Wavelet Trees

02-714
Slides by Carl Kingsford

Following: Navarro,
Wavelet Trees for All, CPM 2012, pp. 2-26

Operations on strings

- $\text{rank}_c(S,i)$:= the number of char c at or before position i in S.
- $\text{select}_c(S,j)$:= the position of the j^{th} occurrence of c in S.
- $S[i]$ = "access character i"

Note: $\text{rank}_c(S, \text{select}_c(S, j)) = j$, so rank and select are inverses of each other.

Goal: rank, select, access in quickly while using small space.

Operations on bit vectors

Our operations will depend on similar operations on bit vectors:

- $\text{rank}_1(S,i)$:= the number of 1 bits at or before position *i* in *S*.
- $\text{select}_1(S,j)$:= the position of the j^{th} 1 bit in S.
- $\text{rank}_0(S,i)$ and $\text{select}_0(S,j)$ are defined analogously.

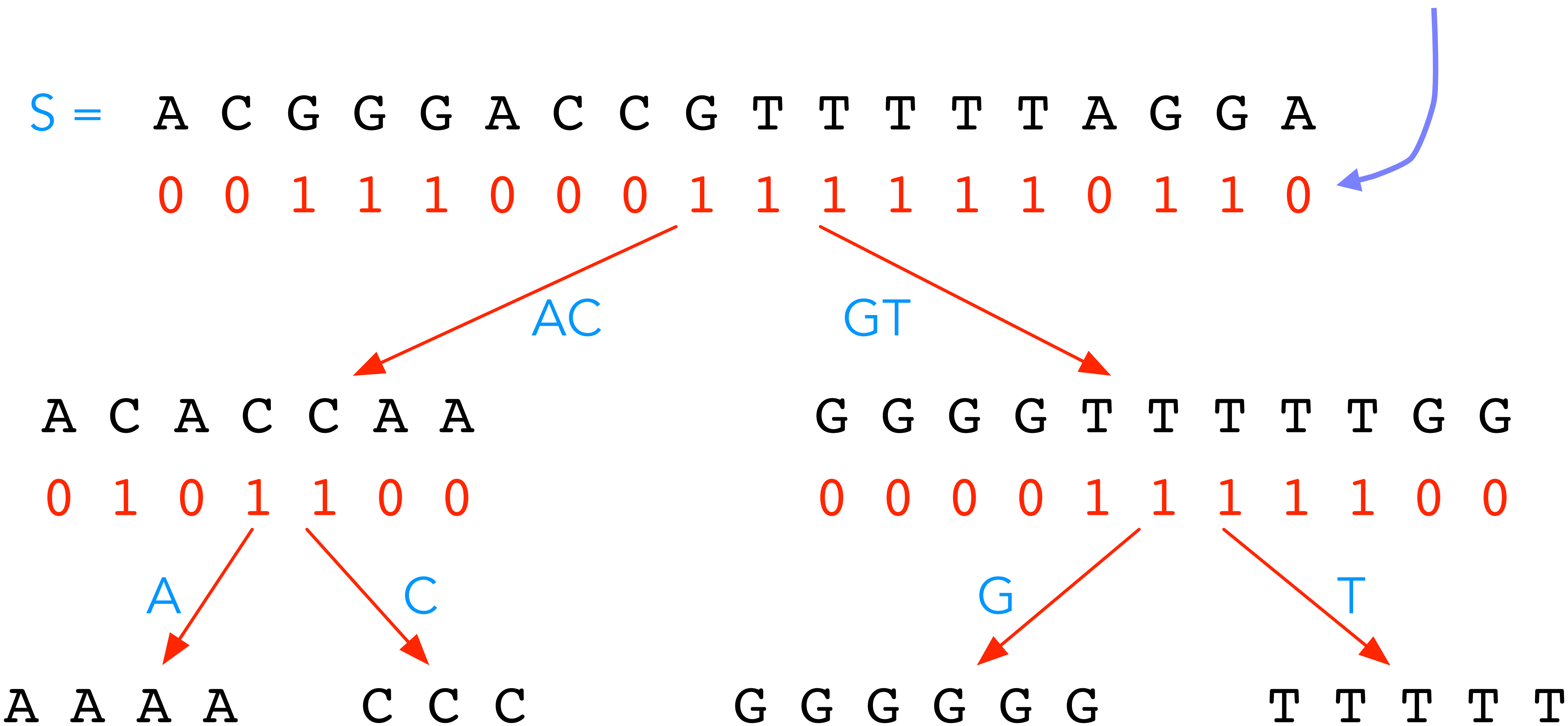
$S[i]$ = "access bit i" = $\text{rank}_1(S, i) - \text{rank}_1(S, i-1)$

We will see later how to implement these operations to run in O(1) time for binary vectors.

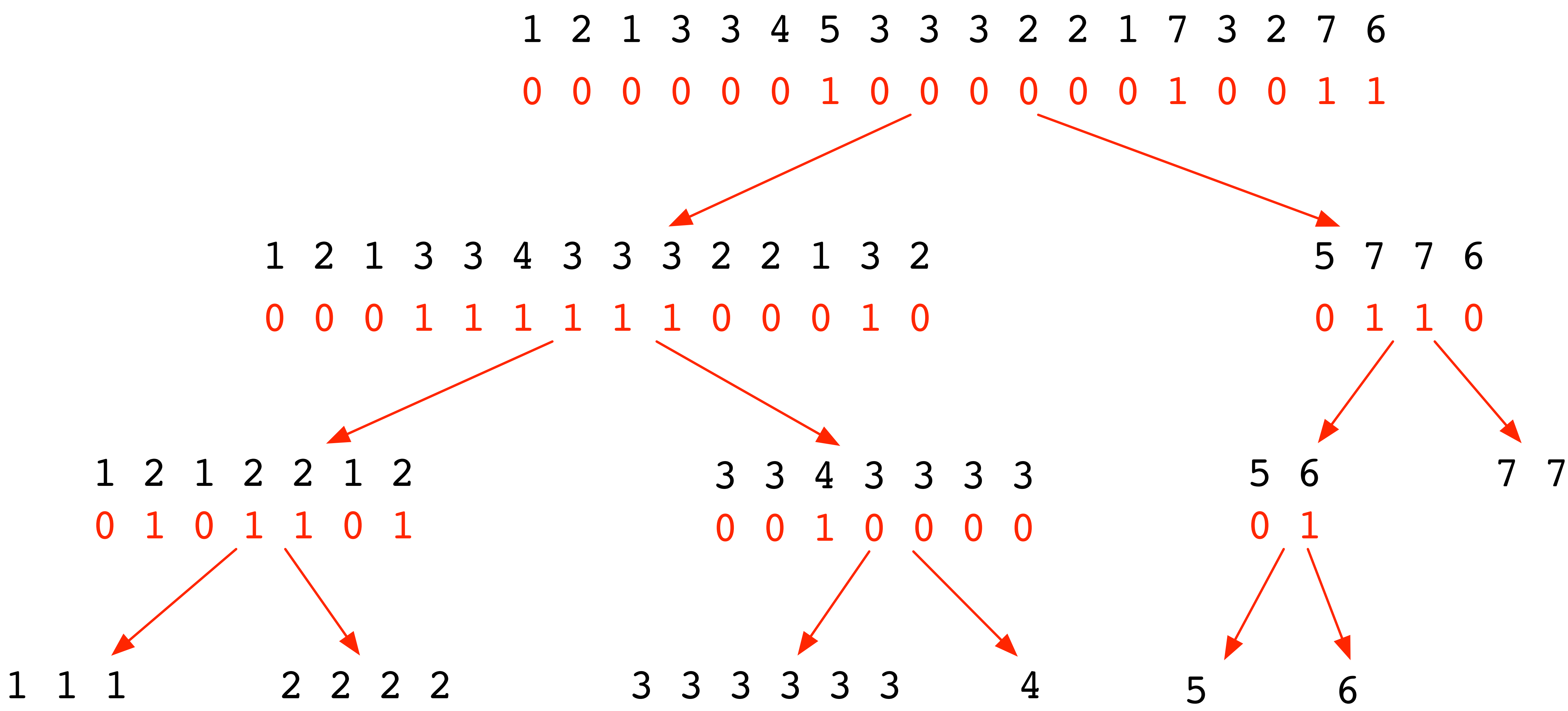
Wavelet Tree

0: letter ∈ first half of alphabet

1: letter ∈ first half of alphabet



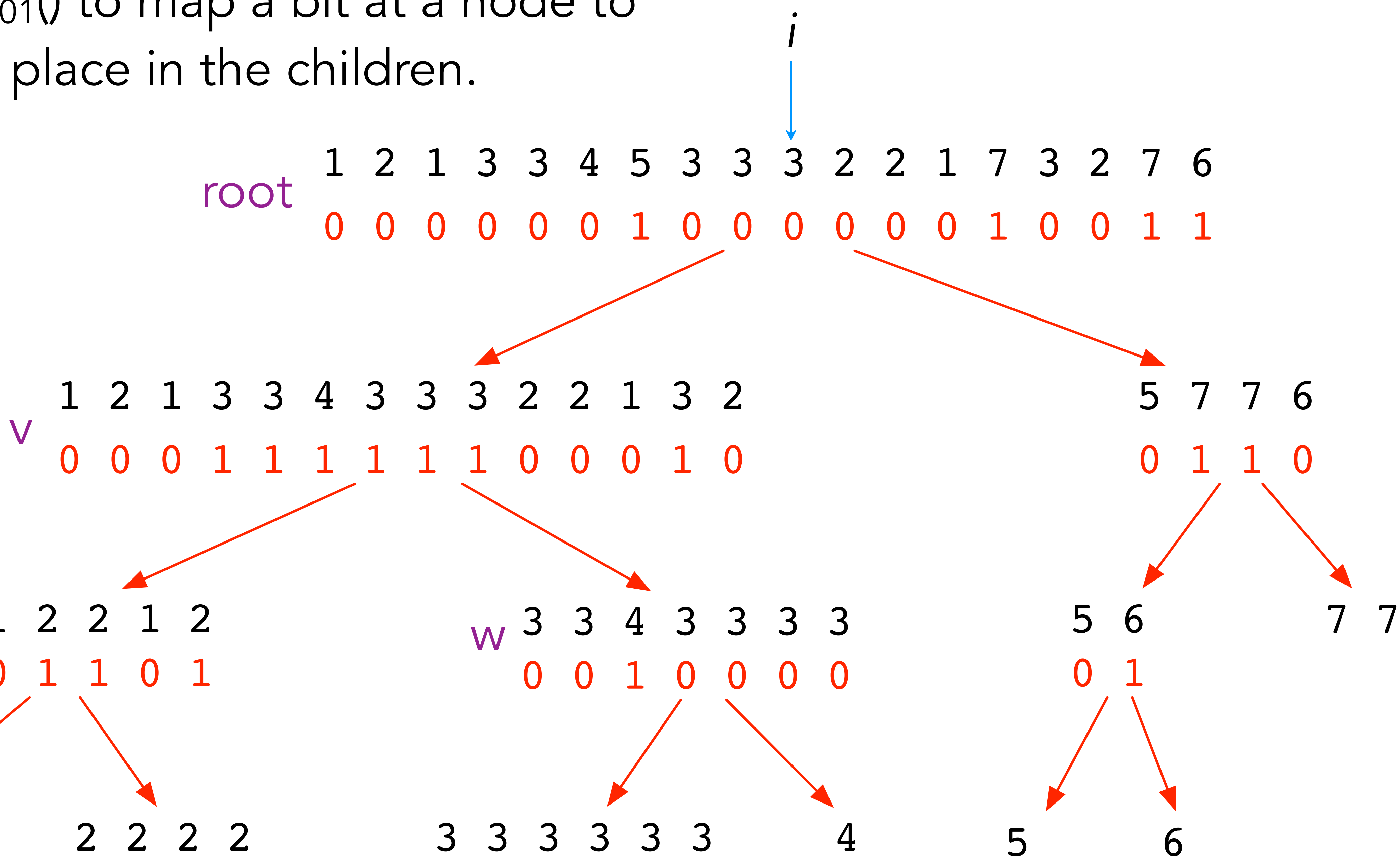
Wavelet Tree, Example 2



S[i]

Go left if bit is 0, go right if bit is 1.

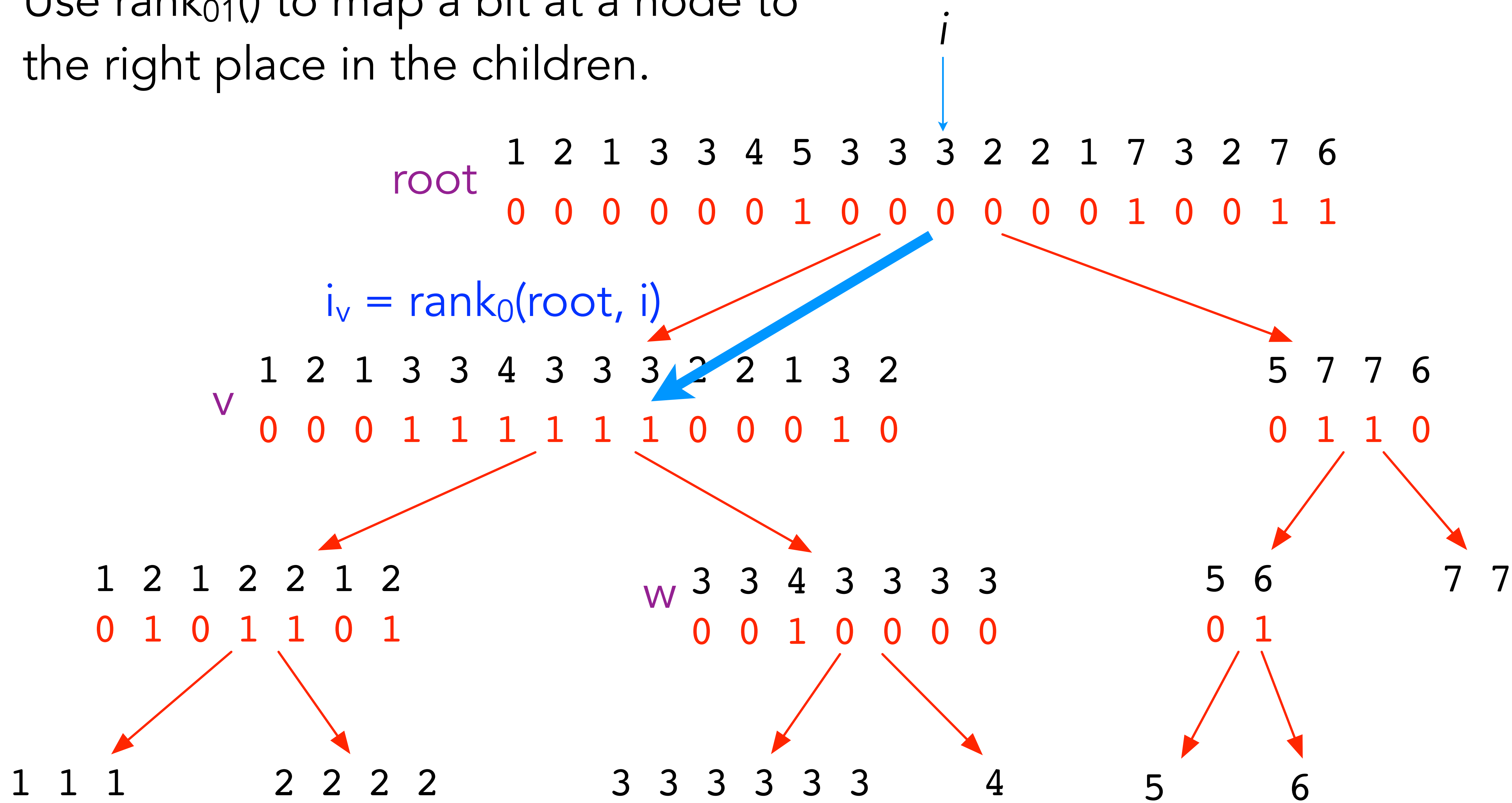
Use rank_{01}() to map a bit at a node to the right place in the children.



S[i]

Go left if bit is 0, go right if bit is 1.

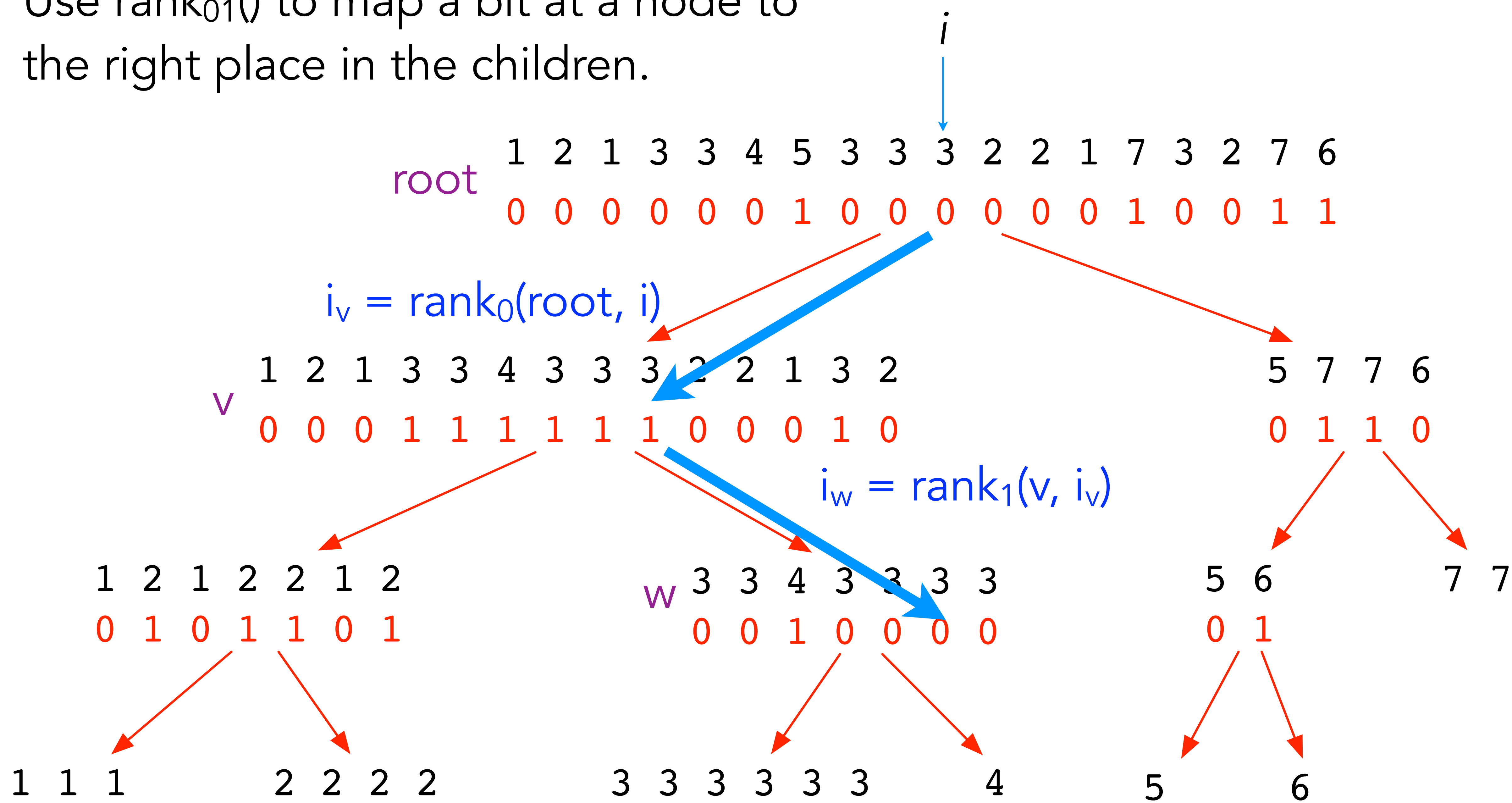
Use $rank_{01}()$ to map a bit at a node to the right place in the children.



S[i]

Go left if bit is 0, go right if bit is 1.

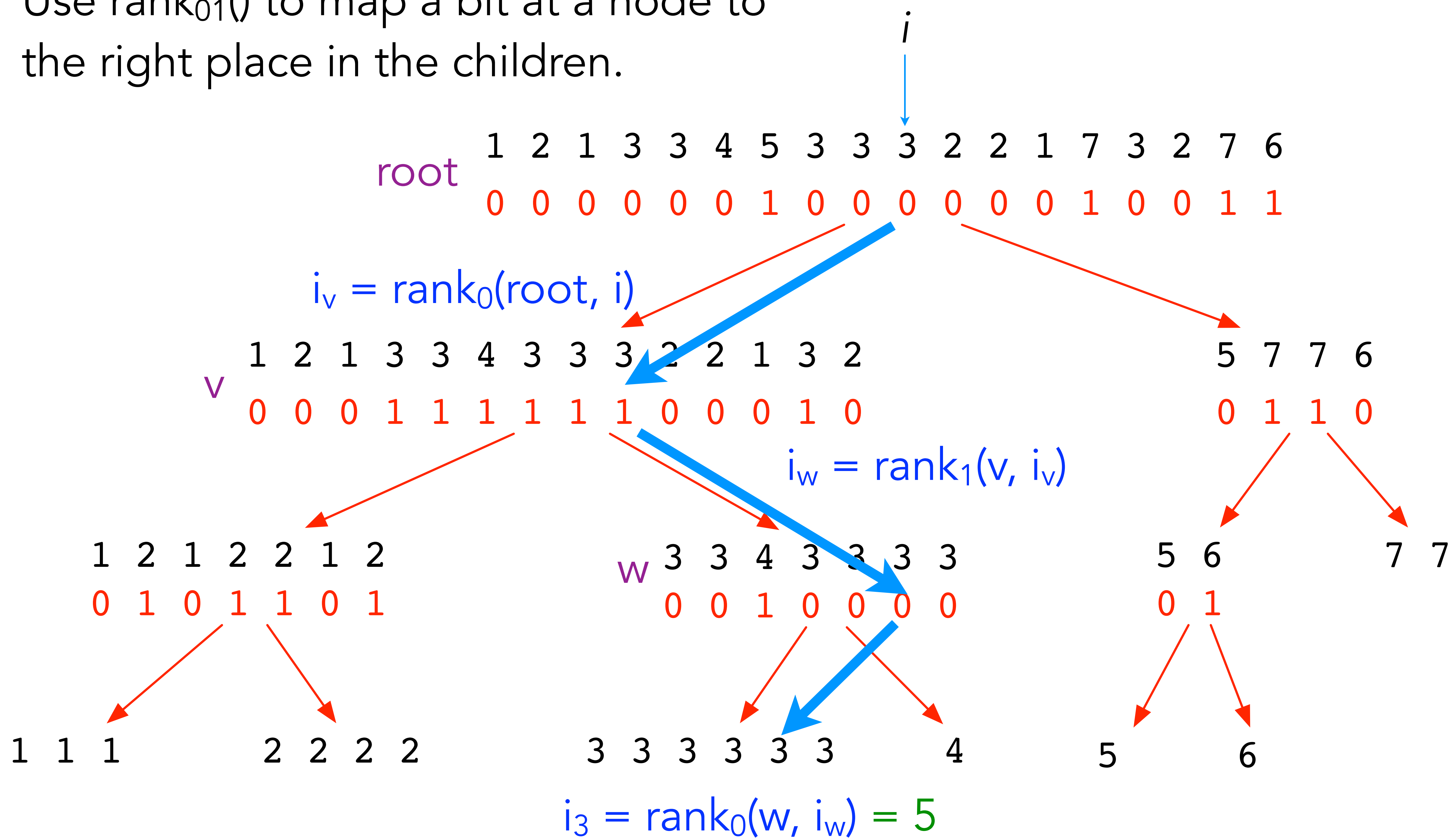
Use $\text{rank}_{01}()$ to map a bit at a node to the right place in the children.



S[i]

Go left if bit is 0, go right if bit is 1.

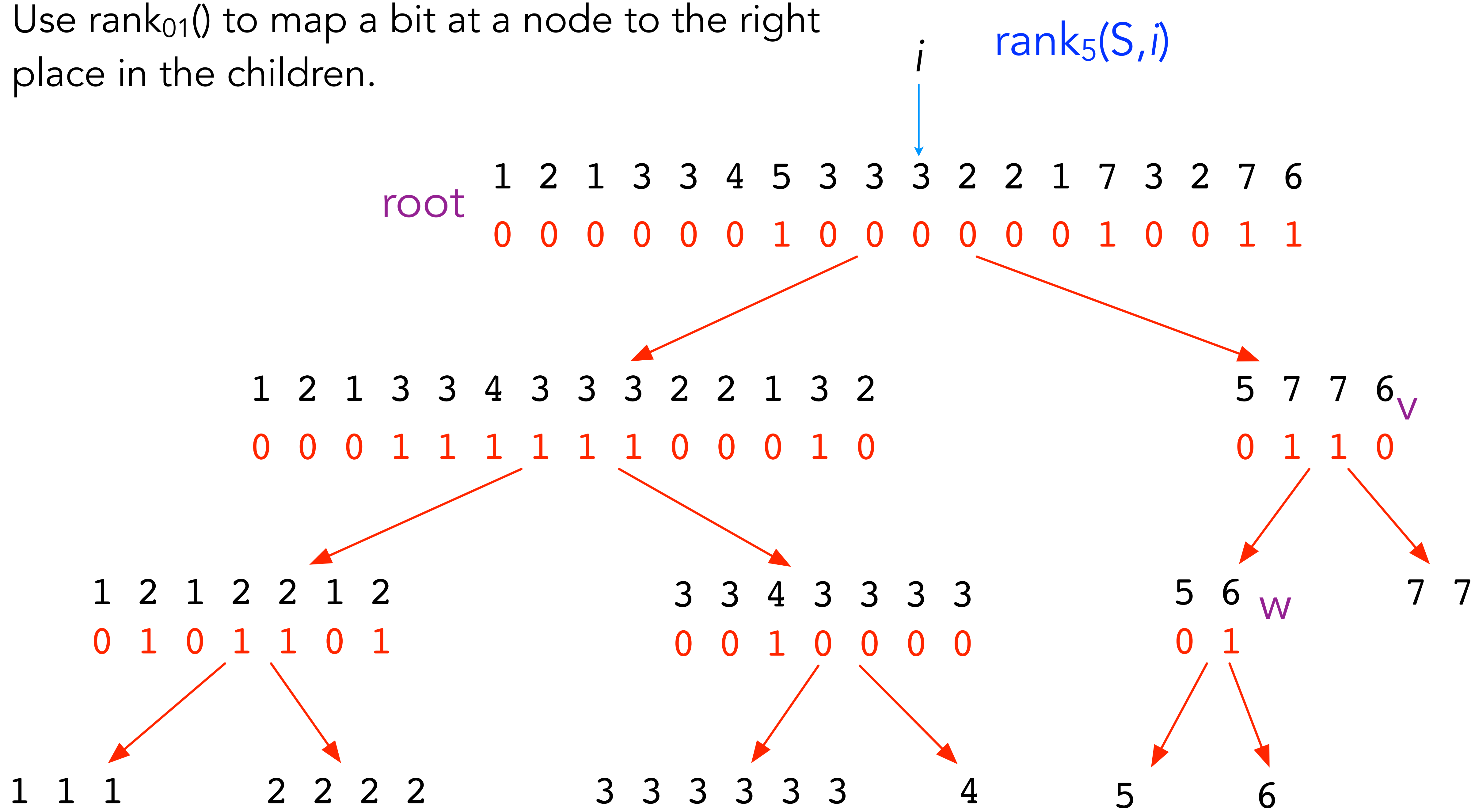
Use $\text{rank}_{01}()$ to map a bit at a node to the right place in the children.



$rank_c(S,i)$

Go left if **c is in first half of alphabet**, go right if c is in second half of alphabet.

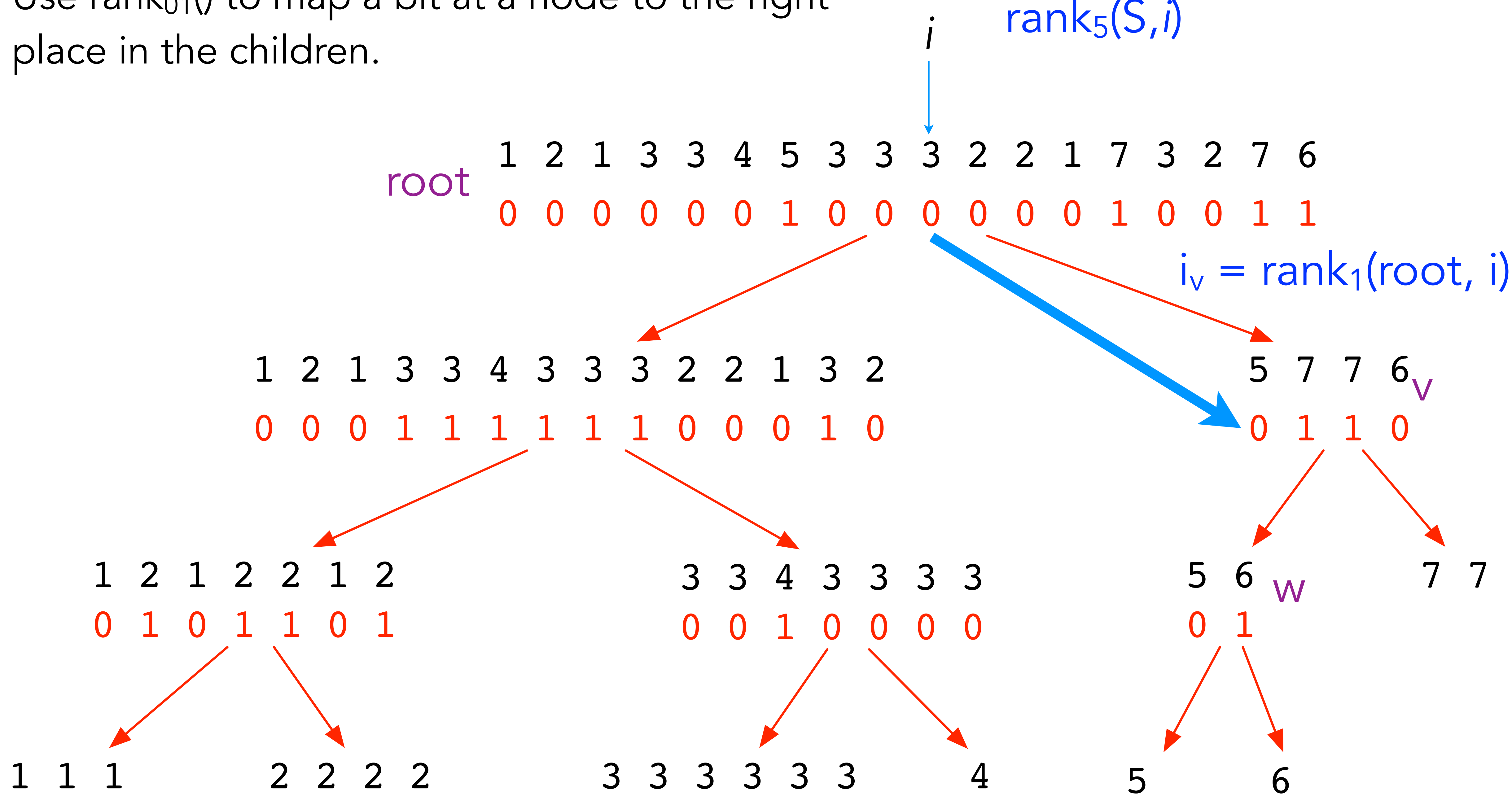
Use $rank_{01}()$ to map a bit at a node to the right place in the children.



$rank_c(S,i)$

Go left if **c is in first half of alphabet**, go right if c is in second half of alphabet.

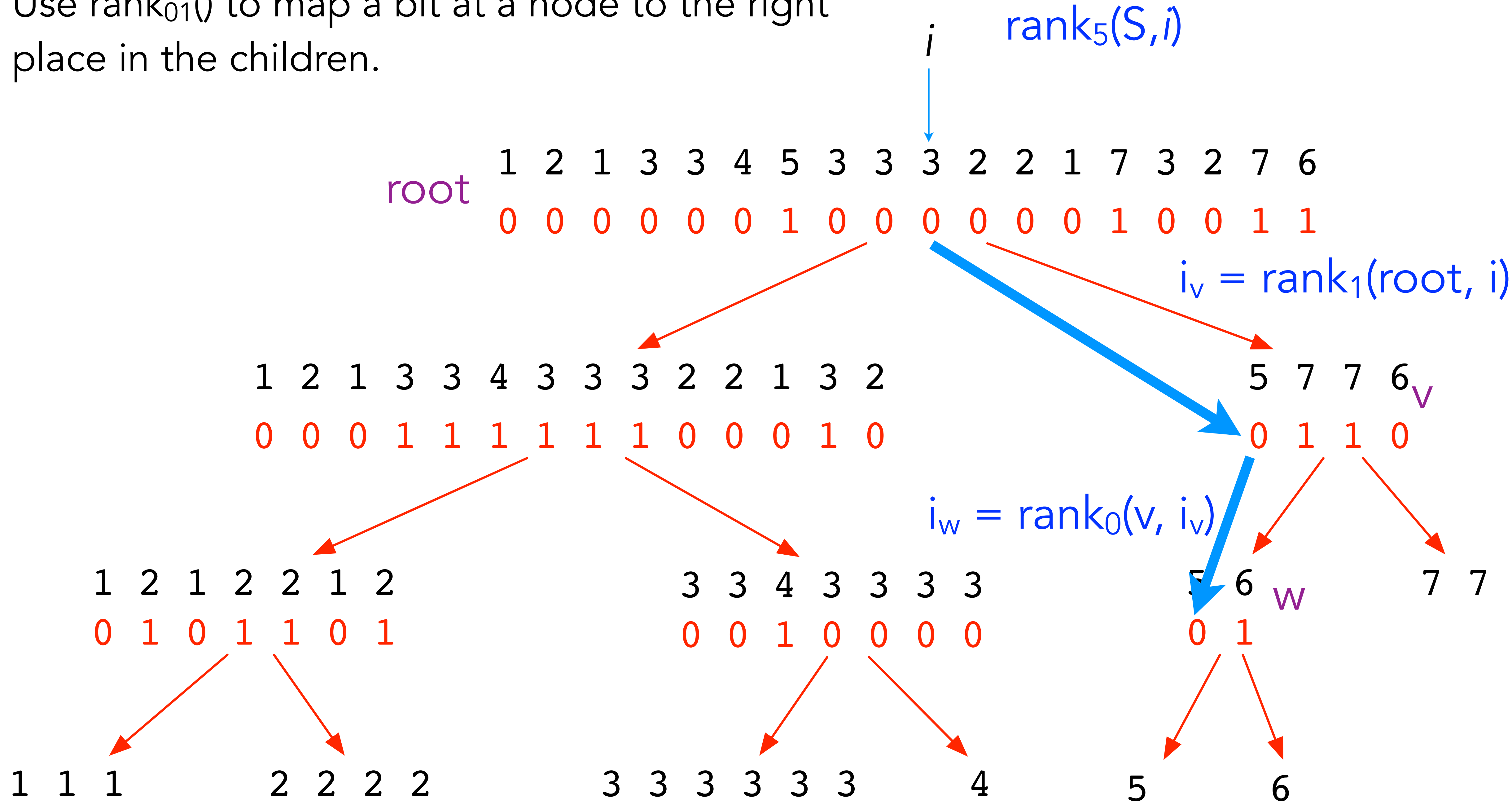
Use $rank_{01}()$ to map a bit at a node to the right place in the children.



$rank_c(S,i)$

Go left if **c is in first half of alphabet**, go right if c is in second half of alphabet.

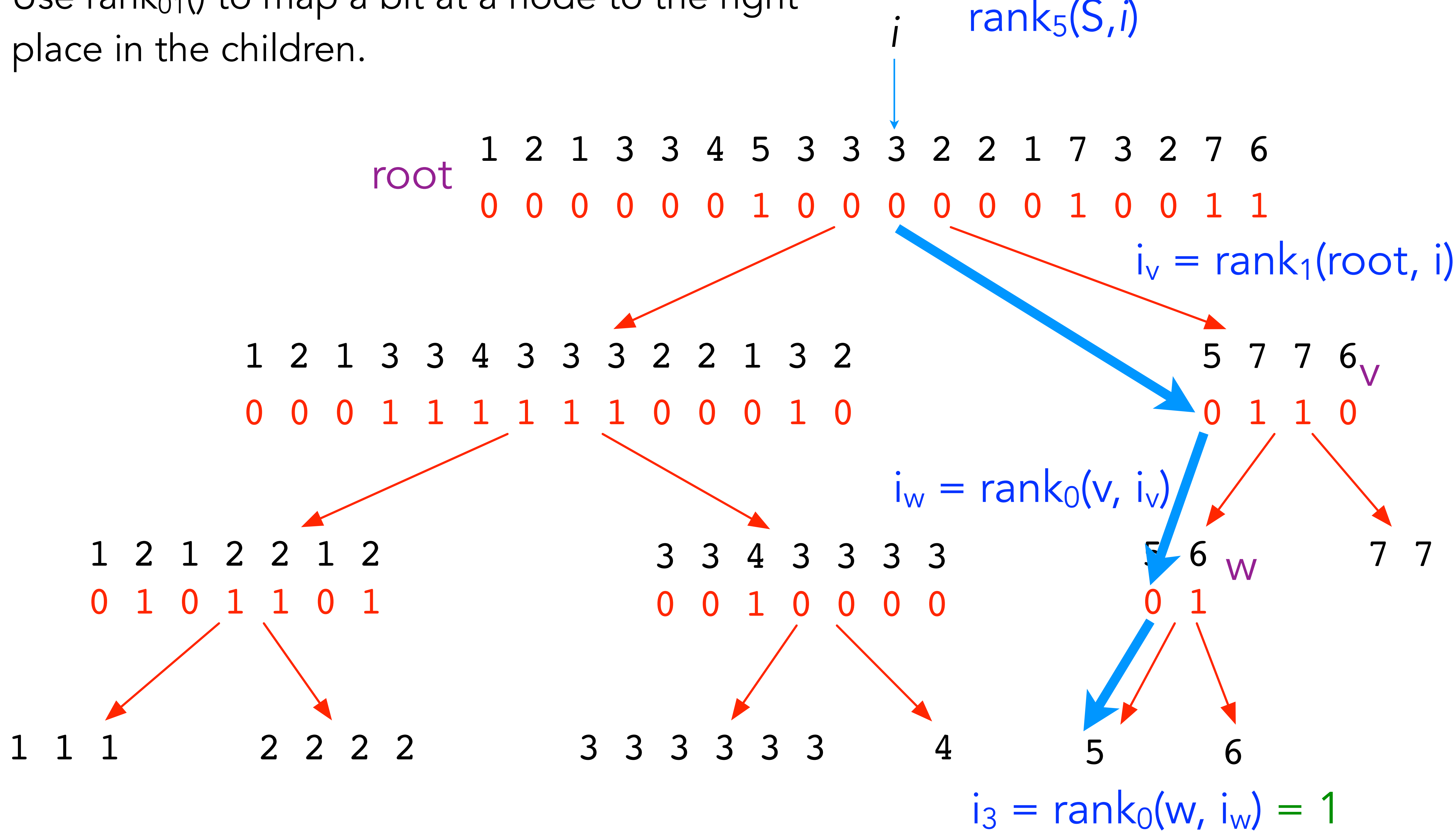
Use $rank_{01}()$ to map a bit at a node to the right place in the children.



$rank_c(S,i)$

Go left if **c is in first half of alphabet**, go right if c is in second half of alphabet.

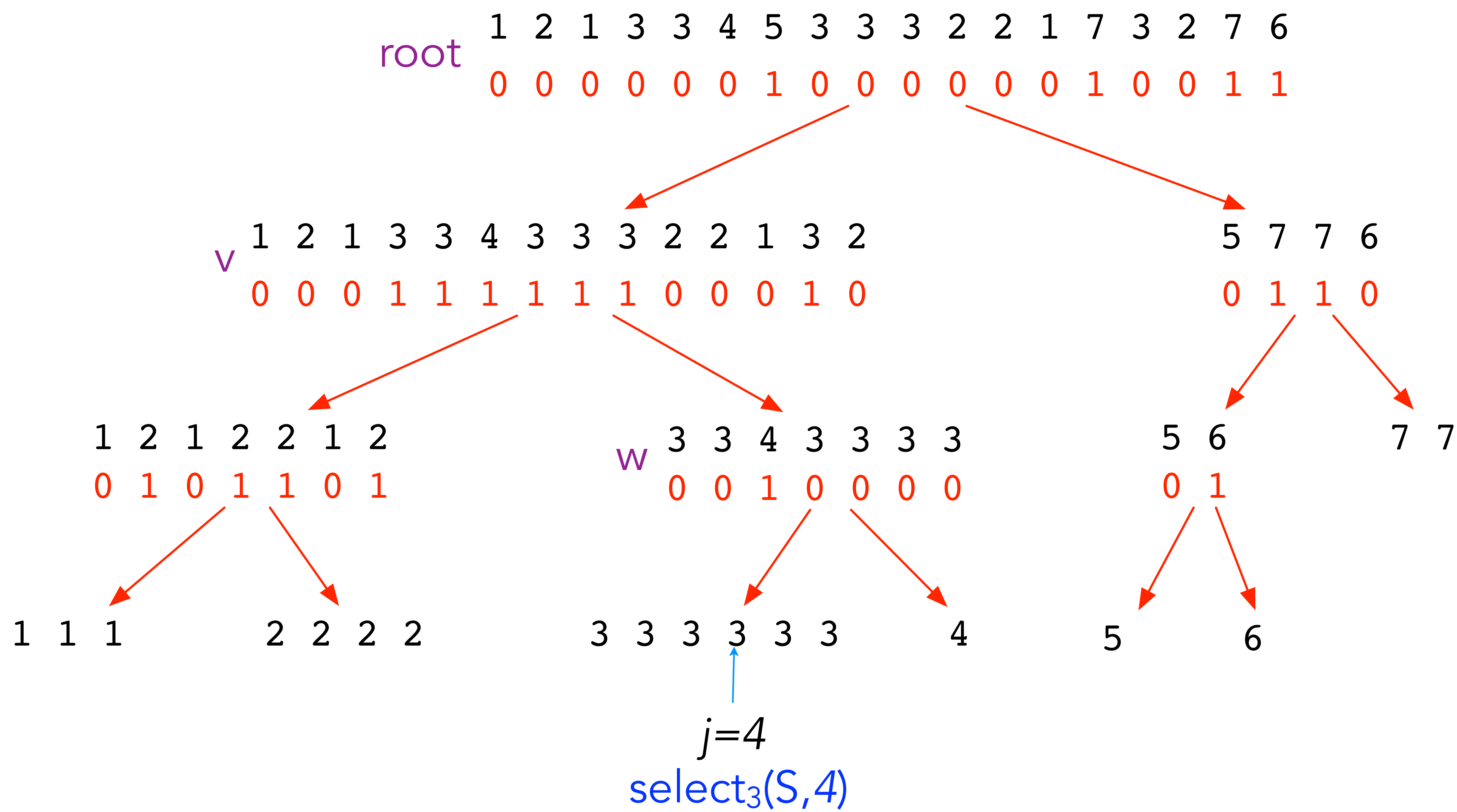
Use $rank_{01}()$ to map a bit at a node to the right place in the children.



$select_c(S,j)$

Start at position *j* in leaf corresponding to *c*.

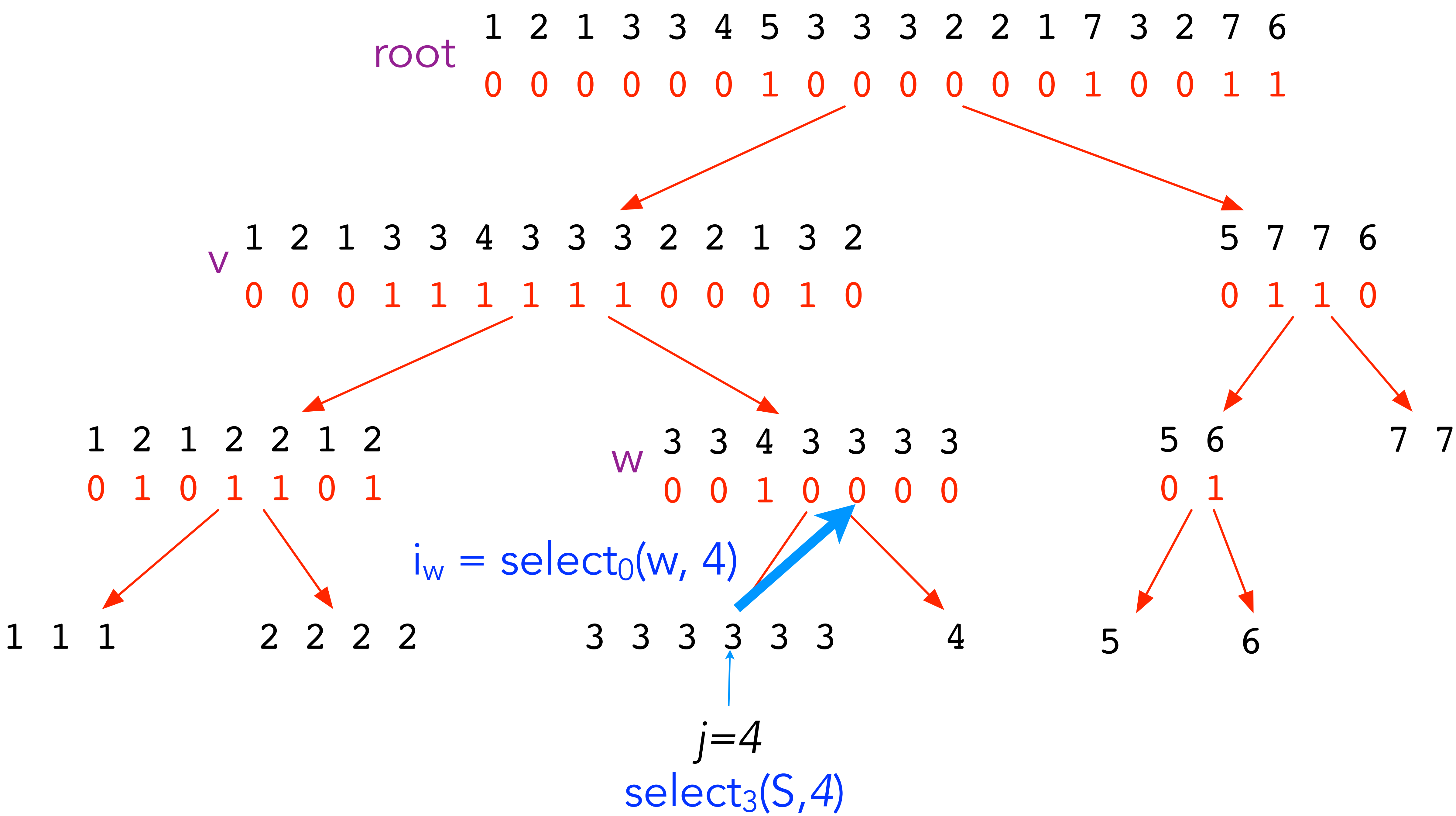
Repeat: new *i* = $select_b(p, i)$ where *p* = parent of current node, *b* = 0 if current node is left child of *p*, 1 if right child



$select_c(S,j)$

Start at position *j* in leaf corresponding to *c*.

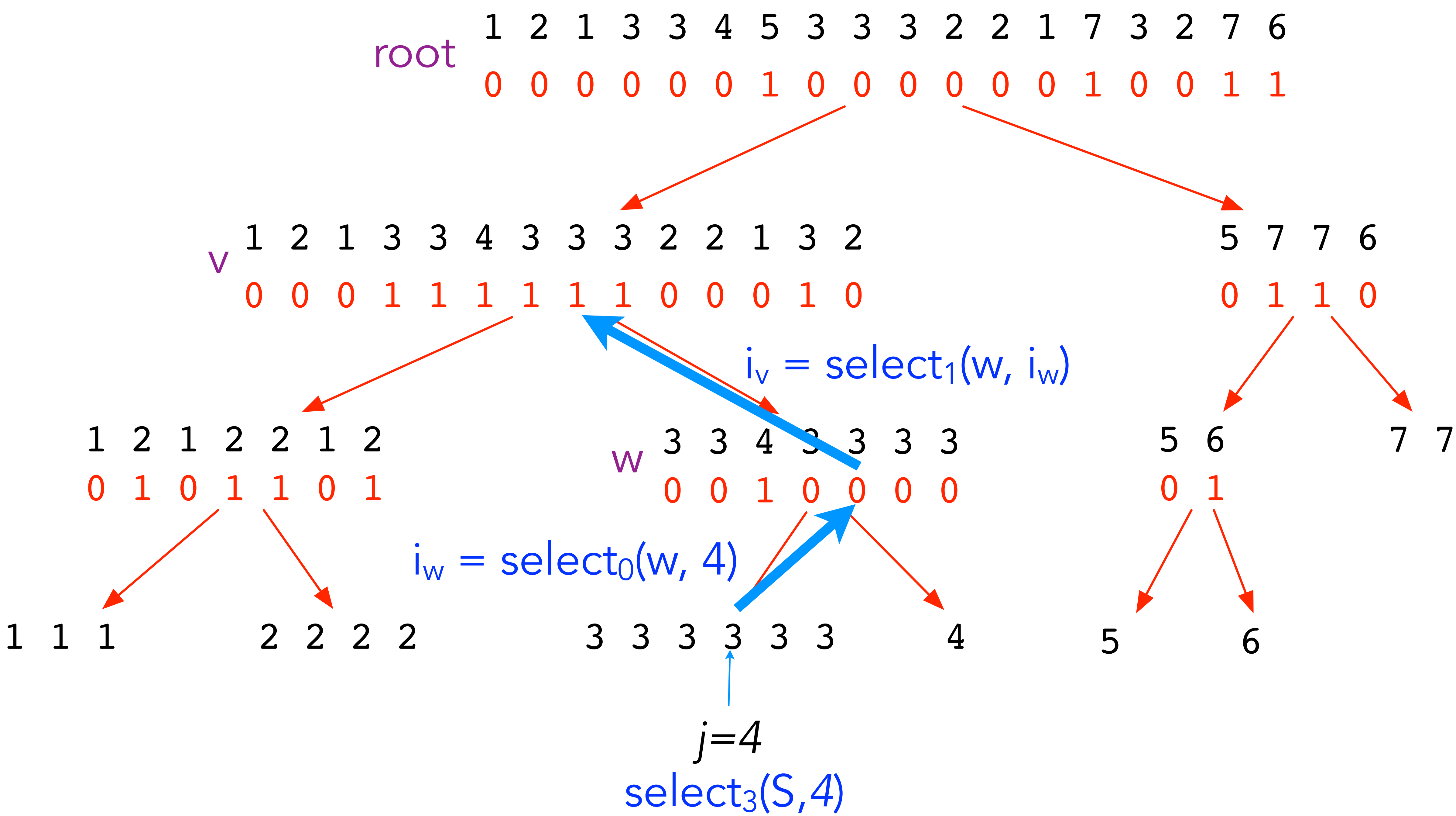
Repeat: new *i* = $select_b(p, i)$ where *p* = parent of current node, *b* = 0 if current node is left child of *p*, 1 if right child



$select_c(S,j)$

Start at position *j* in leaf corresponding to *c*.

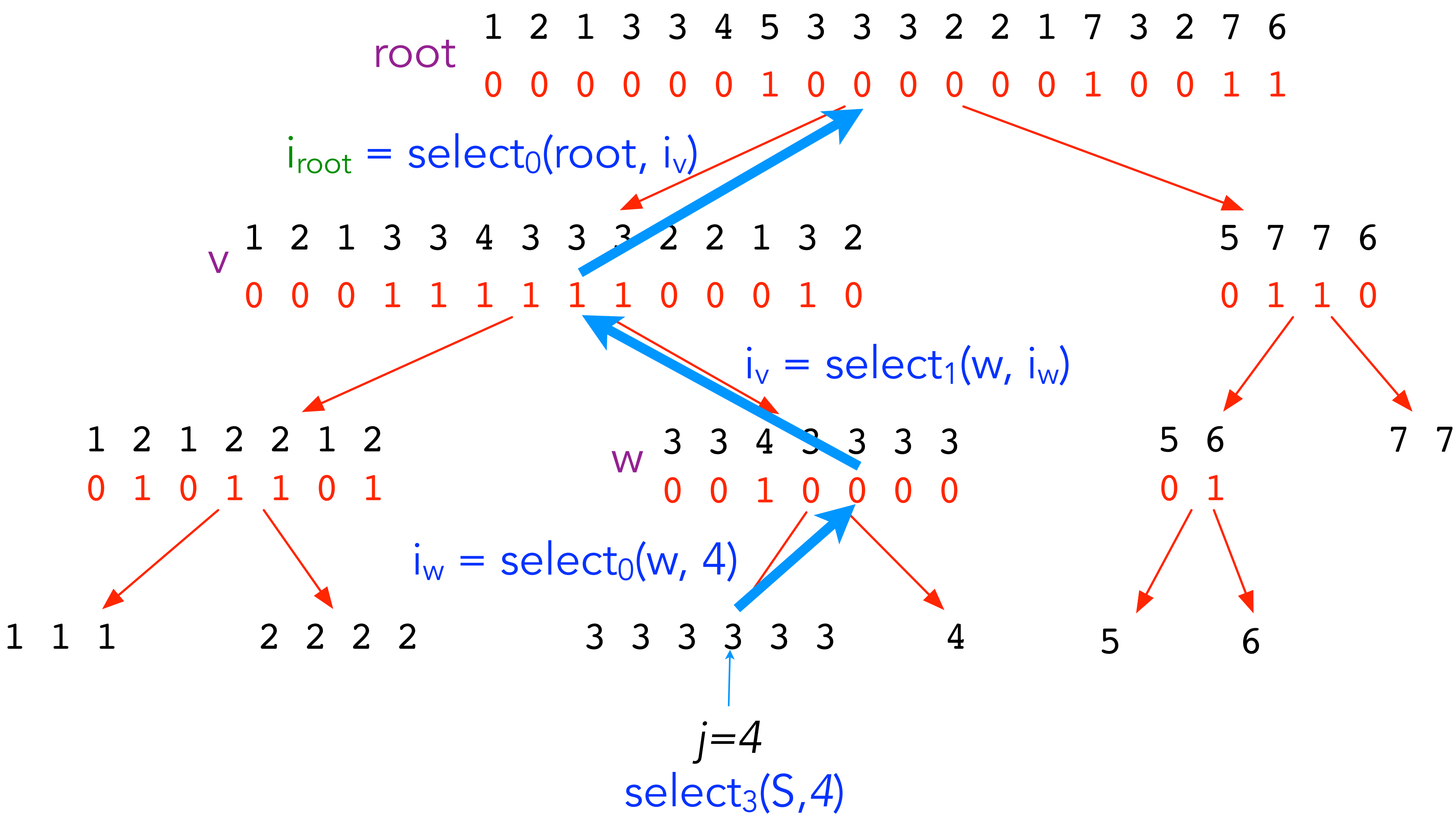
Repeat: new *i* = $select_b(p, i)$ where *p* = parent of current node, *b* = 0 if current node is left child of *p*, 1 if right child



$select_c(S,j)$

Start at position *j* in leaf corresponding to *c*.

Repeat: new $i = select_b(p, i)$ where p = parent of current node, $b = 0$ if current node is left child of p, 1 if right child



Running Times

Tree height = log |∑|, where ∑ is the alphabet.

rank, select, access follow a root-leaf path in the tree, taking O(1) time at each node.

Therefore, rank, select, access take O(log |∑|) to run.

If alphabet size is constant, this is O(1).

Tree Shape

- Don’t have to use balanced tree shape

- Can instead encode using, say, Huffman code tree shape
  - makes accesses to frequent characters faster
  - gives good space usage even without compressed bit vectors.

- Doesn’t have to be binary tree
  - by selecting optimal branching factor, can get query time to $O(\log n / \log \log n)$.

Application: Inverted Indices

```
1  If you really want to hear about it, the first thing you'll
13 probably want to know is where I was born, and what my lousy
26 childhood was like, and how my parents were occupied and all
37 before they had me, and all that David Copperfield kind of crap,
49 but I don't feel like going into it, if you want to know the
63 truth.
```

Traditional inverted index represents the document as an array of lists:

If	1, 57
you	2, 58
really	3
want	4,14,59
and	22,29,35,41
know	16,61
truth	63
⋮	

Good for searching for word *w*

Hard to compute S[i]

Application: Inverted Indices, 2

- Represent text as a string of word ids.
- Store as a wavelet tree.
- S[i] now O(log |∑|) = O(log n)
- $\text{select}_w(S, j)$ now gives the position of the jth occurrence of word w in time O(log n).

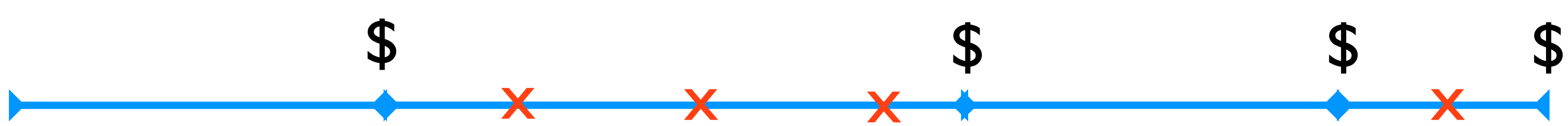
Application: Document Retrieval

Given a collection of documents $D_1, \ldots, D_m$, answer the following types of queries quickly:

- In which documents does word w appear?

Represent documents as strings of word ids.

Concatenate the documents together, separated by a $ word that does not occur elsewhere.




```
i = 1
repeat:
    pᵢ = select_c(i)          # ith occurrence
    dᵢ = rank_$(pᵢ) + 1       # document containing it
    print dᵢ
    p' = select_$(dᵢ)         # end of document
    i = rank_c(p') + 1        # find 1st occurrence after end of doc
```

Application: Graphs

Given a directed graph G, answer the following queries quickly:

- successor(u, i) := the ith vertex v such that edge (v,u) exists.
- predecessor(u, i) := the jth vertex v such that edge (u,v) exists.

Represent G as a concatenation of adjacencies lists, and store in wavelet tree:

	2	3	5	4	3	5	6	2	6	5	4	1	2
bit vector marking list boundaries	1	0	0	1	0	0	1	1	0	0	1	1	0

- successor(u, i): p = $\text{select}_1(B, u)$; return $S[p+i-1]$
- predecessor(u, i): p = $\text{select}_u(S, i)$; return $\text{rank}_1(B, p)$

$|\Sigma|$ = n, so these take O(log n) time.

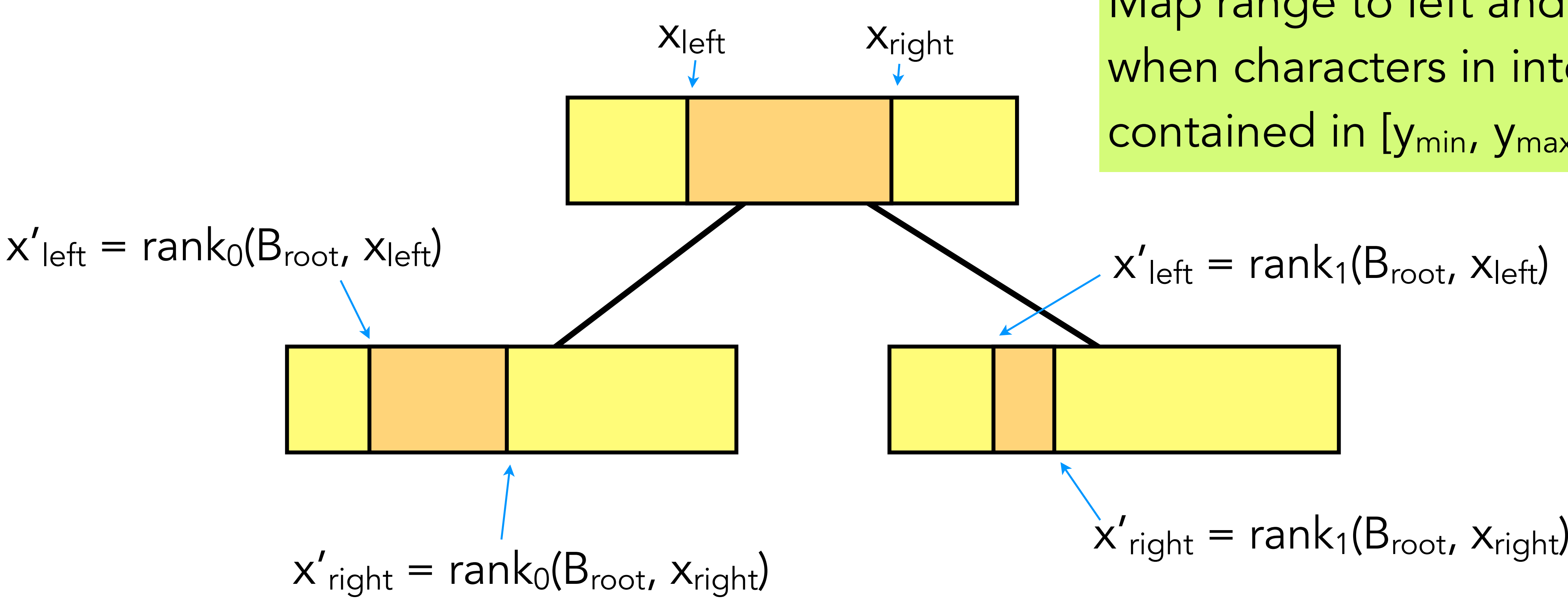
Application: Grid of Points

Suppose you have points $(x_1, y_1), \ldots, (x_n, y_n)$ on an $m \times m$ grid and you want to answer range queries quickly:

- Which points fall inside rectangle $[x_{min}, x_{max}]$ by $[y_{min}, y_{max}]$?

Sort points by x-coordinate, consider $y_{\pi(1)}, y_{\pi(4)}, y_{\pi(3)}, \ldots, y_{\pi(n)}$ as a string and store in a wavelet tree.

Map range to left and right, stop when characters in interval are contained in $[y_{min}, y_{max}]$.



Summary

- Wavelet trees compactly store strings.
- Allowing access almost as fast as for a plain array.
- And allowing for fast rank and select queries too.
- Lots of applications and extensions, often storing things not normally thought of as strings.