The Wavelet Tree data structure

Reminder: rank/select operations:

Given a string s we define the operations:

- $rank_s(c,i)$: # of occ. of c in s[1,i]
- $select_s(c,i)$: position of i-th c in s

Example, s=mississippi

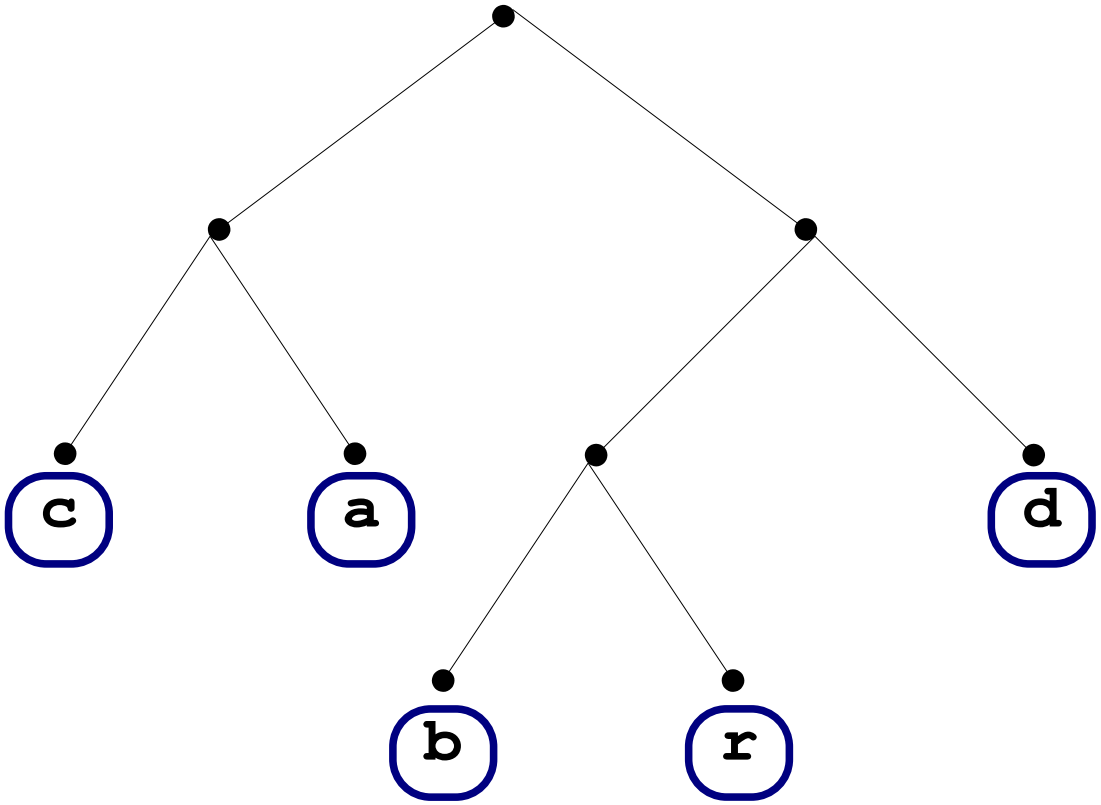
- rank(s,5) = 1
- select(i,3) = 8

Wavelet Trees have been introduced to represent compactly a string supporting rank/select operations.

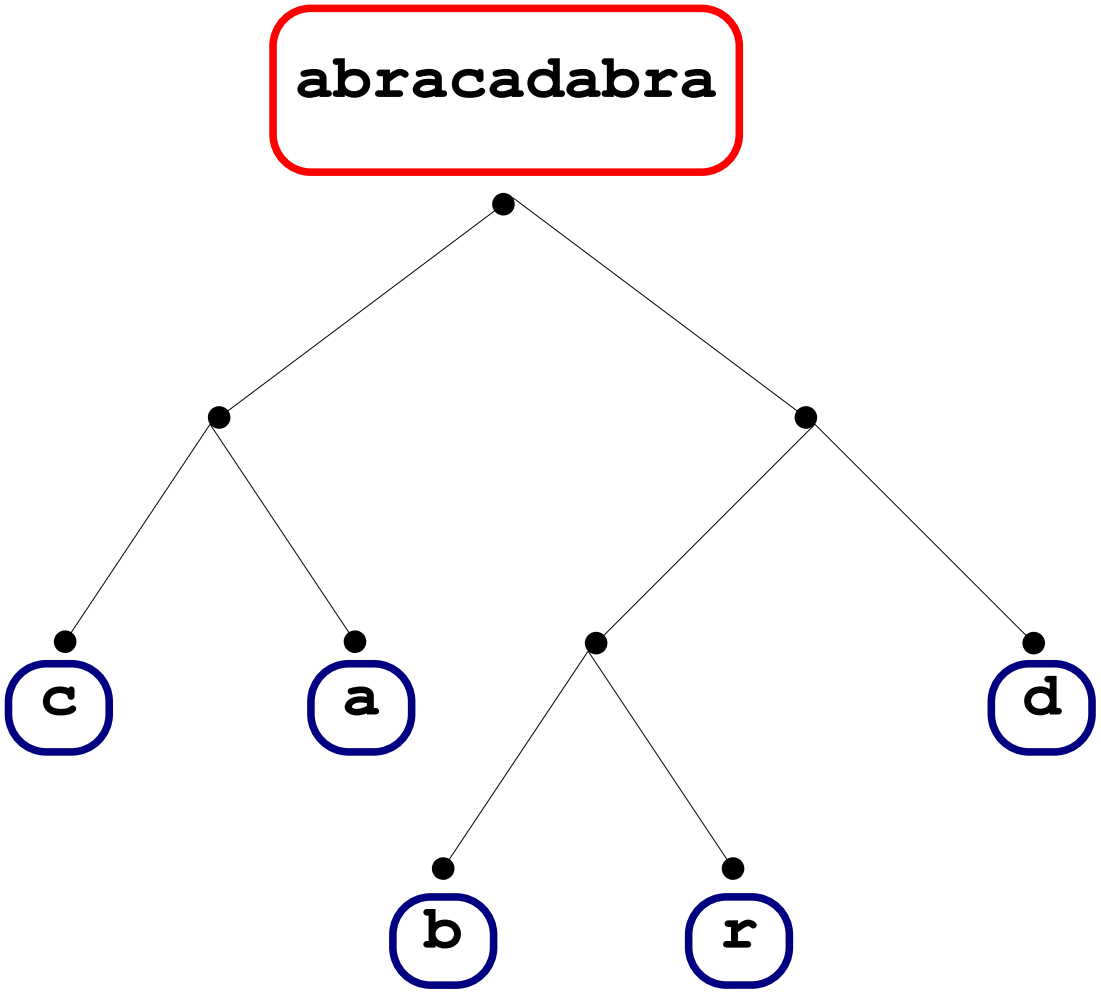
They work assuming we can do rank/select on binary strings,

To build a Wavelet Tree we start with a complete binary tree with a leaf for each alphabet symbol.

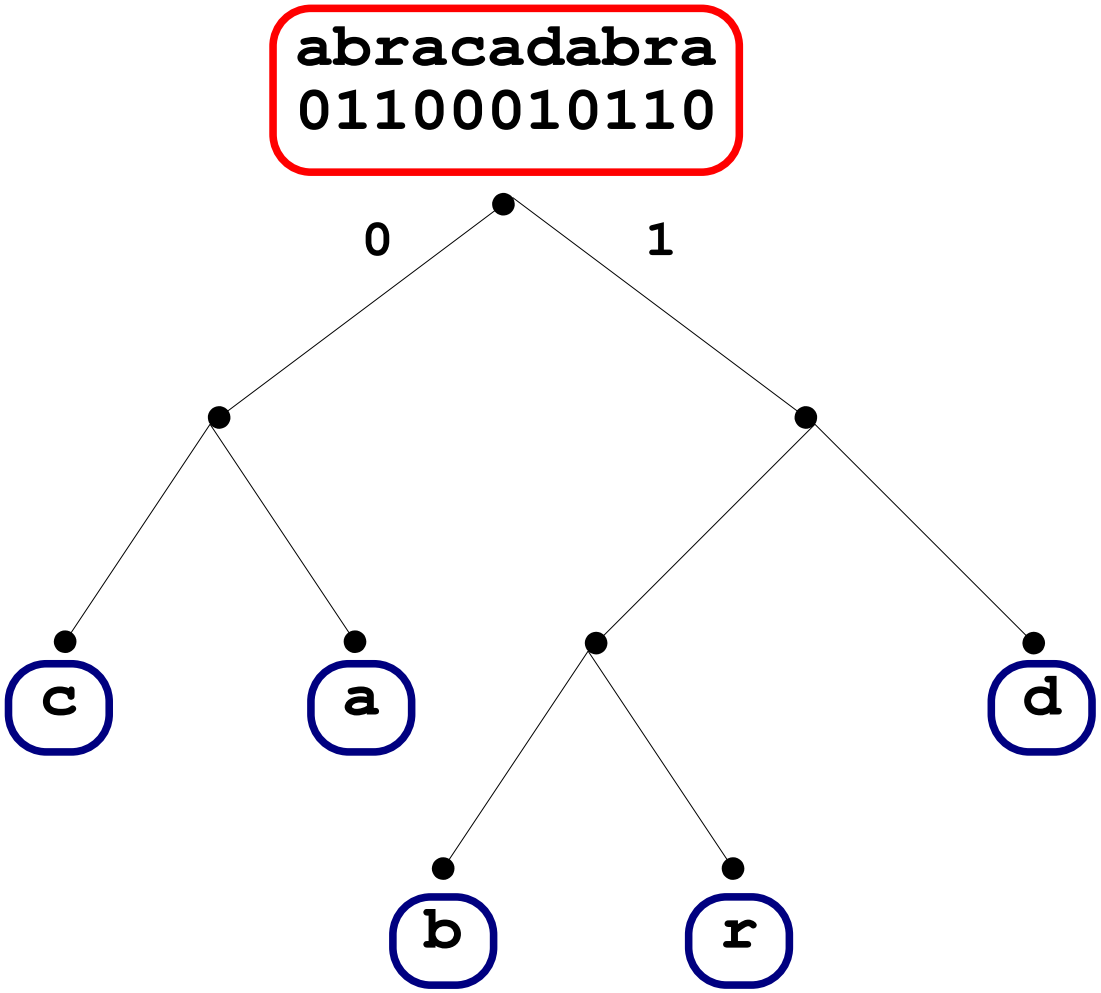
Example. $A = \{a, b, c, d, r\}$



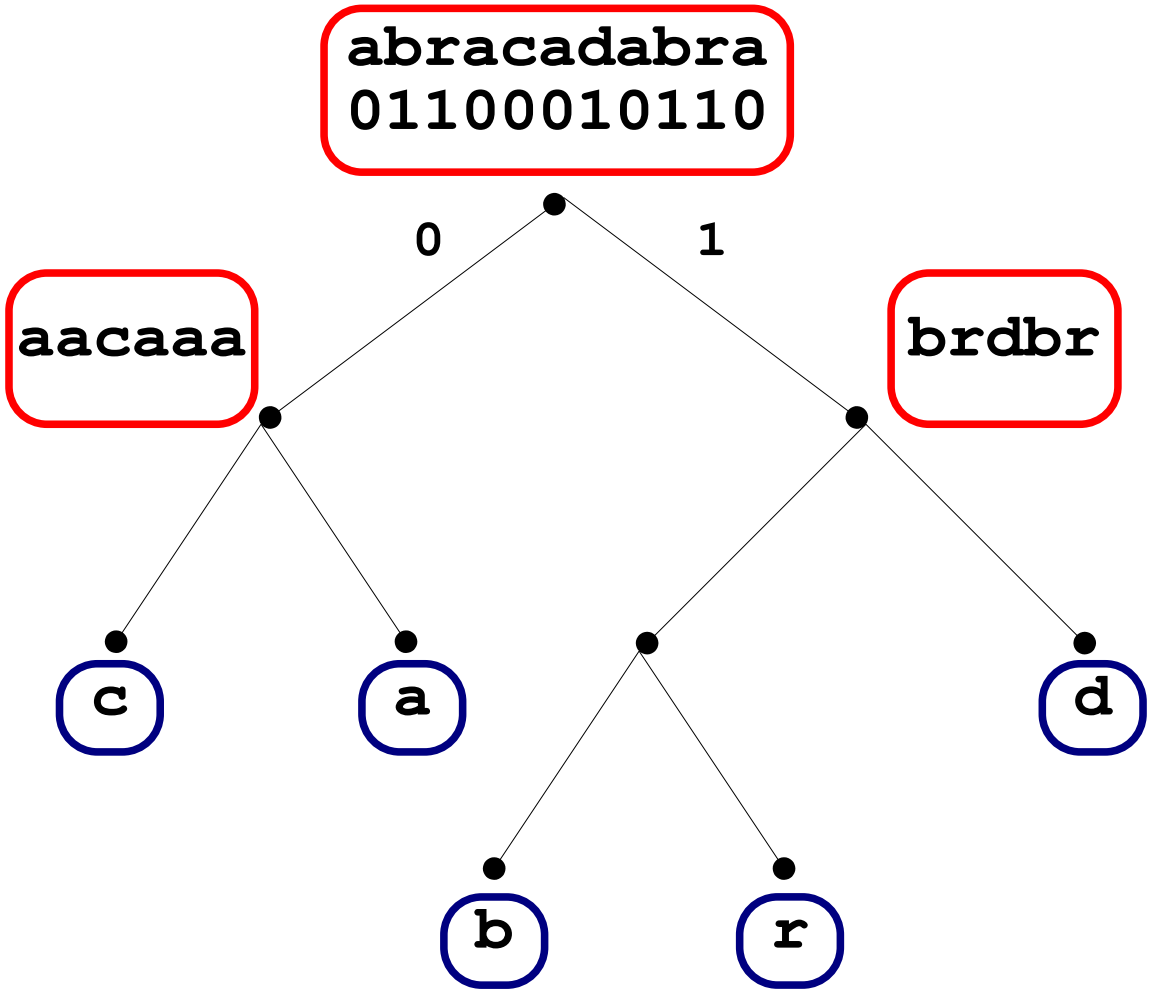
Given the string **abracadabra** we build the corresponding Wavelet Tree as follows:



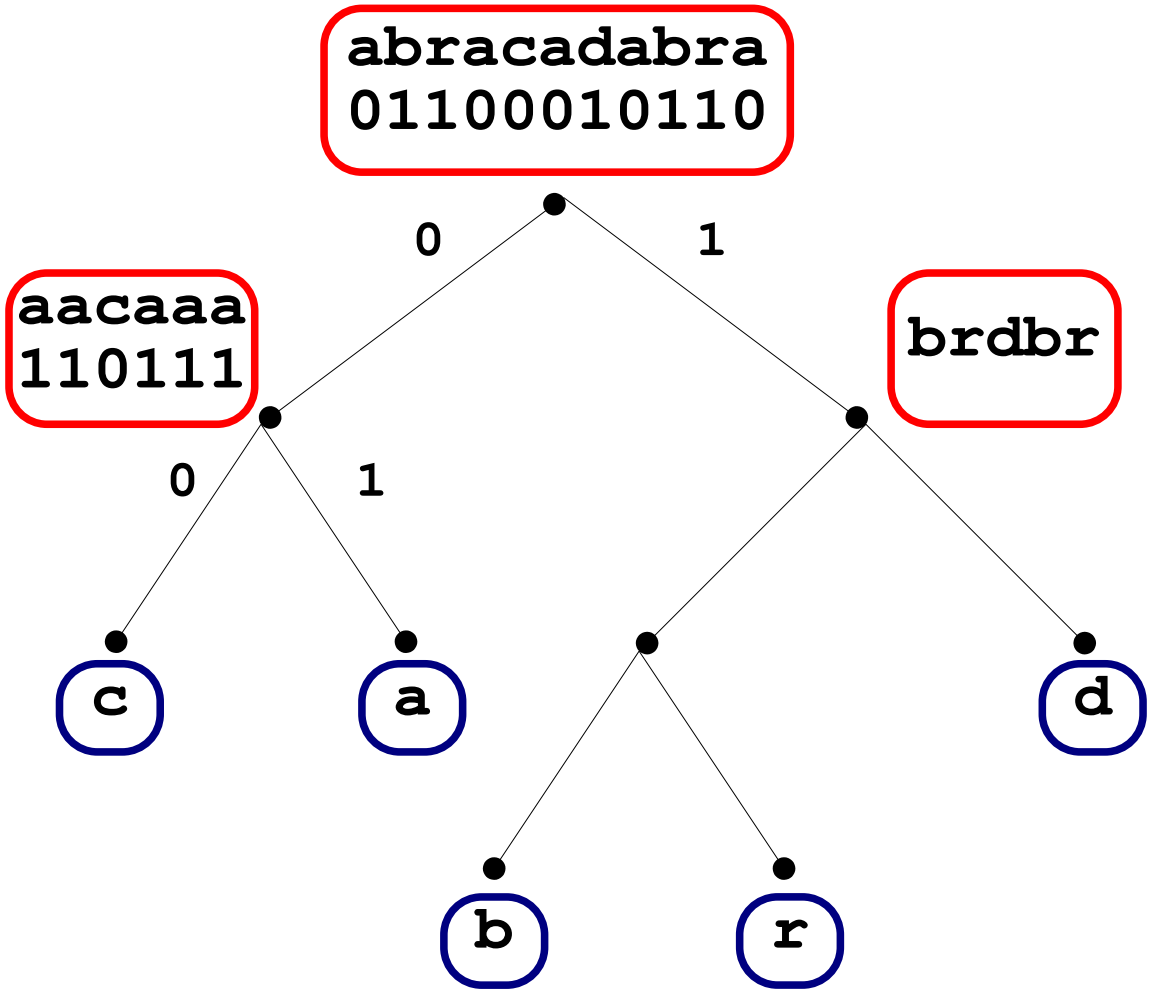
Given the string **abracadabra** we build the corresponding Wavelet Tree as follows:



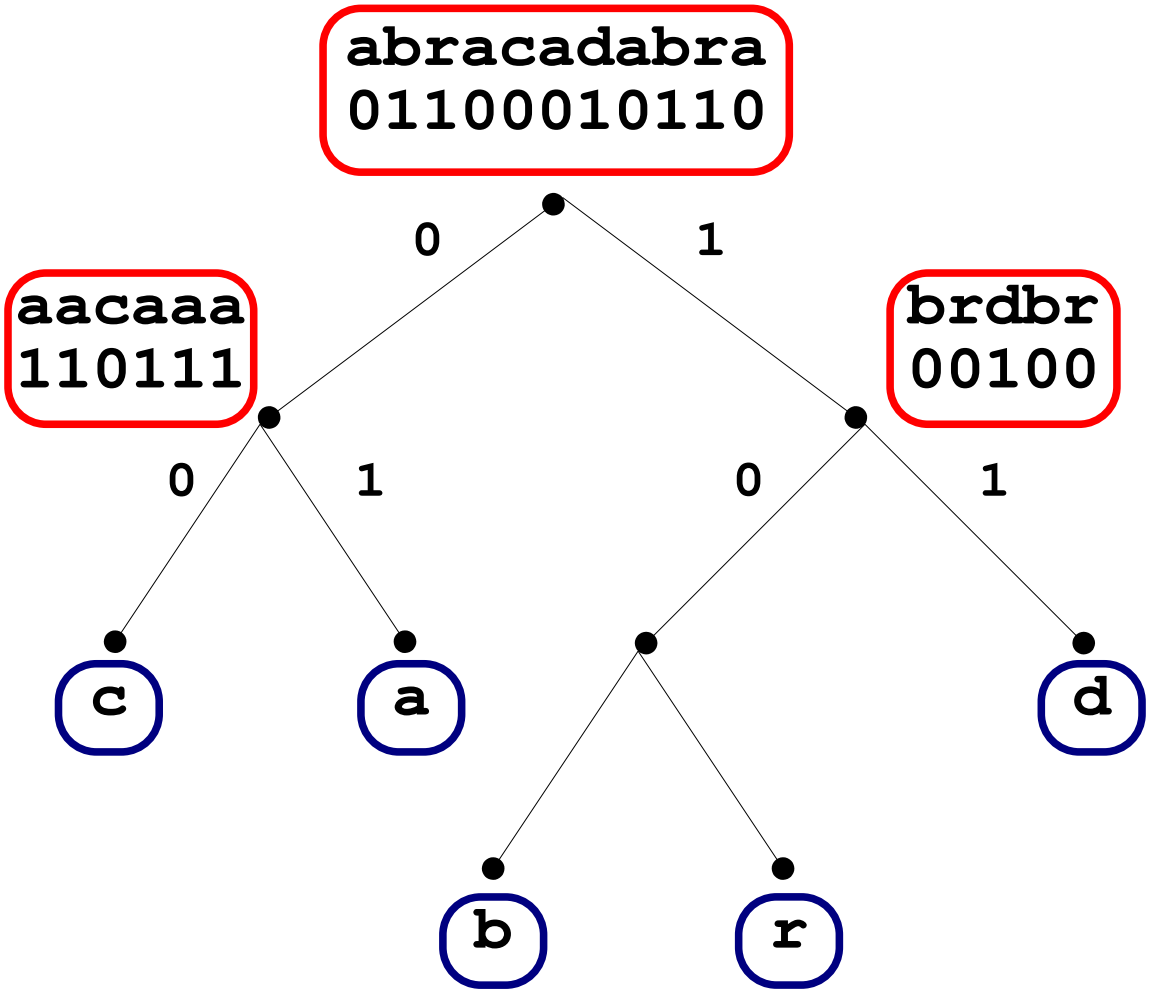
Given the string **abracadabra** we build the corresponding Wavelet Tree as follows:



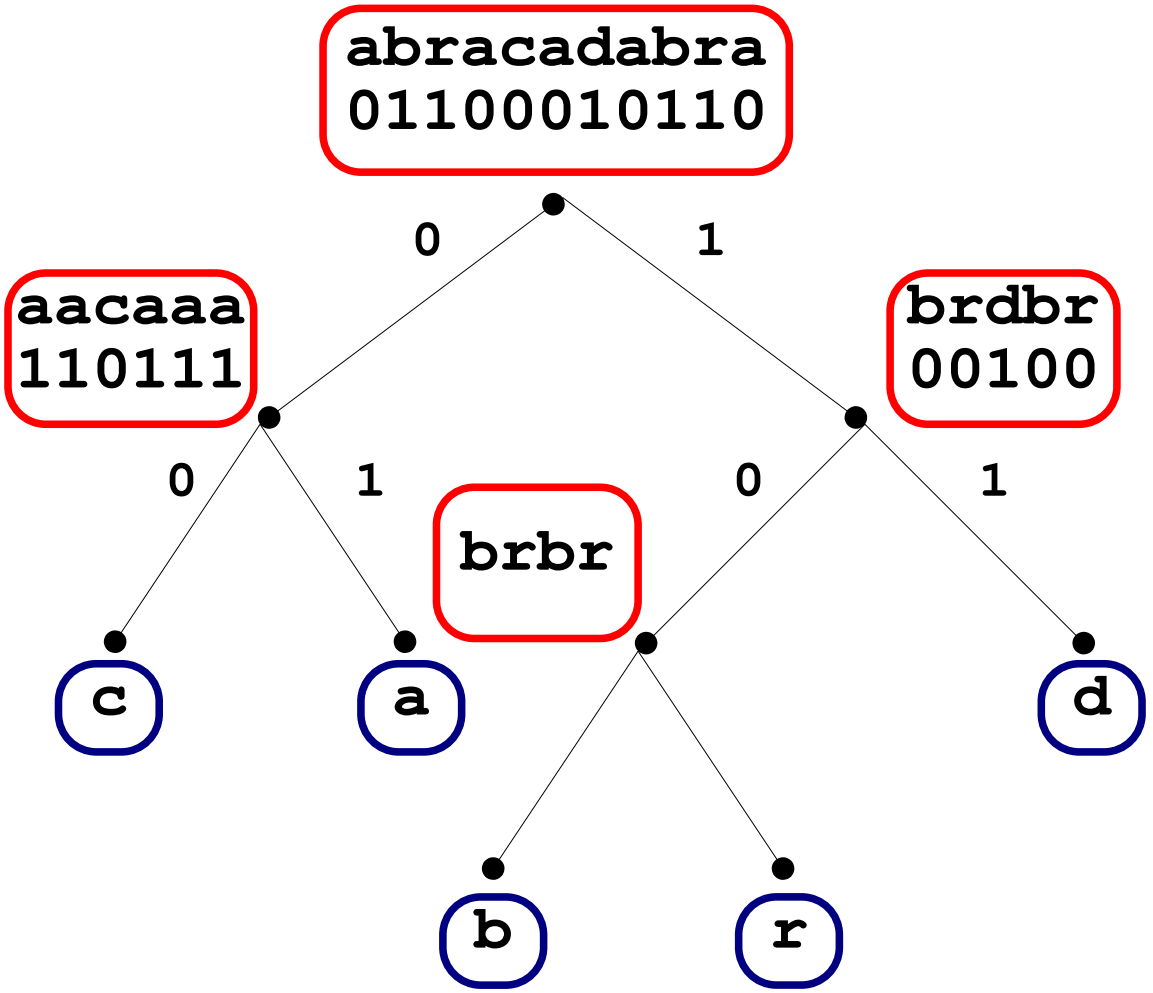
Given the string **abracadabra** we build the corresponding Wavelet Tree as follows:



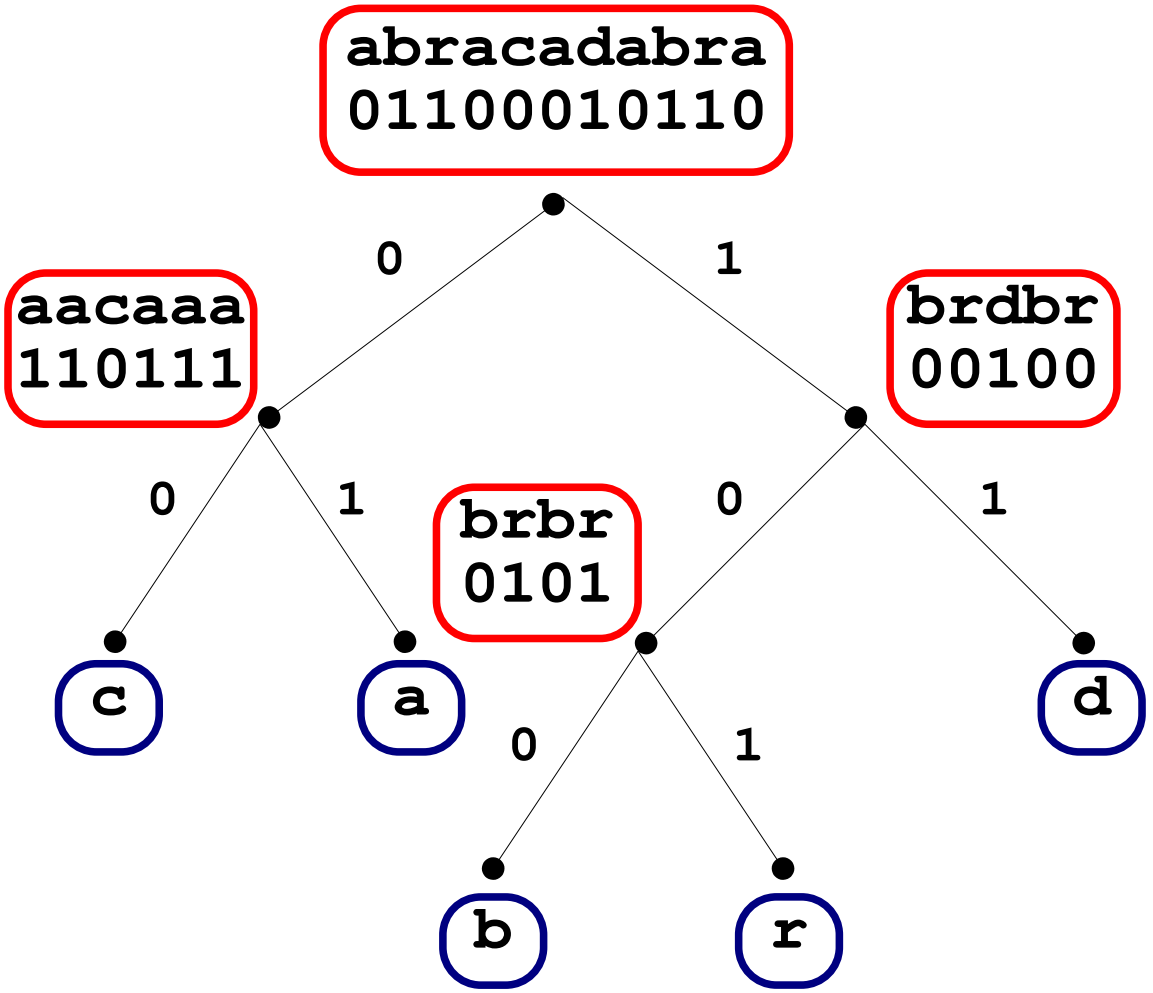
Given the string **abracadabra** we build the corresponding Wavelet Tree as follows:



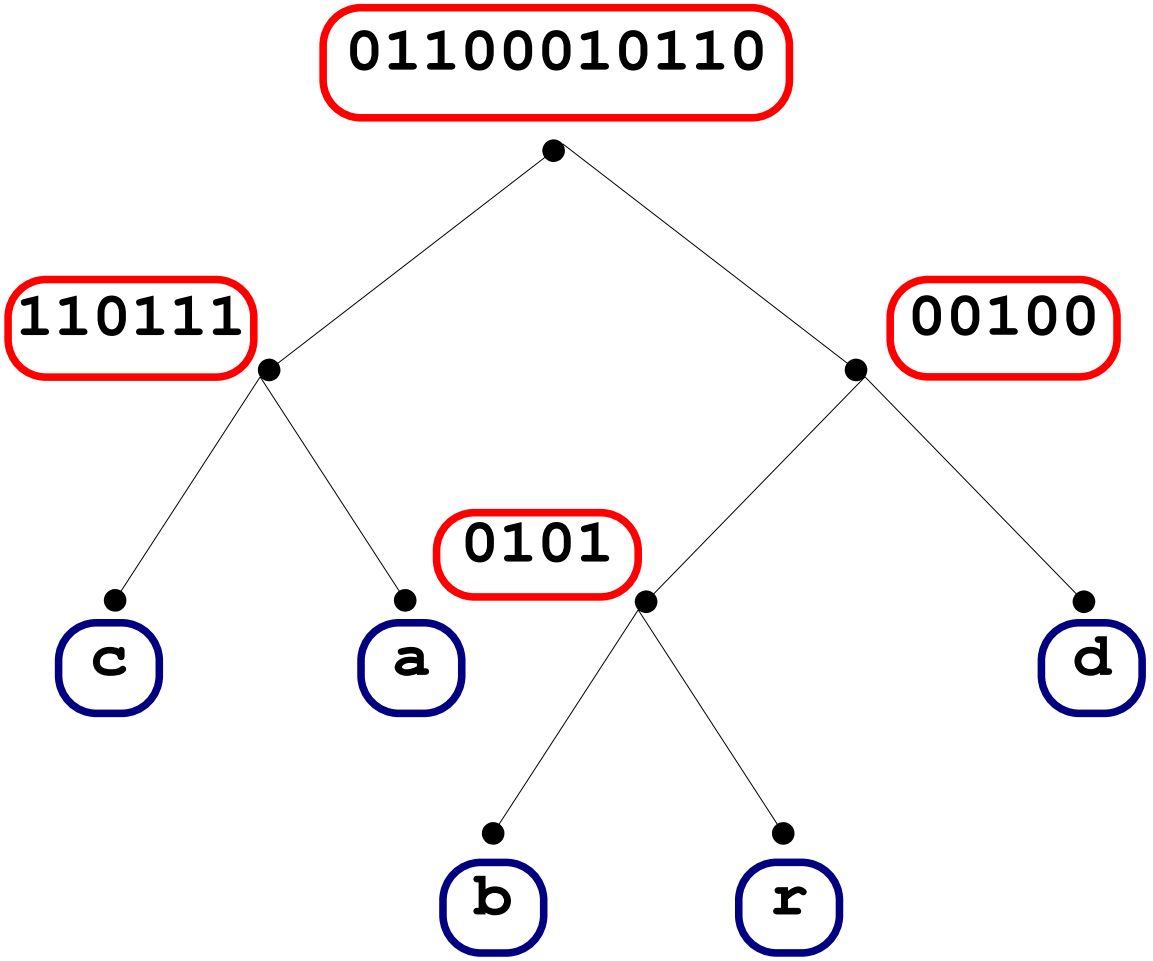
Given the string **abracadabra** we build the corresponding Wavelet Tree as follows:



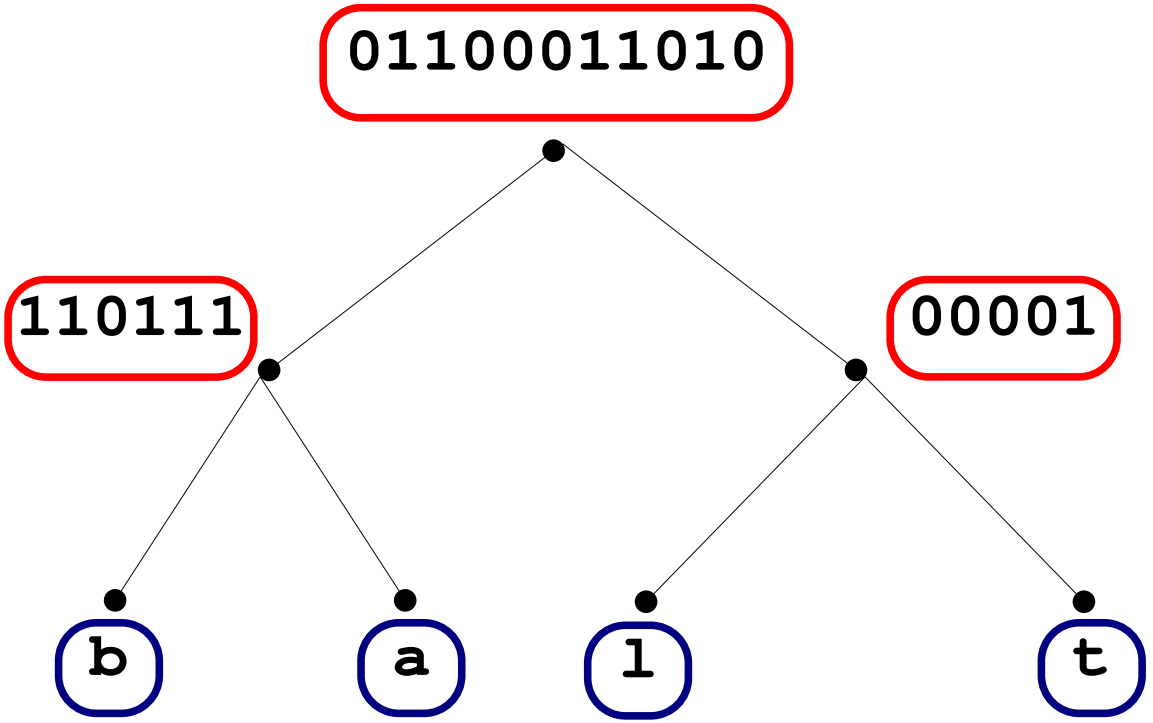
Given the string **abracadabra** we build the corresponding Wavelet Tree as follows:



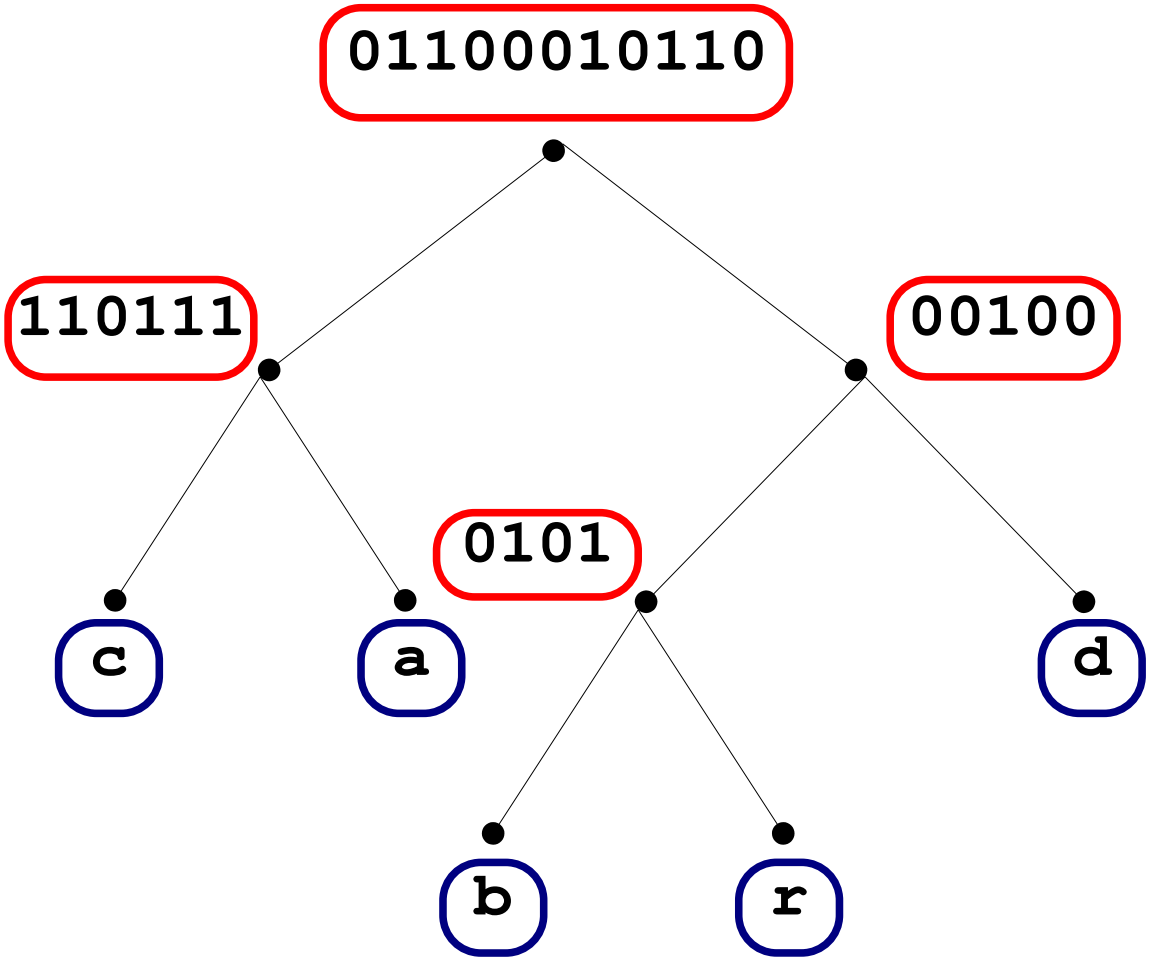
We only store the shape of the tree and the binary strings associated to each node:



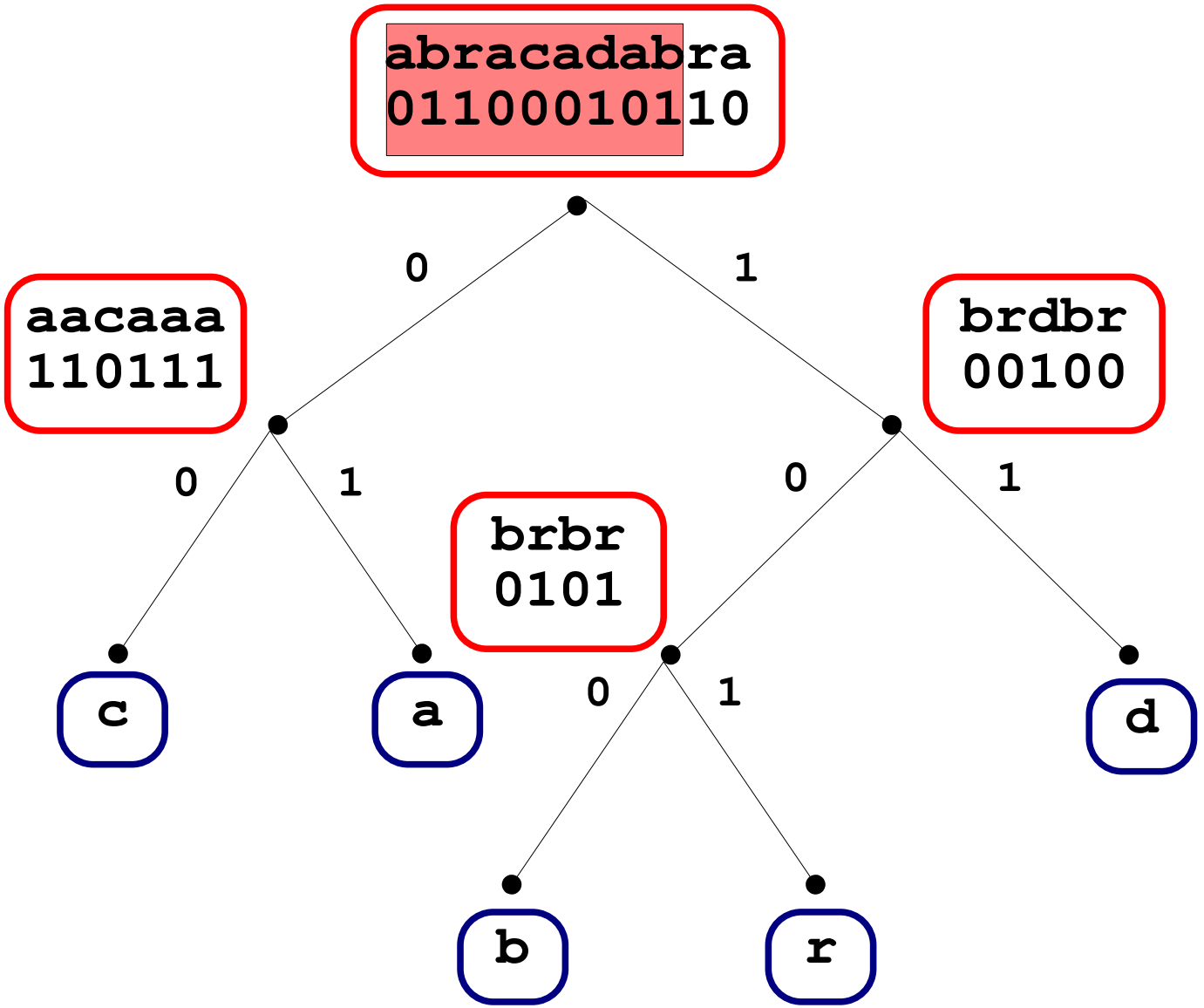
Example: find the string represented by the following Wavelet Tree:



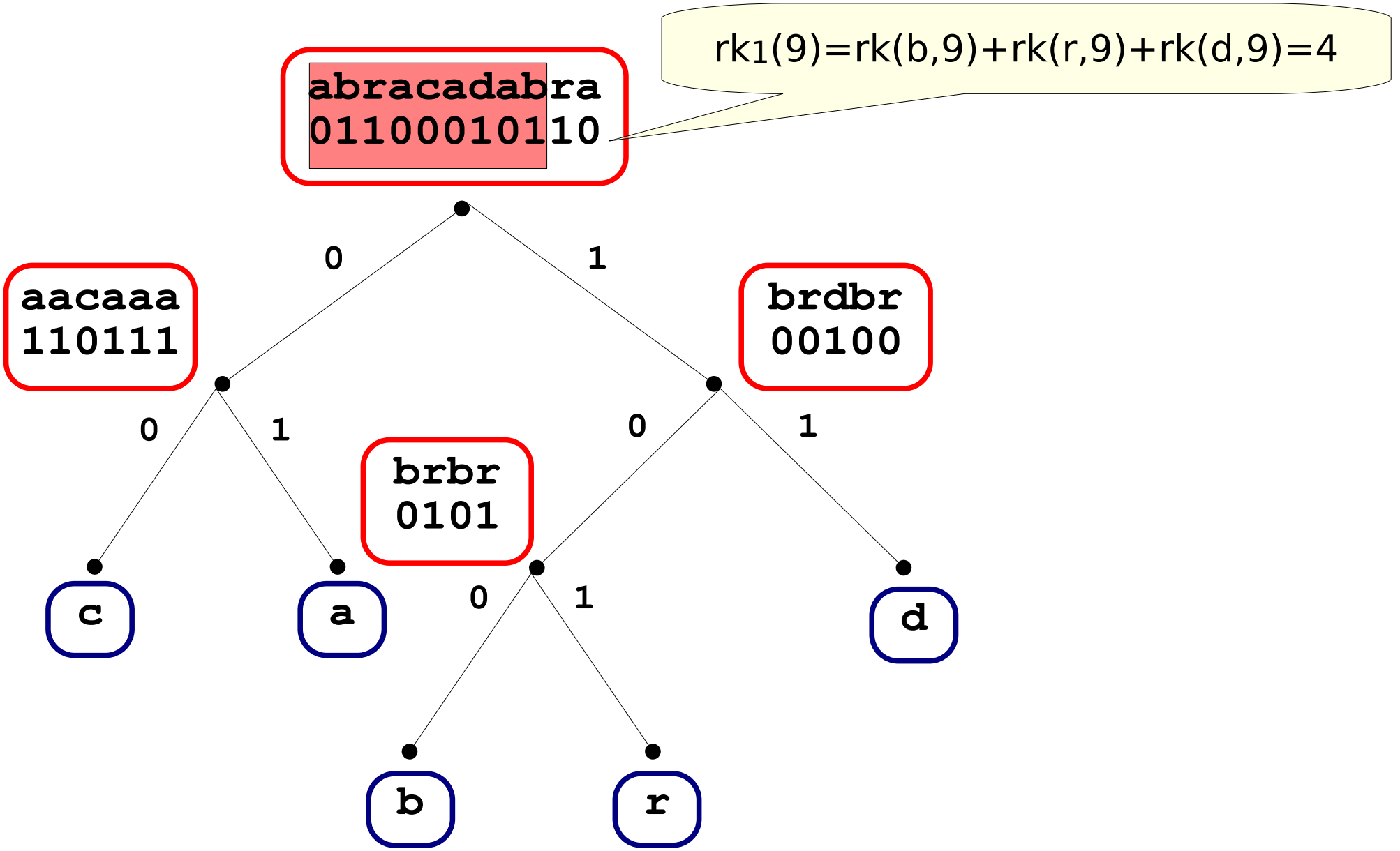
rank/select queries over the original string can be answered via rank/select queries over the binary strings associated to internal nodes.



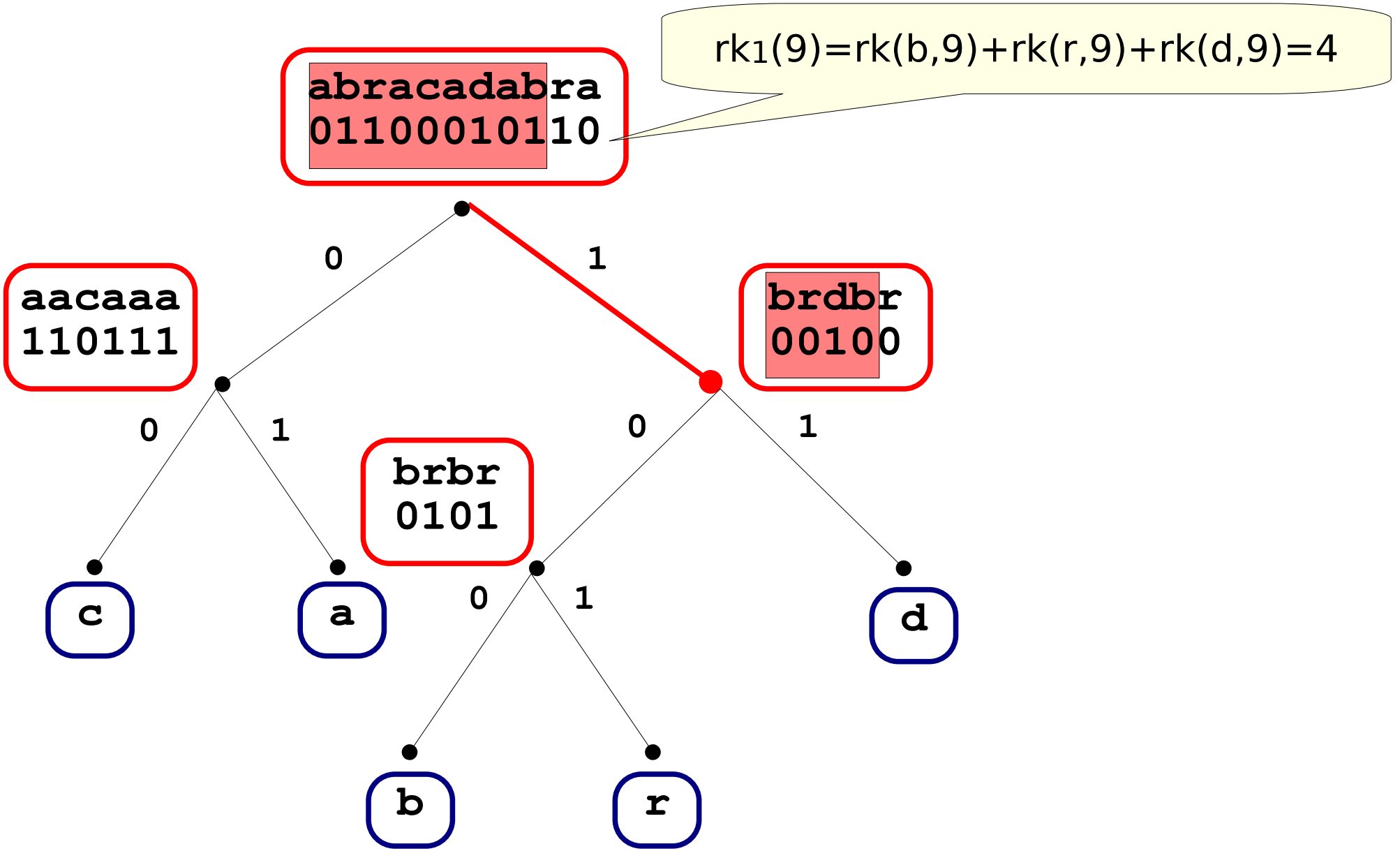
Suppose we want to compute rank(b,9)



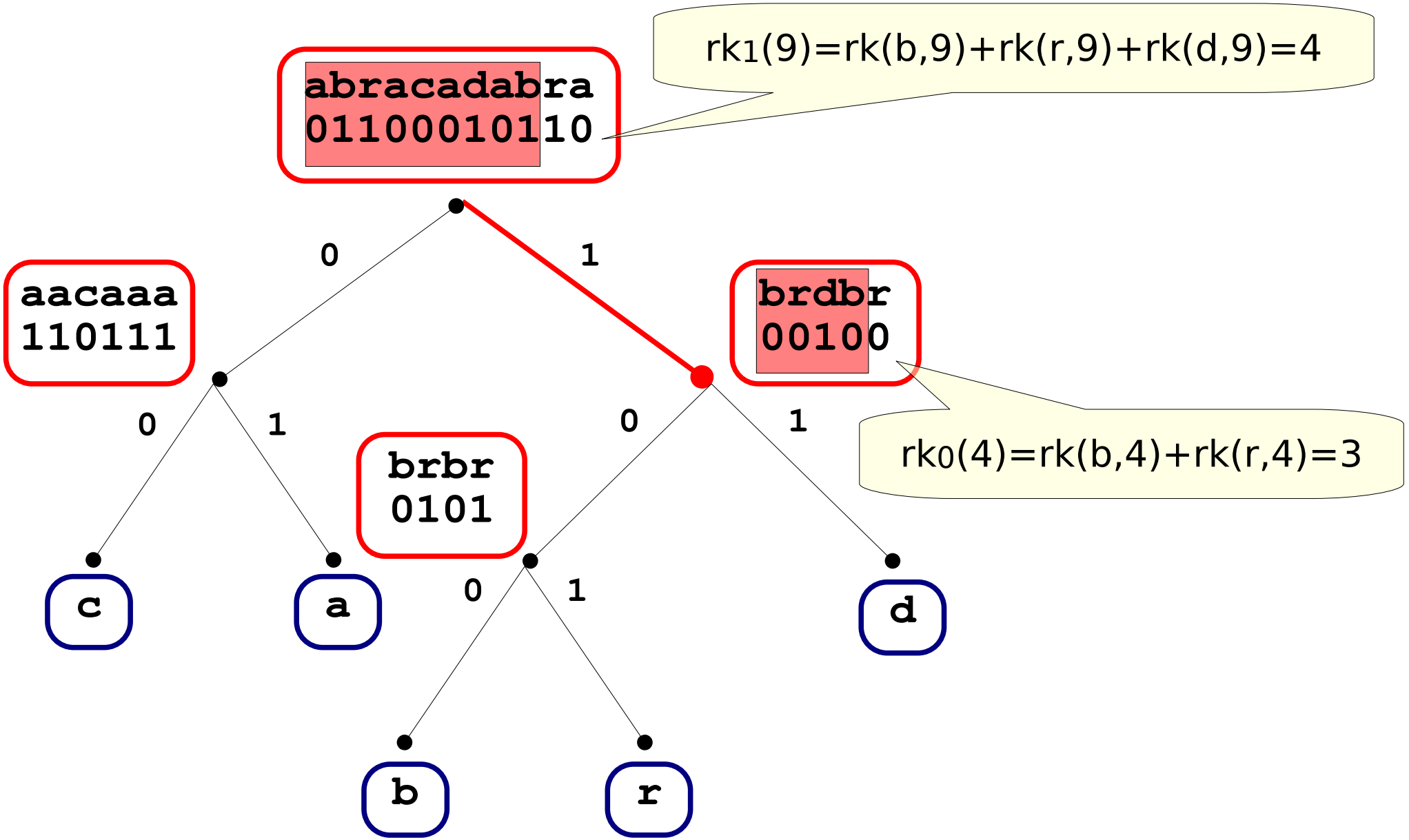
Suppose we want to compute rank(b,9)



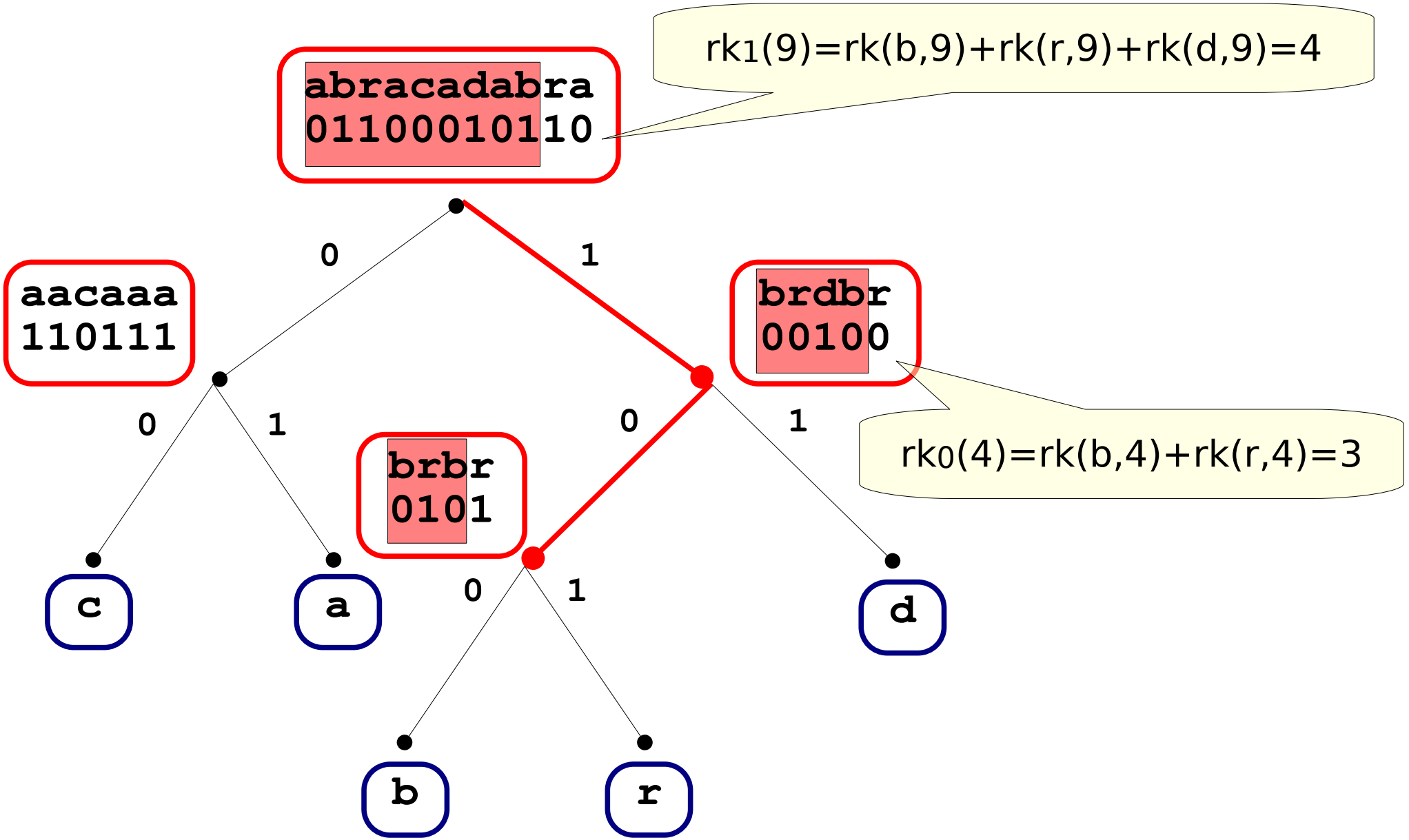
Suppose we want to compute rank(b,9)



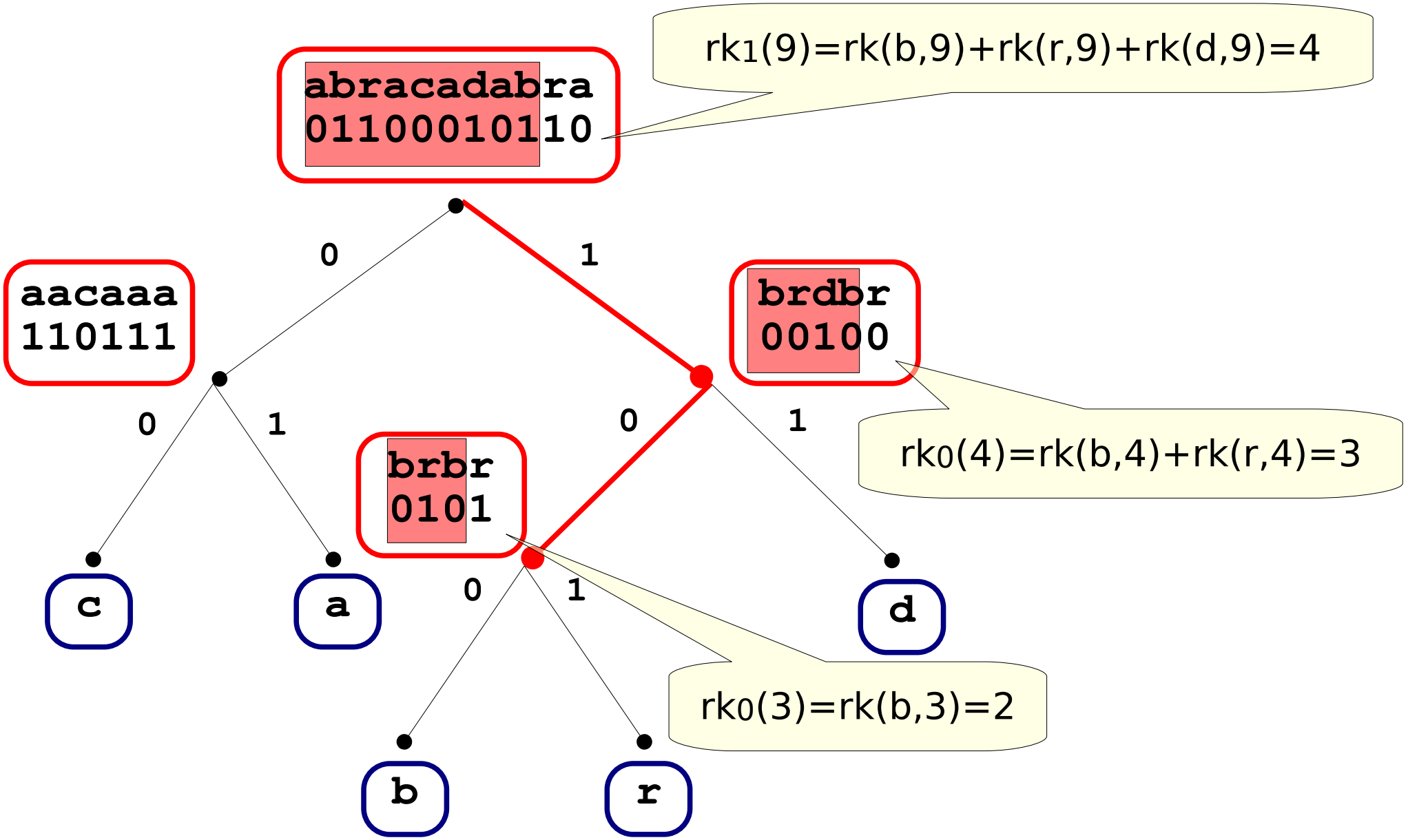
Suppose we want to compute rank(b,9)



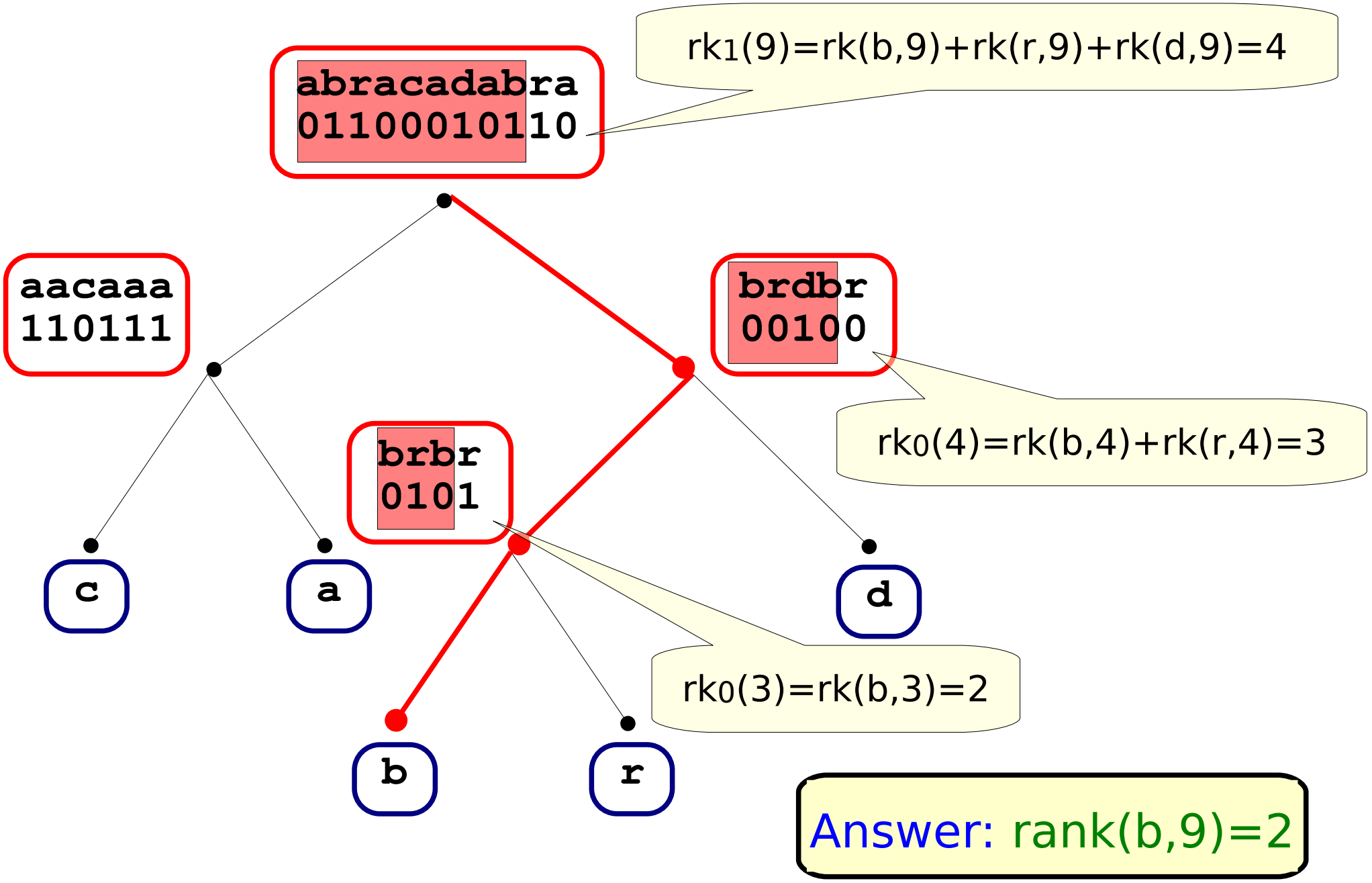
Suppose we want to compute rank(b,9)



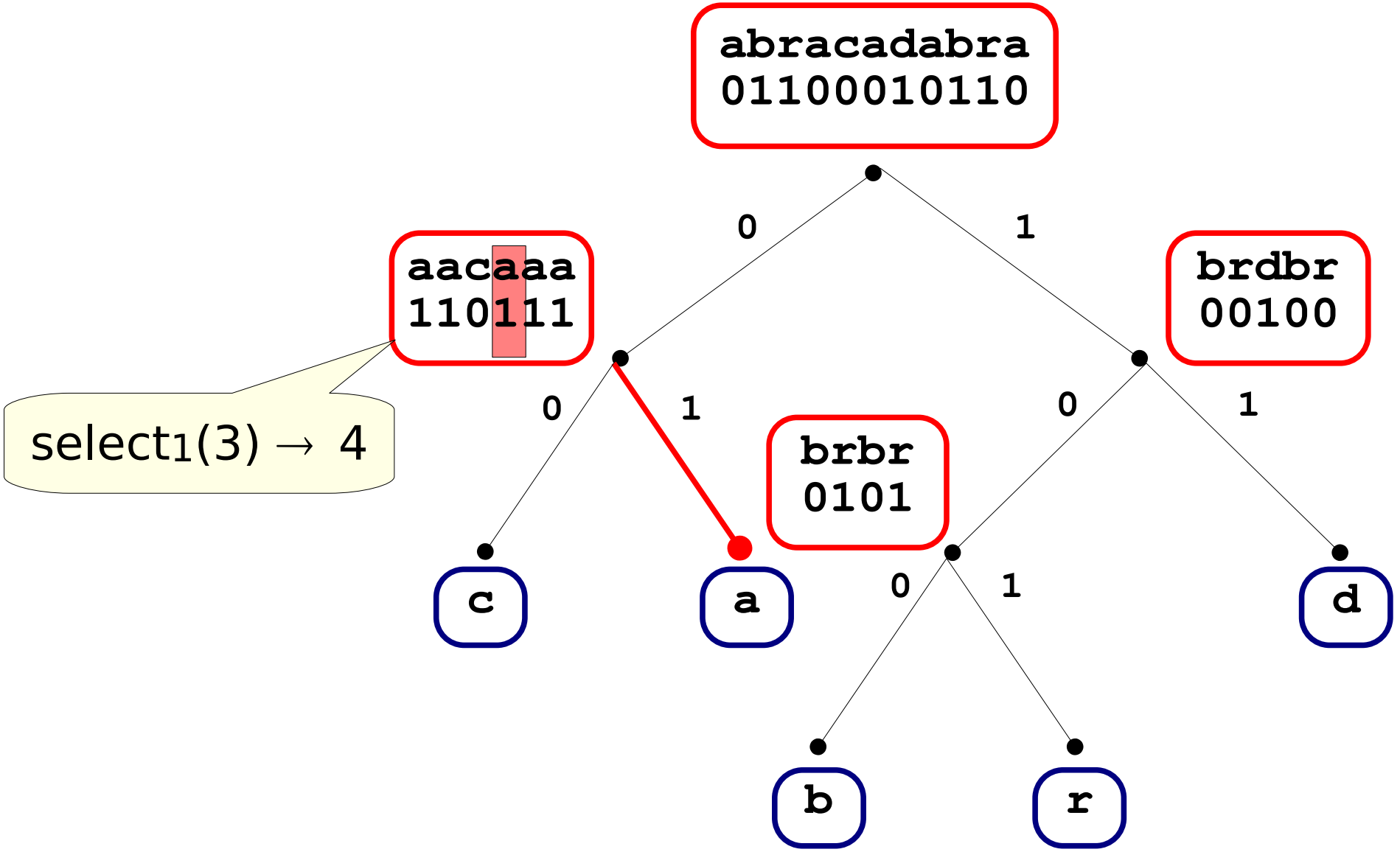
Suppose we want to compute rank(b,9)



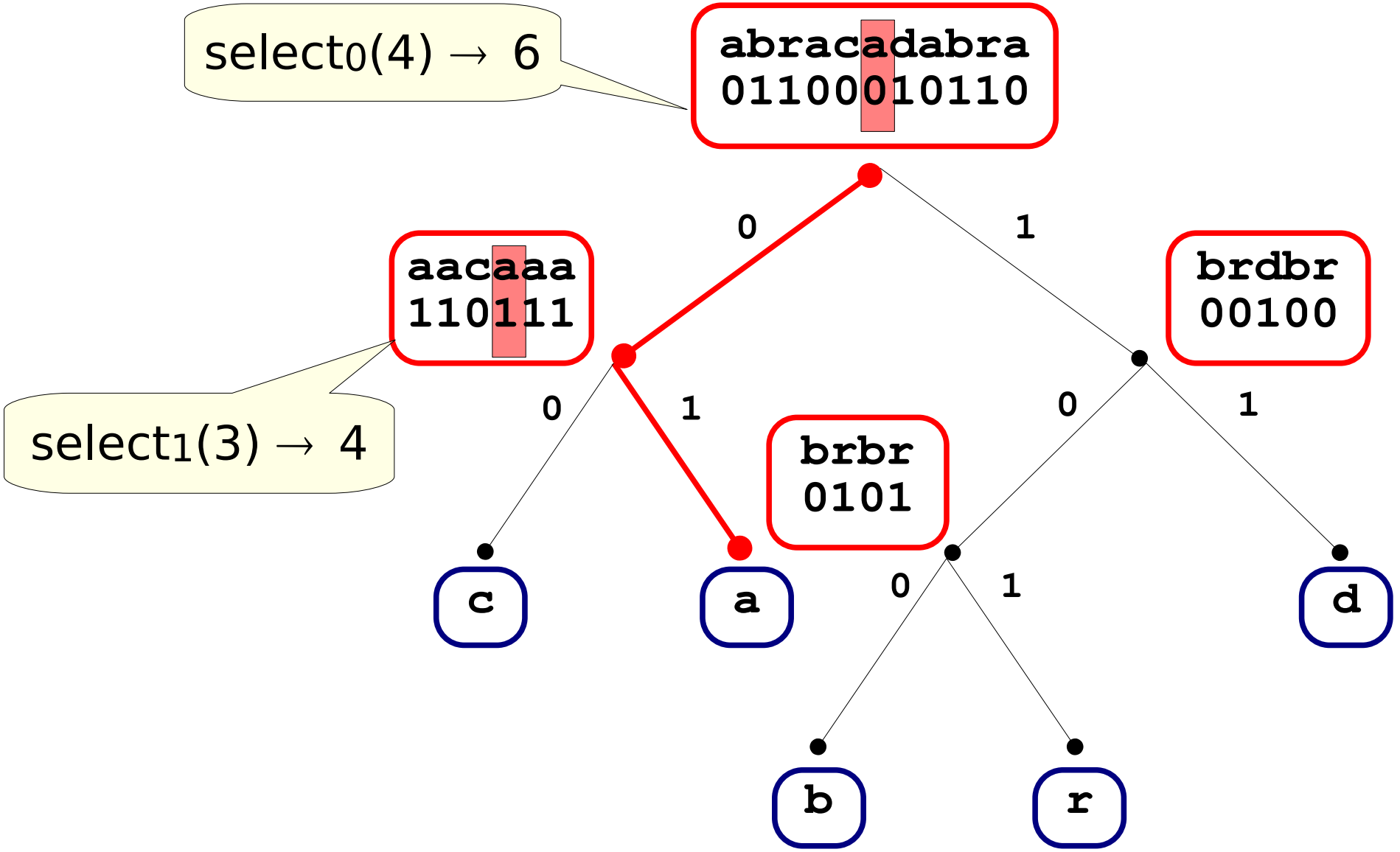
Suppose we want to compute rank(b,9)



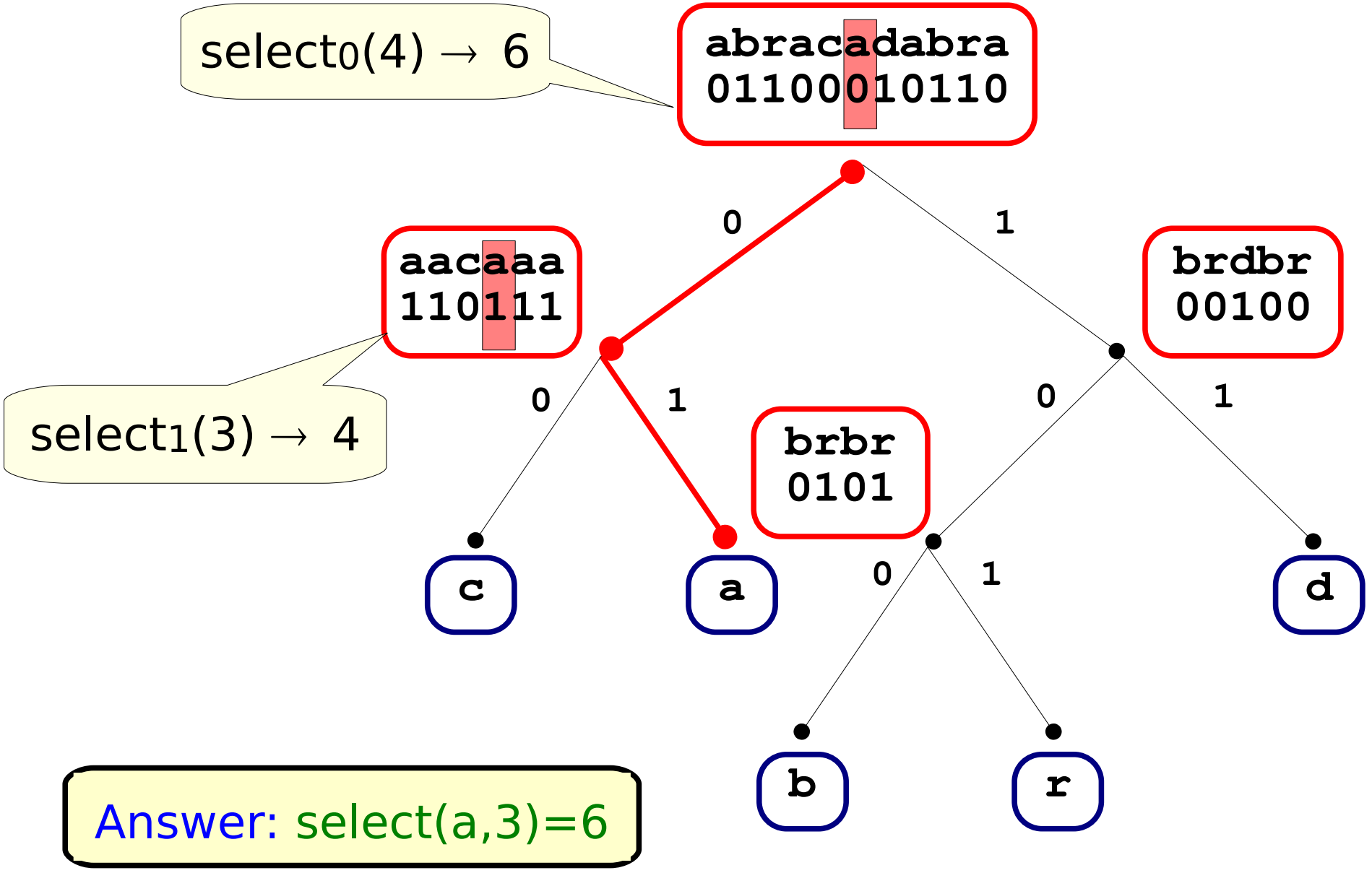
Select queries go upward. To compute select(a,3)



Select queries go upward. To compute select(a,3)

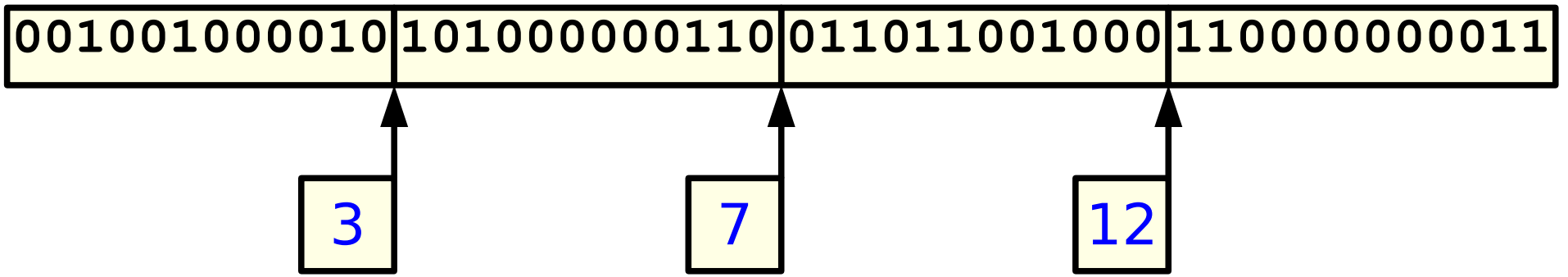


Select queries go upward. To compute select(a,3)

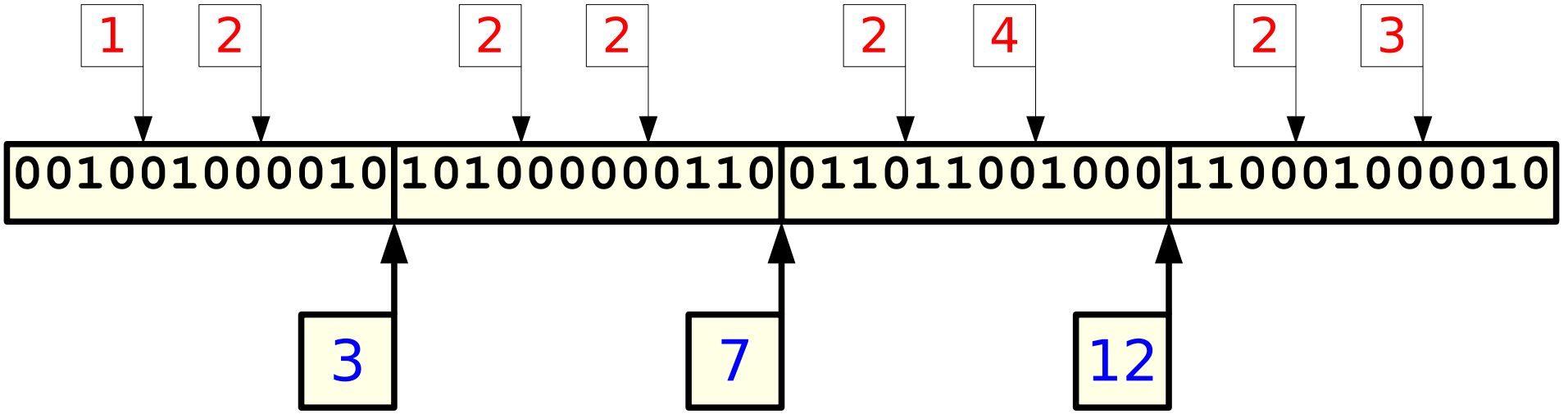


Rank_1 queries on binary strings

Simplest idea: split the binary string into blocks and store a partial sum for each block:



We can refine the idea using blocks and mini-blocks:

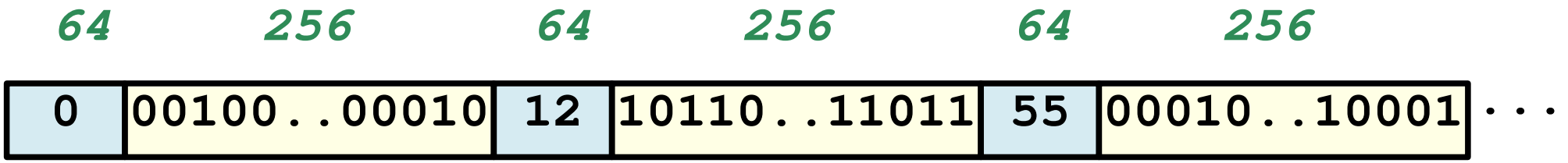


Blocks have length $\log^2 n$, mini-blocks $\log n$, each block entry (blue) takes $O(\log n)$ bits, each mini-block entry (red) takes $O(\log \log n)$ bits. Total space overhead $o(n)$ bits.

Using the pop_count instruction we can count the # of 1s in the prefix of a mini-block in $O(1)$ time. Hence:

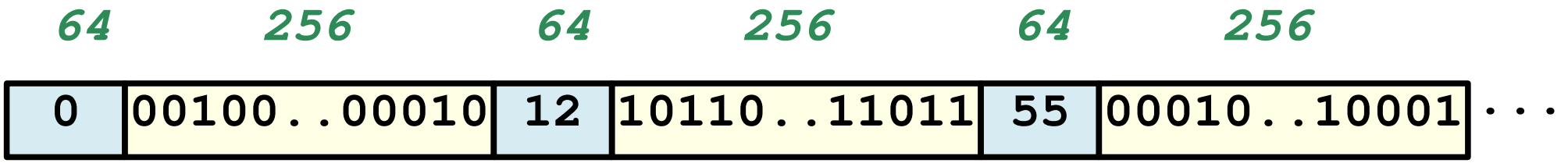
rank_1 = 2 table look-up + 1 pop_count = $O(1)$ time

A more practical implementation: interleave a 64-bit block count every 256 bits of data



- The space overhead is 64/256 = 25%
- The interleaving of counts and actual data reduces cache misses

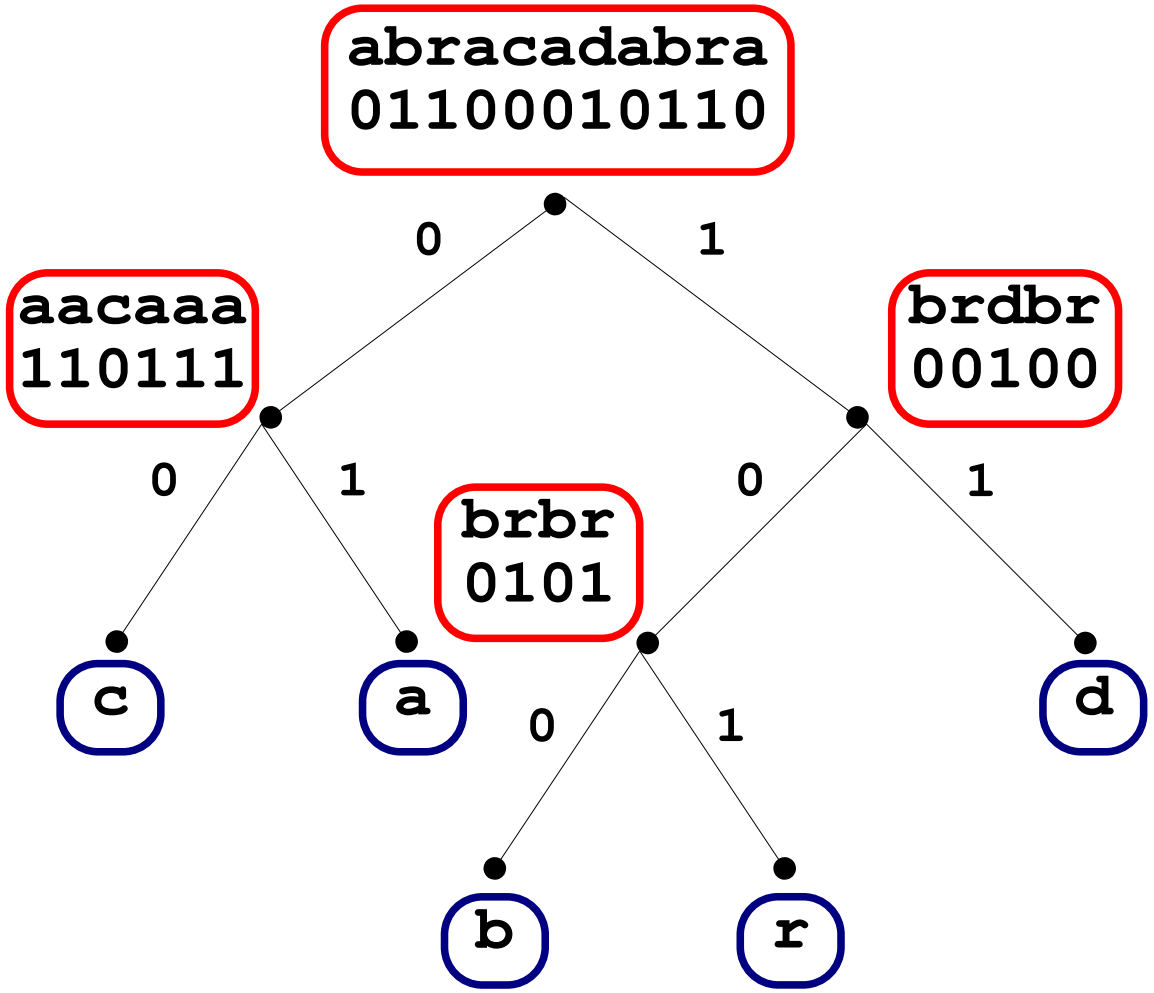
A more practical implementation: interleave a 64-bit block count every 256 bits of data



- The space overhead is 64/256 = 25%
- The interleaving of counts and actual data reduces cache misses

Similar (more complex) techniques for select queries!

Wavelet Trees as scrambled prefix codes



We are implicitly using the encoding:

c→00 a→01 b→100 r→101 d→11

Wavelet Trees as scrambled prefix codes:

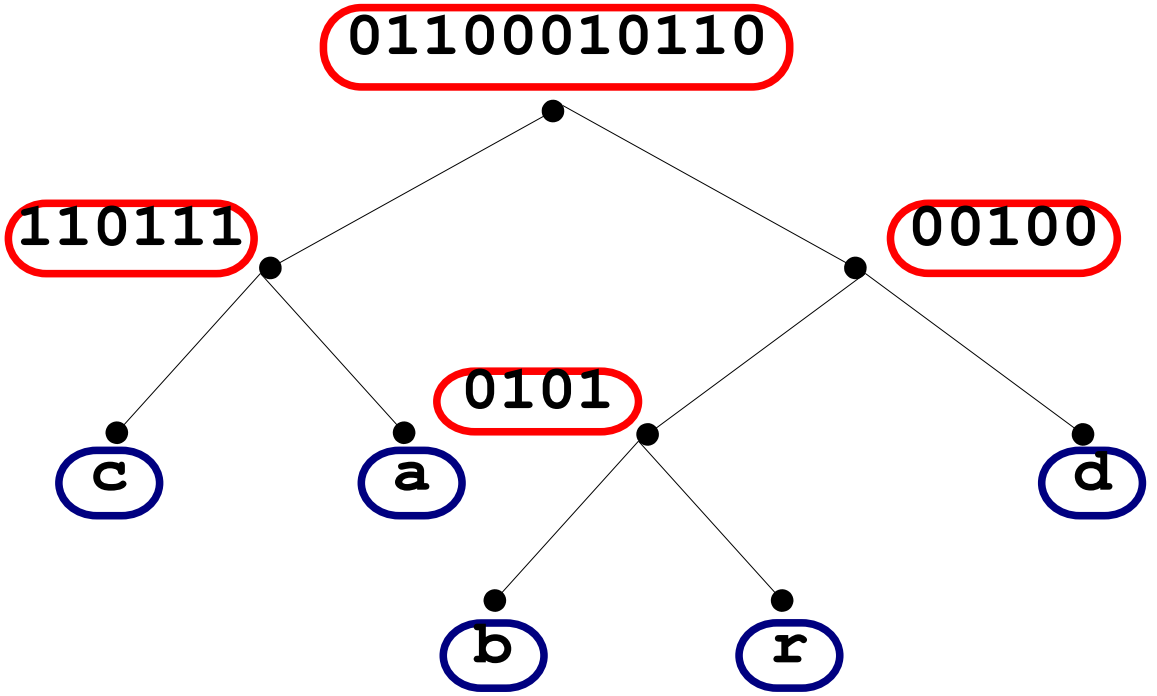
We are implicitly using the encoding:

a*→*01* *c*→*00* *b*→*100* *r*→*101* *d*→*11

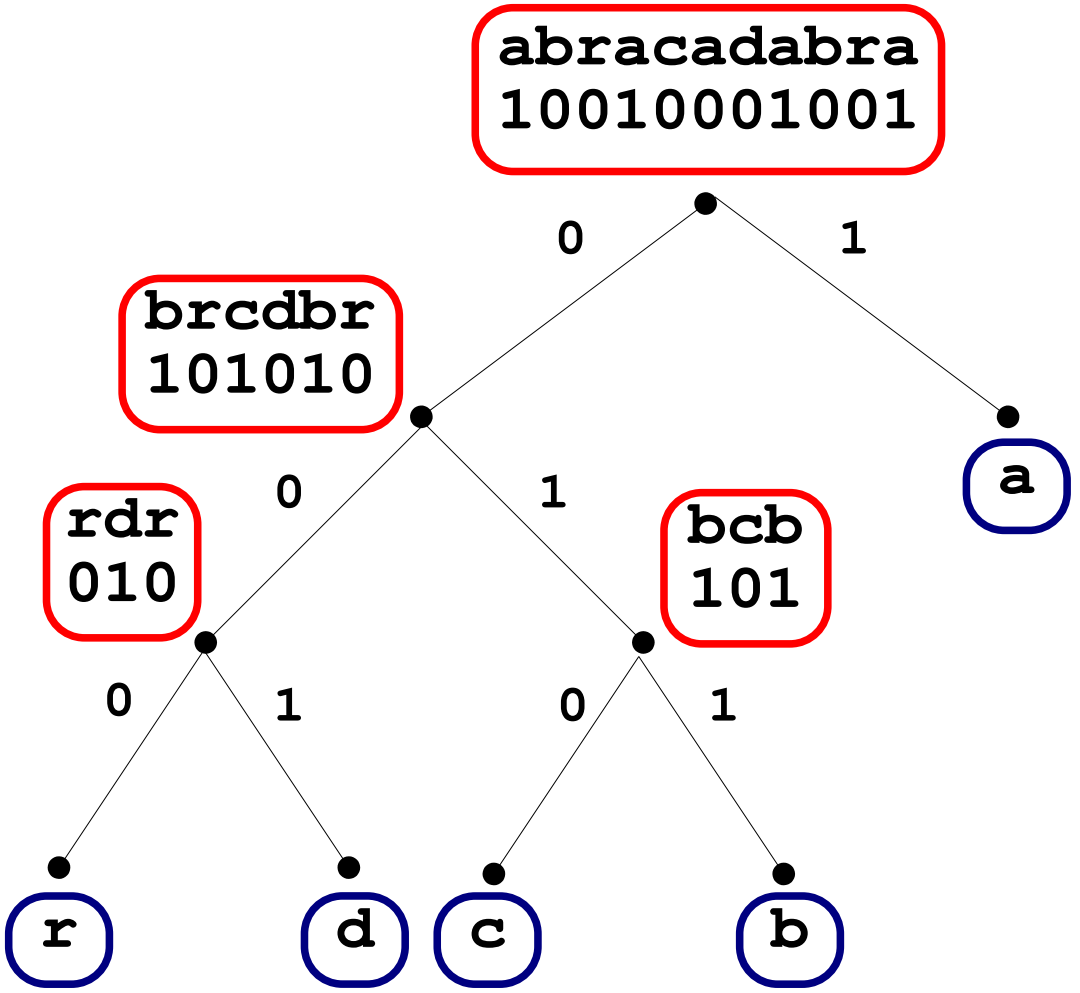
Traditional prefix codes:

abracadabra* → *01 100 101 01 00 01 11 . . .

Wavelets:



Changing the shape of the the Wavelet Tree, all properties mentioned above still hold with a different prefix code.



r→000
d→001
c→010
b→011
a→1

We can choose any prefix-free binary encoding of the characters and build the corresponding Wavelet Tree.

If we choose Huffman codes we have compression for free and direct access to arbitrary positions in the text.

Other Wavelet Trees virtues

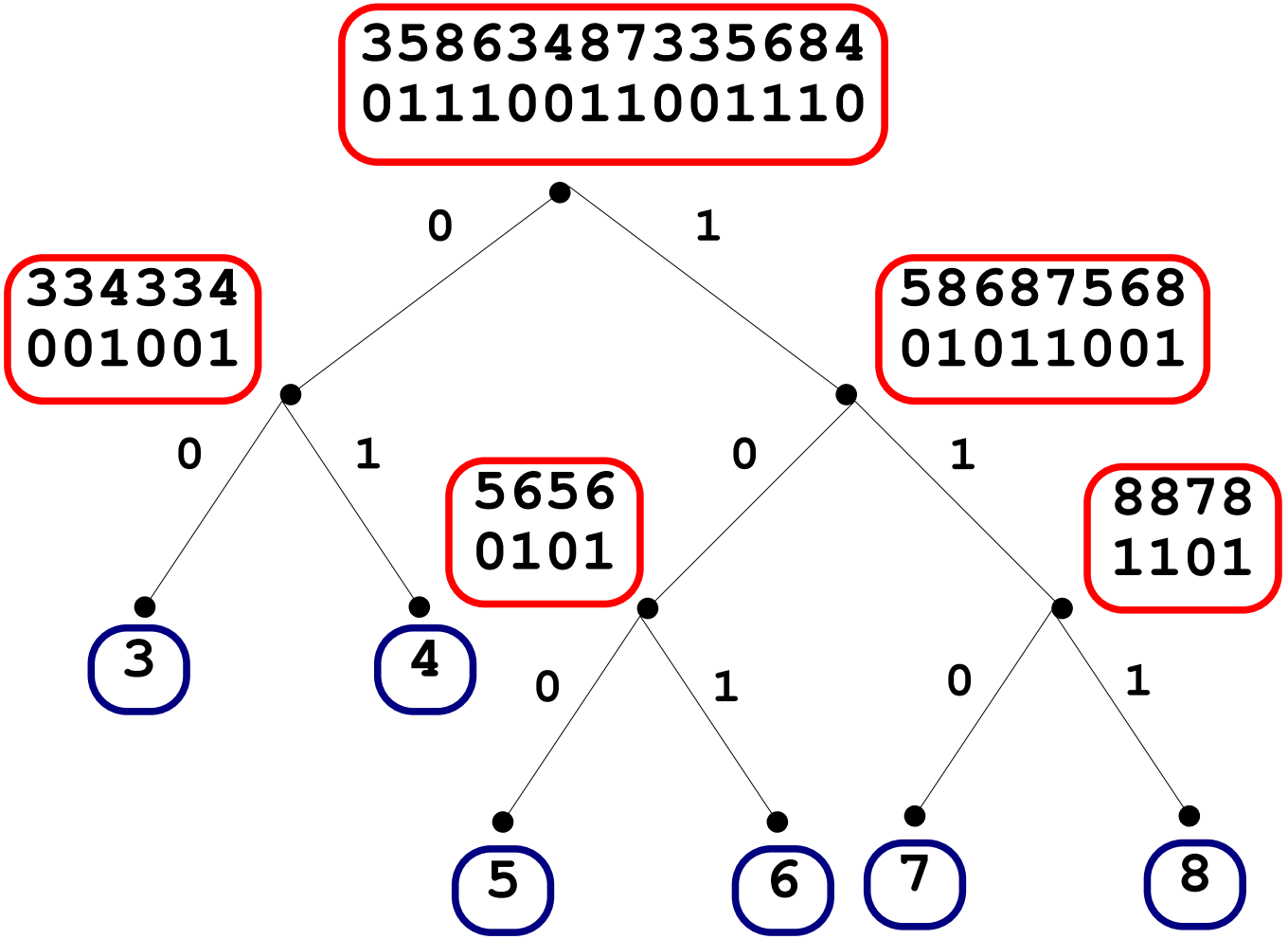
Simple solutions for range queries on an integer sequence S[1,n], for example

range_quantile(S,i,j,k): return the k-th smallest value in S[i⋯j]

range_count(S,i,j): return # of distinct values in S[i⋯j]

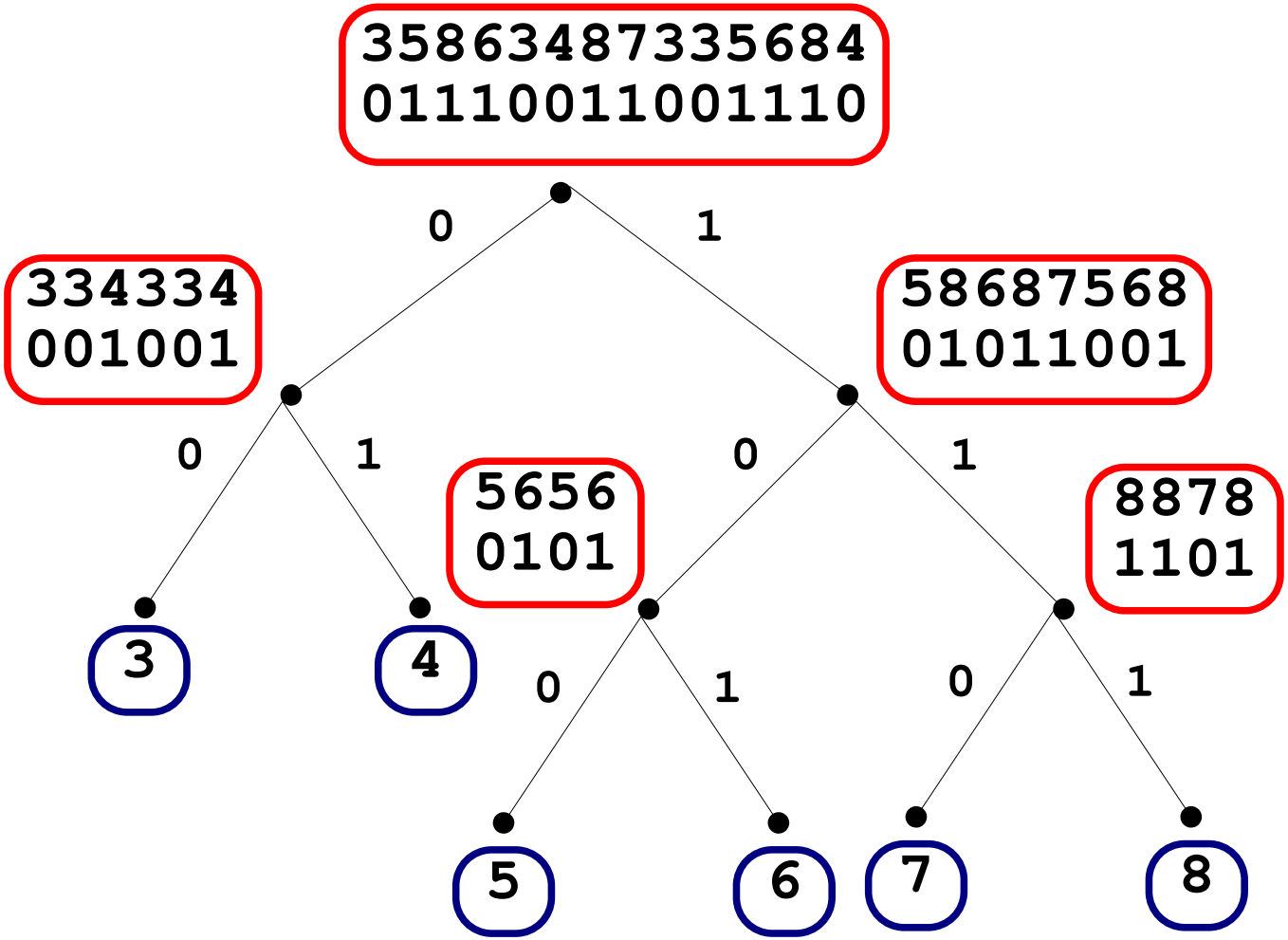
Example.

S = [3,5,8,6,3,4,8,7,3,3,5,6,8,4]



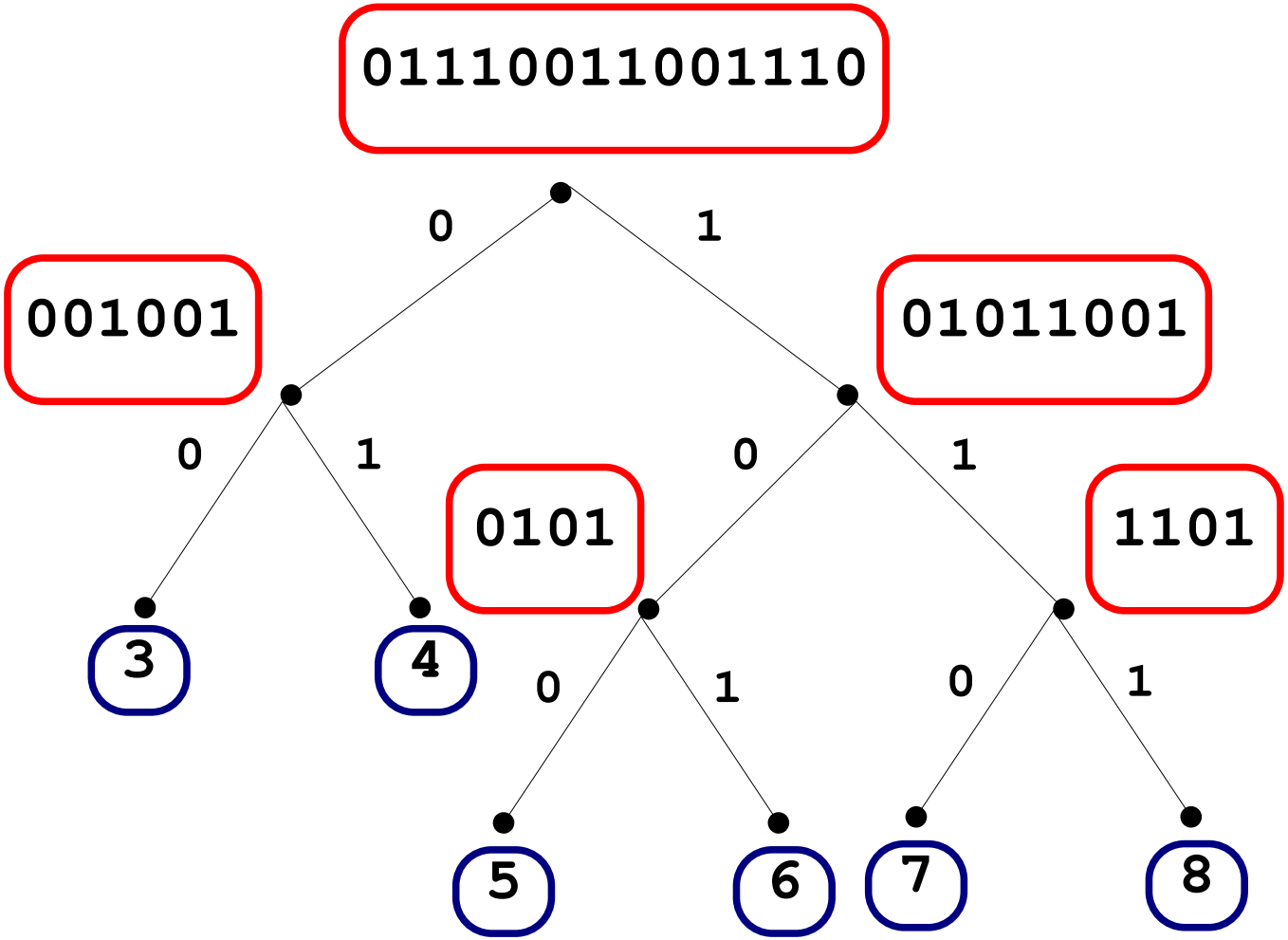
range_quantile(S,2,7,3): 3-rd smallest value in S[2⋯7]

S = [3,5,8,6,3,4,8,7,3,3,5,6,8,4]



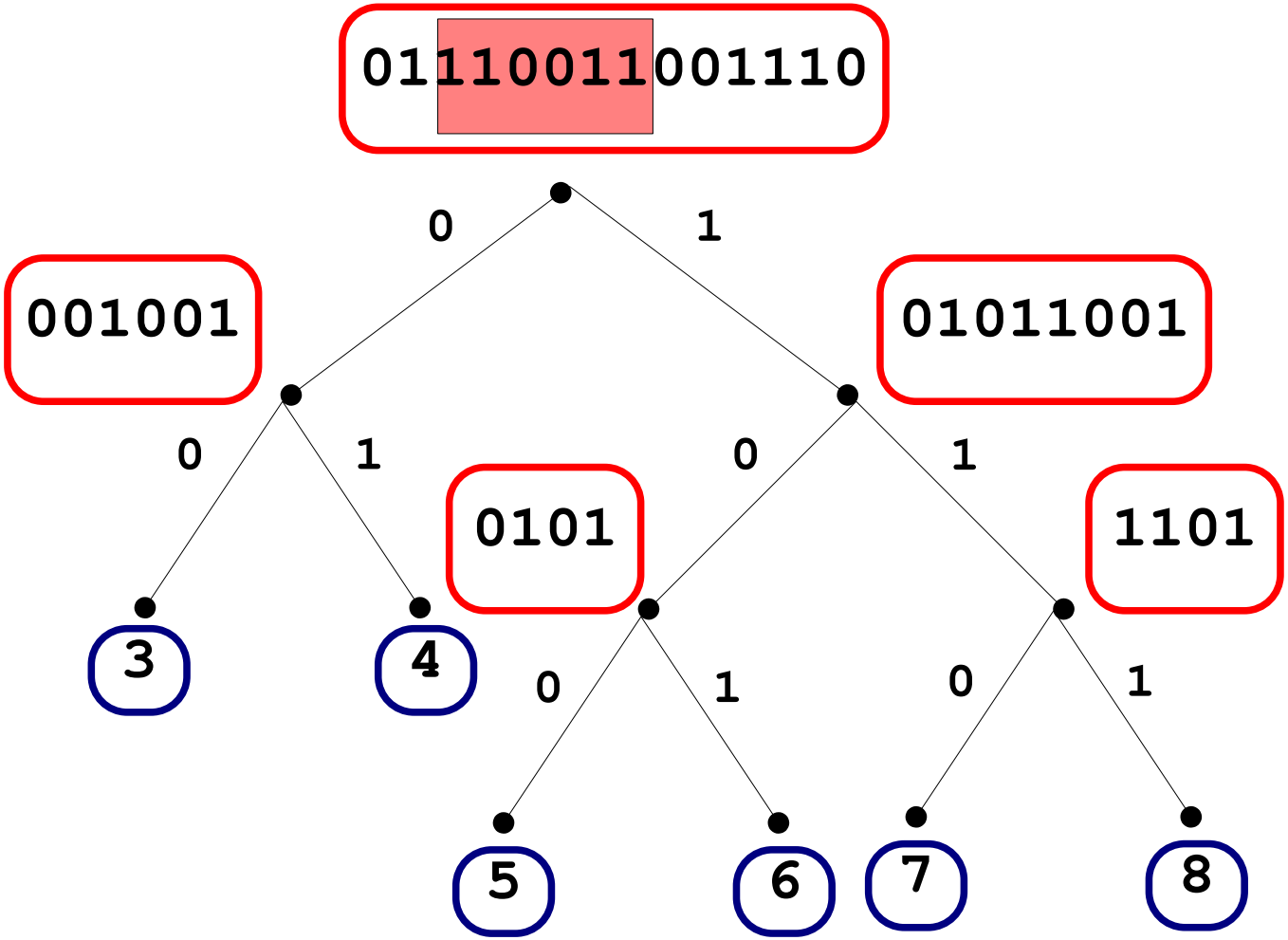
range_quantile(S,2,7,3): 3-rd smallest value in S[2⋯7]

S = [3,5,8,6,3,4,8,7,3,3,5,6,8,4]



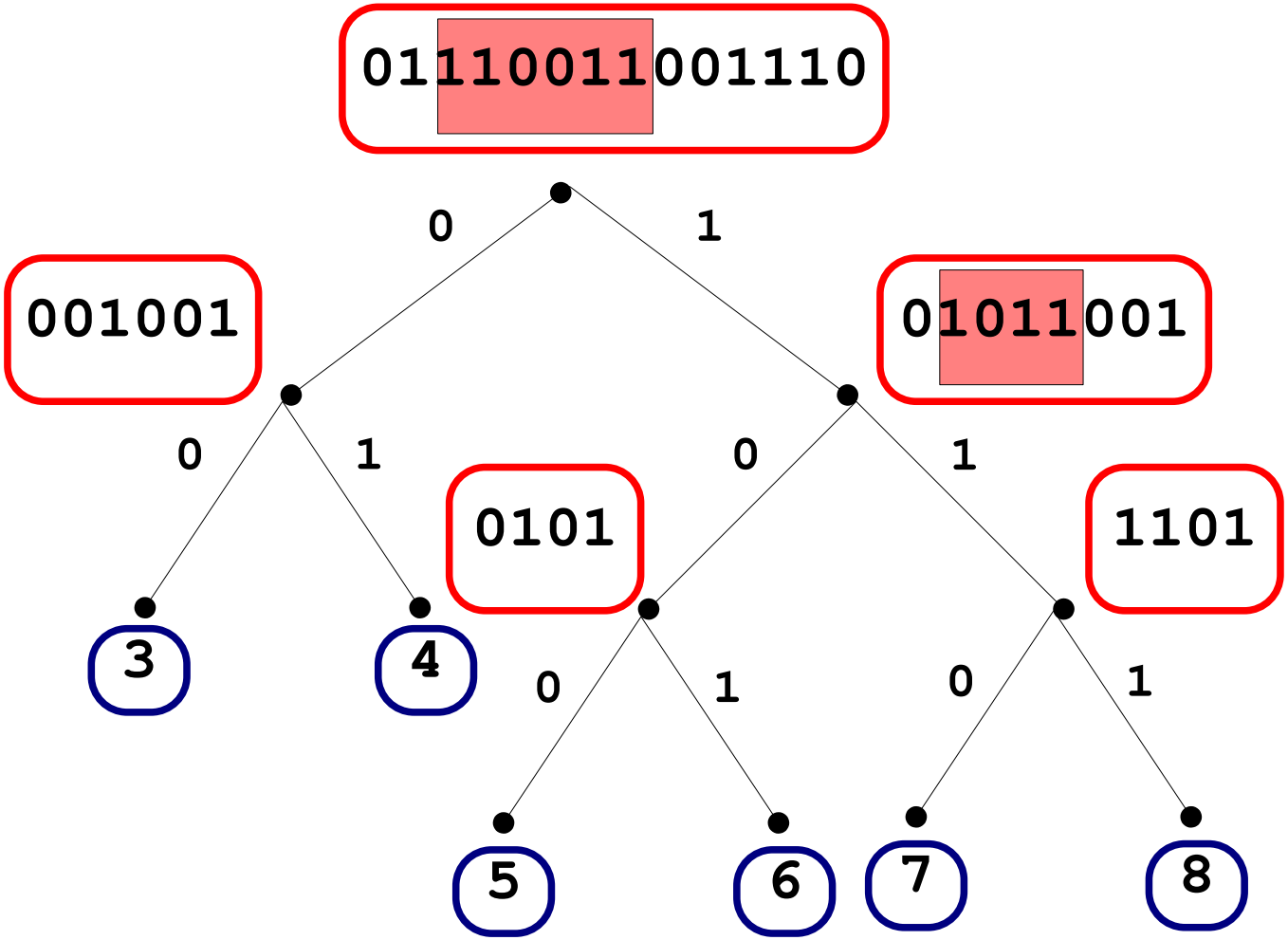
range_quantile(S,2,7,3): 3-rd smallest value in S[2⋯7]

S = [3,5,8,6,3,4,8,7,3,3,5,6,8,4]



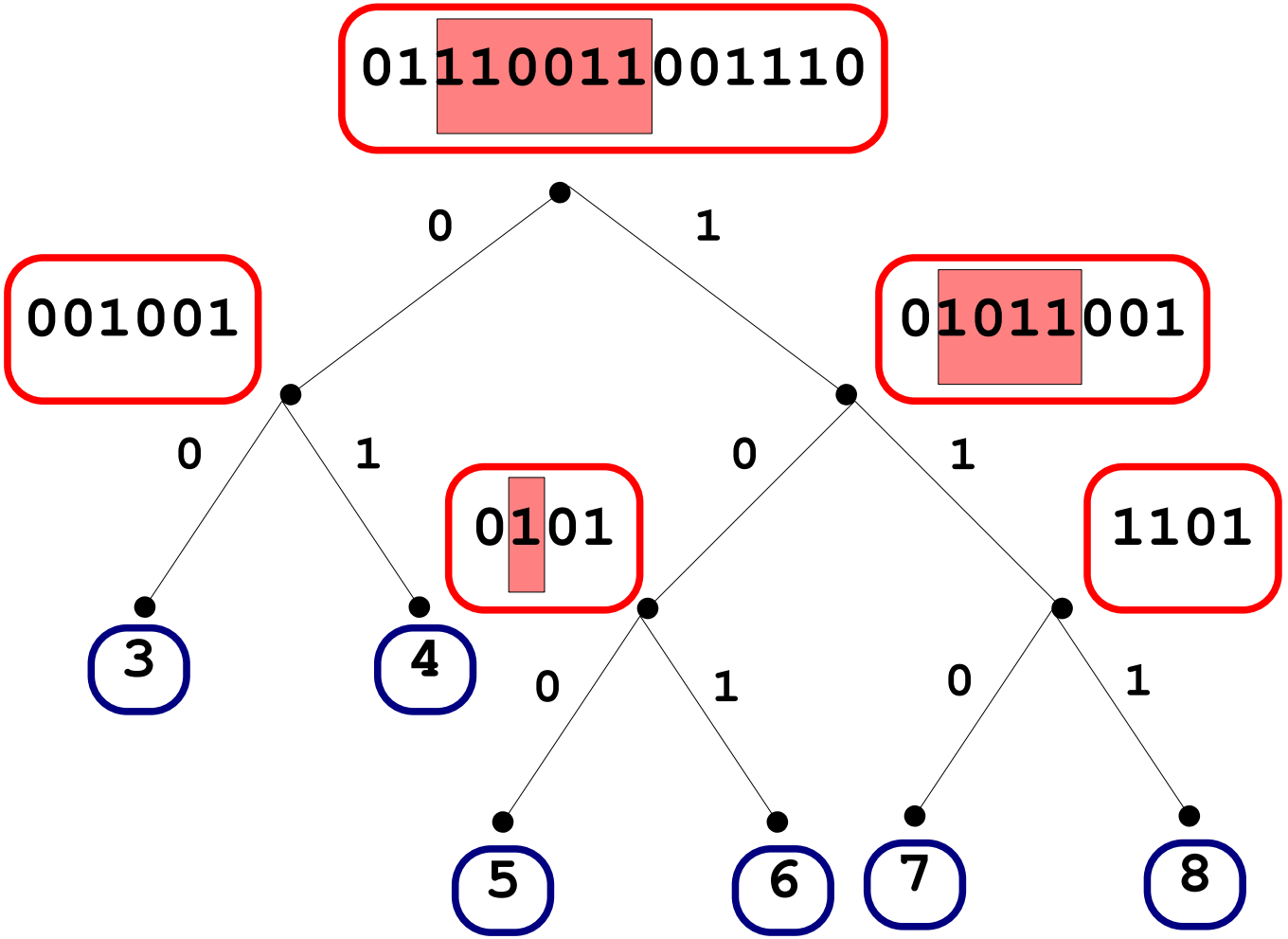
range_quantile(S,2,7,3): 3-rd smallest value in S[2⋯7]

S = [3,5,8,6,3,4,8,7,3,3,5,6,8,4]



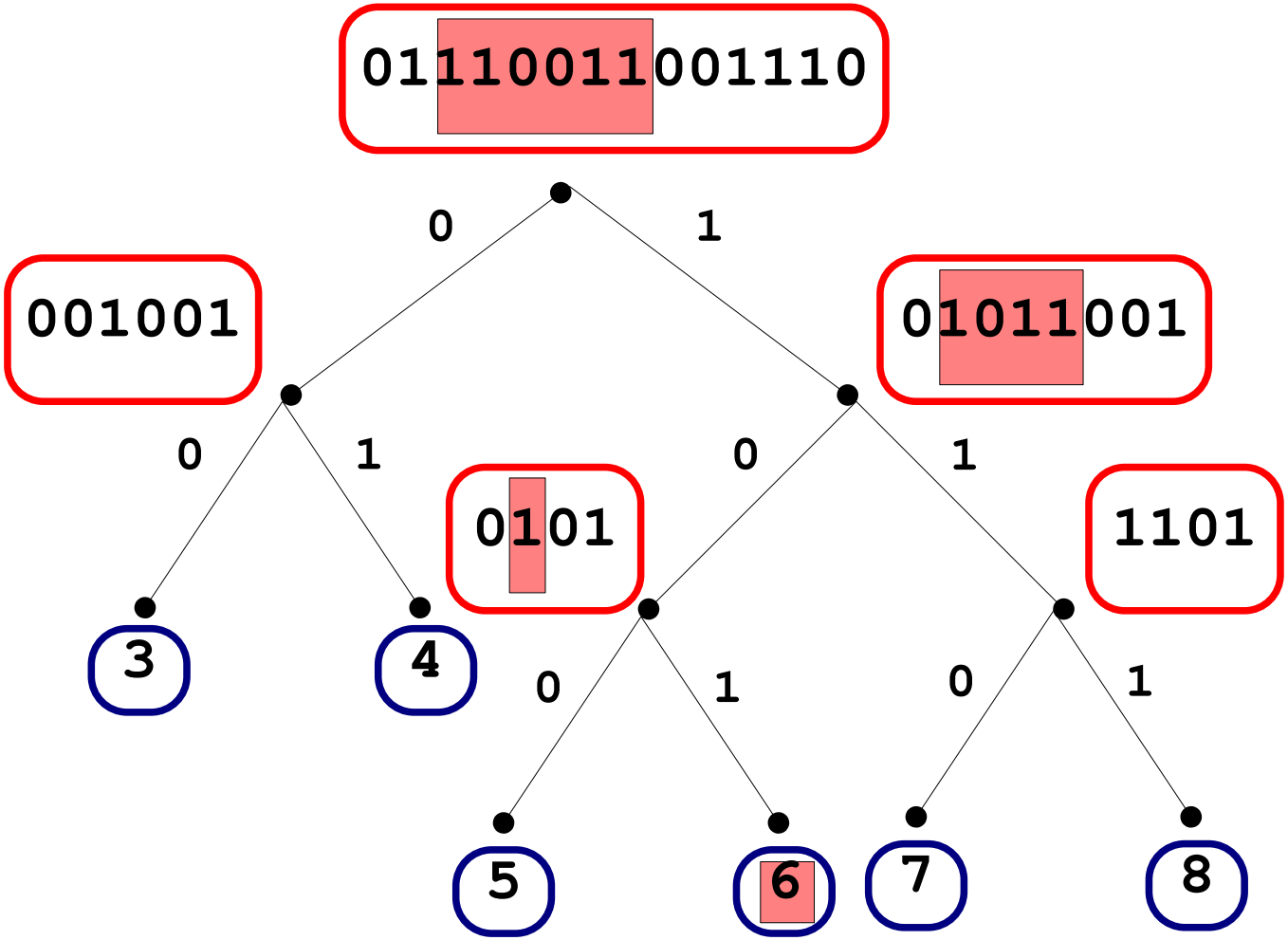
range_quantile(S,2,7,3): 3-rd smallest value in S[2⋯7]

S = [3,5,8,6,3,4,8,7,3,3,5,6,8,4]



range_quantile(S,2,7,3): 3-rd smallest value in S[2⋯7]

S = [3,5,8,6,3,4,8,7,3,3,5,6,8,4]



Wavelet Tree Implementation

- Conceptually simple, but tricky in a few details
- Several implementations cited in the literature, some available on the web
- Ready to use, open source, modular C++ library:

  https://github.com/simongog/sdsl-lite

References

- R. Grossi, A. Gupta, J. Vitter High-order entropy-compressed text indexes. Proc. SODA 2003 (paper introducing WTs: somewhat hard to read)
- G. Navarro, V. Mäkinen. Compressed Full-text indexes. ACM Comp. Surv. 2007 (main reference for WT and compressed indices)
- G. Navarro. Wavelet Trees for All, Journal Discrete Algorithms, 2014 (recent review paper on WTs)