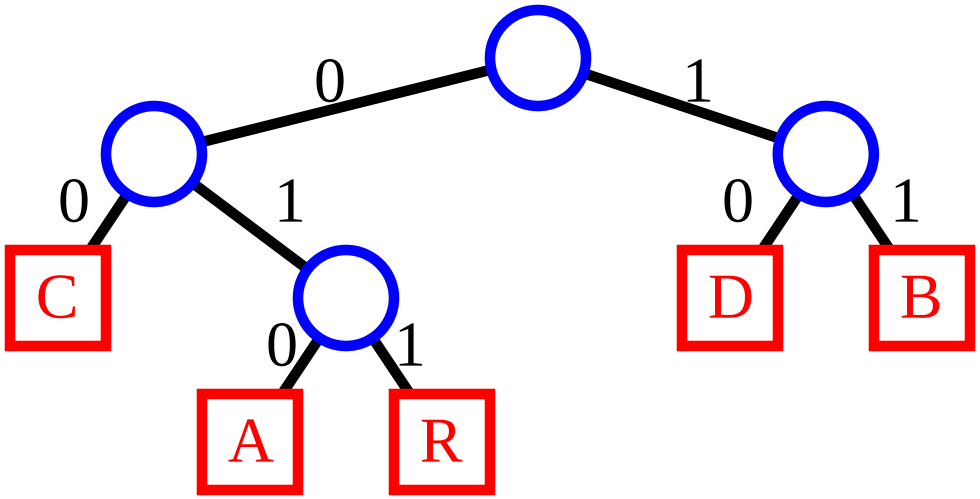
DATA COMPRESSION

- File Compression
- Huffman Tries



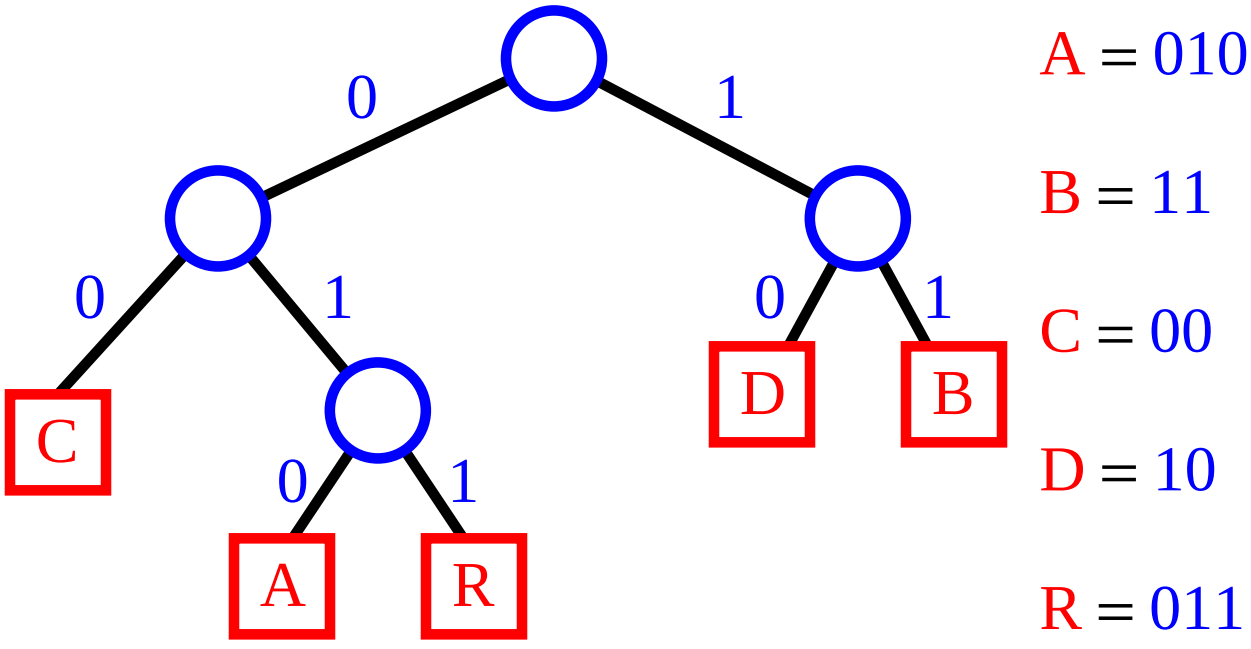
ABRACADABRA
01011011010000101001011011010

File Compression

- text files are usually stored by representing each character with an 8-bit ASCII code (type man ascii in a Unix shell to see the ASCII encoding)
- the ASCII encoding is an example of **fixed-length encoding**, where each character is represented with the same number of bits
- in order to reduce the space required to store a text file, we can exploit the fact that some characters are more likely to occur than others
- **variable-length encoding** uses binary codes of different lengths for different characters; thus, we can assign fewer bits to frequently used characters, and more bits to rarely used characters.
- Example:
  - text: java
  - encoding: a = “0”, j = “11”, v = “10”
  - encoded text: 110100 (6 bits)
- How to decode?
  - a = “0”, j = “01”, v = “00”
  - encoded text: 010000 (6 bits)
  - is this java, jvv, jaaaa ...

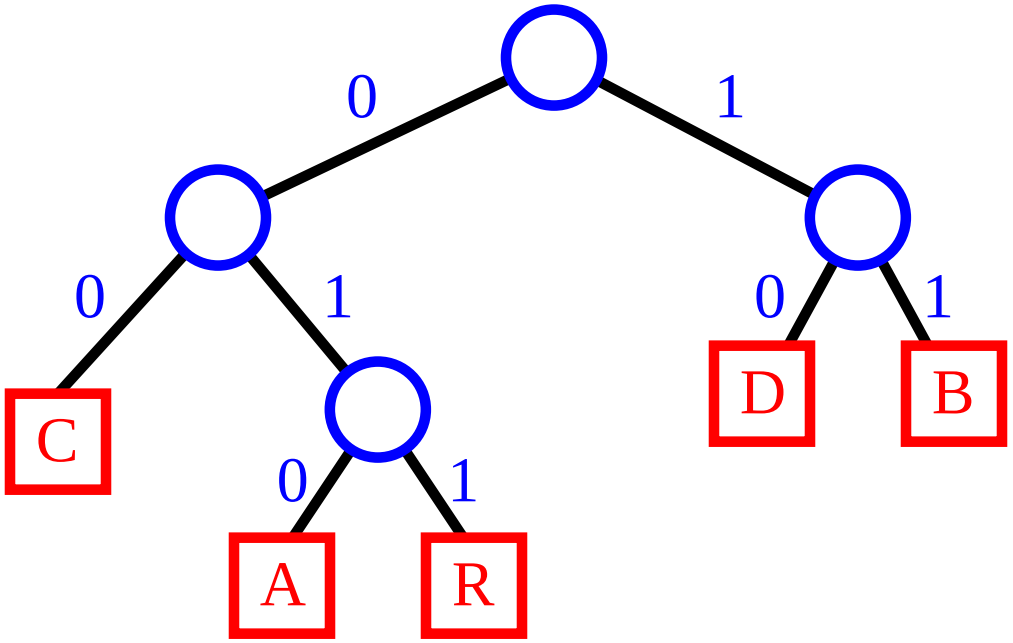
Encoding Trie

- to prevent ambiguities in decoding, we require that the encoding satisfies the **prefix rule**, that is, no code is a prefix of another code
  - a = “0”, j = “11”, v = “10” satisfies the prefix rule
  - a = “0”, j = “01”, v= “00” does **not** satisfy the prefix rule (the code of a is a prefix of the codes of j and v)
- we use an **encoding trie** to define an encoding that satisfies the prefix rule
  - the characters stored at the external nodes
  - a left child (edge) means 0
  - a right child (edge) means 1



Example of Decoding

- trie:



A = 010

B = 11

C = 00

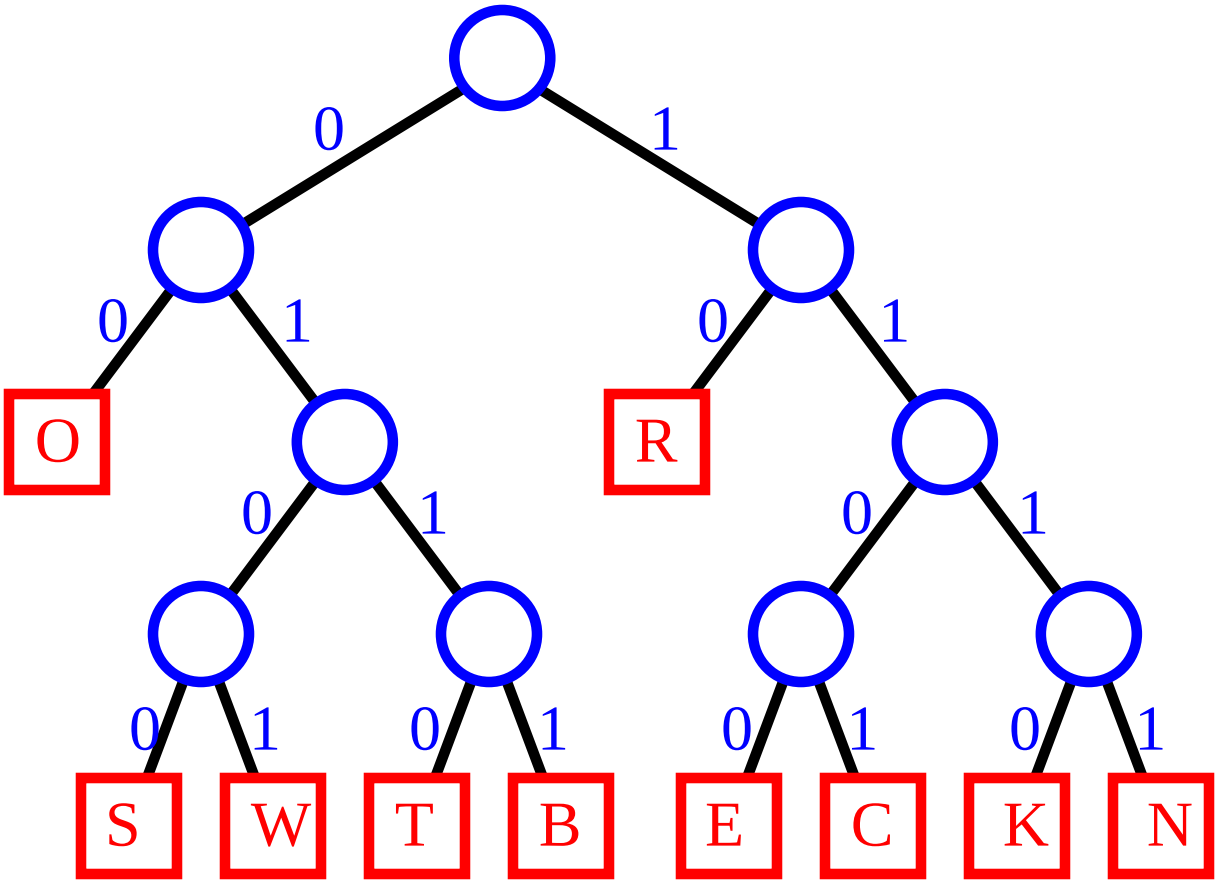
D = 10

R = 011

- encoded text:
  01011011010000101001011011010

- text:

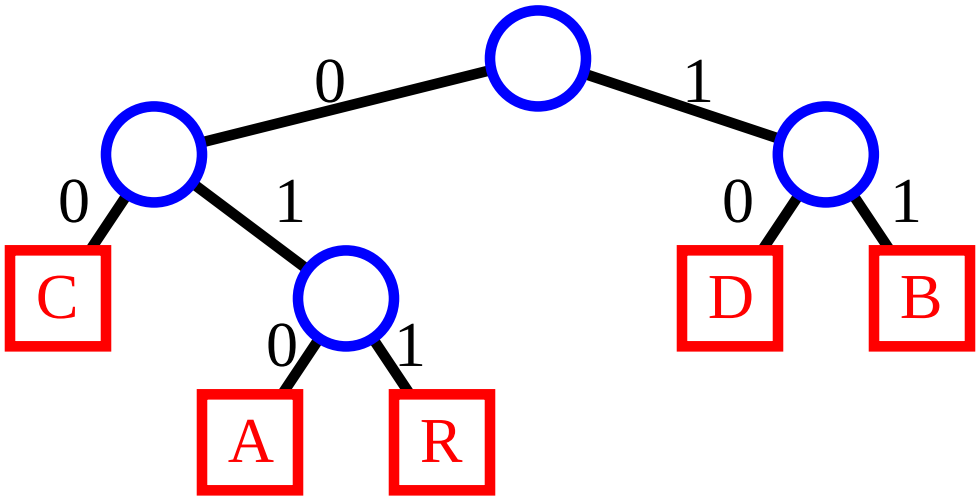
Trie this!



10000111110010011000111011110001010100110101 00

Optimal Compression

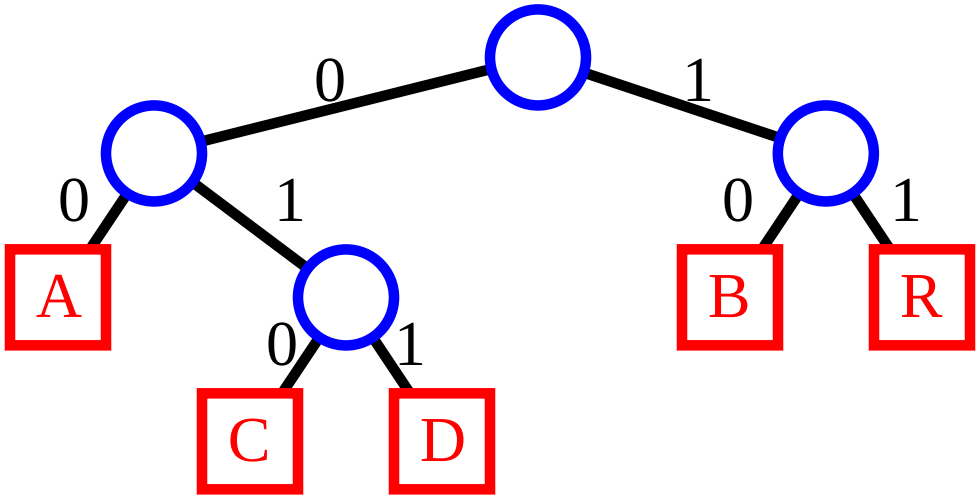
- An issue with encoding tries is to insure that the encoded text is as short as possible:



ABRACADABRA

01011011010000101001011011010

29 bits

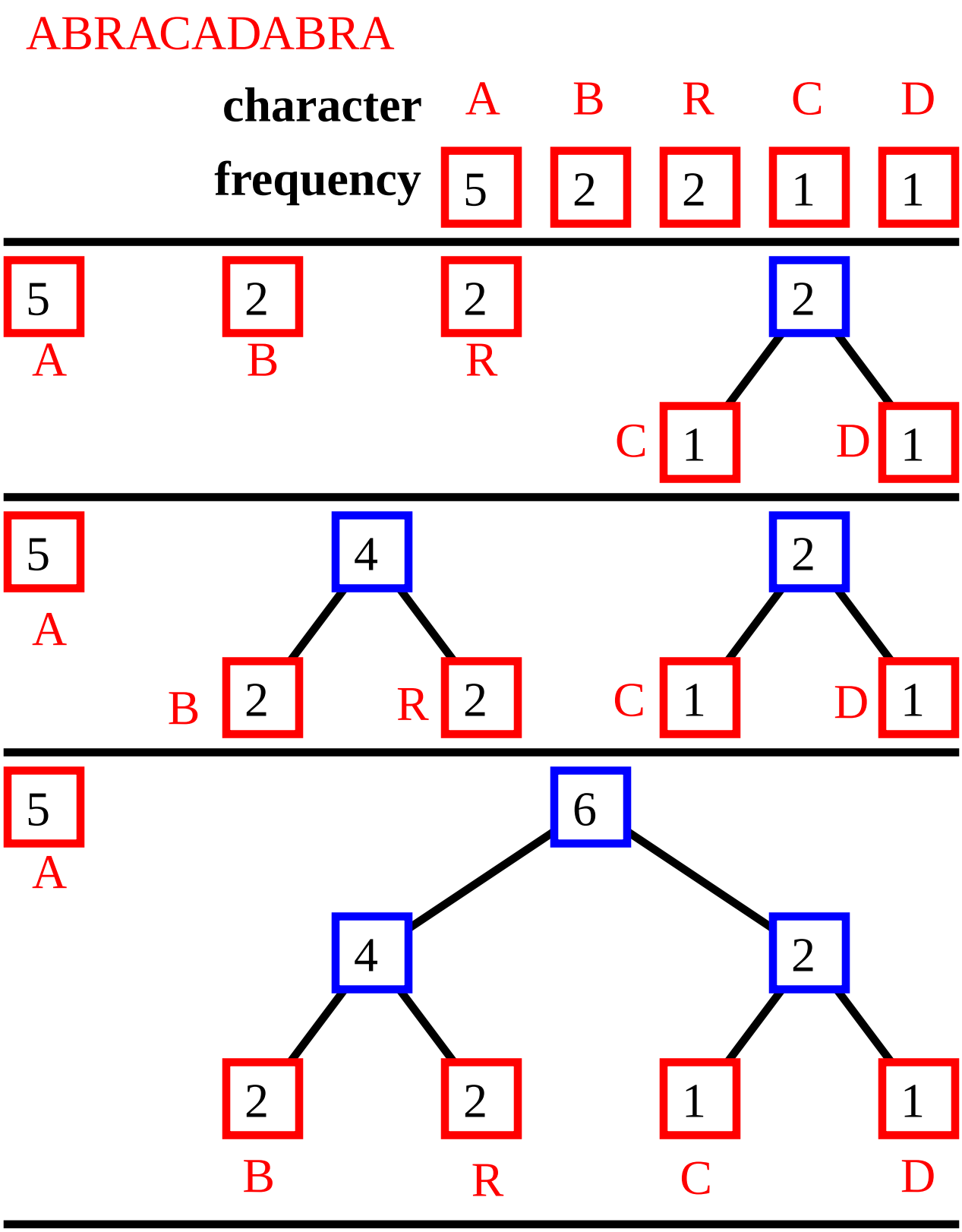


ABRACADABRA

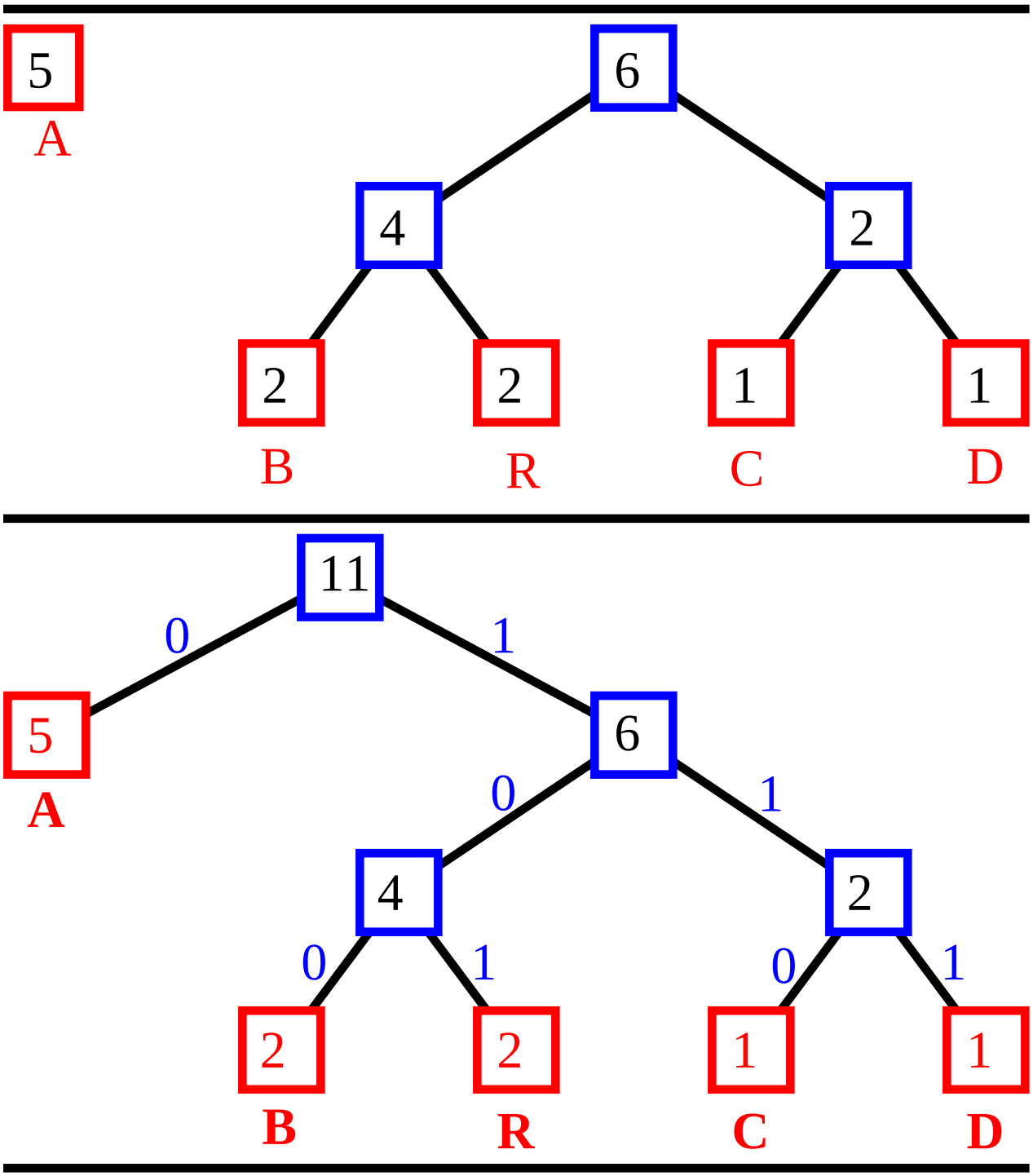
00101100010000110010110 0

24 bits

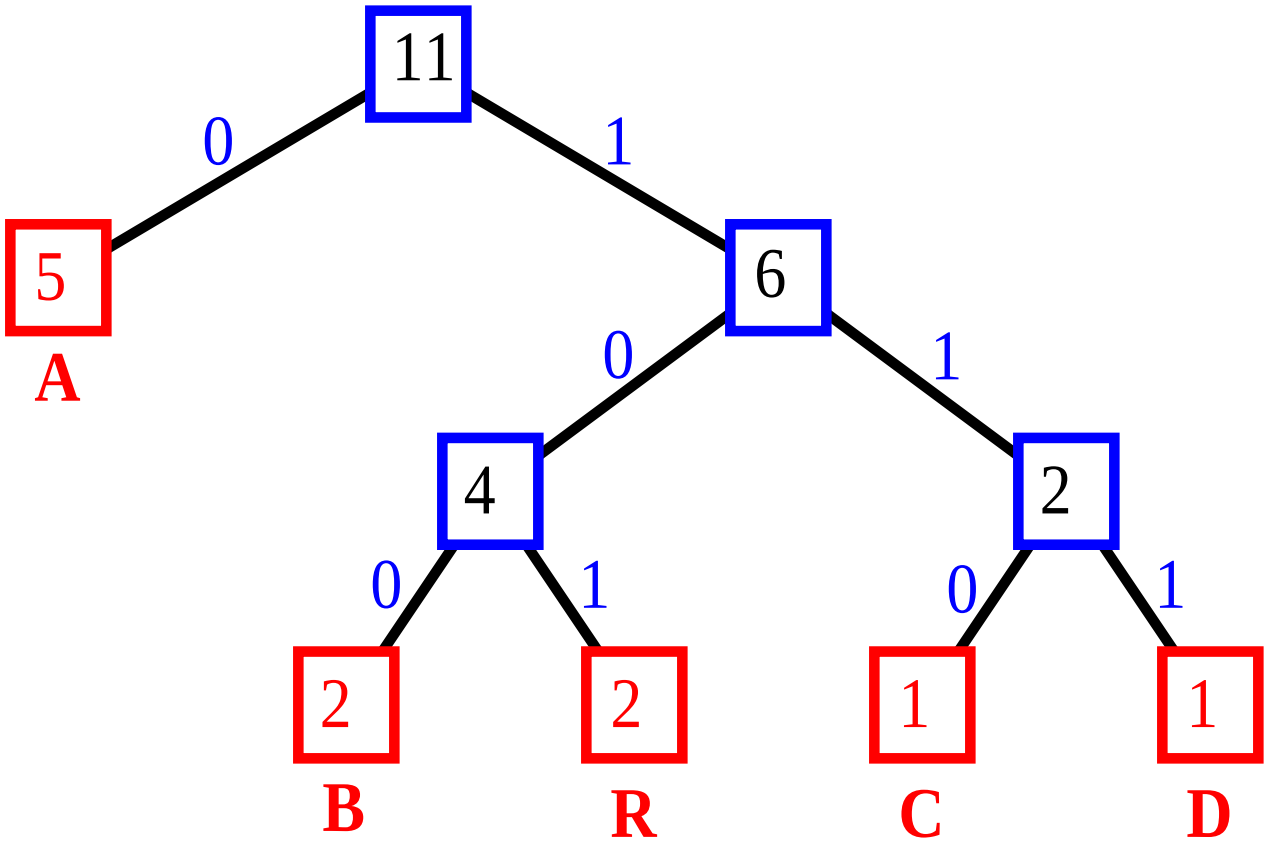
Huffman Encoding Trie



Huffman Encoding Trie (contd.)



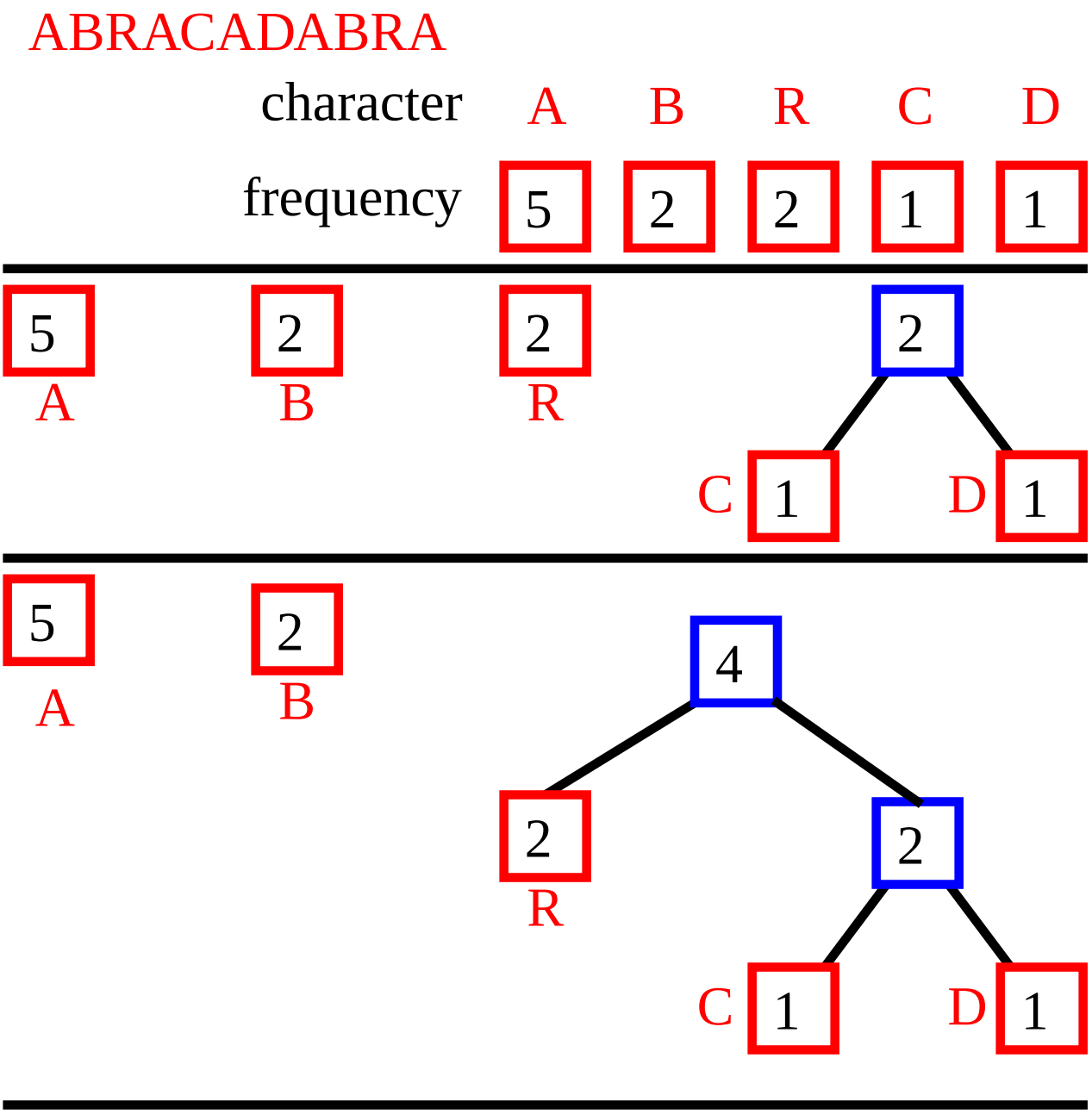
Final Huffman Encoding Trie



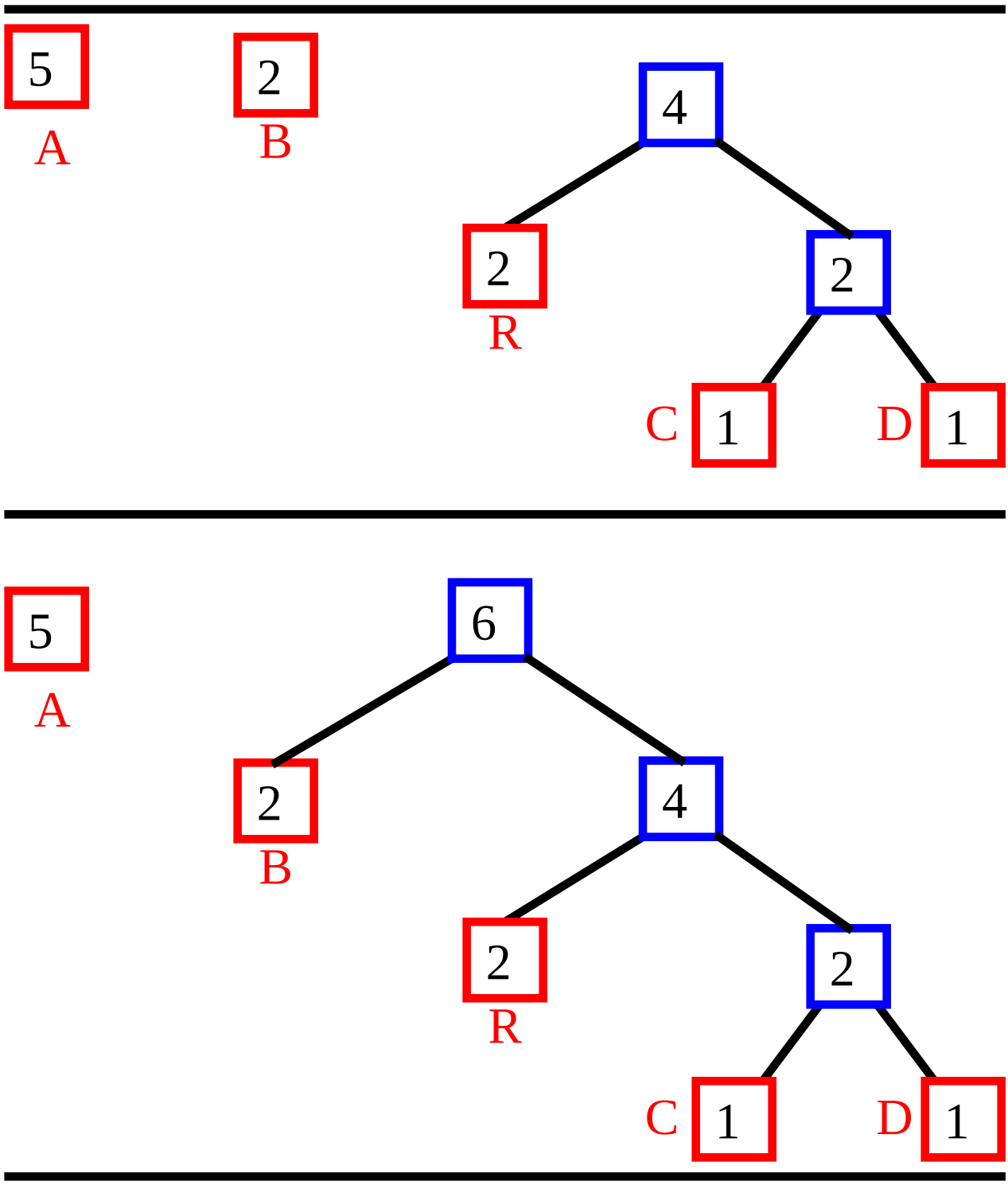
A	B	R	A	C	A	D	A	B	R	A
0	100	101	0	110	0	111	0	100	101	0

23 bits

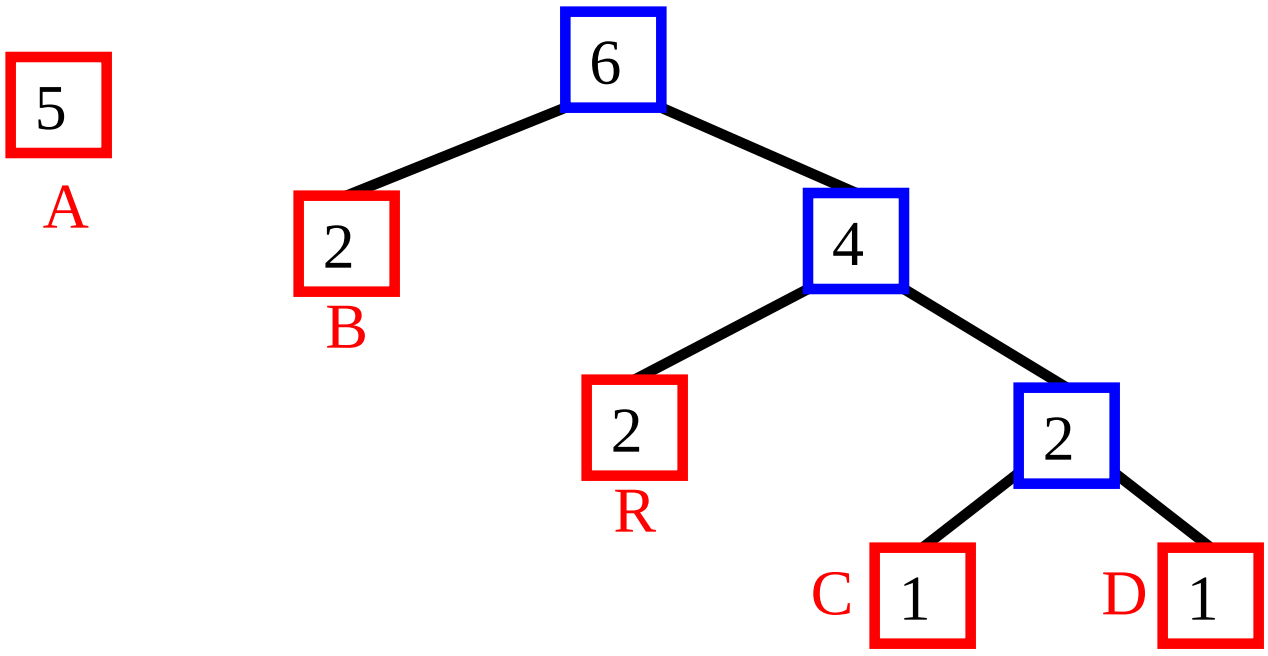
Another Huffman Encoding Trie

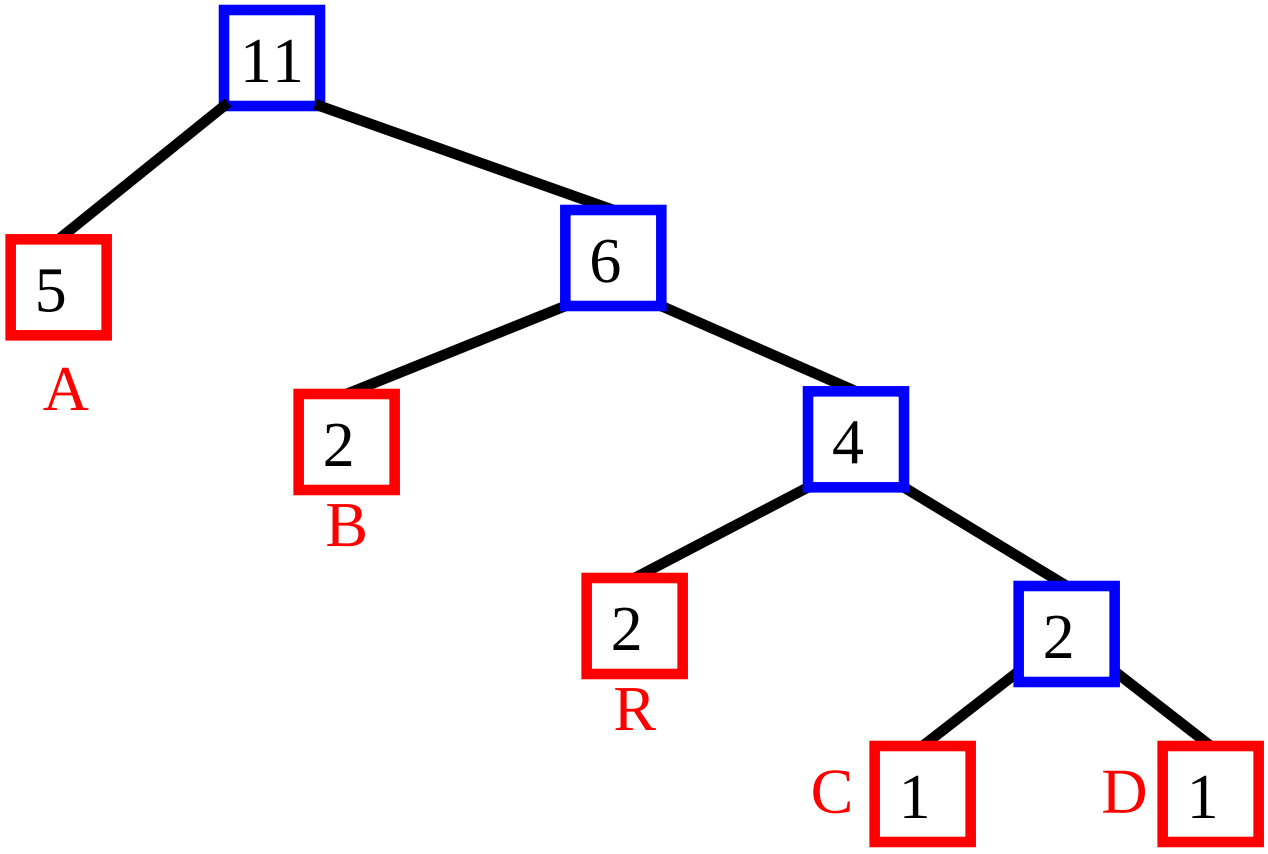


Another Huffman Encoding Trie

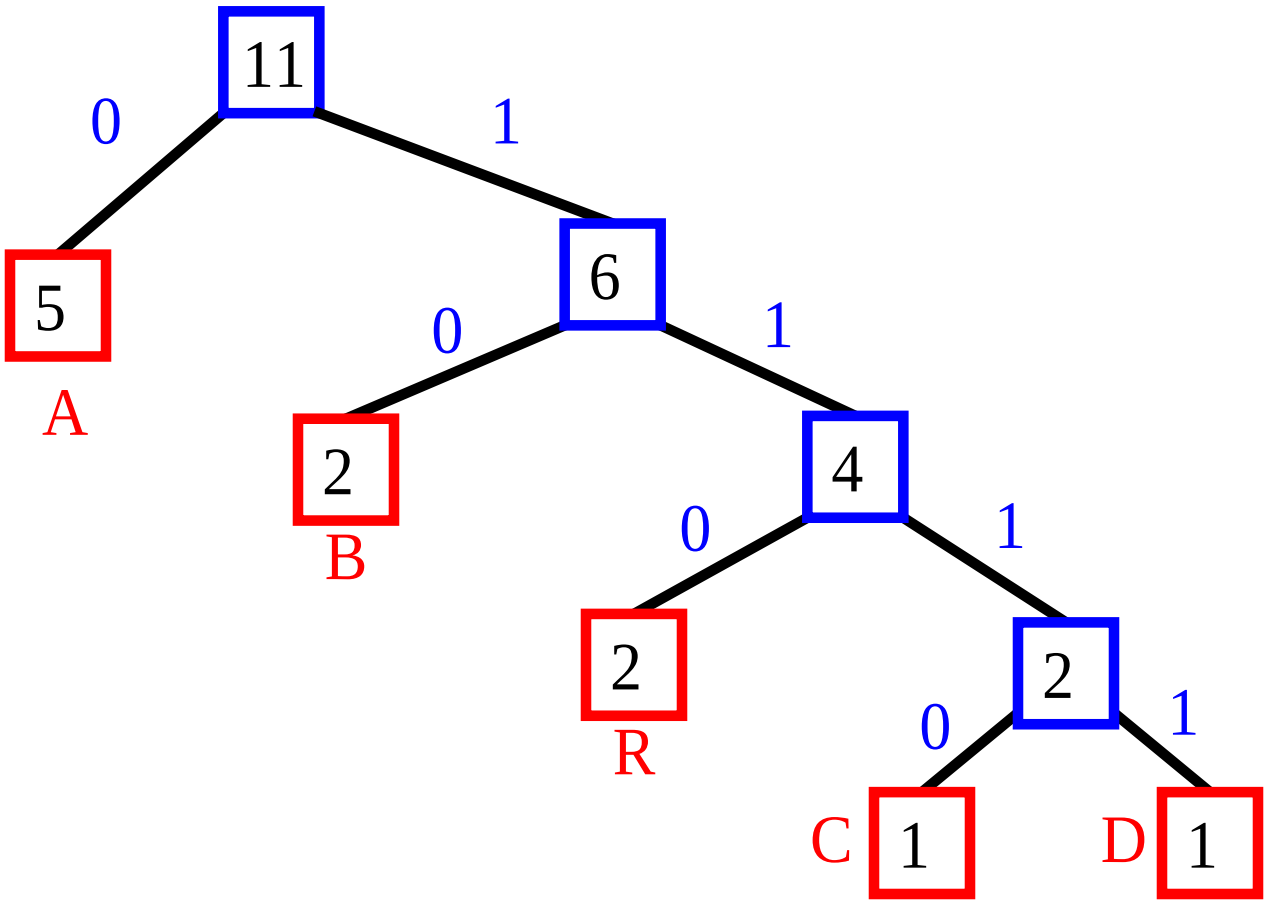


Another Huffman Encoding Trie





Another Huffman Encoding Trie



A	B	R	A	C	A	D	A	B	R	A
0	10	110	0	1100	0	1111	0	10	110	0

23 bits

Construction Algorithm

- with a Huffman encoding trie, the encoded text has minimal length

Algorithm Huffman(X):
  Input: String X of length n
  Output: Encoding trie for X
  Compute the frequency $f(c)$ of each character c of X.
  Initialize a priority queue Q.
  for each character c in X **do**
    Create a single-node tree T storing c
    Q.insertItem($f(c)$, T)
  while Q.size() > 1 **do**
    $f_1 \leftarrow Q$.minKey()
    $T_1 \leftarrow Q$.removeMinElement()
    $f_2 \leftarrow Q$.minKey()
    $T_2 \leftarrow Q$.removeMinElement()
    Create a new tree T with left subtree T_1 and right subtree T_2.
    Q.insertItem($f_1 + f_2$, T)
  return tree Q.removeMinElement()

- runing time for a text of length n with k distinct characters: O(n + k log k)

- typically, k is O(1) (e.g., ASCII characters) and the algorithm runs in O(n) time.

Image Compression

- we can use Huffman encoding also for binary files (bitmaps, executables, etc.)
- common groups of bits are stored at the leaves
- Example of an encoding suitable for b/w bitmaps