

Efficient Rank/Select operations on arbitrary strings

First step: binary rank/select

Given a binary string $b[1,n]$:

$\text{rank}_c(i)$: number of c 's in $b[1,i]$

$\text{select}_c(i)$: position of the i -th c

001001001001110001001000100100000100

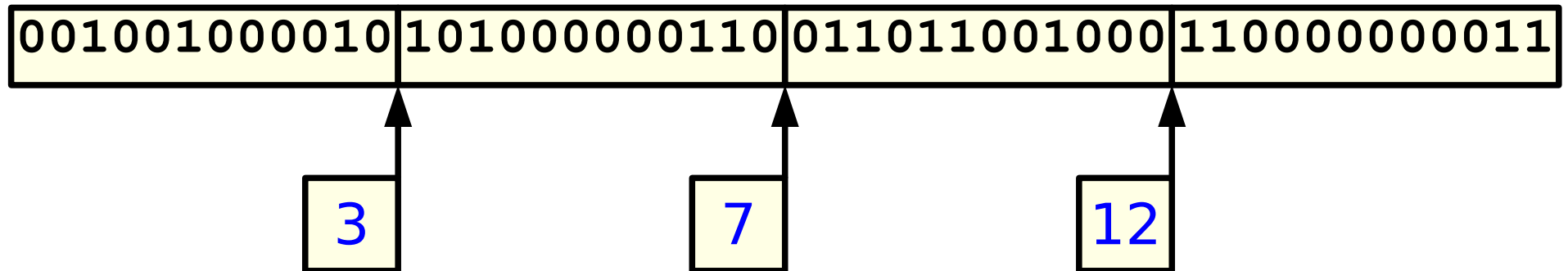
$\text{rank}_0(6)=4$

$\text{select}_1(3)=9$

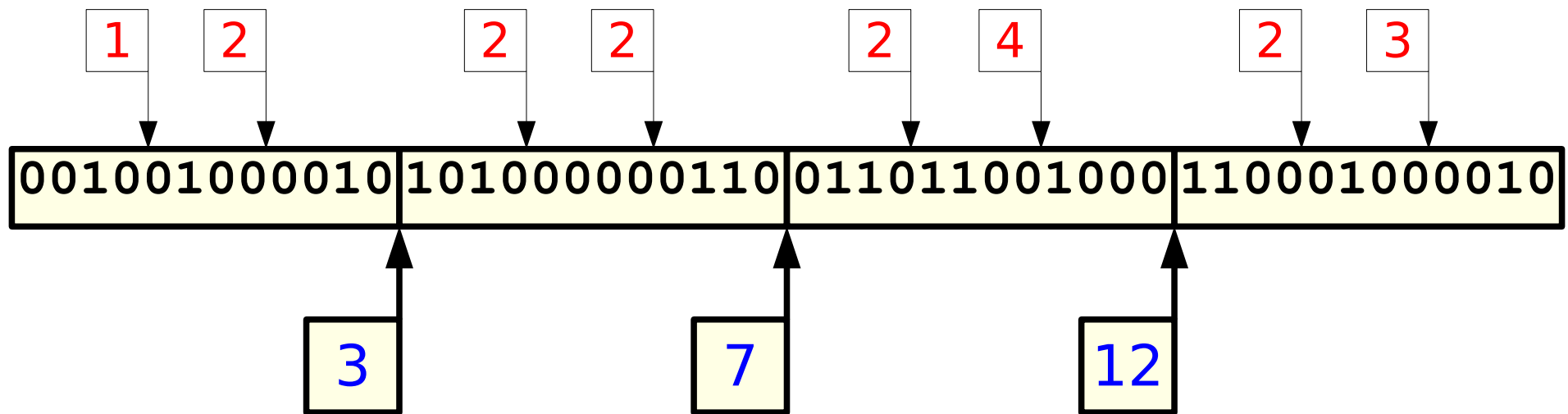
Using additional $o(n)$ bits we can answer
rank/select queries in $O(1)$ time [Clark, Munro '96].

Rank₁ 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**:

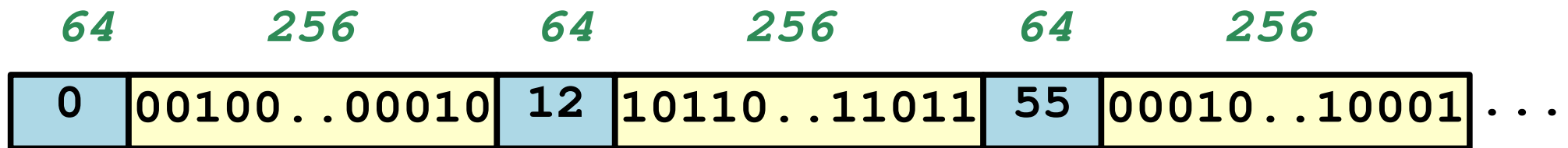


Blocks have length $\log^2 n$, mini-blocks $\log n$, each block entry (blue) takes $O(\log n)$ bits, each mini-block entry (red) takes $O(\log \log n)$ bits. Total space overhead $o(n)$ bits.

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

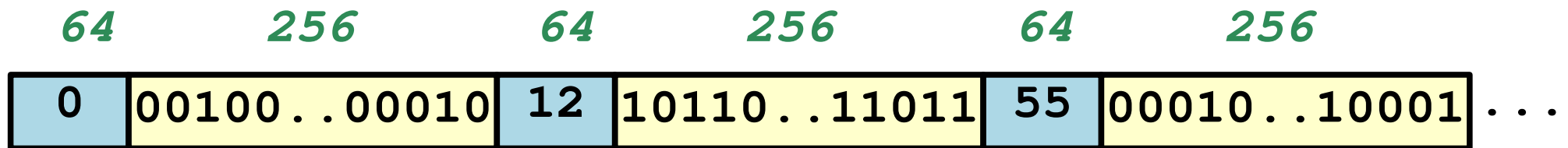
$$\text{rank}_1 = 2 \text{ table look-up} + 1 \text{ pop_count} = O(1) \text{ time}$$

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!

The Wavelet Tree data structure

rank/select operations on arbitrary strings

Given a string s we define the operations:

- $\text{rank}(c,i)$: # of occ. of c in $s[1,i]$
- $\text{select}(c,i)$: position of i -th c in s

Example, $s=\text{mississippi}$

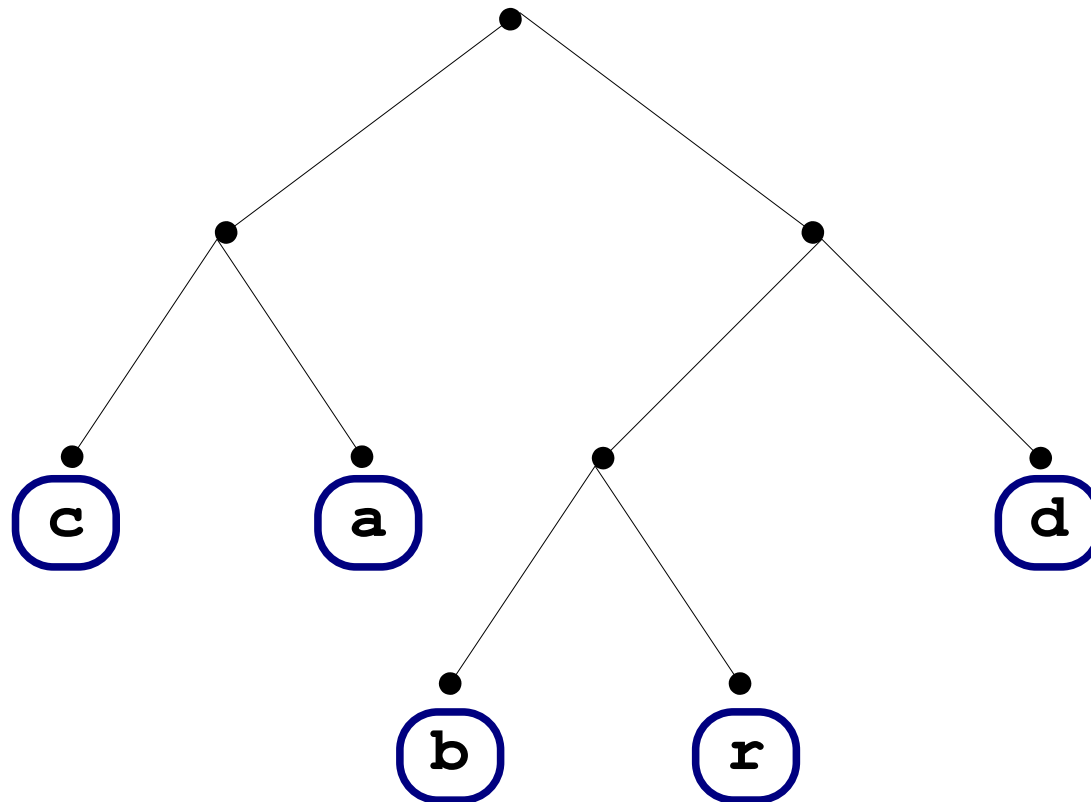
- $\text{rank}(s,5) = 2$
- $\text{select}(i,3) = 8$

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

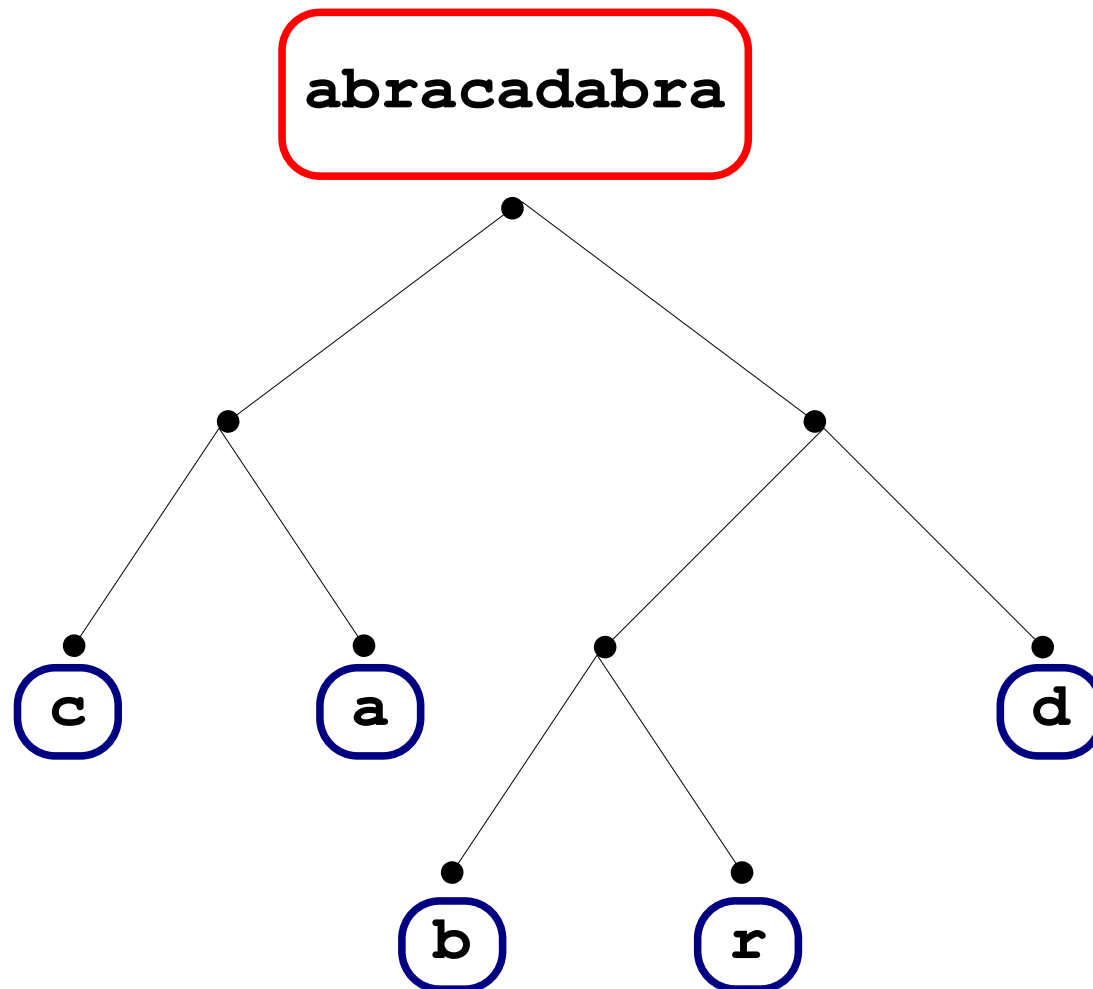
They work assuming we can do rank/select on binary strings,

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

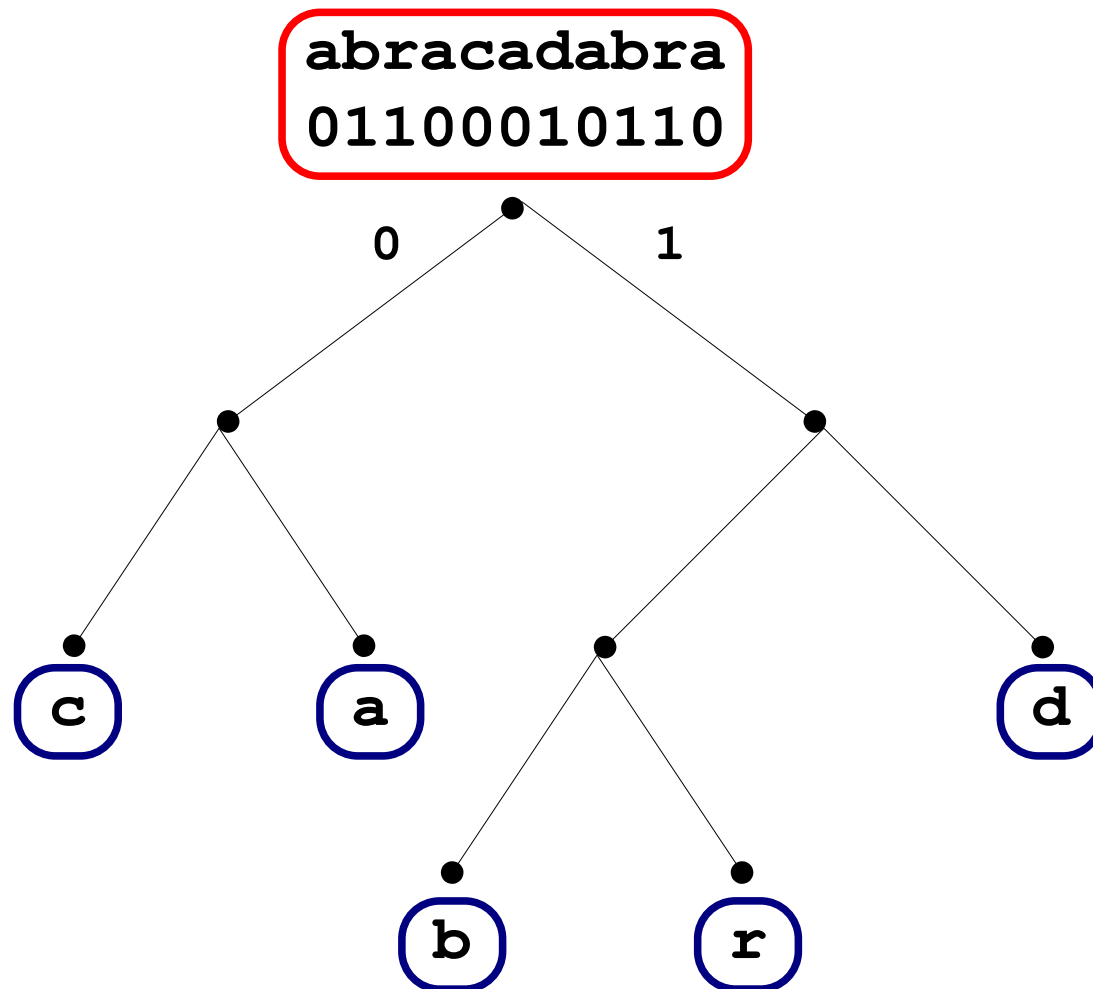
Example. $A = \{a, b, c, d, r\}$



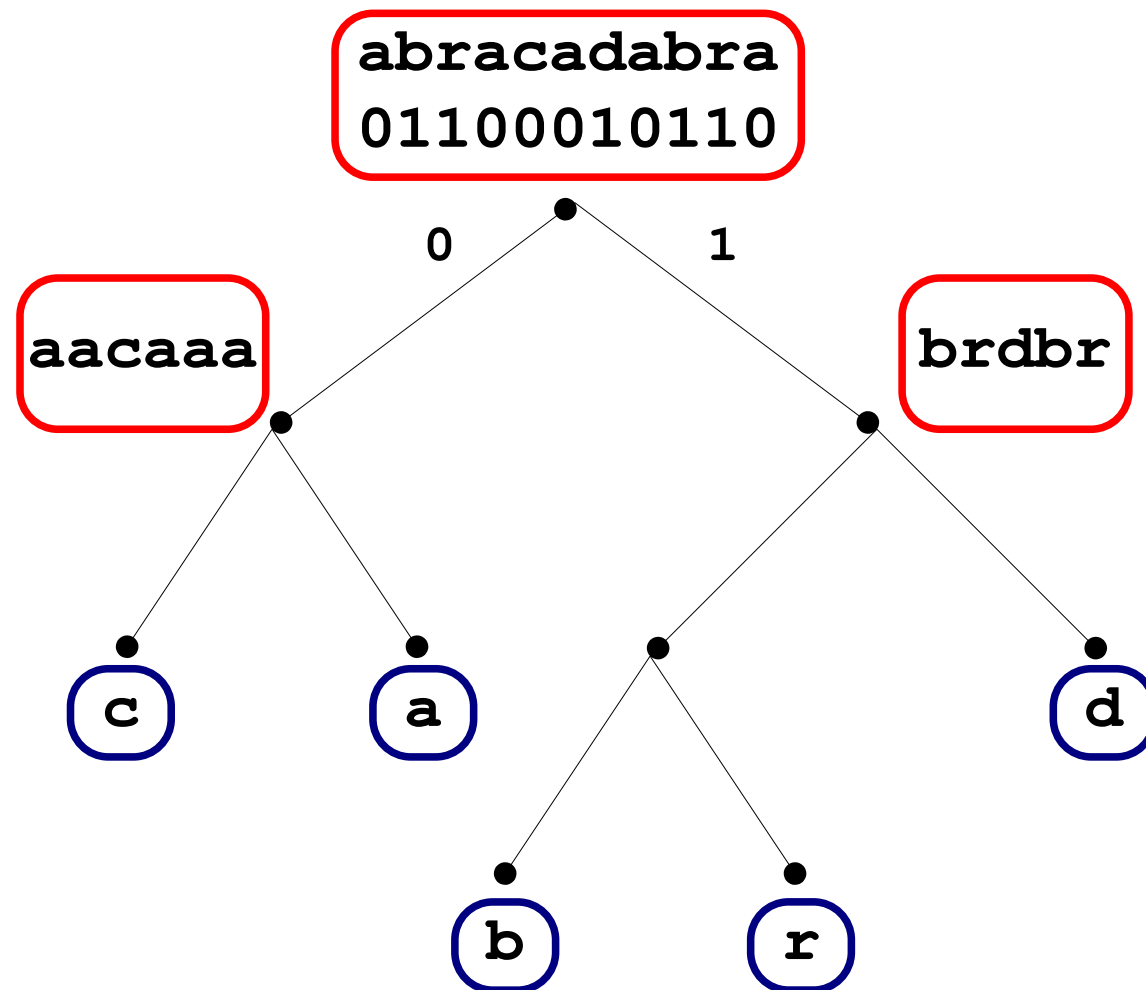
Given the string **abracadabra** we build the corresponding **Wavelet Tree** as follows:



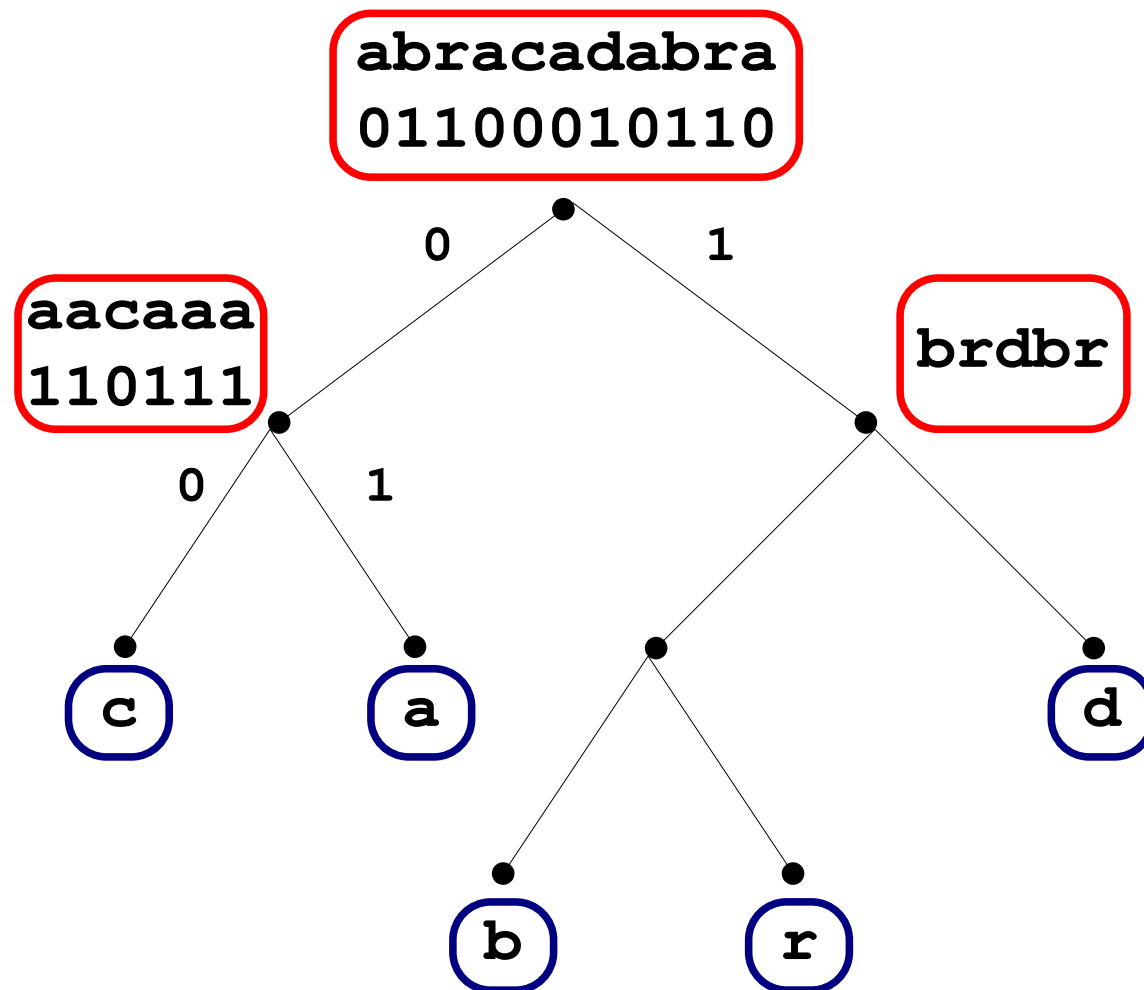
Given the string **abracadabra** we build the corresponding **Wavelet Tree** as follows:



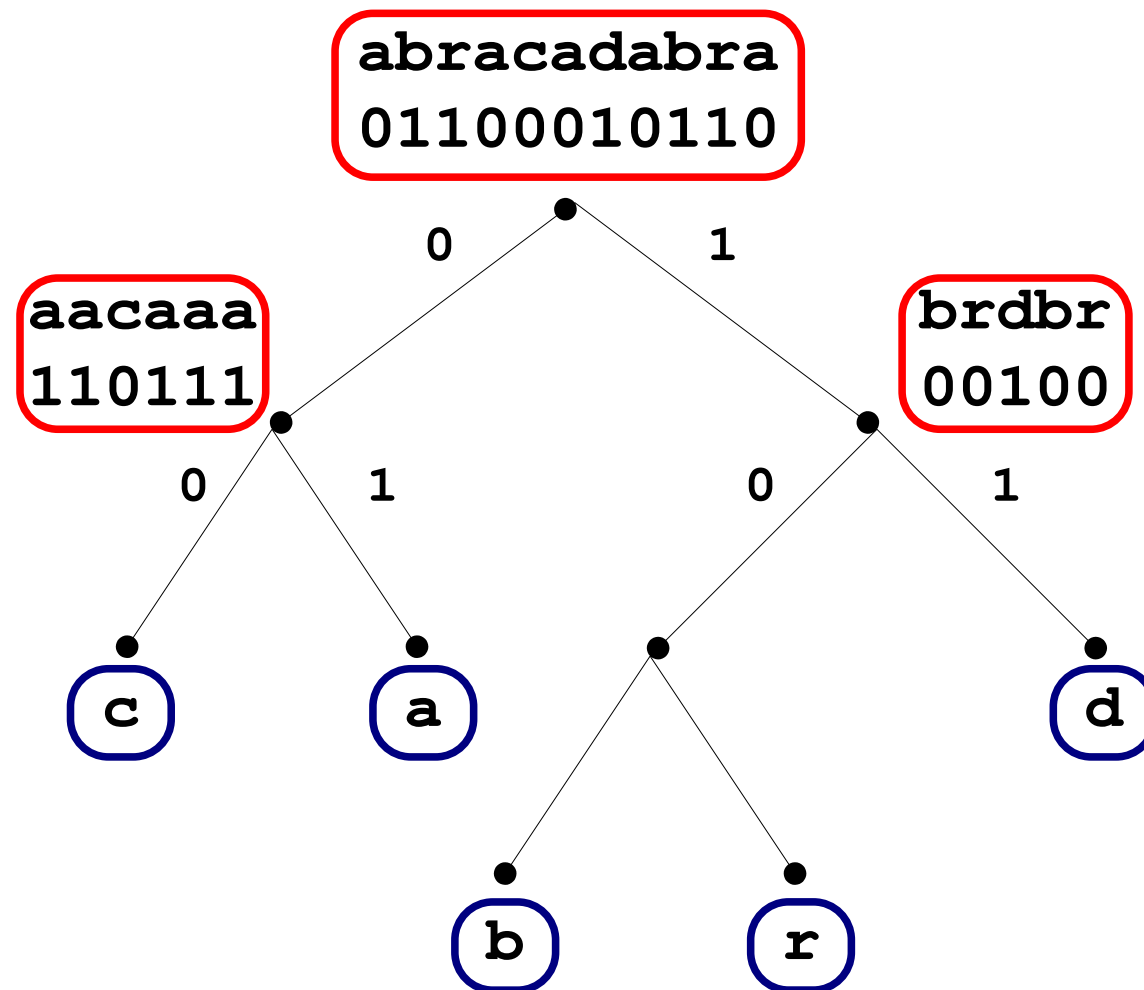
Given the string **abracadabra** we build the corresponding **Wavelet Tree** as follows:



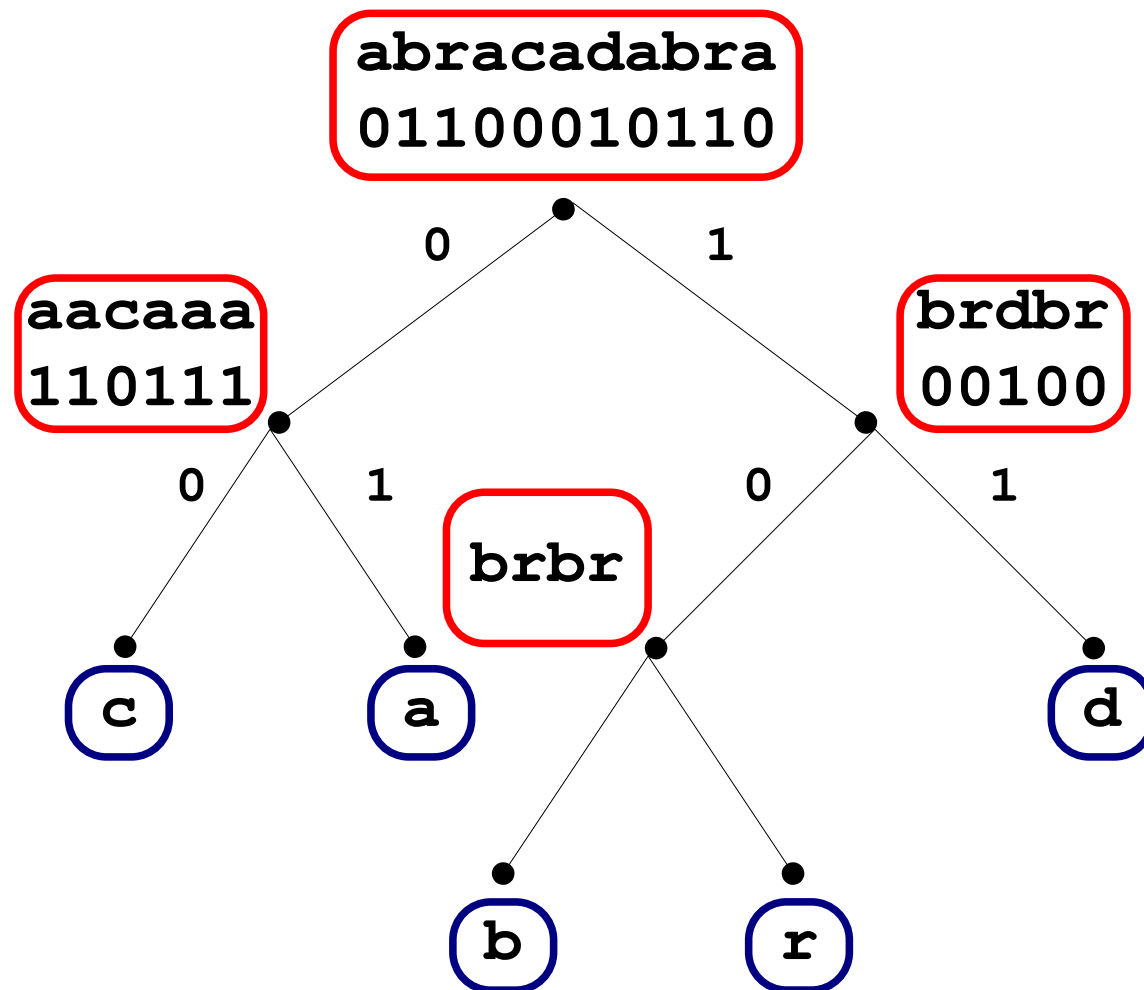
Given the string **abracadabra** we build the corresponding **Wavelet Tree** as follows:



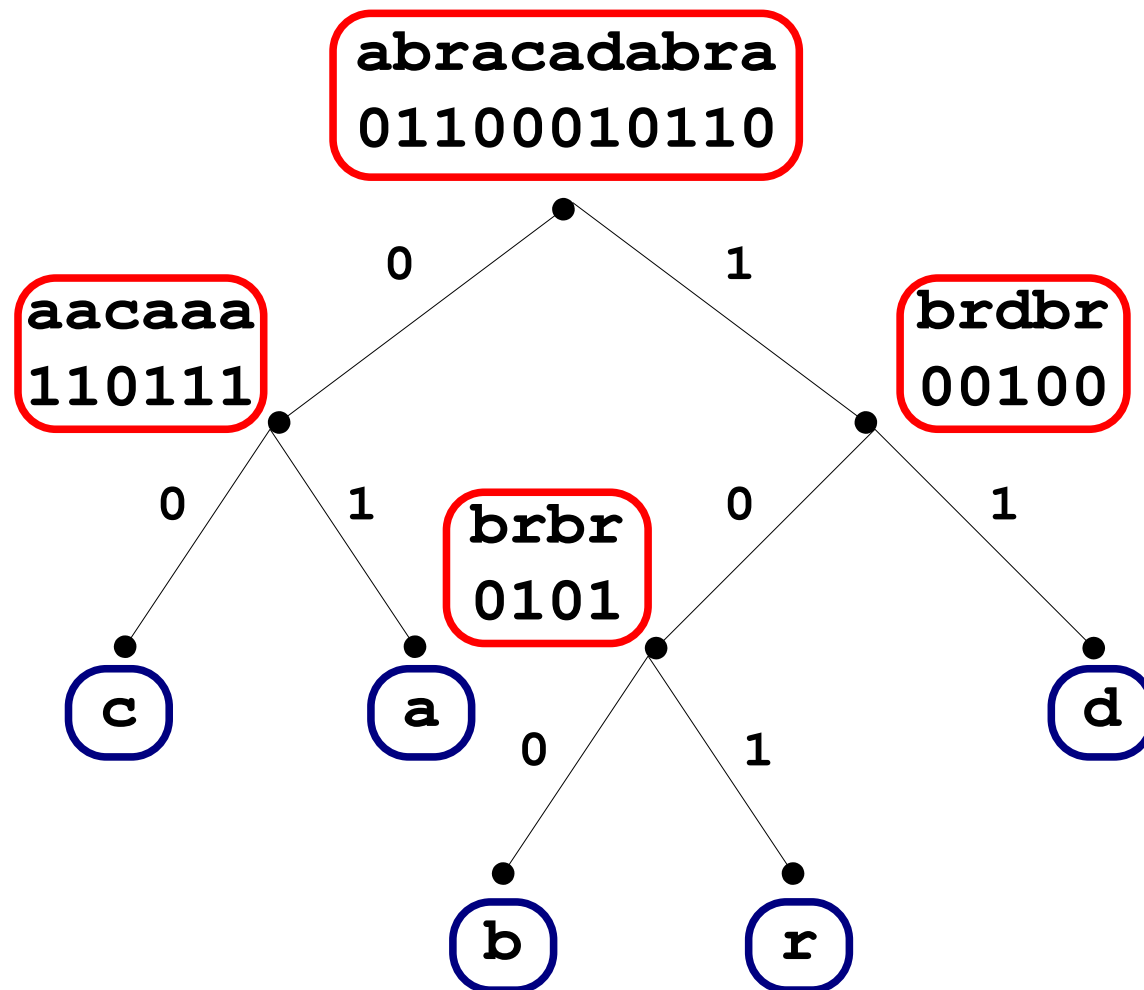
Given the string **abracadabra** we build the corresponding **Wavelet Tree** as follows:



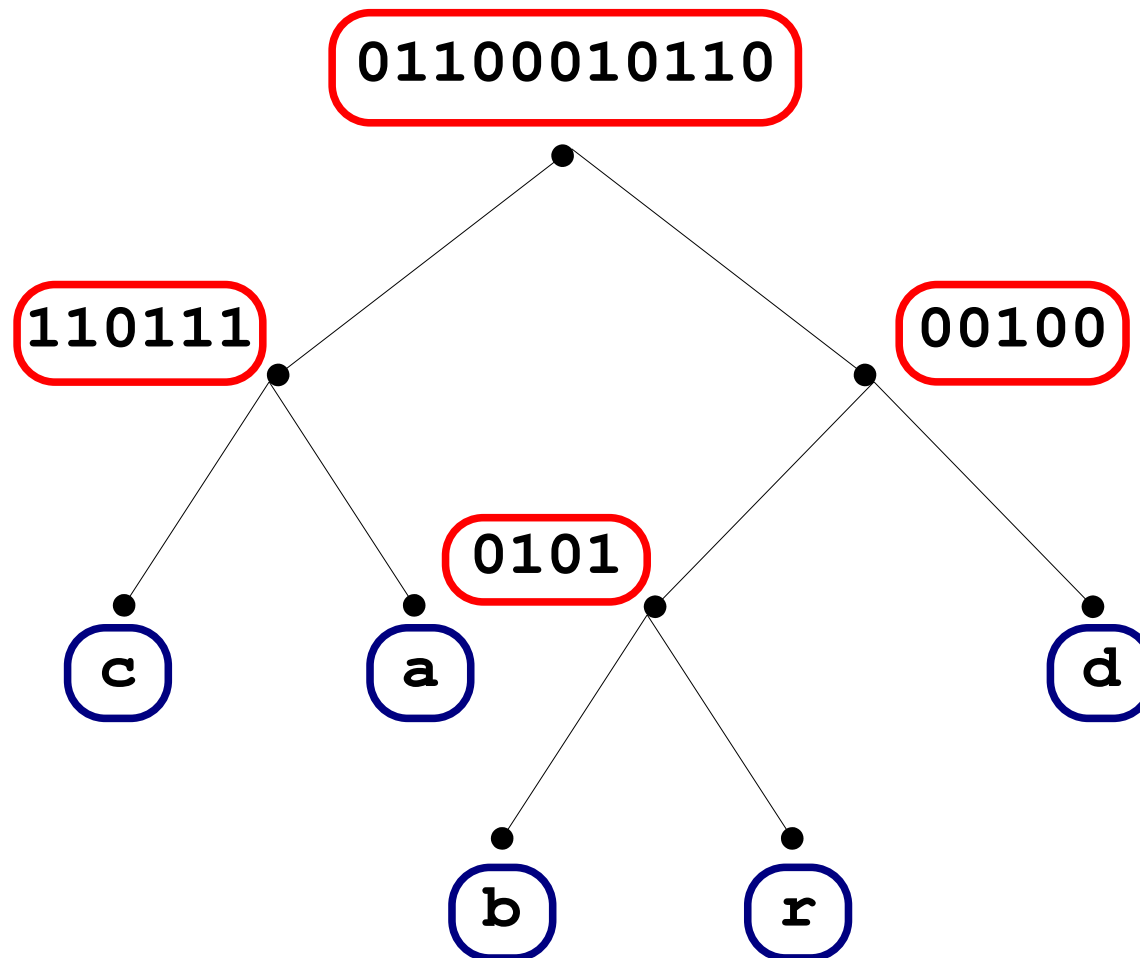
Given the string **abracadabra** we build the corresponding **Wavelet Tree** as follows:



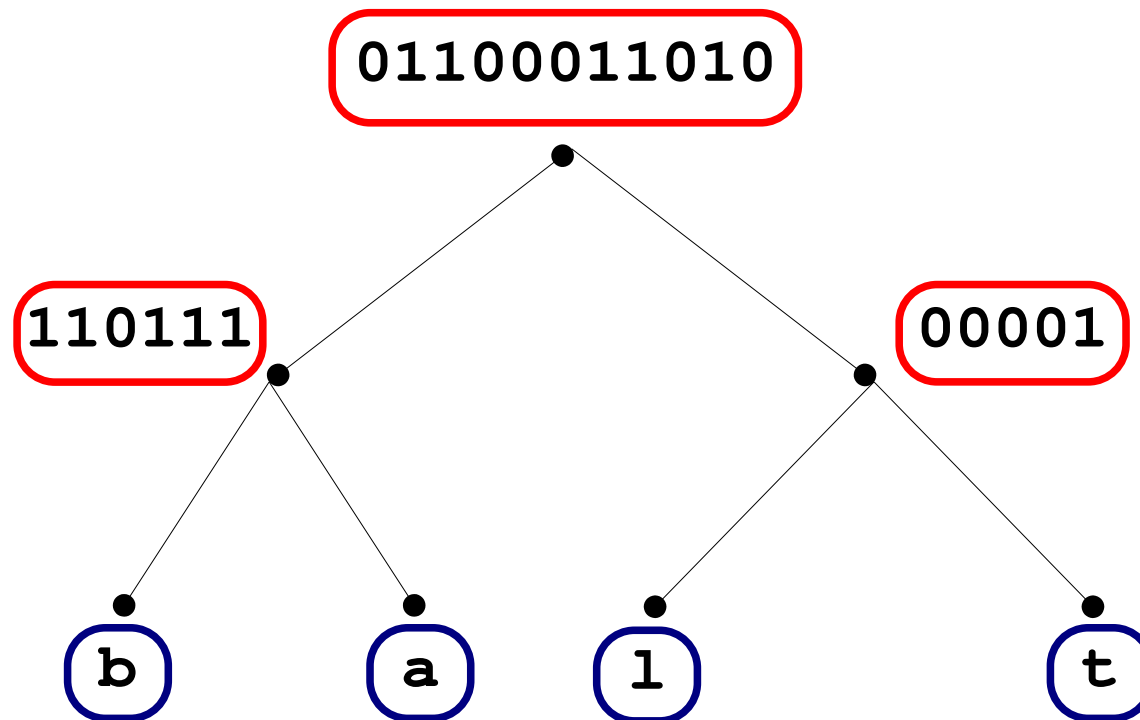
Given the string **abracadabra** we build the corresponding **Wavelet Tree** as follows:



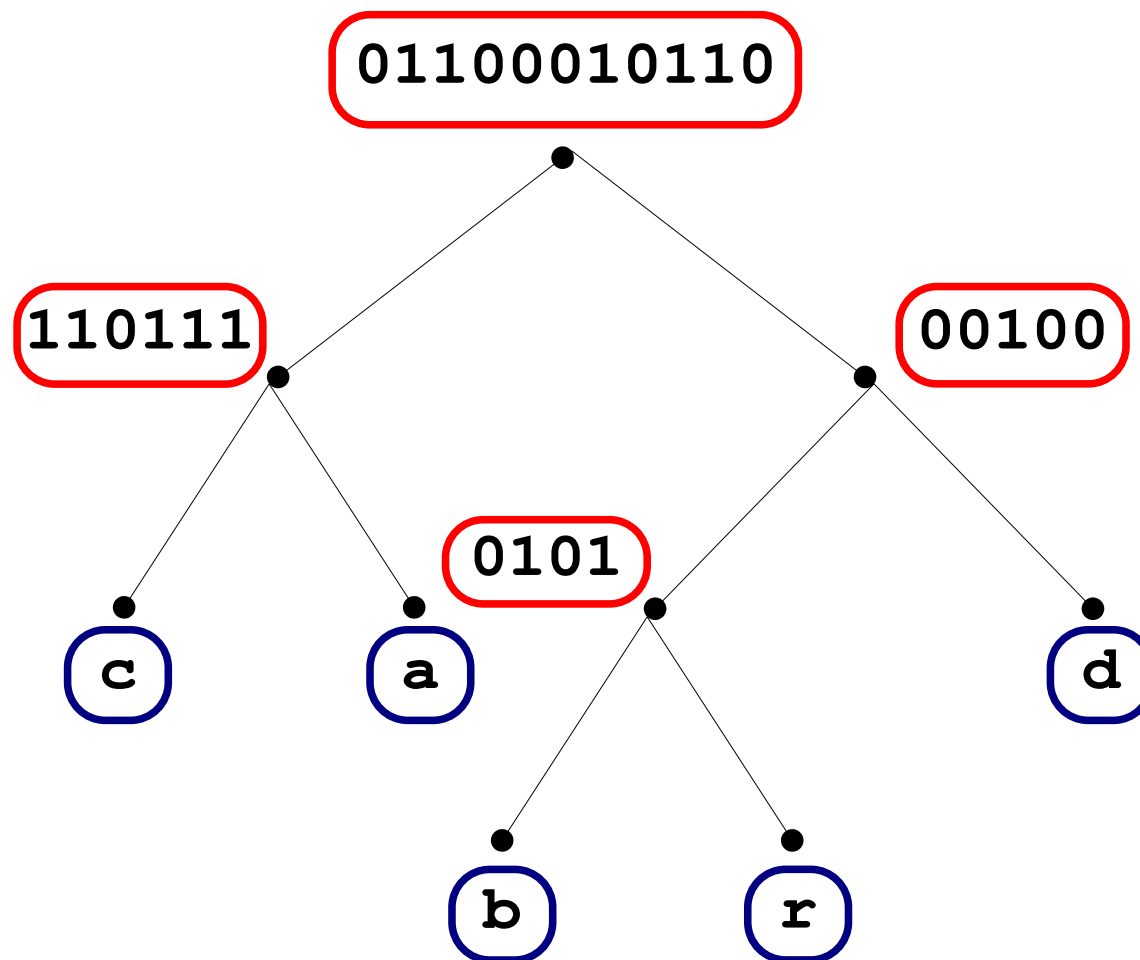
We only store the shape of the tree and the binary strings associated to each node:



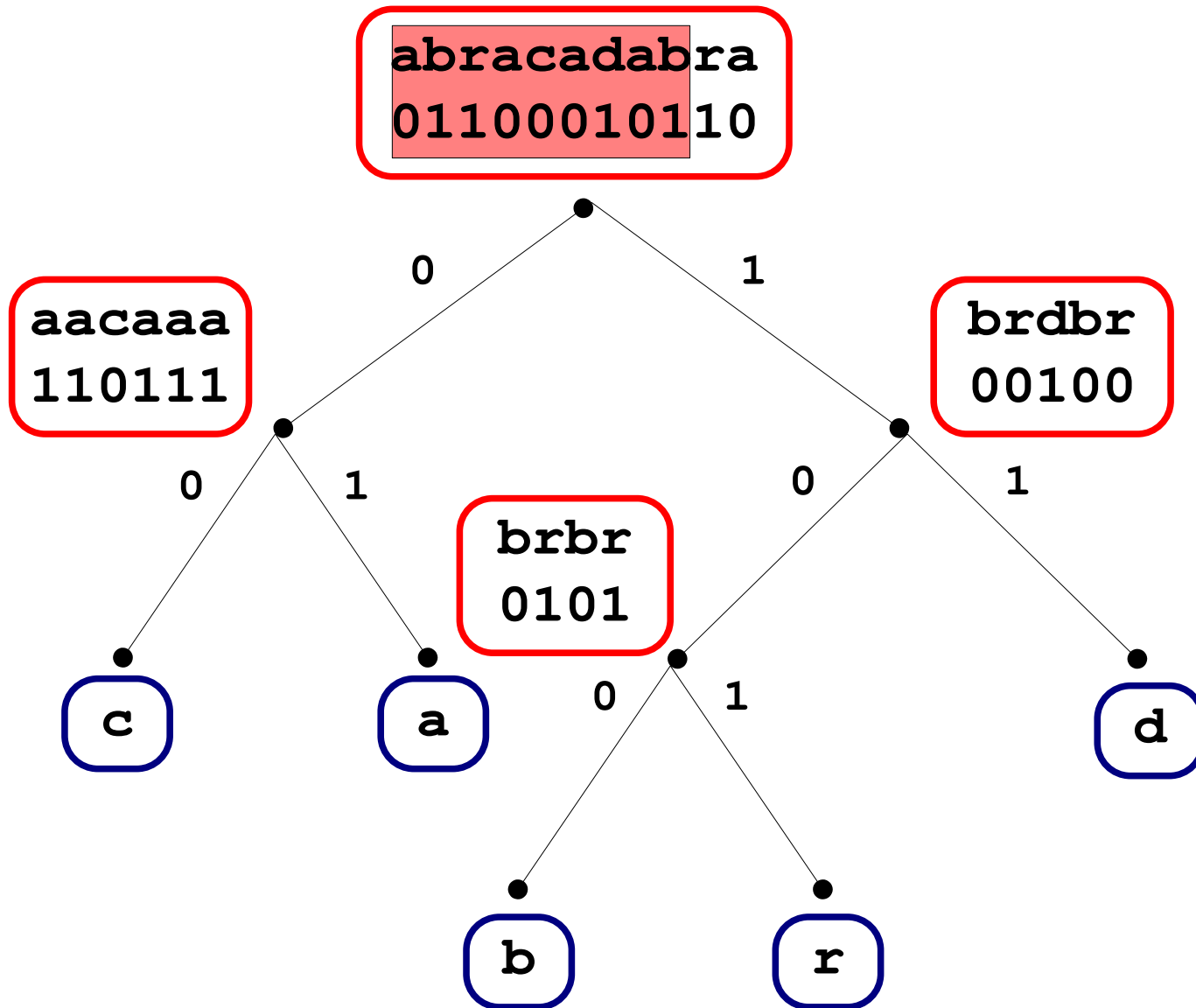
Example: find the string represented by the following Wavelet Tree:



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



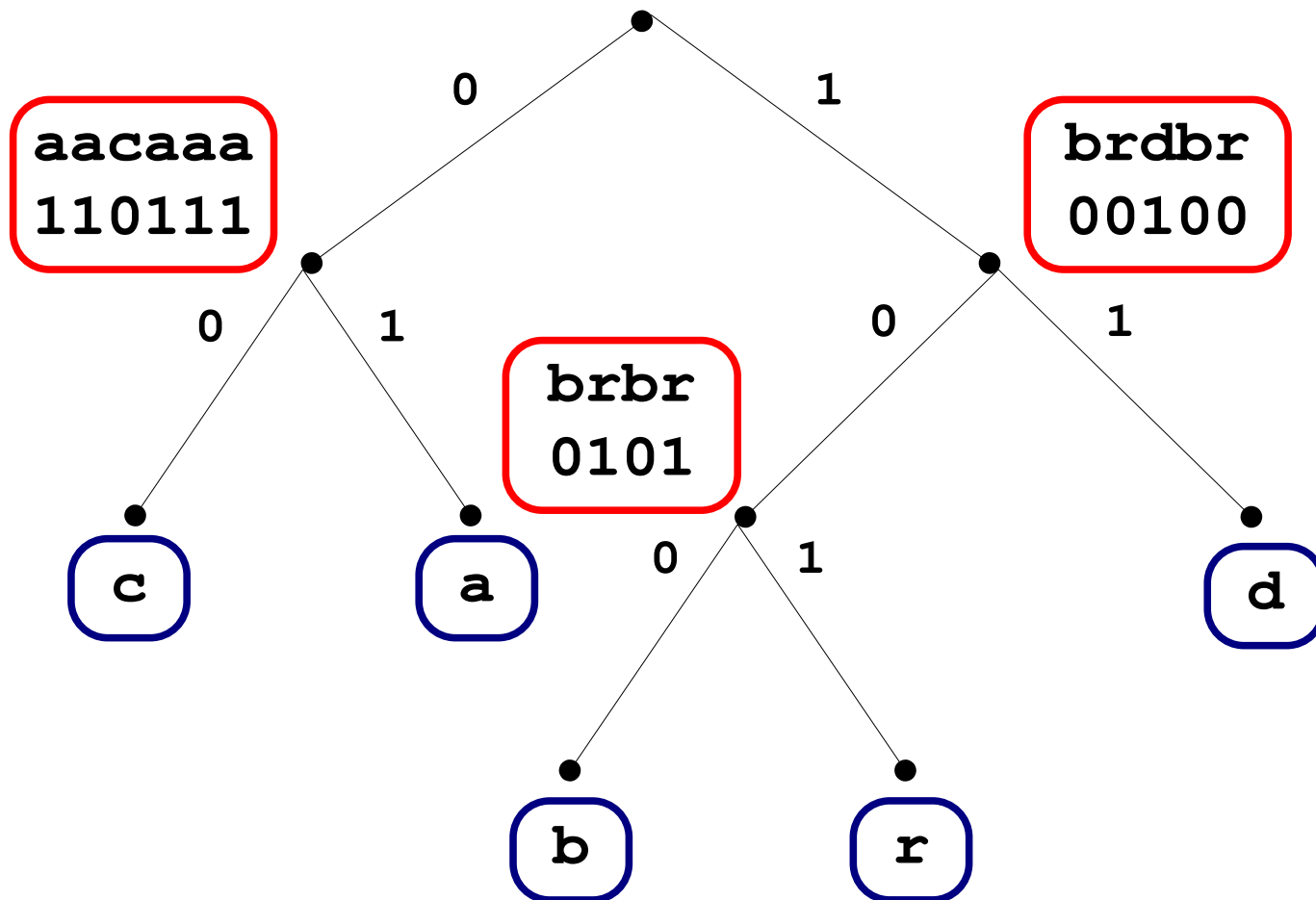
Suppose we want to compute $\text{rank}(b, 9)$



Suppose we want to compute $\text{rank}(b, 9)$

abracadabra
01100010110

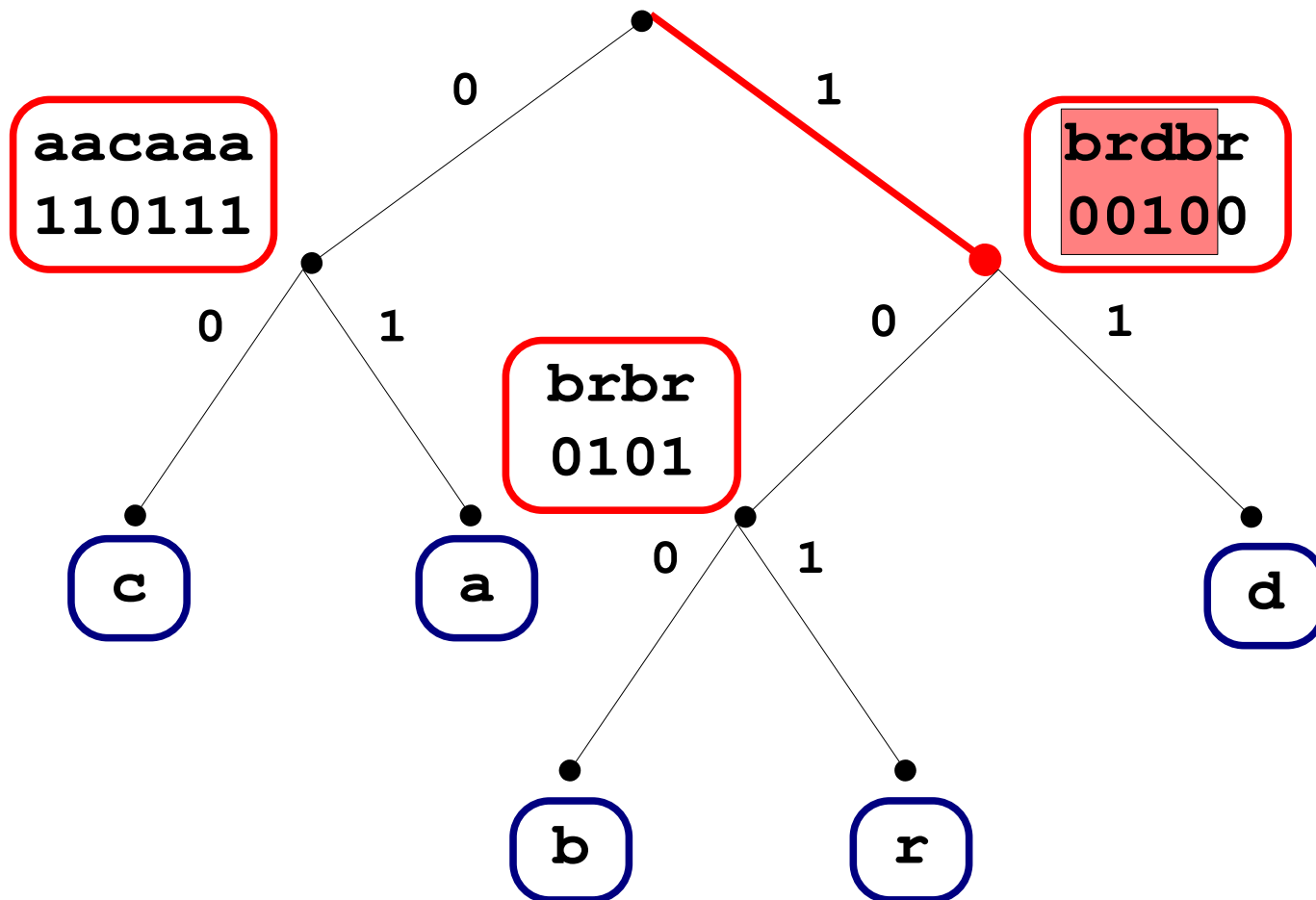
$$\text{rk}_1(9) = \text{rk}(b, 9) + \text{rk}(r, 9) + \text{rk}(d, 9) = 4$$



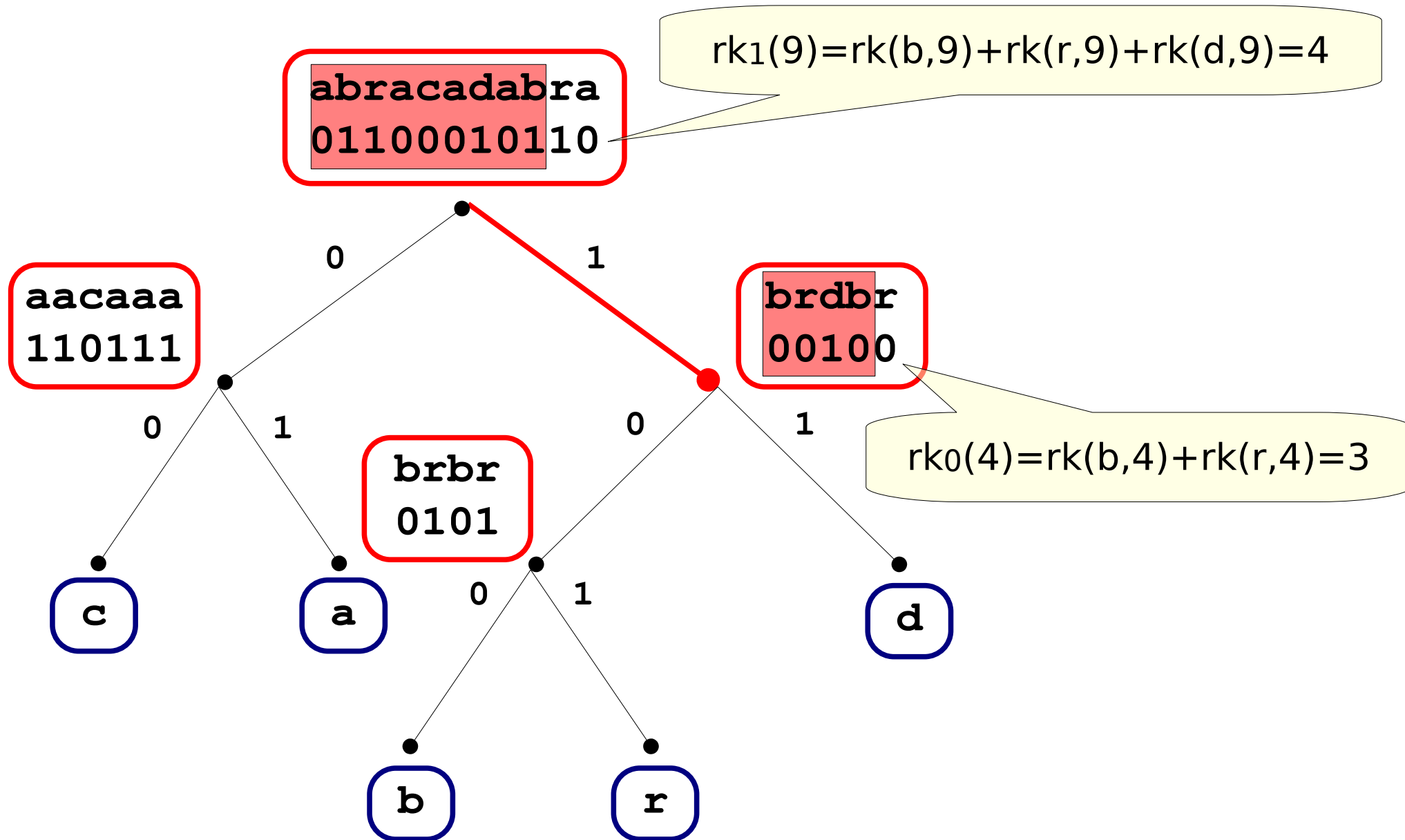
Suppose we want to compute $\text{rank}(b, 9)$

abracadabra
01100010110

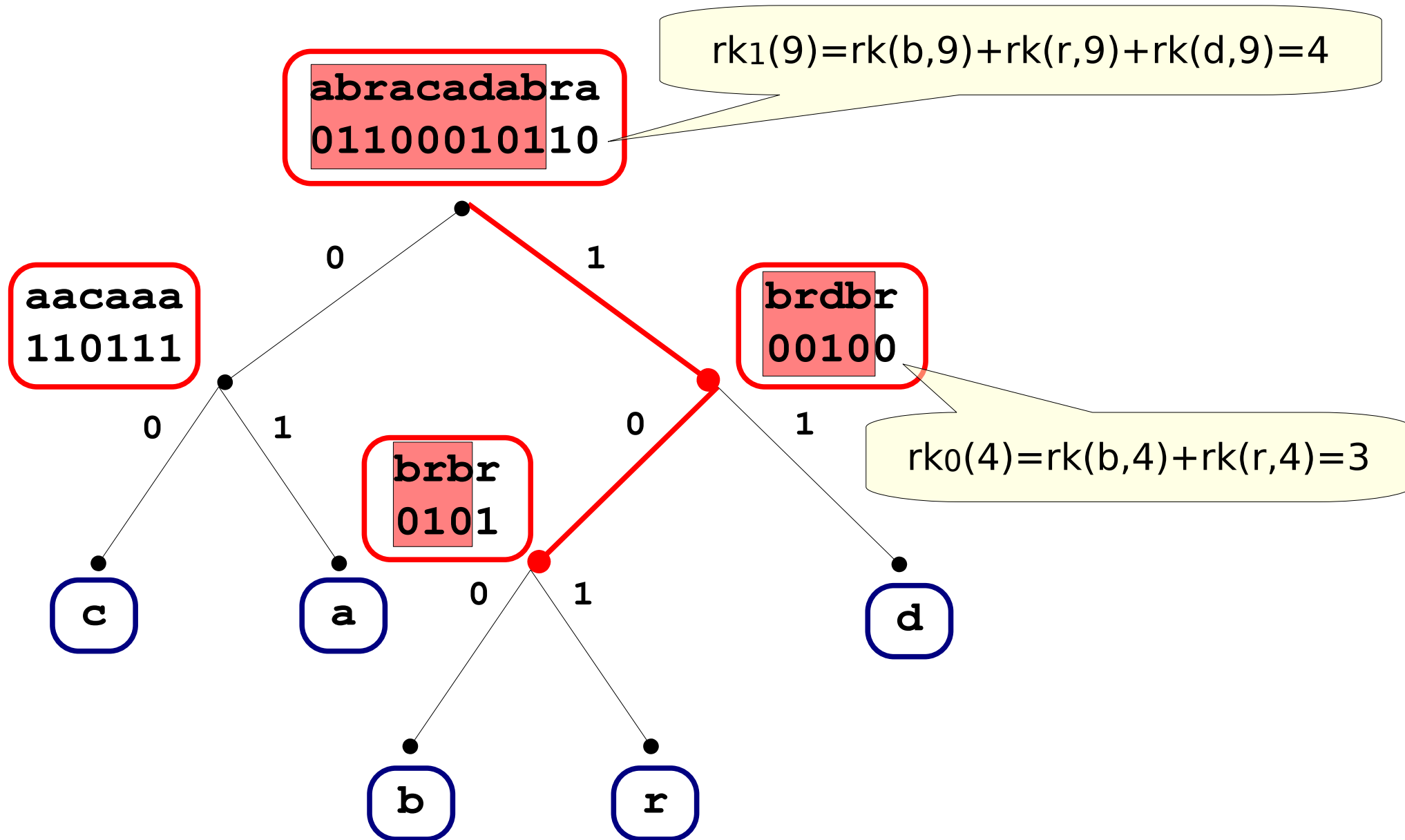
$$\text{rk}_1(9) = \text{rk}(b, 9) + \text{rk}(r, 9) + \text{rk}(d, 9) = 4$$



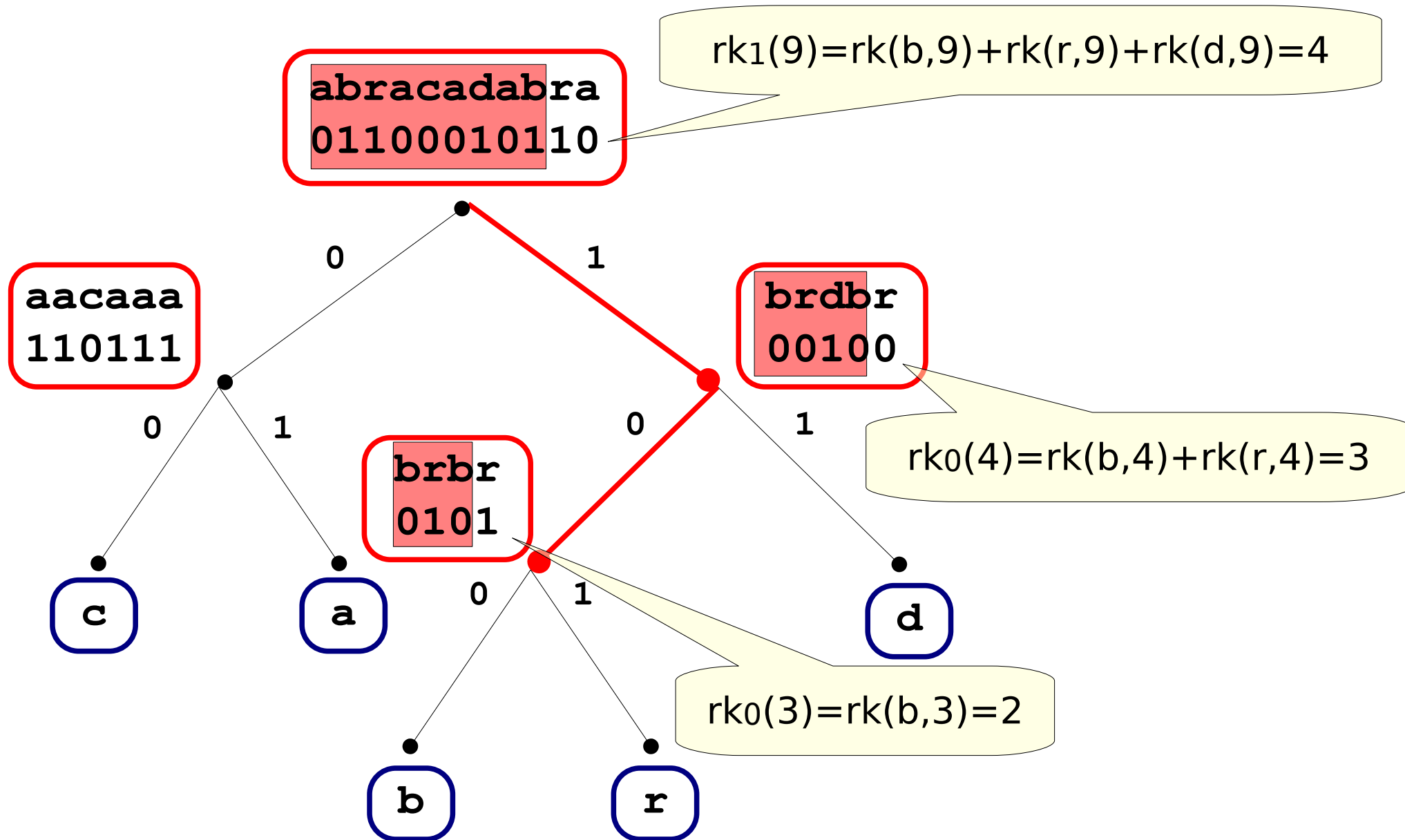
Suppose we want to compute $\text{rank}(b,9)$



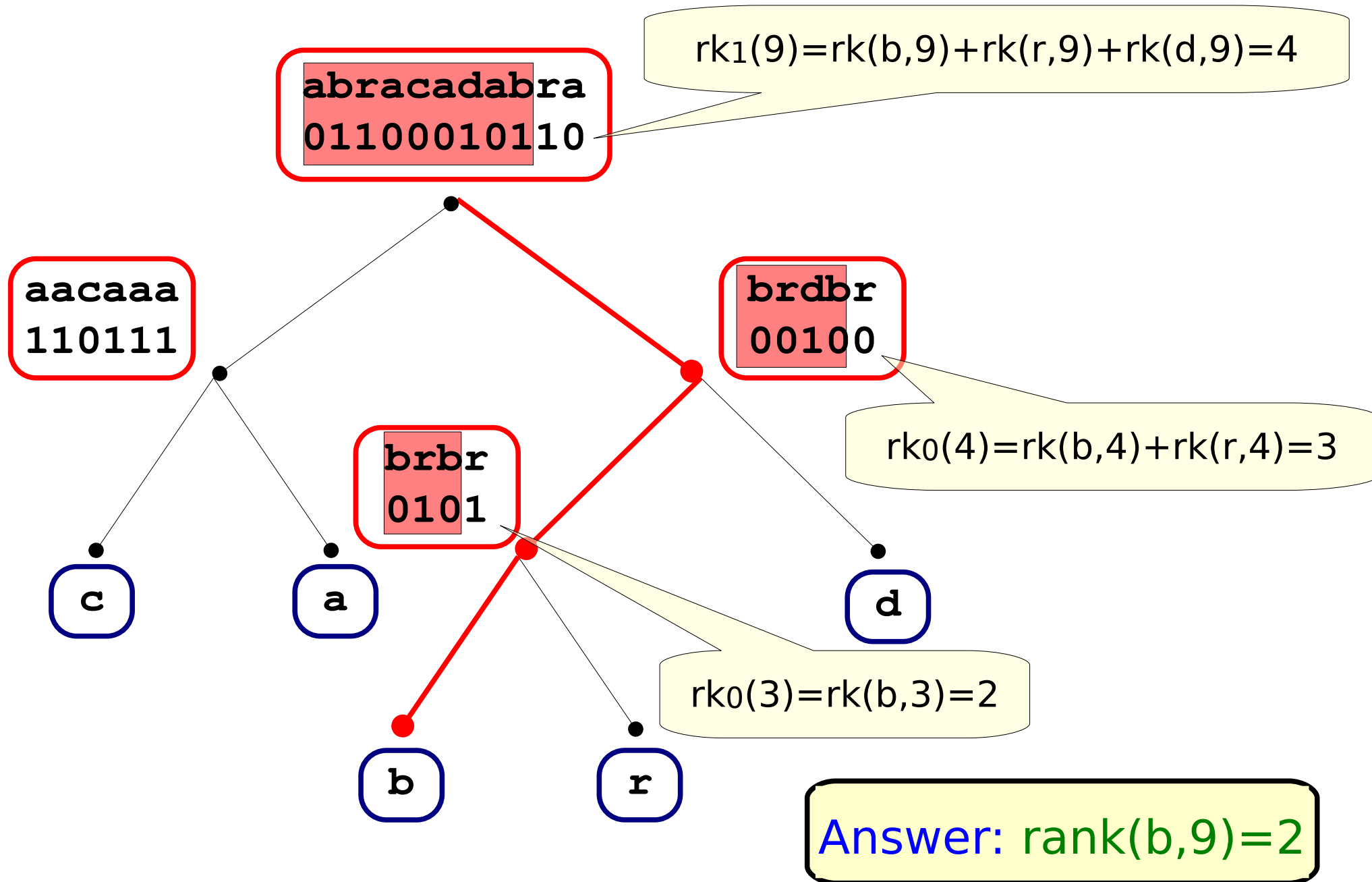
Suppose we want to compute $\text{rank}(b, 9)$



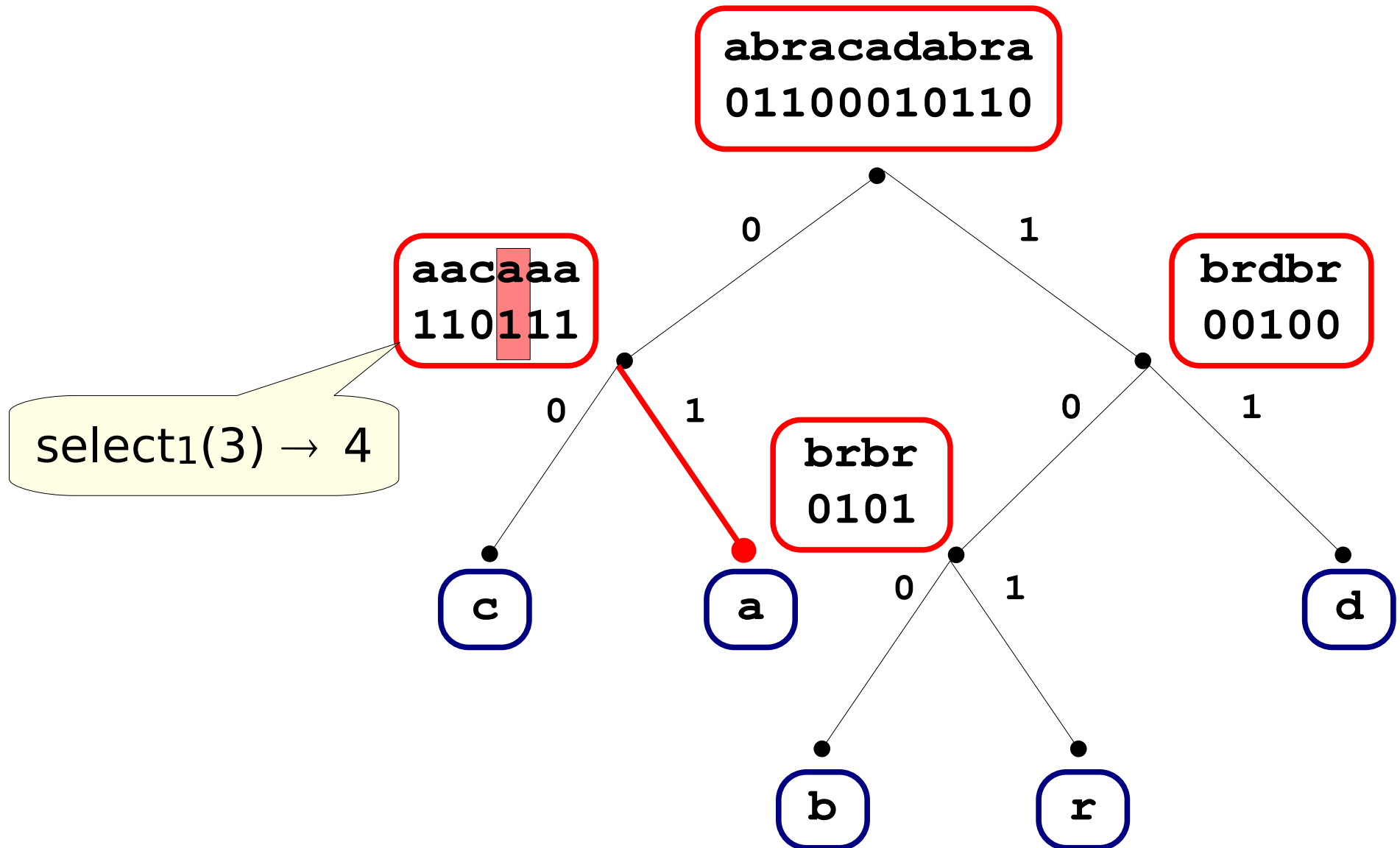
Suppose we want to compute $\text{rank}(b, 9)$



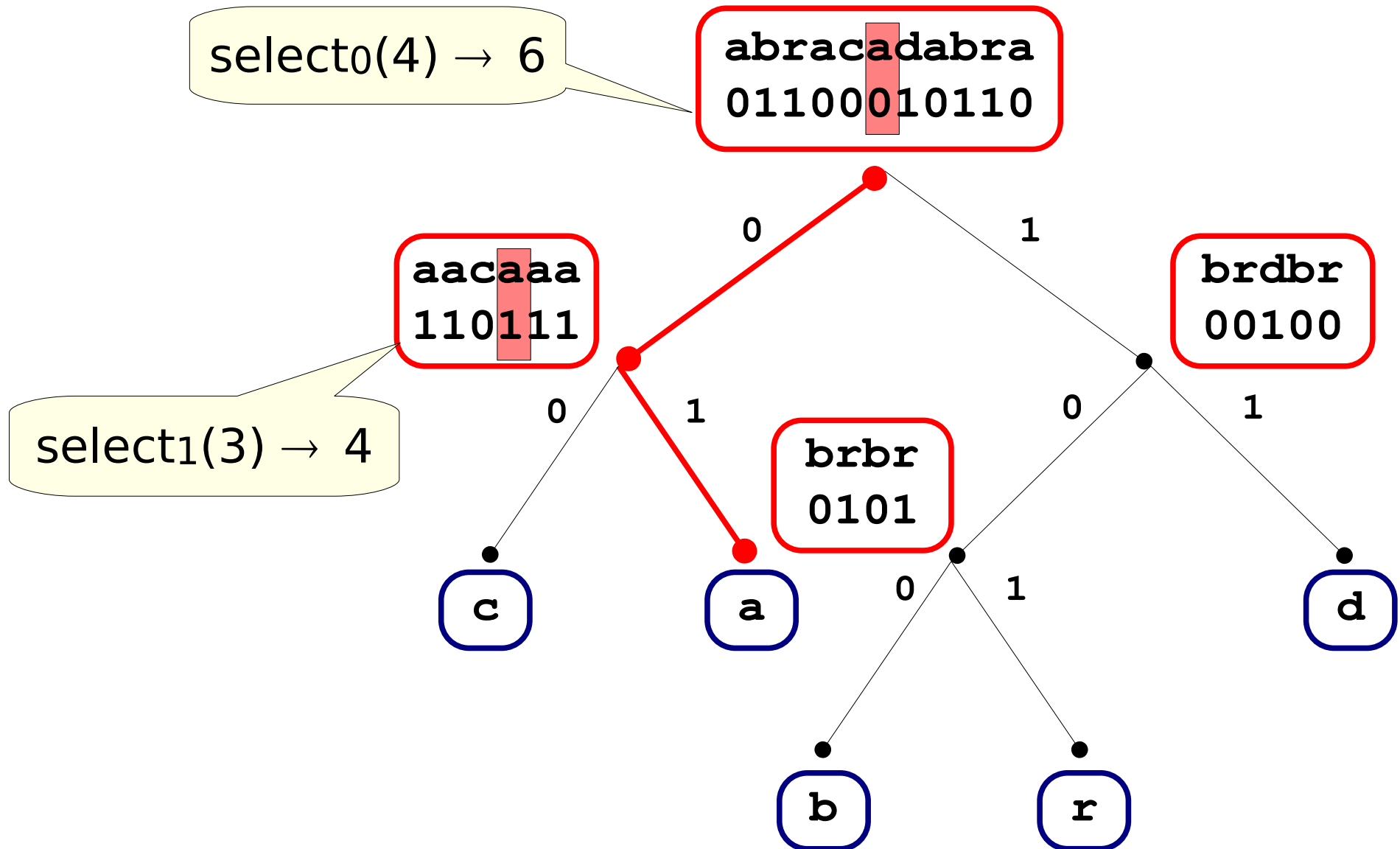
Suppose we want to compute $\text{rank}(b, 9)$



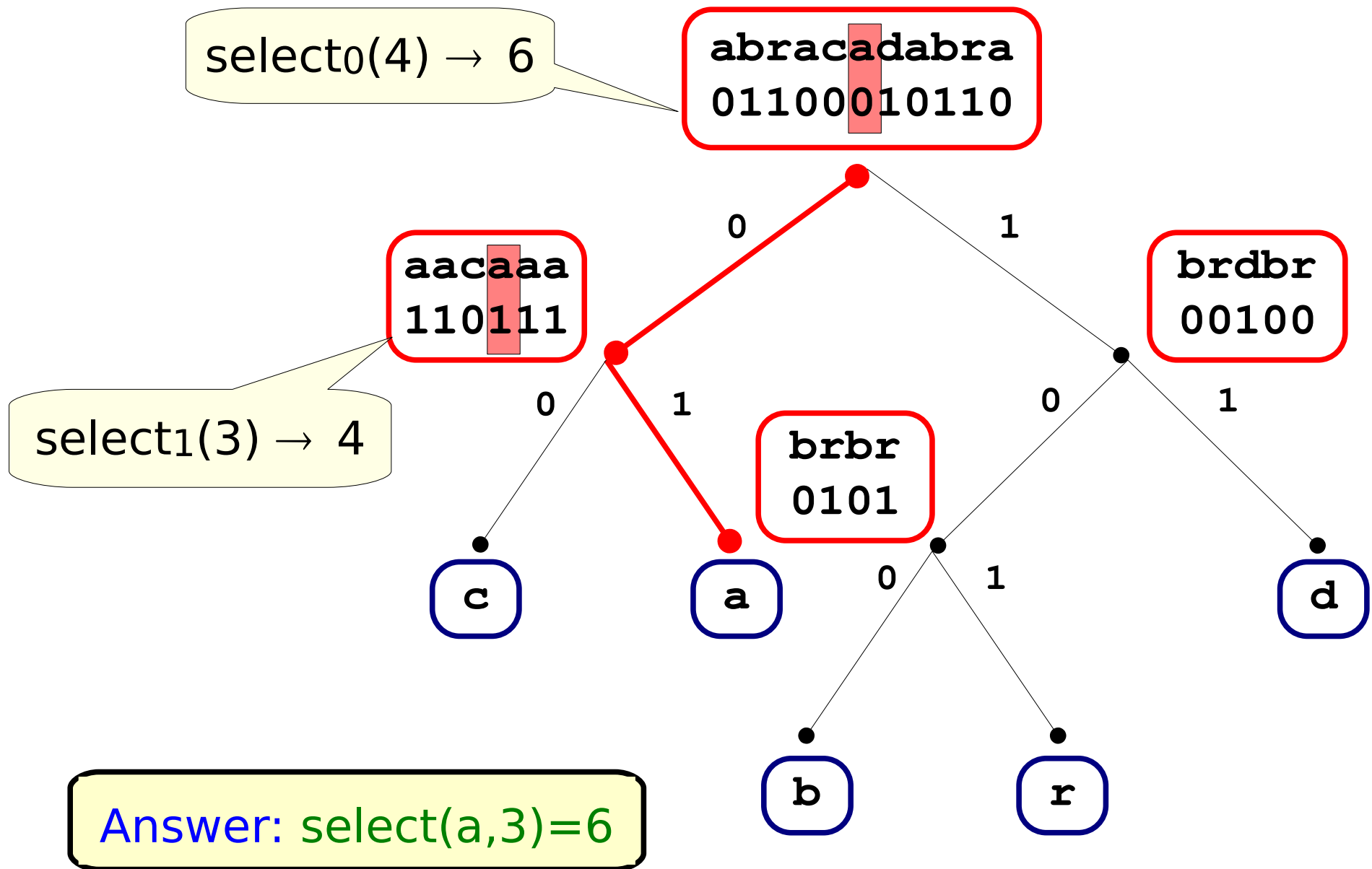
Select queries go upward. To compute $\text{select}(a, 3)$



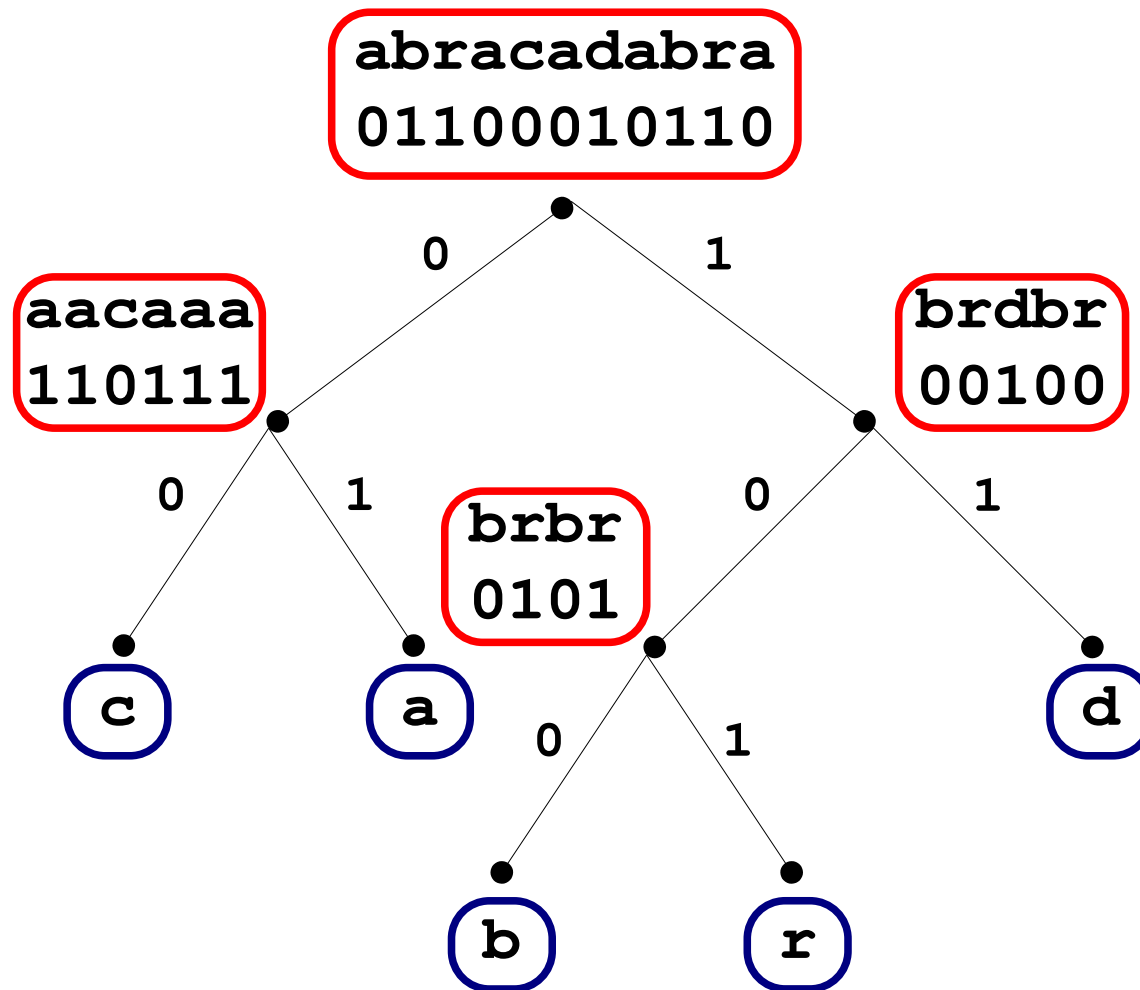
Select queries go upward. To compute $\text{select}(a, 3)$



Select queries go upward. To compute $\text{select}(a,3)$



Wavelet Trees as scrambled prefix codes



We are implicitly using the encoding:

$c \rightarrow 00$ $a \rightarrow 01$ $b \rightarrow 100$ $r \rightarrow 101$ $d \rightarrow 11$

Wavelet Trees as scrambled prefix codes:

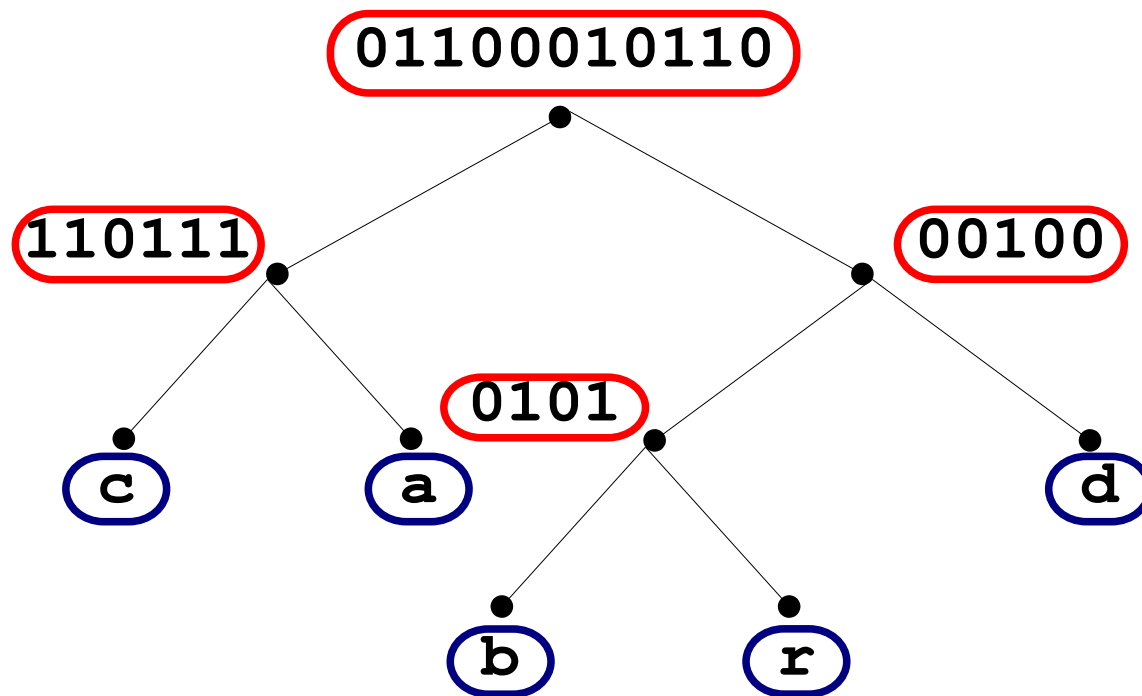
We are implicitly using the encoding:

$a \rightarrow 01$ $c \rightarrow 00$ $b \rightarrow 100$ $r \rightarrow 101$ $d \rightarrow 11$

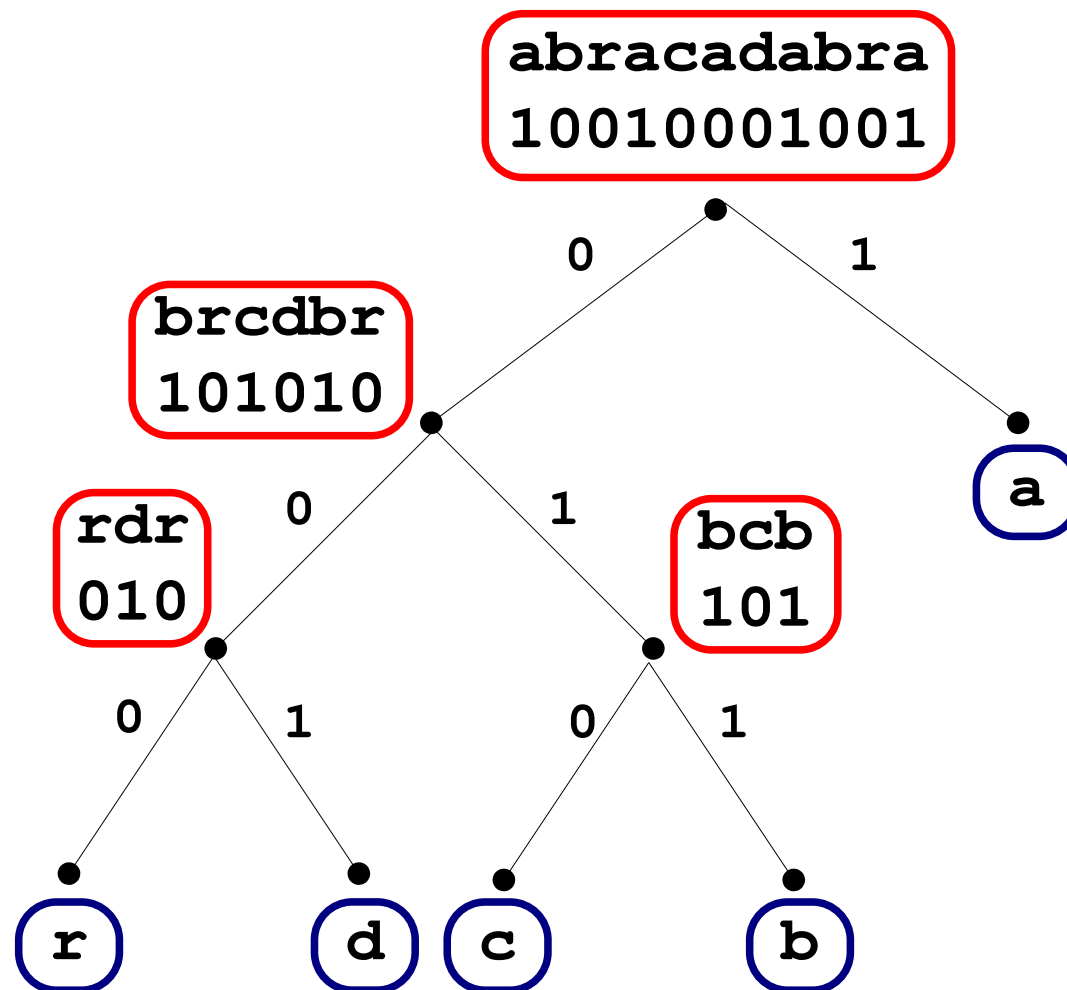
Traditional prefix codes:

abracadabra $\rightarrow$ *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.

The cost of the rank/select operations is proportional to the tree height.

With a tree shaped as a Huffman tree the cost is smaller for frequent symbols and larger for less frequent symbols...

...or we can use a balanced Wavelet tree: if $|A|$ is the alphabet size the height is bounded by $\lceil \log |A| \rceil$ and we use $\approx \log |A|$ bits per symbol.

Other Wavelet Trees virtues

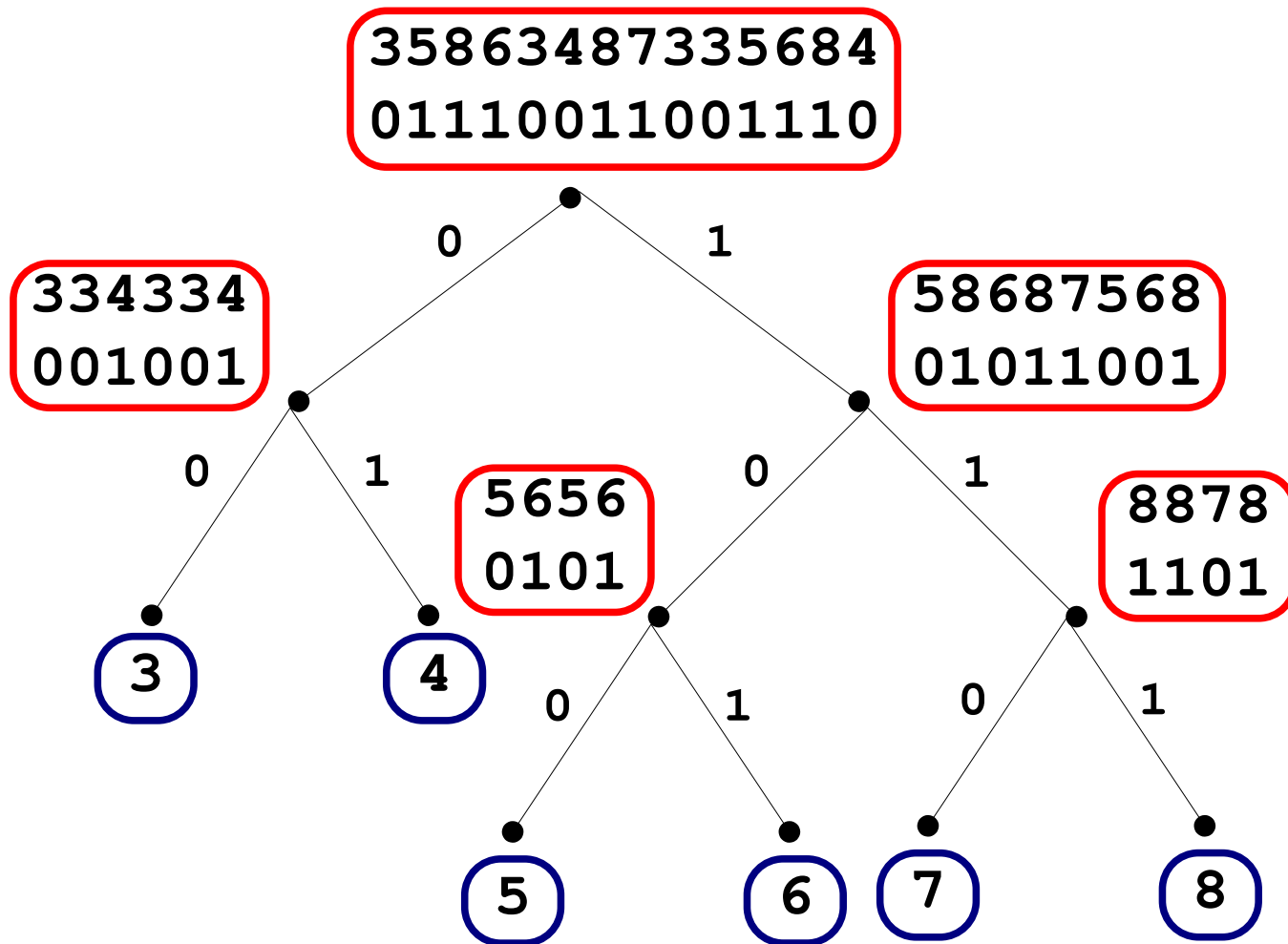
Simple solutions for range queries on an integer sequence $S[1,n]$, for example

`range_quantile(S,i,j,k)`: return the k -th smallest value in $S[i \cdots j]$

`range_count(S,i,j)`: return # of distinct values in $S[i \cdots j]$

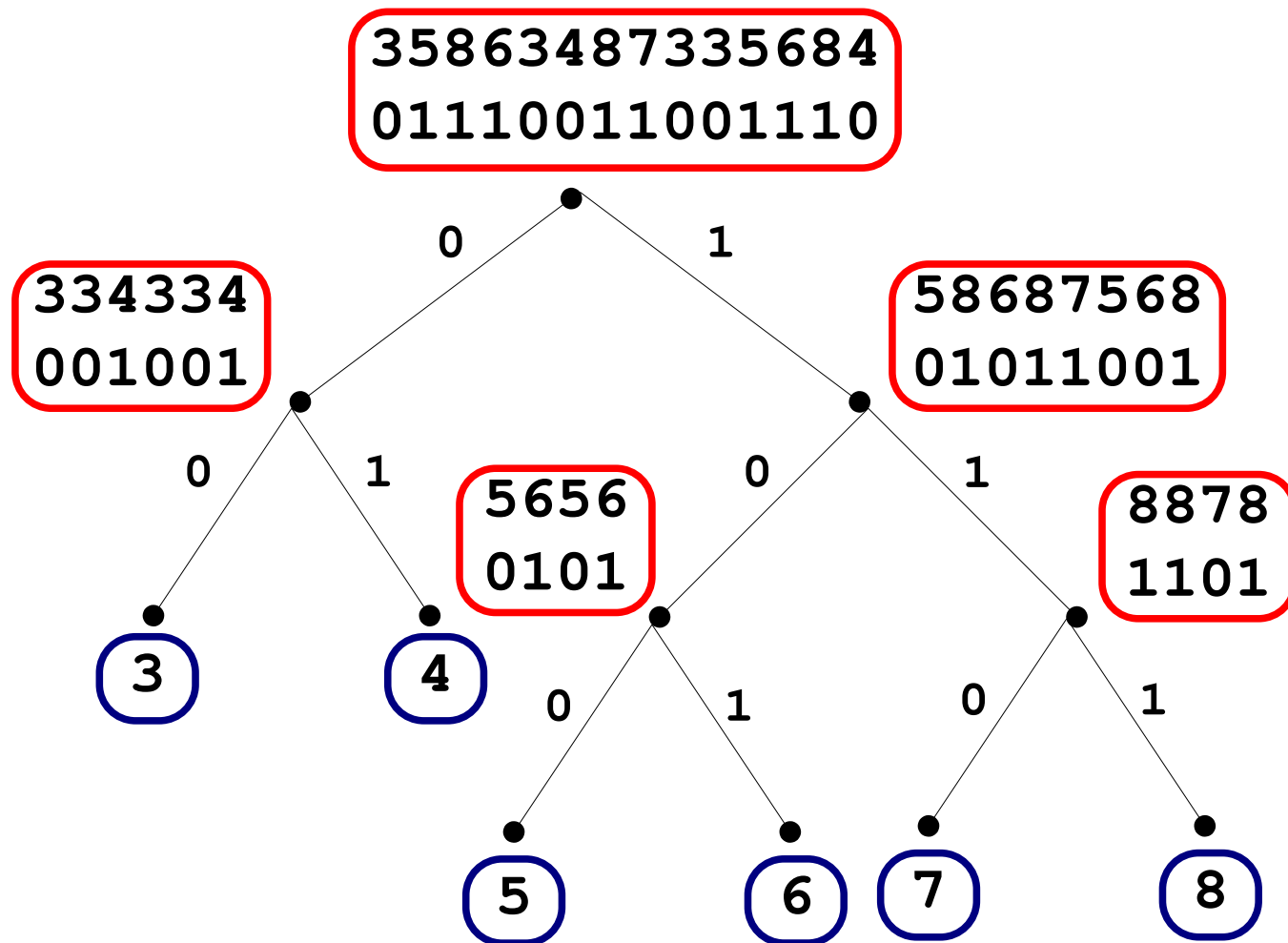
Example.

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



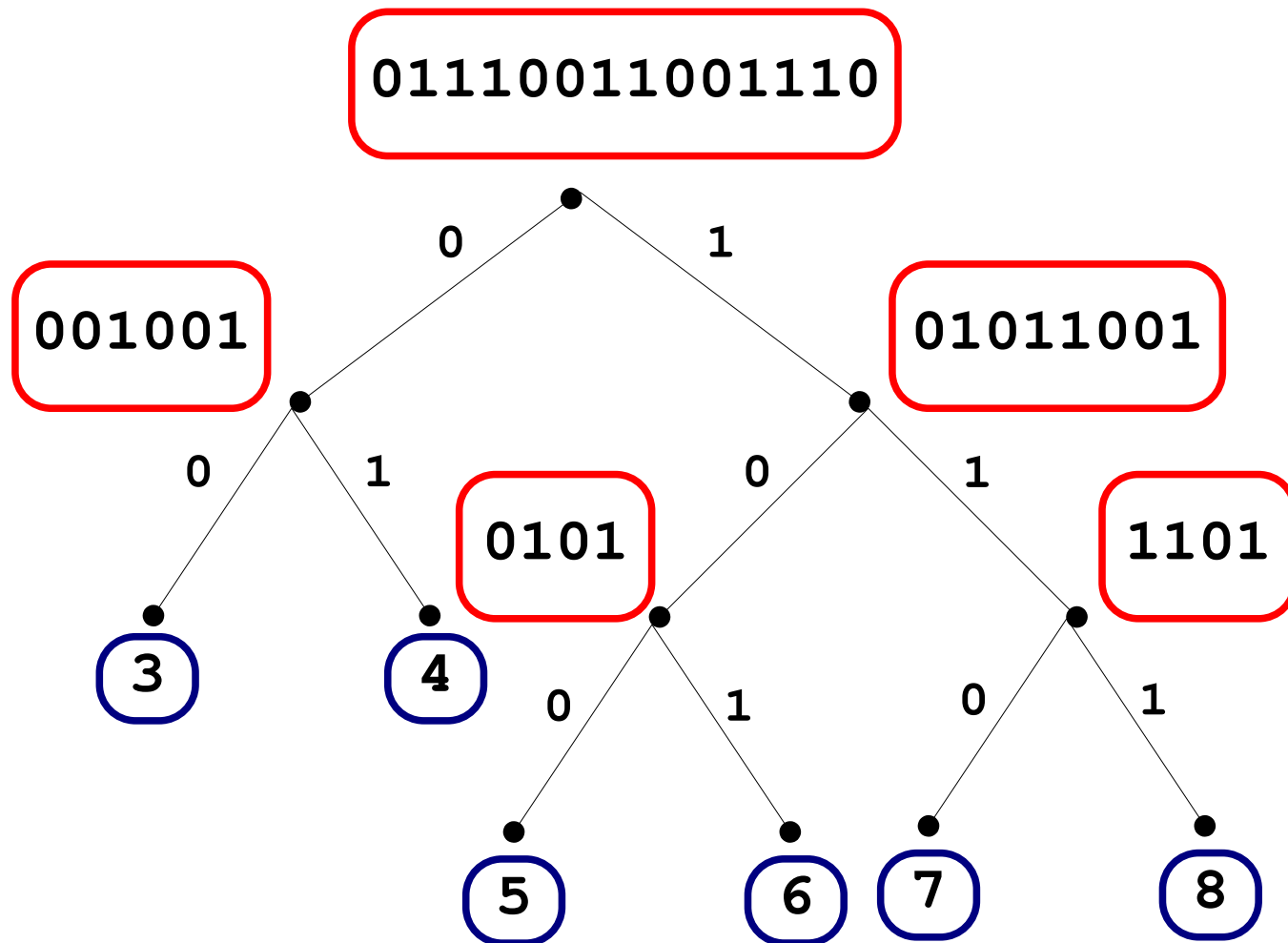
range_quantile(S,2,7,3): 3-rd smallest value in S[2...7]

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



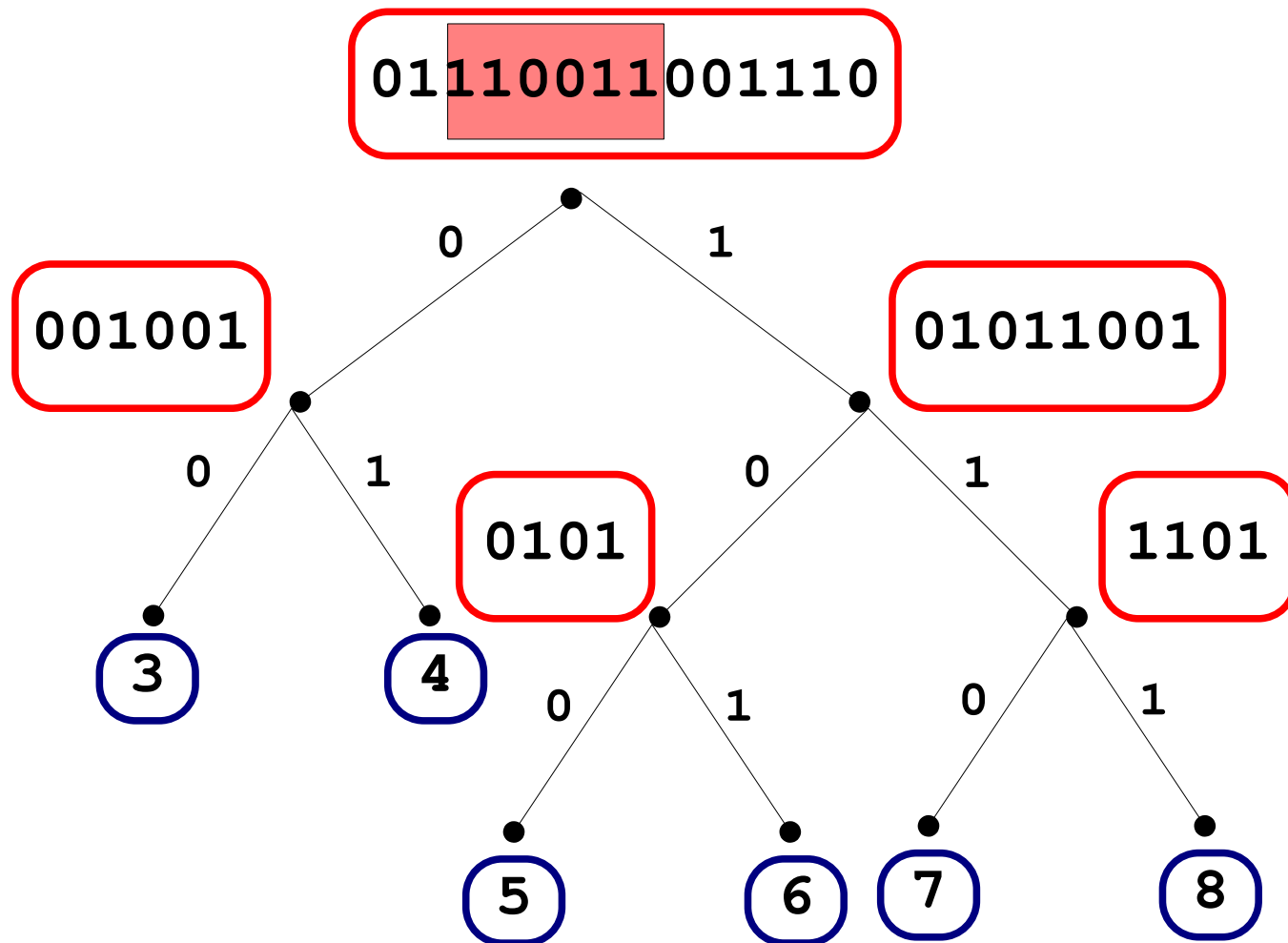
`range_quantile(S,2,7,3)`: 3-rd smallest value in $S[2\cdots 7]$

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



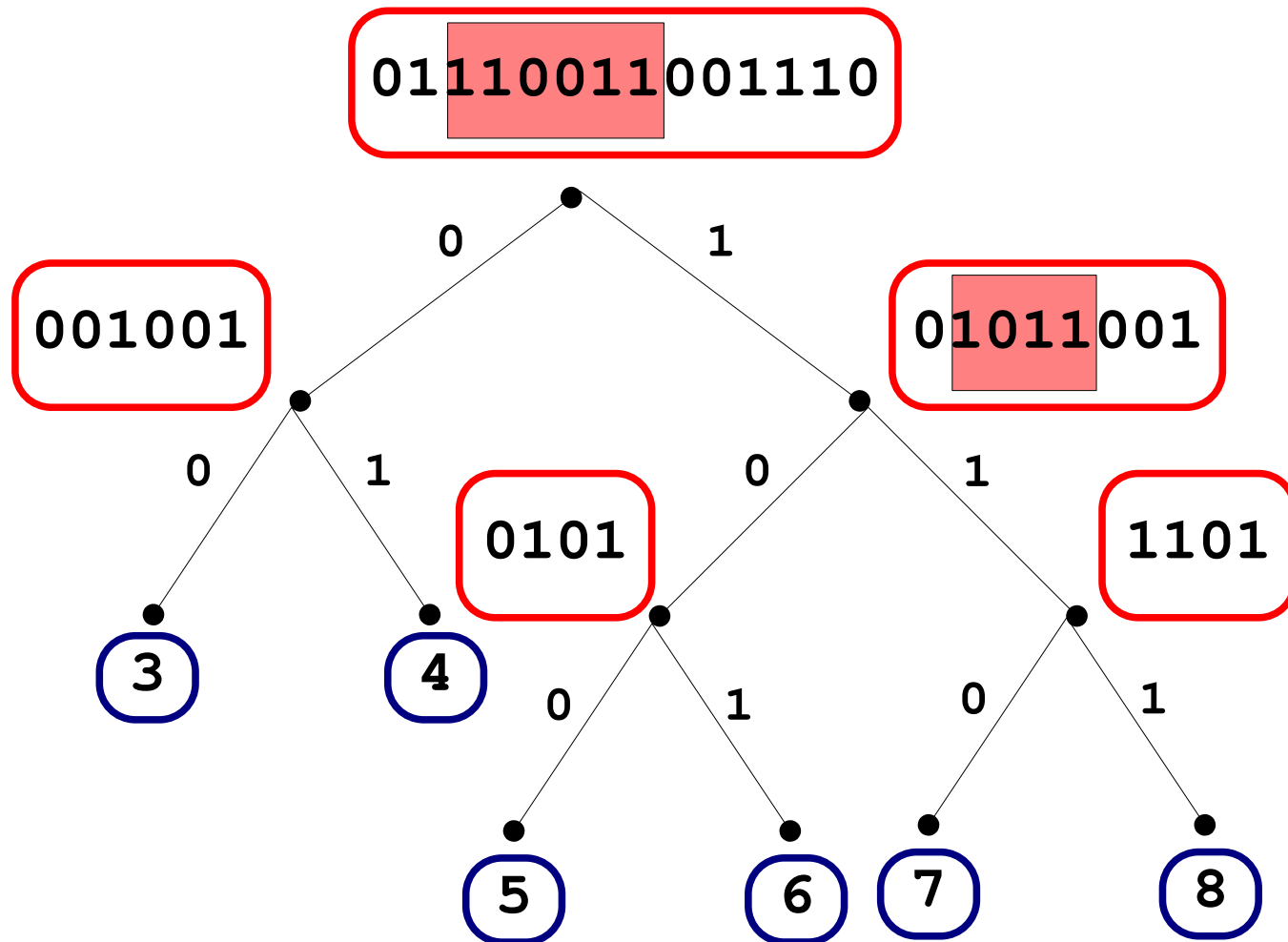
range_quantile(S,2,7,3): 3-rd smallest value in S[2...7]

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



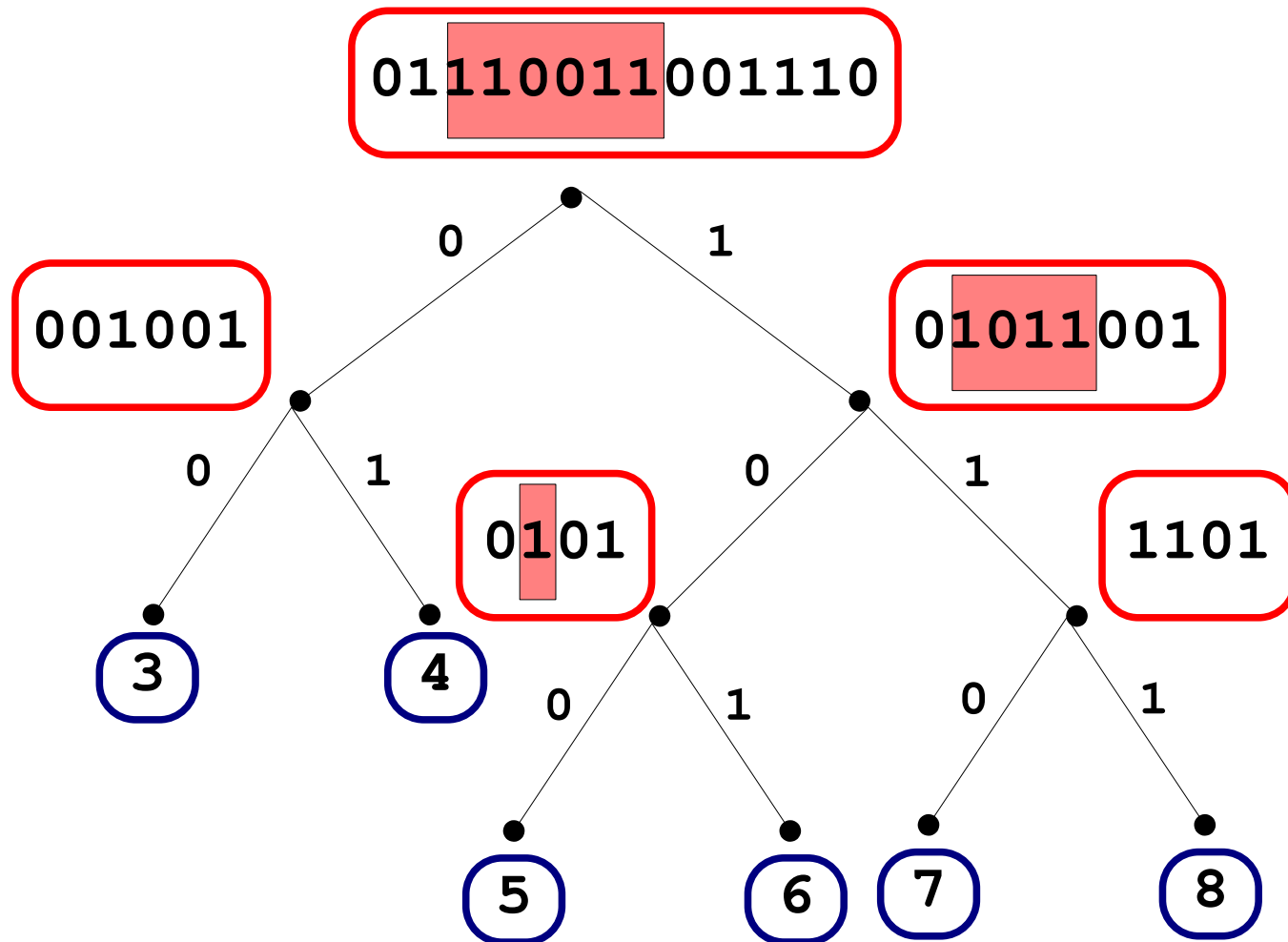
range_quantile(S,2,7,3): 3-rd smallest value in S[2...7]

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



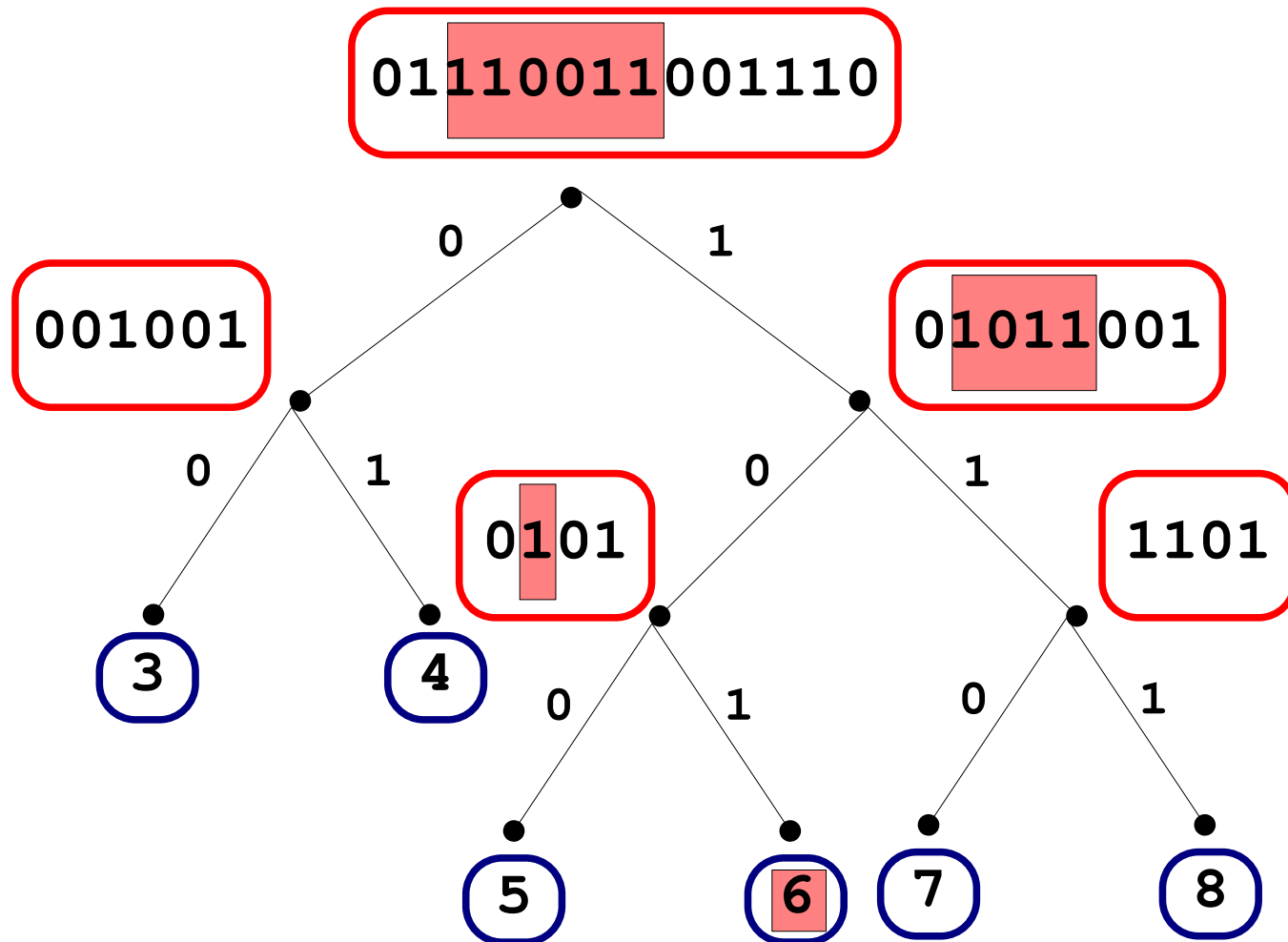
range_quantile(S,2,7,3): 3-rd smallest value in S[2...7]

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



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 (review paper on WTs describing many applications)