Monotone Minimal Perfect Hashing

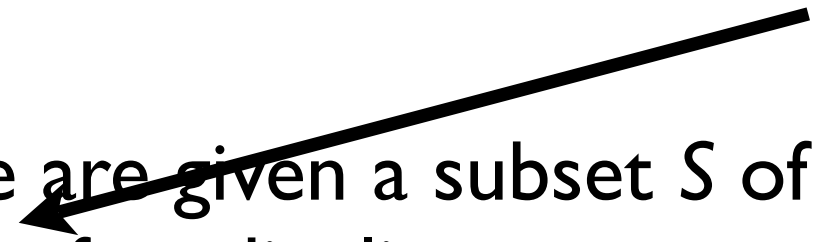
Monotone Minimal Perfect Hashing

Monotone Minimal Perfect Hashing

- We are given a subset *S* of the universe $\Omega = [u]$ of cardinality *n*

Monotone Minimal Perfect Hashing

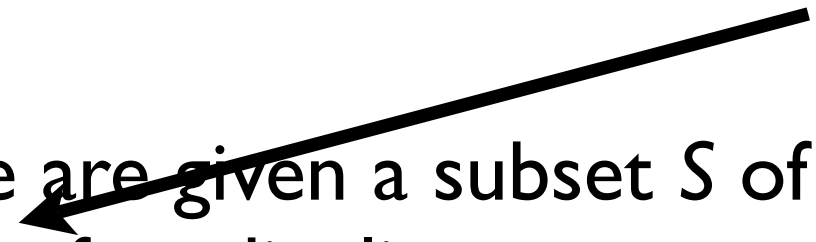
Note: von Neumann's notation: $u = \{ 0, 1, 2, ..., u - 1 \}$



- We are given a subset S of the universe $\Omega = [u]$ of cardinality n

Monotone Minimal Perfect Hashing

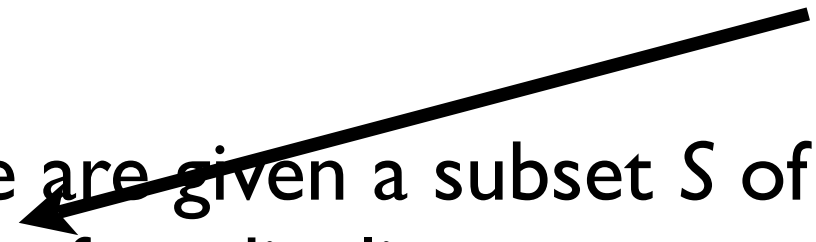
Note: von Neumann's notation: $u = \{ 0, 1, 2, ..., u - 1 \}$



- We are given a subset S of the universe $\Omega = [u]$ of cardinality n
- We want a bijection $S \rightarrow [n]$ (Keys are binary strings of constant length $w = \log u$)

Monotone Minimal Perfect Hashing

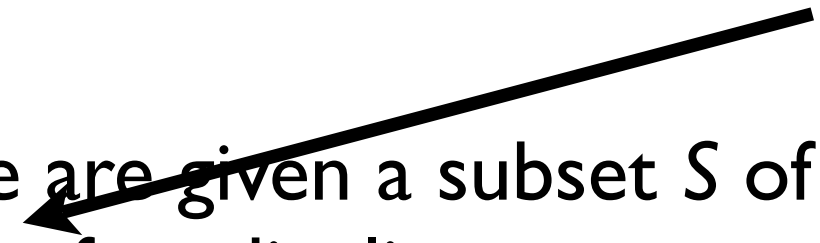
Note: von Neumann's notation: $u = \{ 0, 1, 2, ..., u - 1 \}$



- We are given a subset S of the universe $\Omega = [u]$ of cardinality n
- We want a bijection $S \rightarrow [n]$ (Keys are binary strings of constant length $w = \log u$)
- We want the bijection to preserve the *lexicographical* ordering of [u]

Monotone Minimal Perfect Hashing

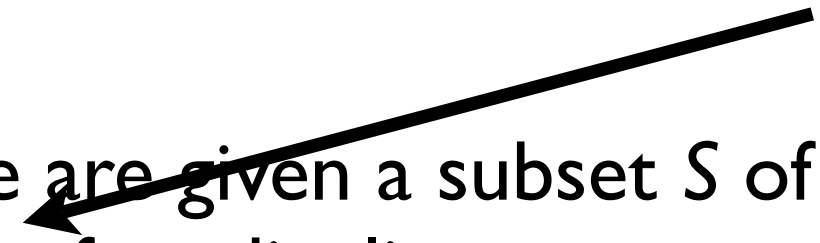
Note: von Neumann's notation: $u = \{ 0, 1, 2, ..., u - 1 \}$



- We are given a subset S of the universe $\Omega = [u]$ of cardinality n
- We want a bijection $S \rightarrow [n]$ (Keys are binary strings of constant length $w = \log u$)
- We want the bijection to preserve the *lexicographical* ordering of [u]
- We want to use very little space, and do it very quickly (constant or $O(\log w)$ time)

Monotone Minimal Perfect Hashing

Note: von Neumann's notation: $u = \{ 0, 1, 2, ..., u - 1 \}$



- We are given a subset S of the universe $\Omega = [u]$ of cardinality n
- We want a bijection $S \rightarrow [n]$ (Keys are binary strings of constant length $w = \log u$)
- We want the bijection to preserve the *lexicographical* ordering of [u]
- We want to use very little space, and do it very quickly (constant or $O(\log w)$ time)

Basic idea: Bucketing

Basic idea: Bucketing

- Bucketing means splitting the key set *S* into *m* buckets

Basic idea: Bucketing

- Bucketing means splitting the key set S into m buckets
- Assume we have a monotone *distributor* d: $S \rightarrow m$

Basic idea: Bucketing

- Bucketing means splitting the key set S into m buckets
- Assume we have a monotone *distributor* d: $S \rightarrow m$
- Assume we have a monotone minimal perfect hash function g_i on $d^{-1}(i)$

Basic idea: Bucketing

- Bucketing means splitting the key set S into m buckets
- Assume we have a monotone *distributor* d: $S \rightarrow m$
- Assume we have a monotone minimal perfect hash function g_i on $d^{-1}(i)$
- Assume we have a function s: $m \rightarrow n$ such that

Basic idea: Bucketing

- Bucketing means splitting the key set S into m buckets
- Assume we have a monotone *distributor* d: $S \rightarrow m$
- Assume we have a monotone minimal perfect hash function g_i on $d^{-1}(i)$
- Assume we have a function s: $m \rightarrow n$ such that
- $$s(i) = \sum_{i<j} |d^{-1}(j)|$$

Basic idea: Bucketing

- Bucketing means splitting the key set S into m buckets
- Assume we have a monotone *distributor* d: $S \rightarrow m$
- Assume we have a monotone minimal perfect hash function g_i on $d^{-1}(i)$
- Assume we have a function s: $m \rightarrow n$ such that
- $$s(i) = \sum_{i<j} |d^{-1}(j)|$$
- Then, $h(x)=s(d(x))+g_{d(x)}(x)$ is a monotone minimal perfect hash function on S

First idea: LCPs

First idea: LCPs

- Divide S into buckets of equal size

First idea: LCPs

- Divide S into buckets of equal size
- For each bucket, compute the LCP

First idea: LCPs

- Divide S into buckets of equal size
- For each bucket, compute the LCP
- Lemma: all LCPs are distinct

First idea: LCPs

- Divide S into buckets of equal size
- For each bucket, compute the LCP
- Lemma: all LCPs are distinct
- We can identify a bucket by its LCP

First idea: LCPs

- Divide S into buckets of equal size
- For each bucket, compute the LCP
- Lemma: all LCPs are distinct
- We can identify a bucket by its LCP
- We can compute the LCP of the bucket of a key storing a map from S to $w = \log u$

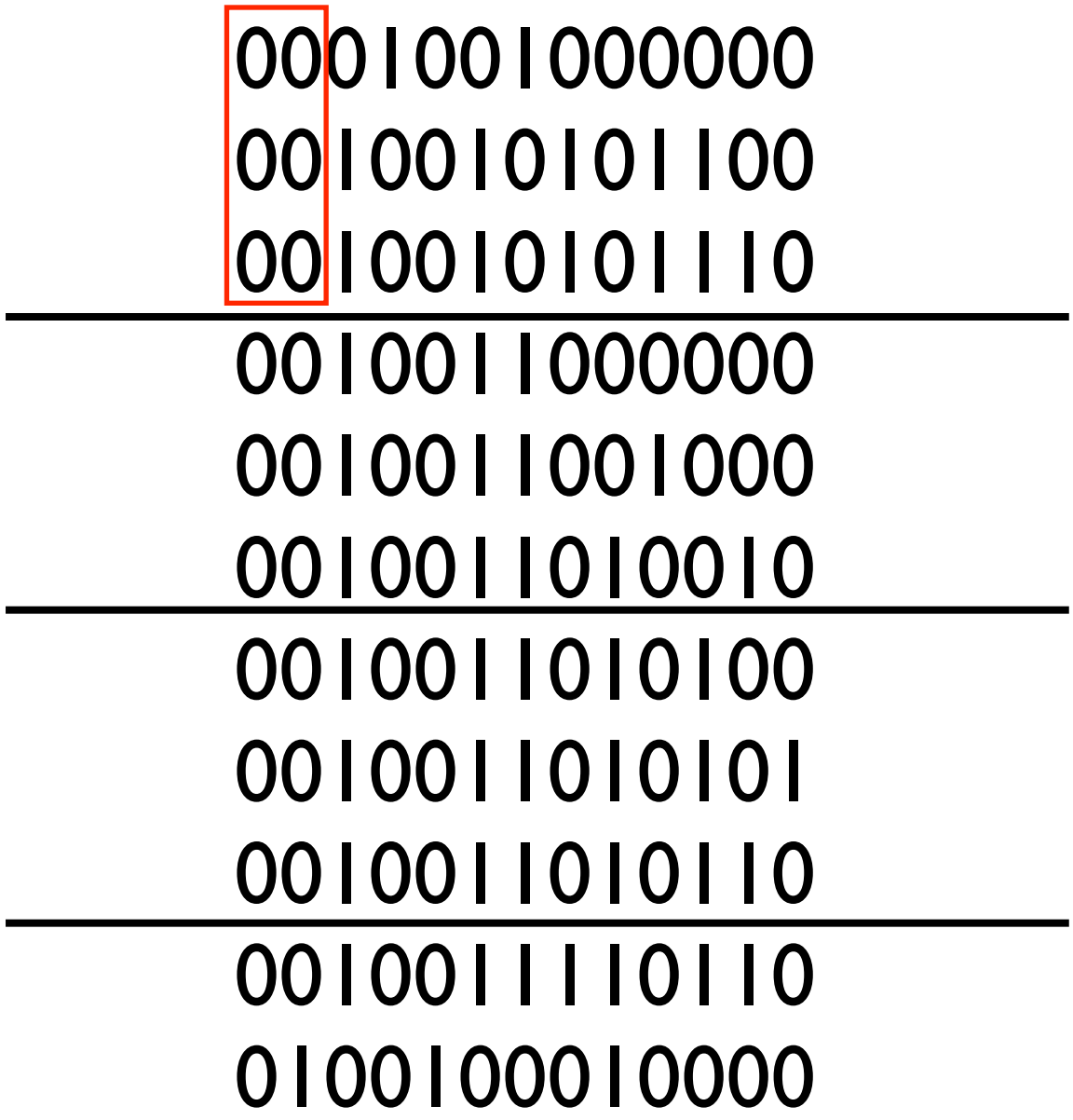
000I00I000000
00I00I0I0II00
00I00I0I0III0
00I00II000000
00I00II00I000
00I00II0I00I0
00I00II0I0I00
00I00II0I0I0I
00I00II0I0II0
00I00IIIIOIIO
0I00I000I0000

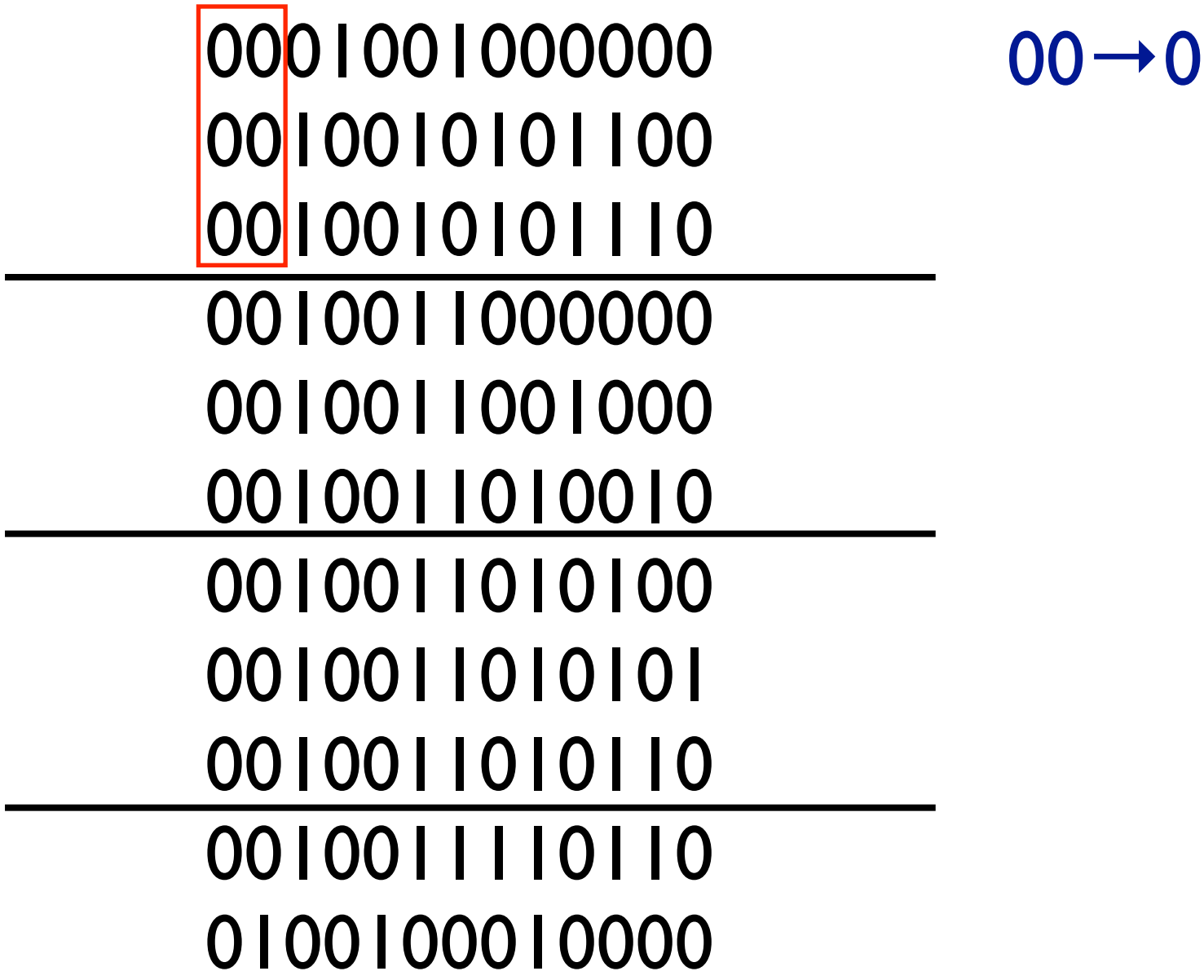
000|00|000000
00|00|0|0|1|00
00|00|0|0|1|10

00|00|1|000000
00|00|1|00|000
00|00|1|0|00|0

00|00|1|0|0|00
00|00|1|0|0|01
00|00|1|0|0|10

00|00|1|1|0|10
0|00|000|0000


000|00|000000
00|00|0|0|1|00
00|00|0|0|1|1|0
00|00|1|000000
00|00|1|00|000
00|00|1|0|00|0
00|00|1|0|0|00
00|00|1|0|0|01
00|00|1|0|0|1|0
00|00|1|1|0|1|0
0|00|000|0000


00|00|000000
00|00|0|0|1|00
00|00|0|0|1|1|0
00|00|1|000000
00|00|1|00|000
00|00|1|0|00|0
00|00|1|0|0|00
00|00|1|0|0|1
00|00|1|0|1|10
00|00|1|1|0|1|0
0|00|000|0000
00→0

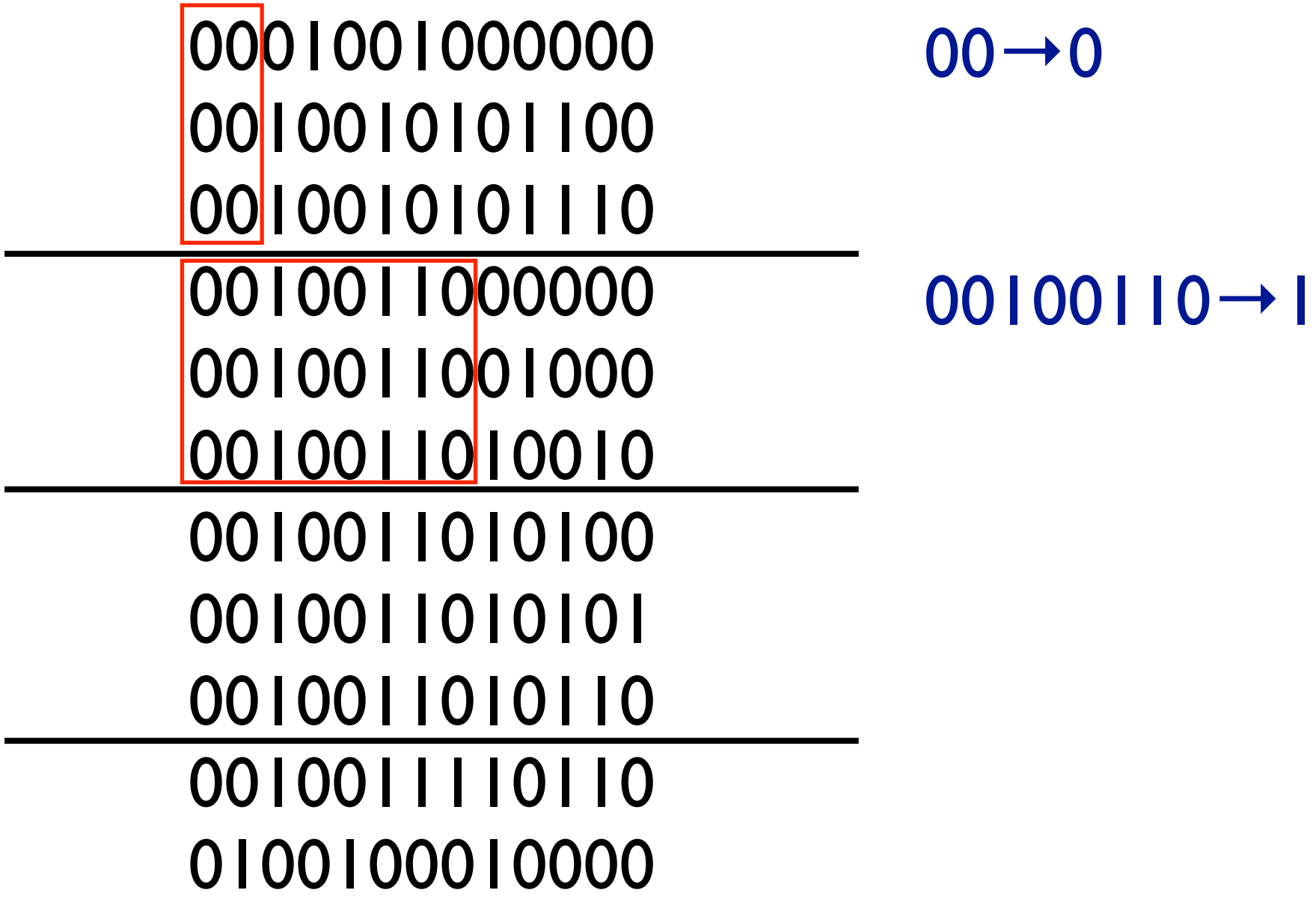
00→0

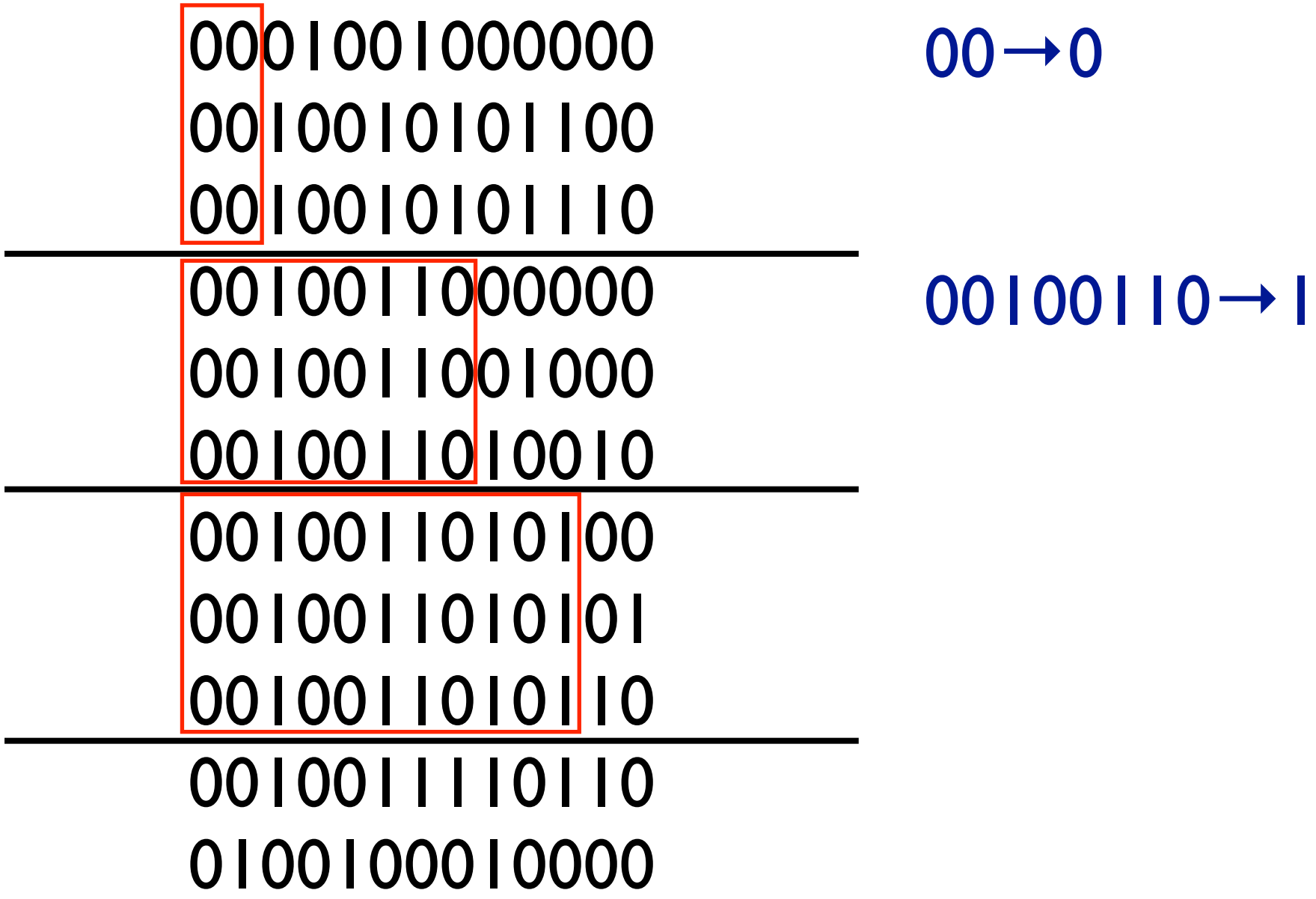
000|00|000000
00|00|0|0|1|00
00|00|0|0|1|1|0

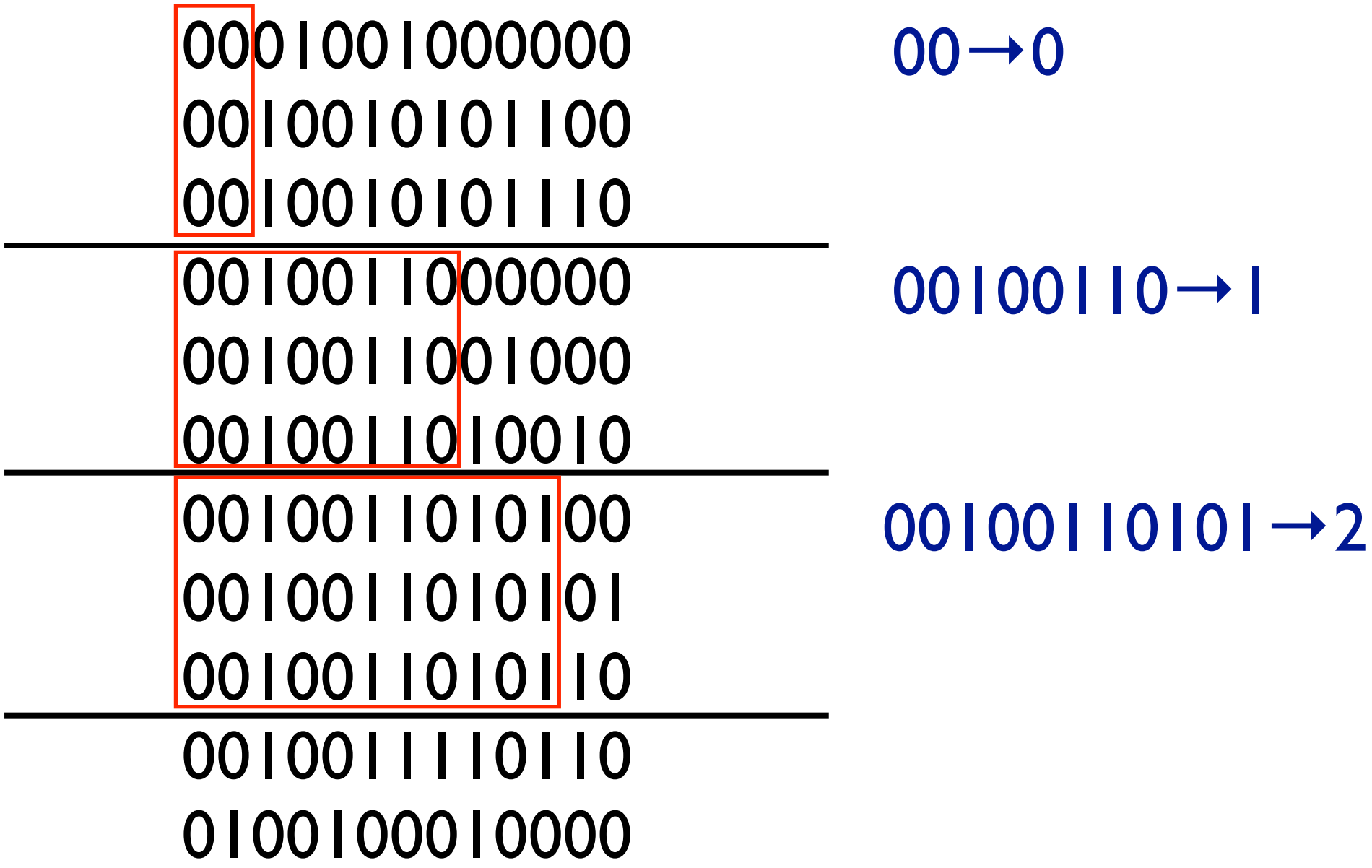
00|00|10|00000
00|00|10|0|000
00|00|10|00|0

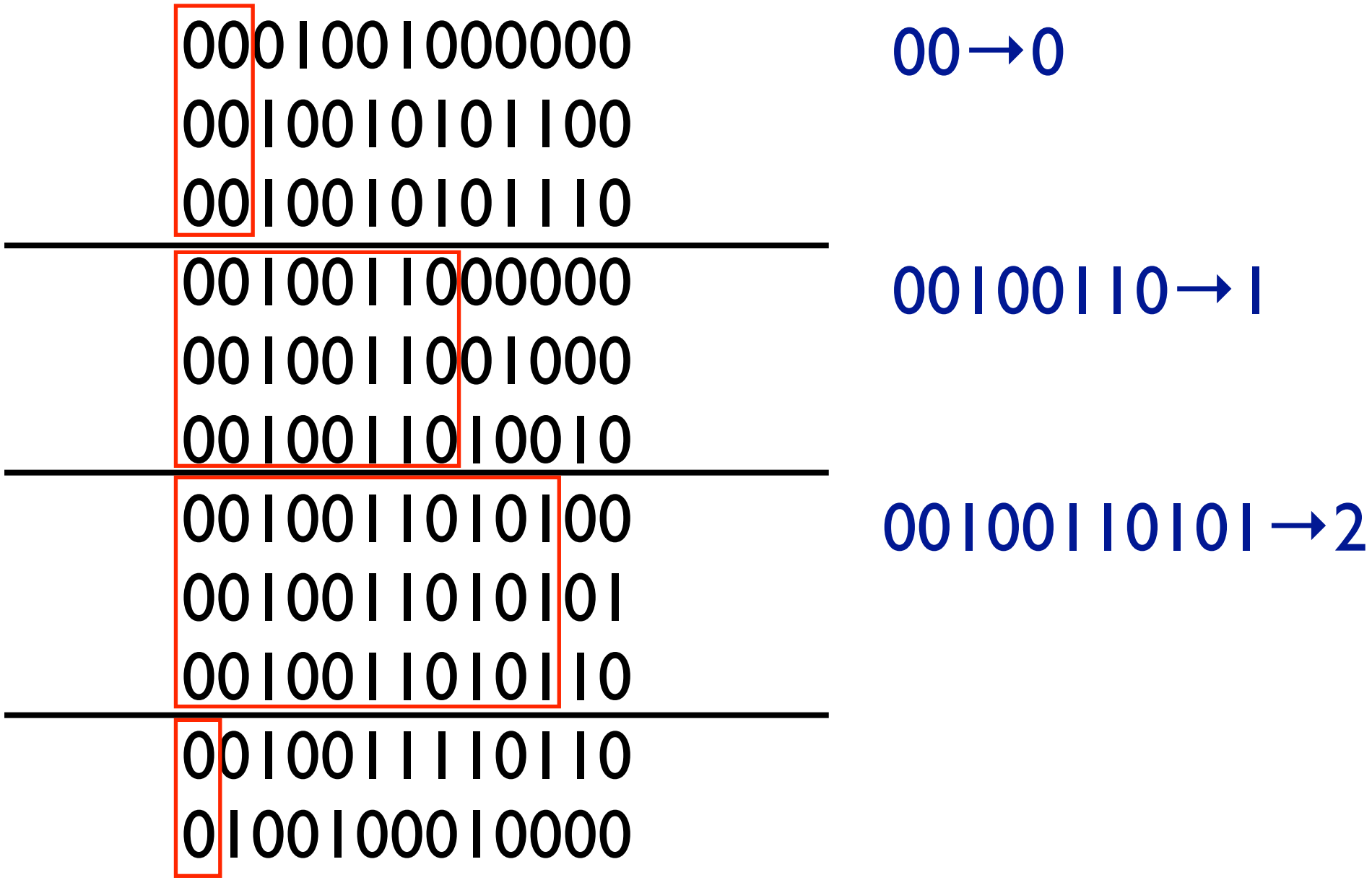
00|00|10|0|00
00|00|10|0|0|
00|00|10|0|1|0

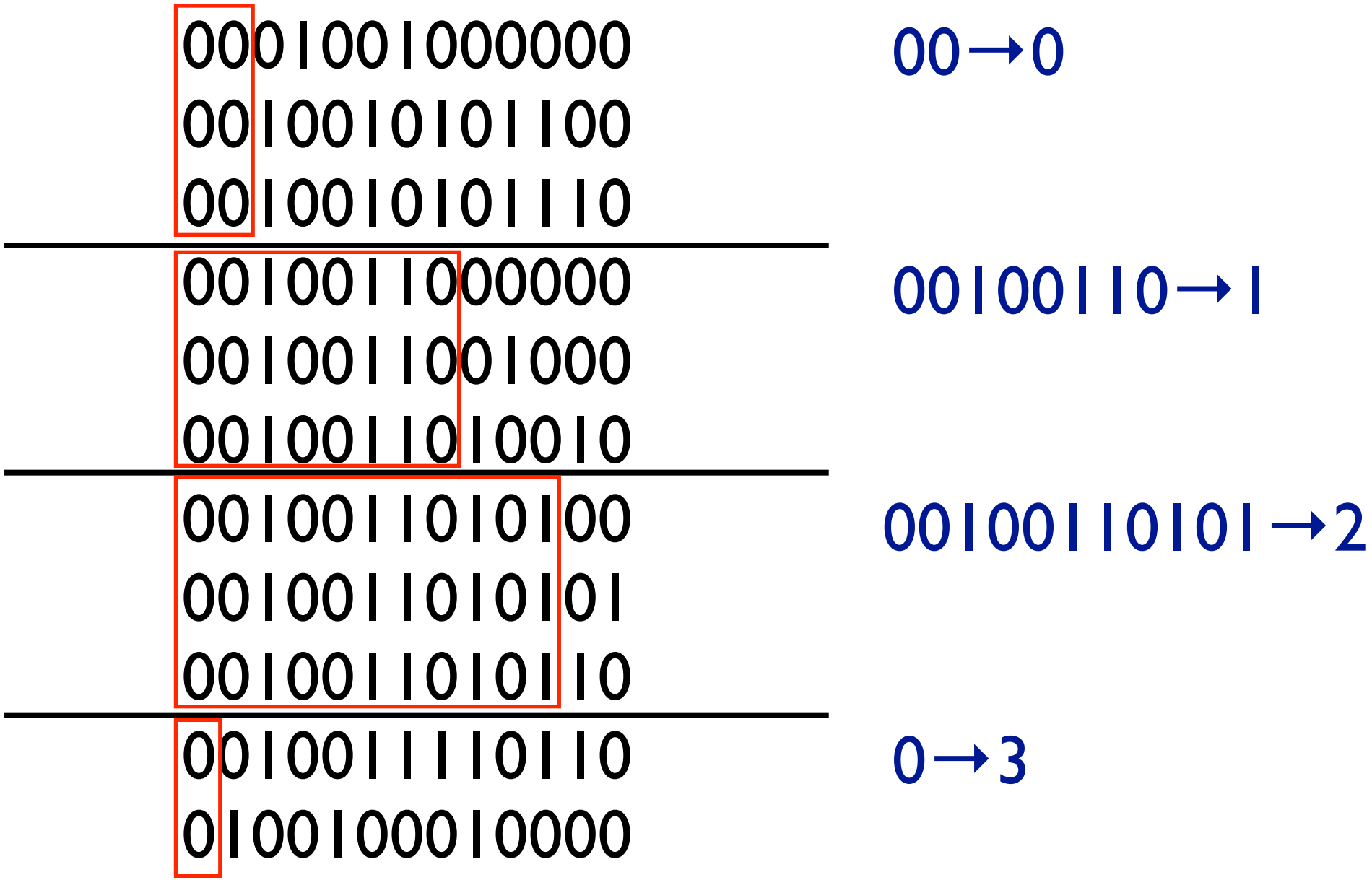
00|00|111|0|1|0
0|00|000|0000

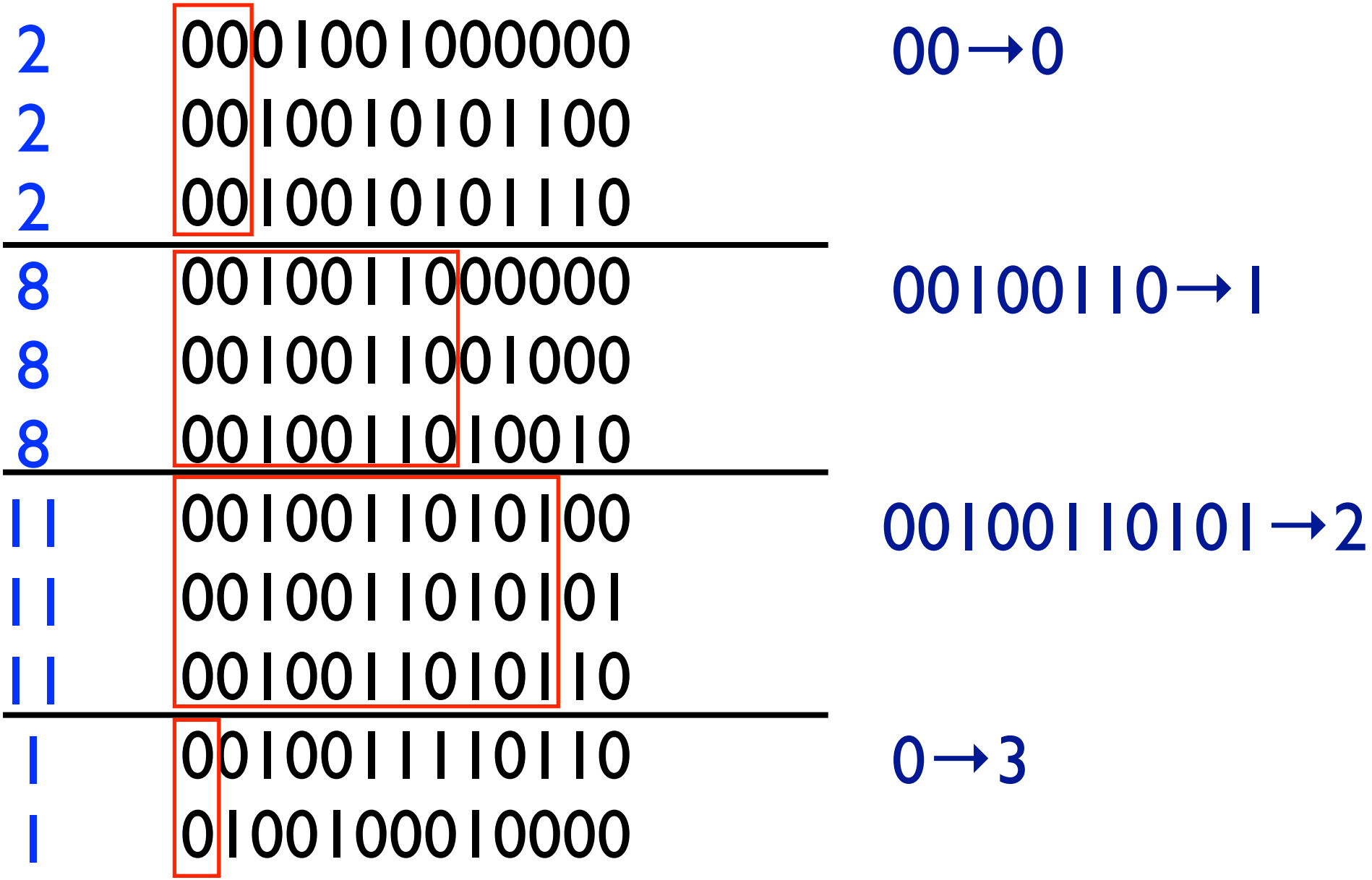

00|00|000000
00|00|0|0|1|00
00|00|0|0|1|1|0
00|00|1|000000
00|00|1|001000
00|00|1|0|00|0
00|00|1|0|0|00
00|00|1|0|0|0|1
00|00|1|0|1|1|0
00|00|1|1|1|0|1|0
0|00|000|0000
00→0
00|00|1|0→1

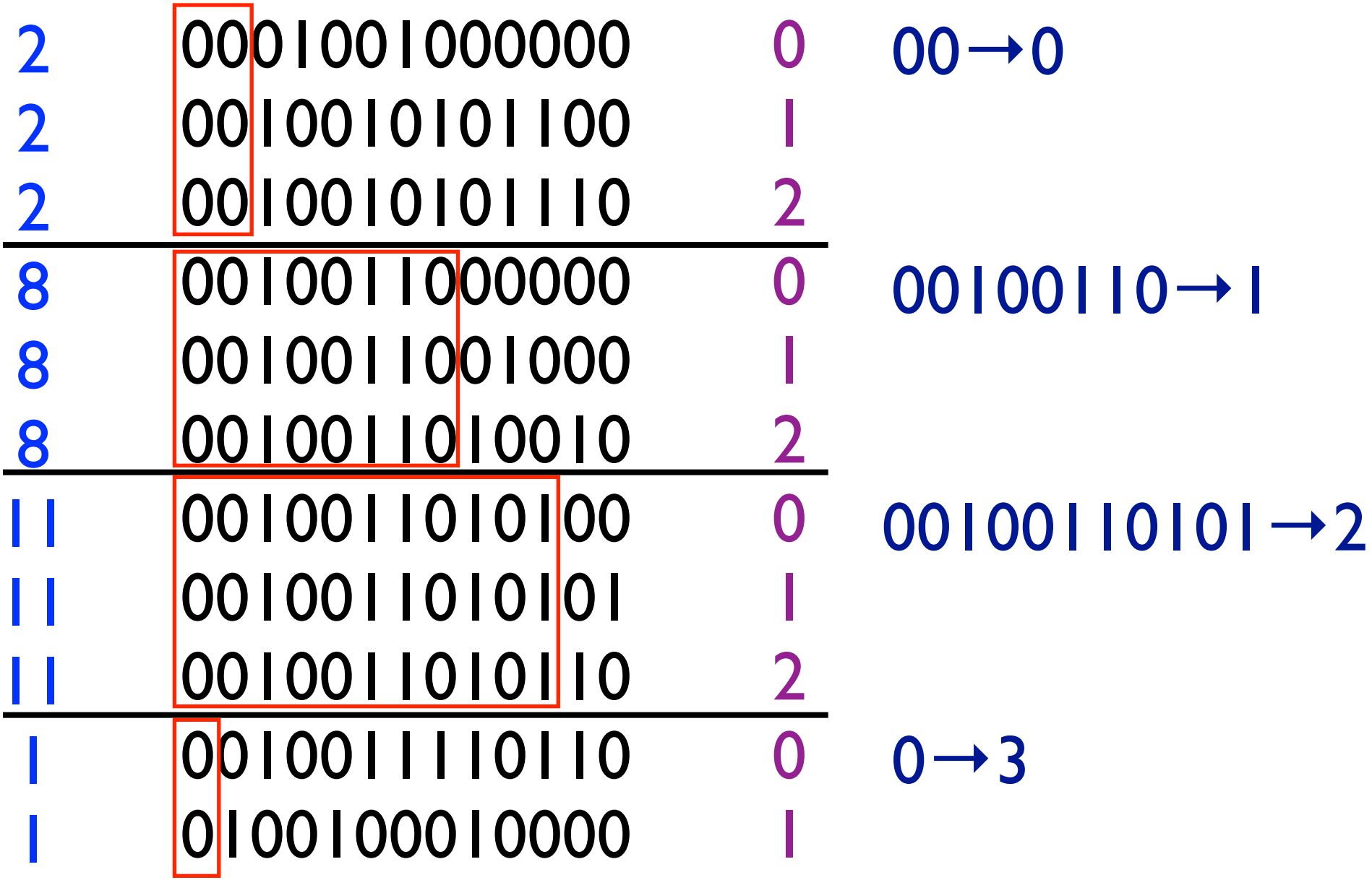

00|00|000000
00|00|0|0|1|00
00|00|0|0|1|1|0
00|00|1|000000
00|00|1|001000
00|00|1|0|00|0
00|00|1|0|0|00
00|00|1|0|0|01
00|00|1|0|0|1|0
00|00|1|1|1|0|1|0
0|00|1|000|0000
00→0
00|00|1|0→1

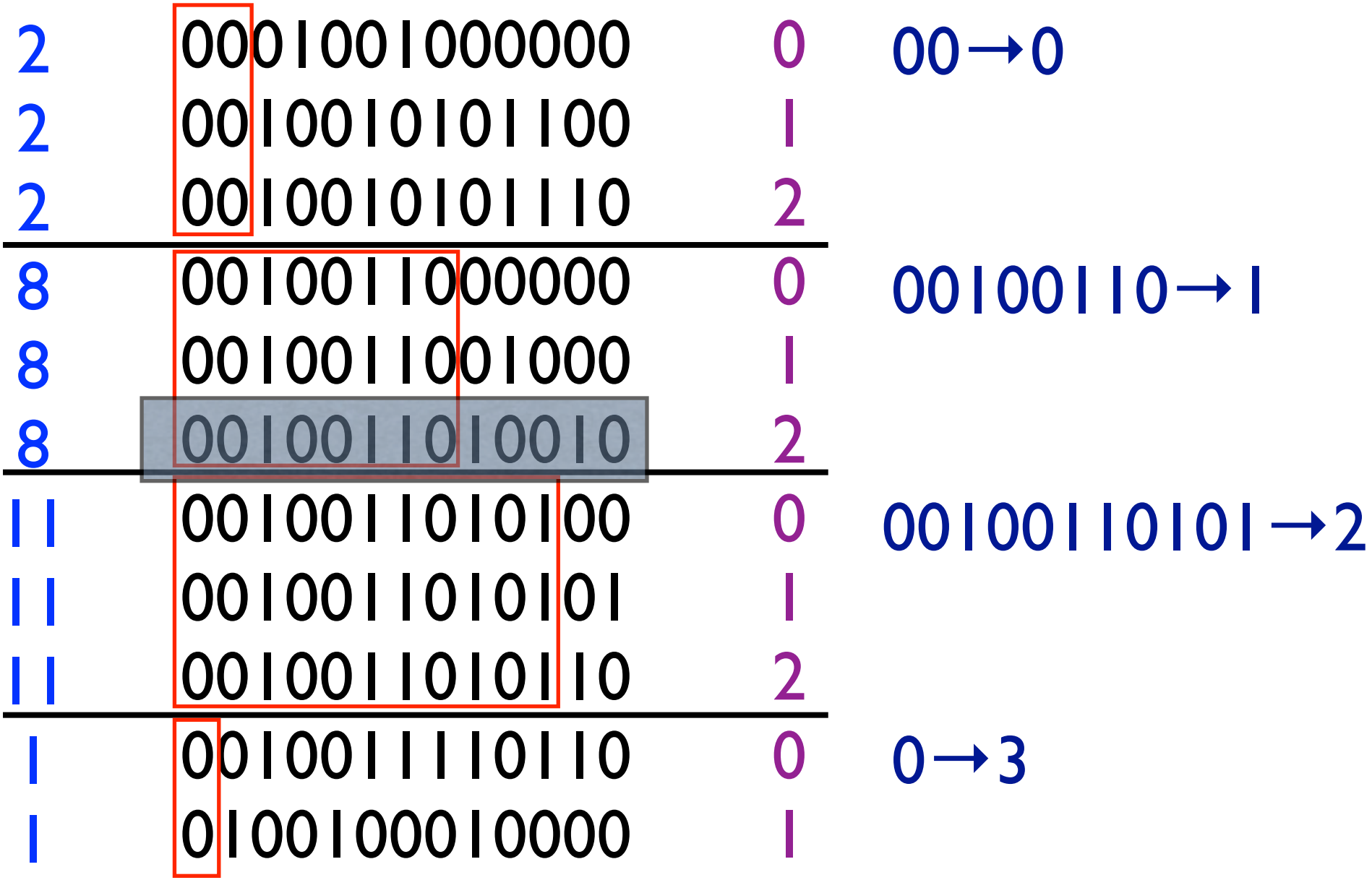

00|00|000000
00|00|0|0|1|00
00|00|0|0|1|1|0
00|00|1|0|00000
00|00|1|0|01000
00|00|1|0|0|00|0
00|00|1|0|0|00
00|00|1|0|0|1
00|00|1|0|0|1|0
00|00|1|1|1|0|1|0
0|00|1|000|1|0000
00|00→0
00|00|1|0→1
00|00|1|0|0|1→2

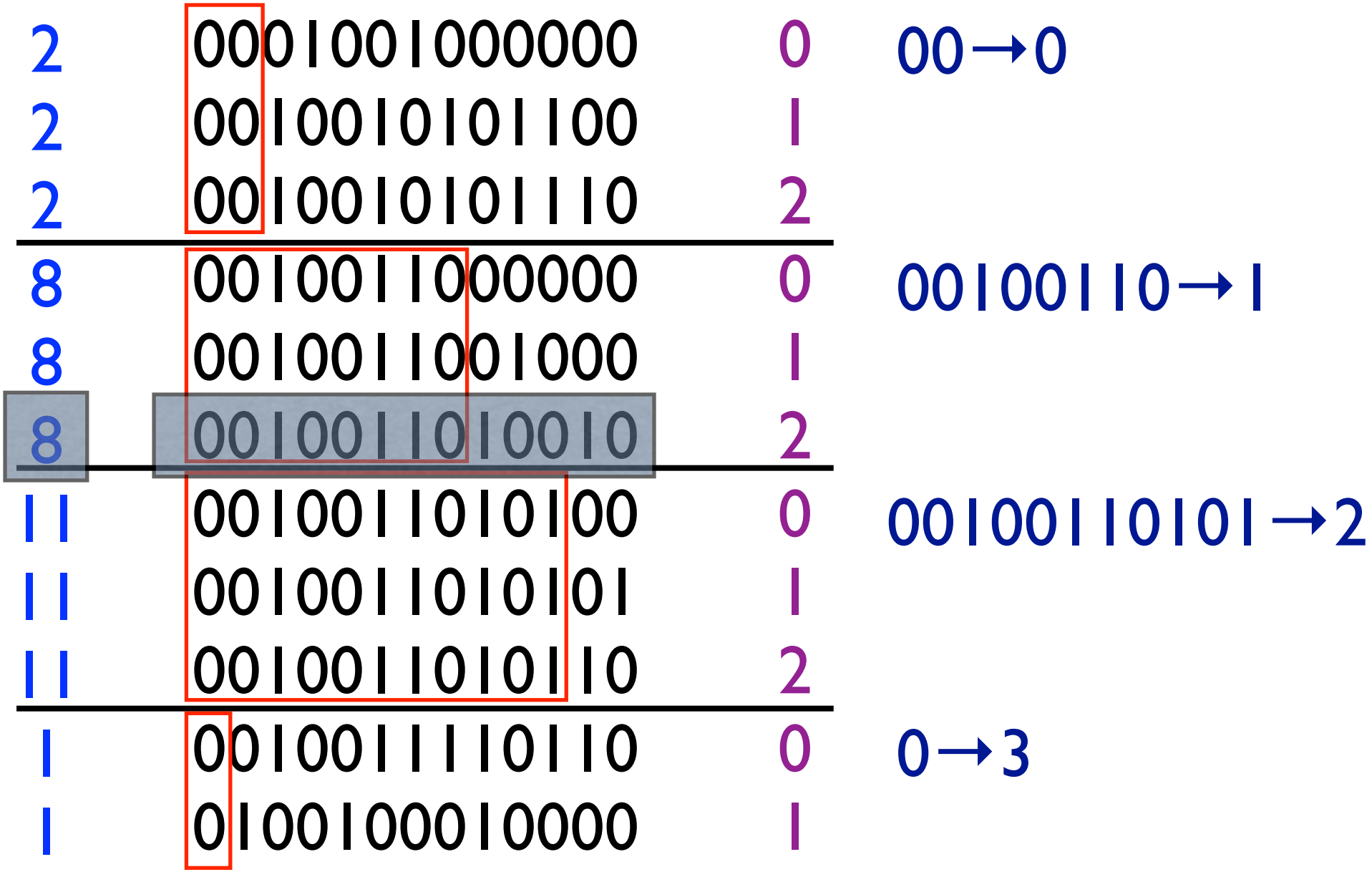

00|00|000000
00|00|0|0|1|00
00|00|0|0|1|1|0
00|00|1|000000
00|00|1|001000
00|00|1|0|00|0
00|00|1|0|0|00
00|00|1|0|0|01
00|00|1|0|0|1|0
00|00|1|1|1|0|1|0
0|00|1|000|0000
00|00|1|0→1
00|00|1|0|0→2

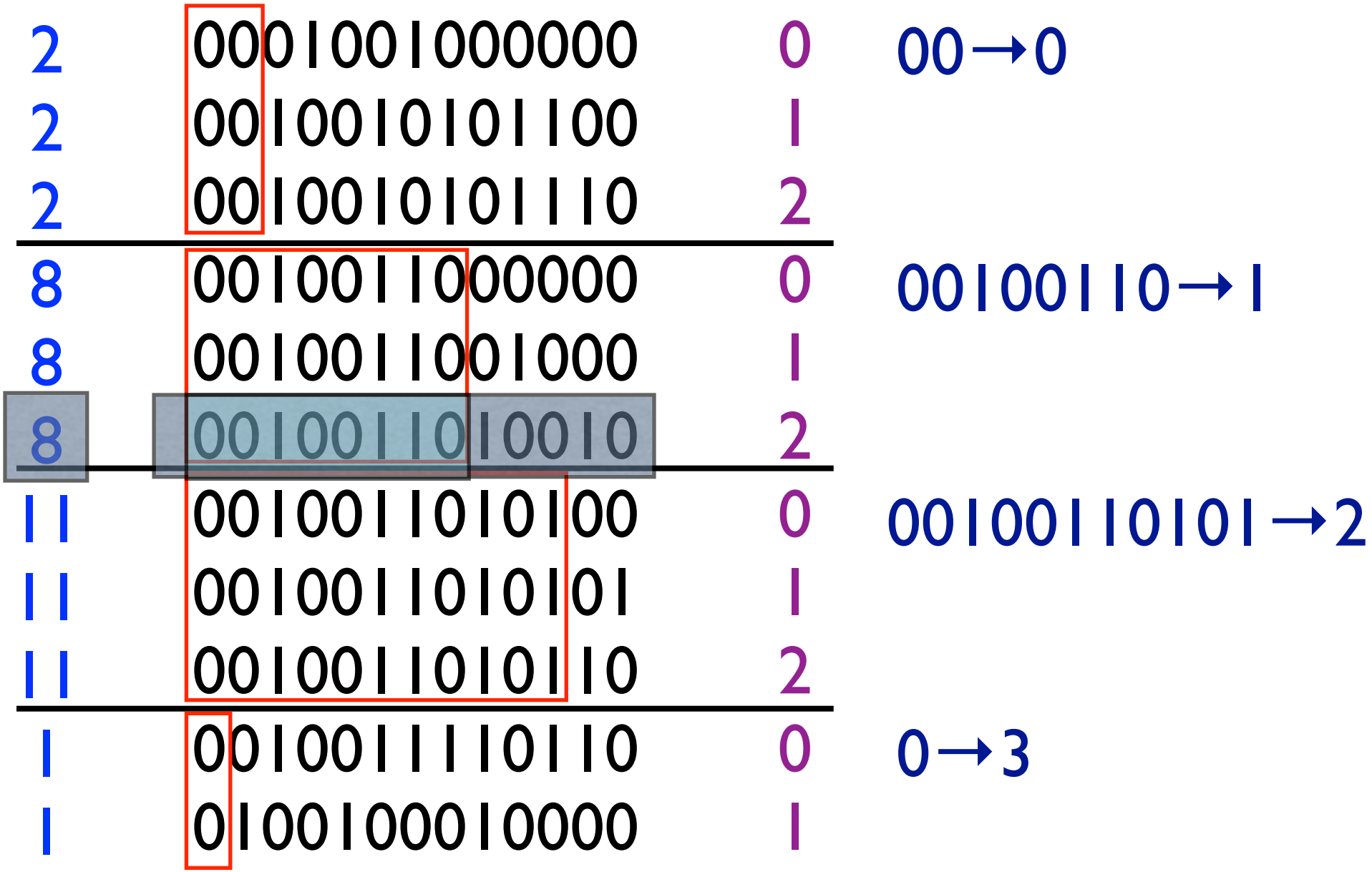

00|00|000000
00|00|0|0|1|00
00|00|0|0|1|1|0
00|00|1|000000
00|00|1|001000
00|00|1|0|00|0
00|00|1|0|0|00
00|00|1|0|0|01
00|00|1|0|0|1|0
00|00|1|1|1|0|1|0
0|00|0|00|0000
00|00 → 0
00|00|1|0 → 1
00|00|1|0|0|1 → 2
0 → 3

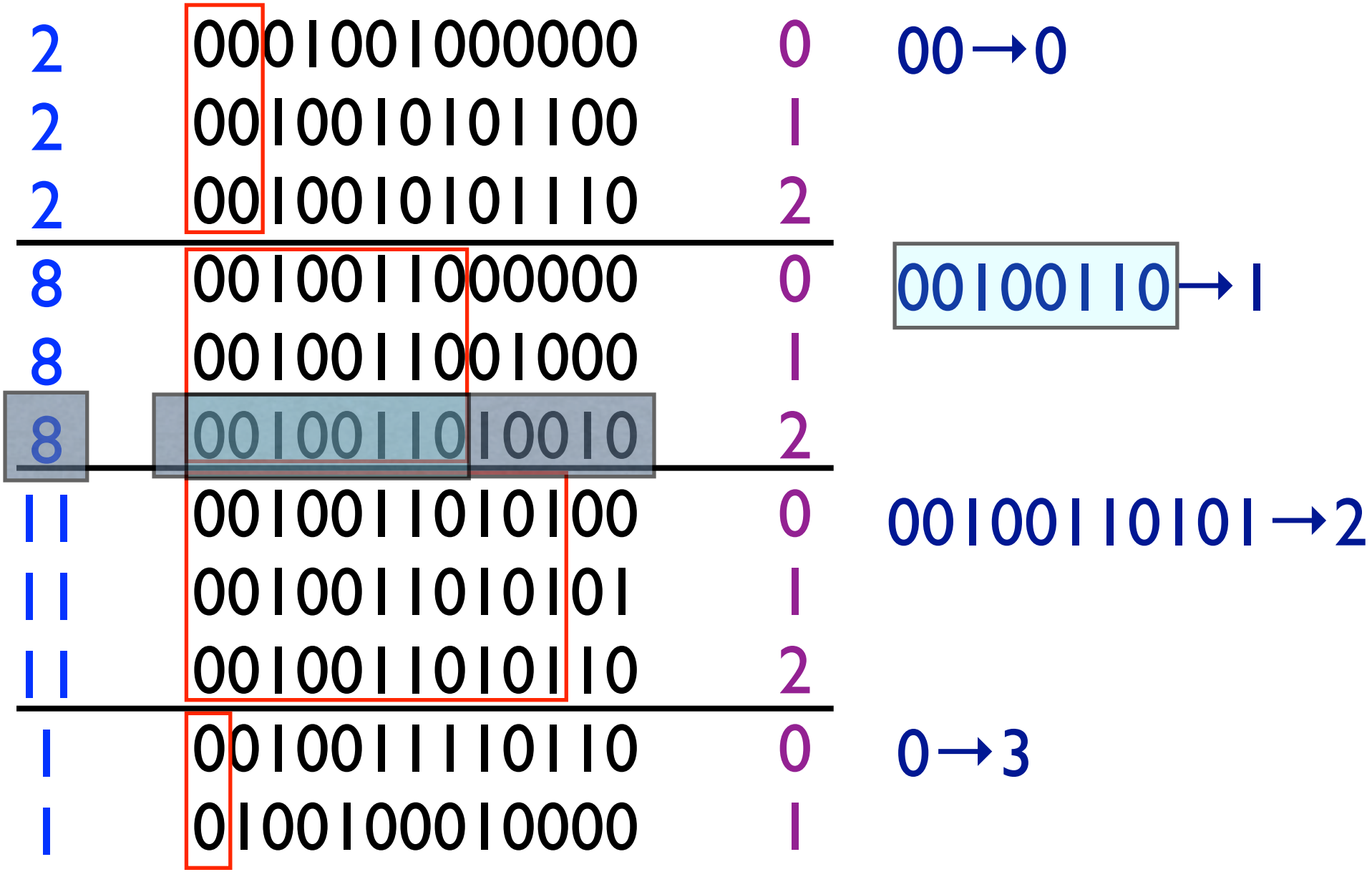

2 0001001000000
2 0010010101100
2 0010010101110
00→0
8 0010011000000
8 0010011001000
8 0010011001010
0010011001→1
11 0010011010100
11 0010011010101
11 0010011010110
0011→2
1 0010011110110
1 0100100010000
0→3

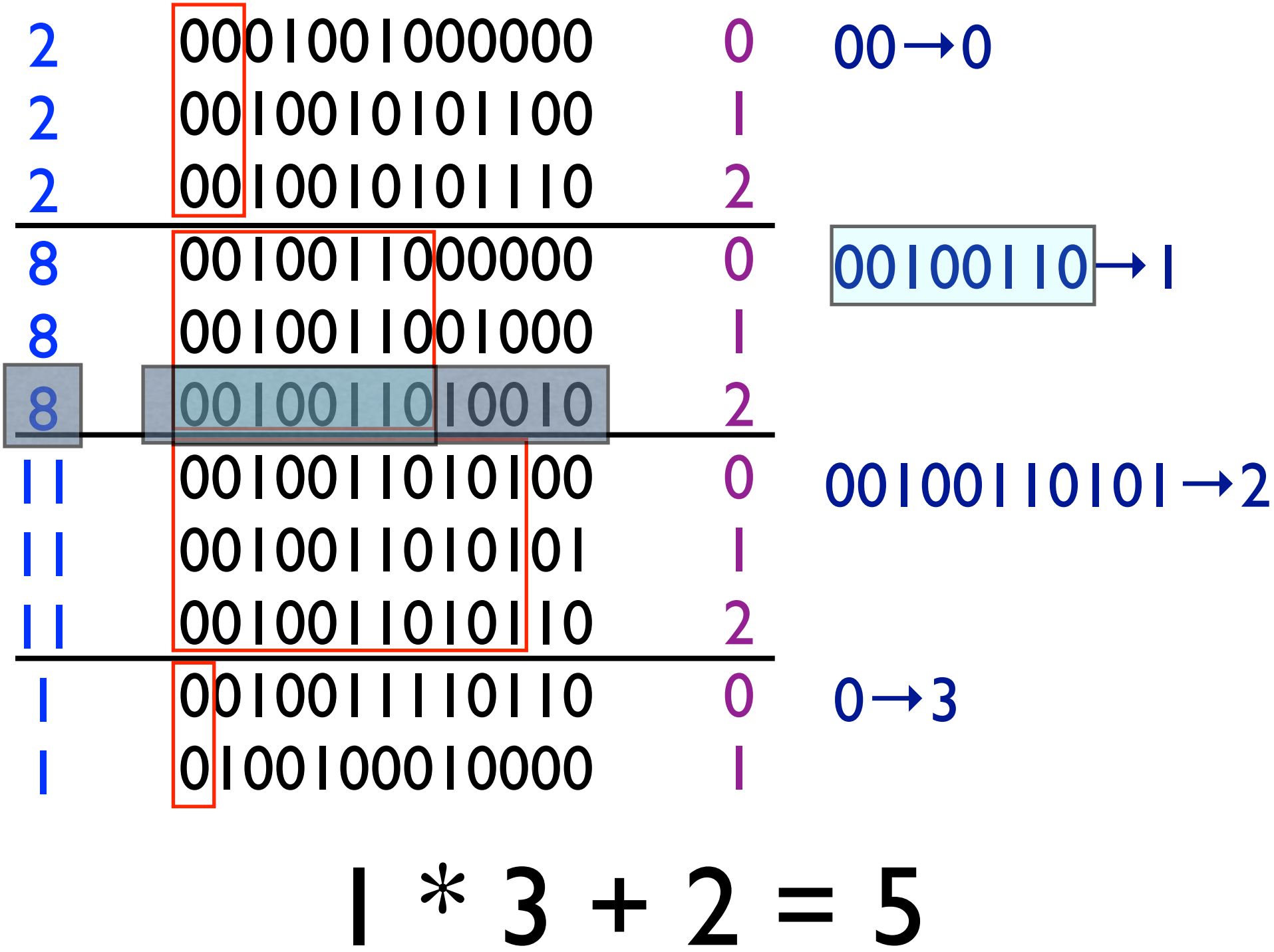

2 0001001000000 0
2 0010010101100 1
2 0010010101110 2
8 0010011000000 0
8 0010011001000 1
8 0010011001010 2
11 0010011010100 0
11 0010011010101 1
11 0010011010110 2
1 0010011110110 0
1 0100100010000 1
00→0
0010011→1
001001101→2
0→3

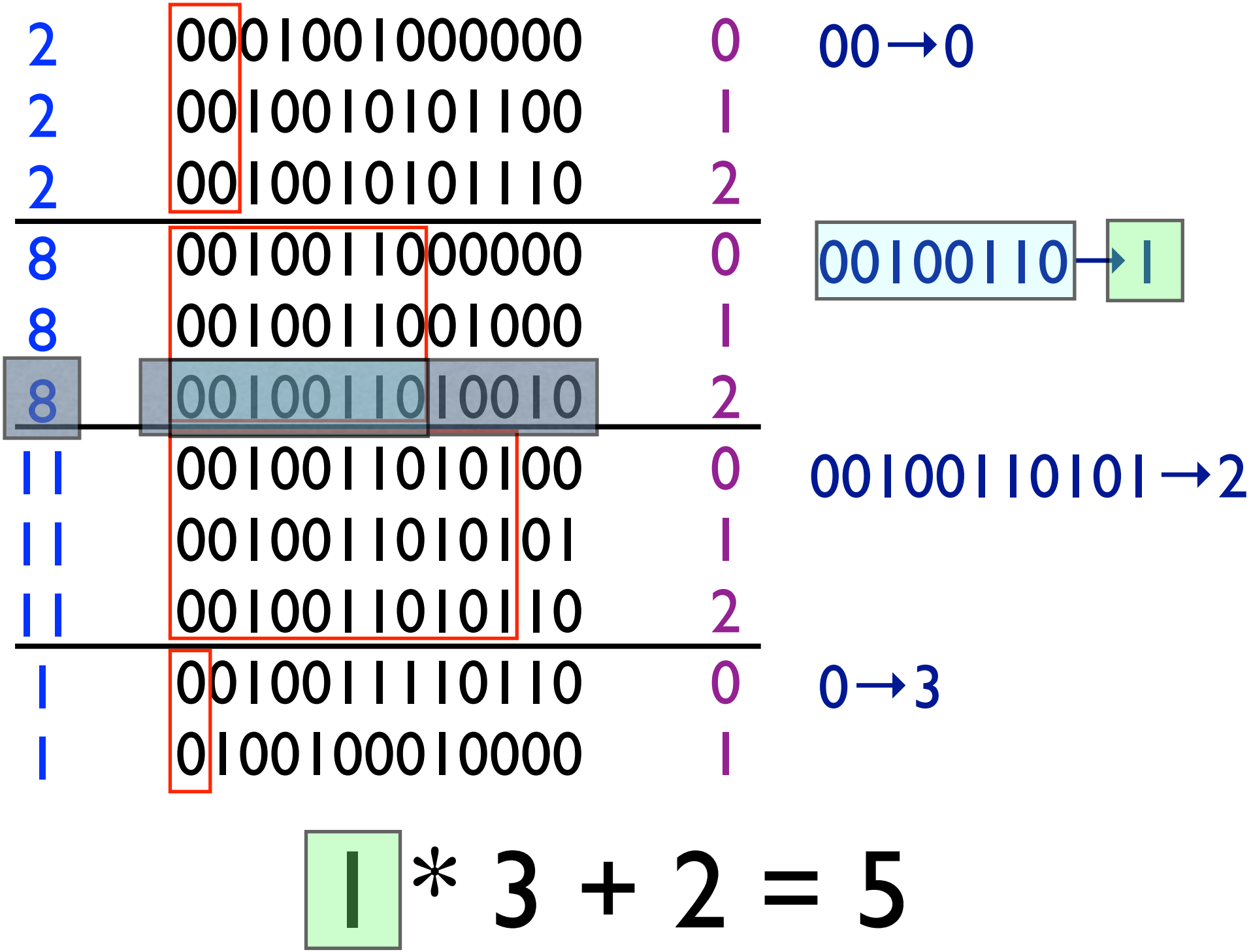

2 0001001000000 0
2 0010010101100 1
2 0010010101110 2
8 0010011000000 0
8 0010011001000 1
8 0010011001010 2
11 0010011010100 0
11 0010011010101 1
11 0010011010110 2
1 0010011110110 0
1 0100100010000 1
00→0
0010011→1
001001101101→2
0→3

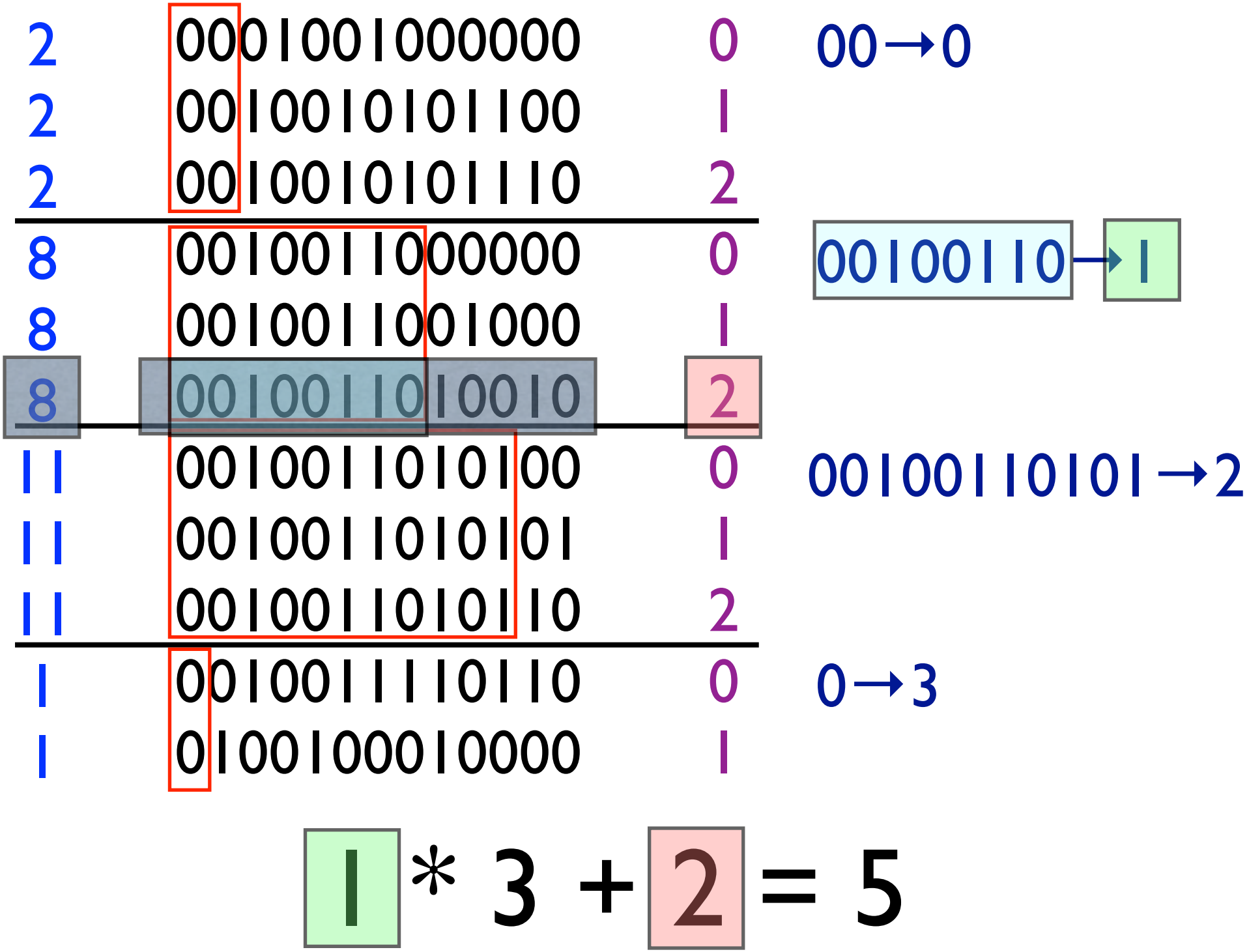

2 0001001000000 0
2 0010010101100 1
2 0010010101110 2
8 0010011000000 0
8 0010011001000 1
8 0010011001010 2
11 0010011010100 0
11 0010011010101 1
11 0010011010110 2
1 0010011110110 0
1 0100100010000 1
00→0
0010011→1
001001101101→2
0→3


2 0001001000000 0
2 0010010101100 1
2 0010010101110 2
8 0010011000000 0
8 0010011001000 1
8 0010011001010 2
11 0010011010100 0
11 0010011010101 1
11 0010011010110 2
1 0010011110110 0
1 0100100010000 1
00→0
0010011→1
001001101101→2
0→3


2 0001001000000 0
2 0010010101100 1
2 0010010101110 2
8 0010011000000 0
8 0010011001000 1
8 0010011001010 2
11 0010011010100 0
11 0010011010101 1
11 0010011010110 2
1 0010011110110 0
1 0100100010000 1
00→0
00100110→1
001001101101→2
0→3


2 0001001000000 0
2 0010010101100 1
2 0010010101110 2
8 0010011000000 0
8 0010011001000 1
8 0010011001010 2
11 0010011010100 0
11 0010011010101 1
11 0010011010110 2
1 0010011110110 0
1 0100100010000 1
00→0
00100110→1
001001101101→2
0→3
1 * 3 + 2 = 5


2 0001001000000 0
2 0010010101100 1
2 0010010101110 2
8 0010011000000 0
8 0010011001000 1
8 0010011001010 2
11 0010011010100 0
11 0010011010101 1
11 0010011010110 2
1 0010011110110 0
1 0100100010000 1
00→0
00100110→1
001001101101→2
0→3
1 * 3 + 2 = 5


2 0001001000000 0
2 0010010101100 1
2 0010010101110 2
8 0010011000000 0
8 0010011001000 1
8 0010011001010 2
11 0010011010100 0
11 0010011010101 1
11 0010011010110 2
1 0010011110110 0
1 0100100010000 1
00→0
00100110→1
001001101101→2
0→3
1 * 3 + 2 = 5

What do we get?

What do we get?

- If buckets are sized as log *n*...

What do we get?

- If buckets are sized as log n...
  - the LCP-length function takes $O(n \log w)$ bits

What do we get?

- If buckets are sized as log n...
  - the LCP-length function takes $O(n \log w)$ bits
  - the offset function takes $O(n \log \log n) = O(n \log w)$ bits

What do we get?

- If buckets are sized as log n...
  - the LCP-length function takes $O(n \log w)$ bits
  - the offset function takes $O(n \log \log n)=O(n \log w)$ bits
  - the LCP-to-bucket function takes $O((n/\log n) \log w)=o(n)$ bits

What do we get?

- If buckets are sized as log n...
  - the LCP-length function takes $O(n \log w)$ bits
  - the offset function takes $O(n \log \log n)=O(n \log w)$ bits
  - the LCP-to-bucket function takes $O((n/\log n) \log w)=o(n)$ bits
  - ...we get constant-time monotone minimal perfect hashing in $O(n \log w)$ bits

Can we do better?

Can we do better?

- Not absolutely better, but we propose a different space/time tradeoff using a new variant of the "fast trie" family of data structure that has (really!) linear space occupancy and query time log *w*

Can we do better?

- Not absolutely better, but we propose a different space/time tradeoff using a new variant of the "fast trie" family of data structure that has (really!) linear space occupancy and query time log *w*
- Consider the *delimiter set D*—the set containing, for each bucket, the last string

Can we do better?

- Not absolutely better, but we propose a different space/time tradeoff using a new variant of the "fast trie" family of data structure that has (really!) linear space occupancy and query time log *w*
- Consider the *delimiter set D*—the set containing, for each bucket, the last string
- The basic idea is to use a z-fast trie on *D* as a distributor instead of LCPs (clearly, a trie on *D* would be fine, but too big and too slow!)

Can we do better?

- Not absolutely better, but we propose a different space/time tradeoff using a new variant of the "fast trie" family of data structure that has (really!) linear space occupancy and query time log *w*
- Consider the *delimiter set D*—the set containing, for each bucket, the last string
- The basic idea is to use a z-fast trie on *D* as a distributor instead of LCPs (clearly, a trie on *D* would be fine, but too big and too slow!)
- The structure is of independent interest; many other applications are coming...

Z-Fast Tries

Z-Fast Tries

- Call f in [$x..y$] the *2-fattest number* in [$x..y$] if f has the largest number of trailing zeroes in its binary representation; equivalently, if $f = a2^b$ with maximum b in [$x..y$]

Z-Fast Tries

- Call f in $[x..y]$ the *2-fattest number* in $[x..y]$ if f has the largest number of trailing zeroes in its binary representation; equivalently, if $f = a2^b$ with maximum b in $[x..y]$
- Consider the *compacted trie* on S. Call:

Z-Fast Tries

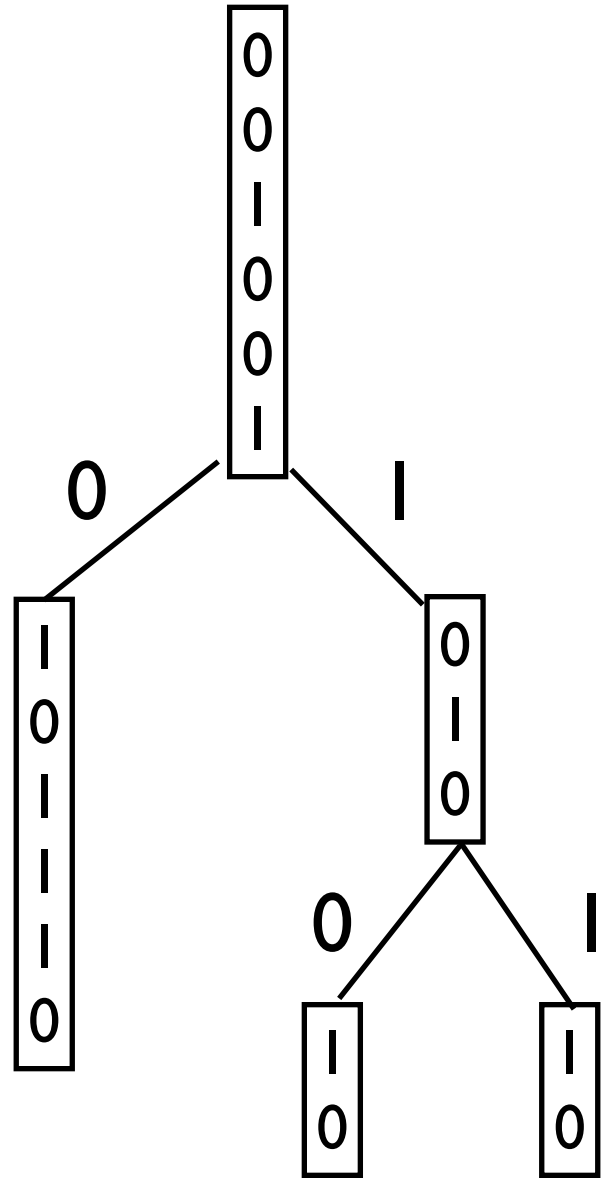
- Call *f* in [*x*..*y*] the *2-fattest number* in [*x*..*y*] if *f* has the largest number of trailing zeroes in its binary representation; equivalently, if $f = a2^b$ with maximum *b* in [*x*..*y*]
- Consider the *compacted trie* on *S*. Call:
  - *name* of a node the path leading to the node

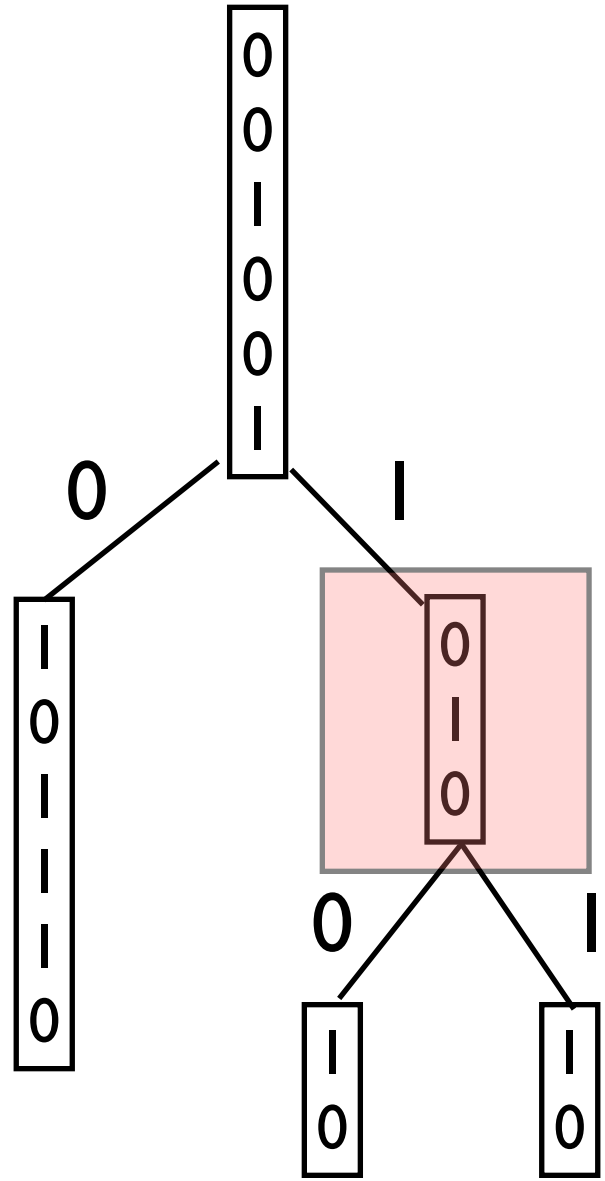
Z-Fast Tries

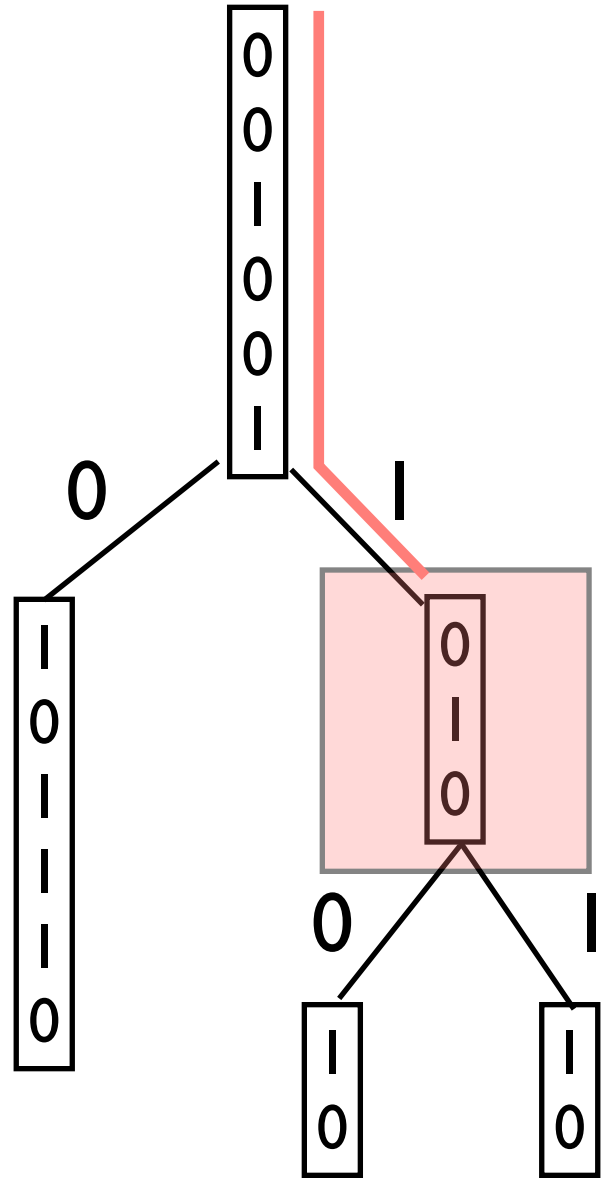
- Call *f* in [*x*..*y*] the *2-fattest number* in [*x*..*y*] if *f* has the largest number of trailing zeroes in its binary representation; equivalently, if $f = a2^b$ with maximum *b* in [*x*..*y*]
- Consider the *compacted trie* on *S*. Call:
  - *name* of a node the path leading to the node
  - *extent* of a node the path *represented* by the node

Z-Fast Tries

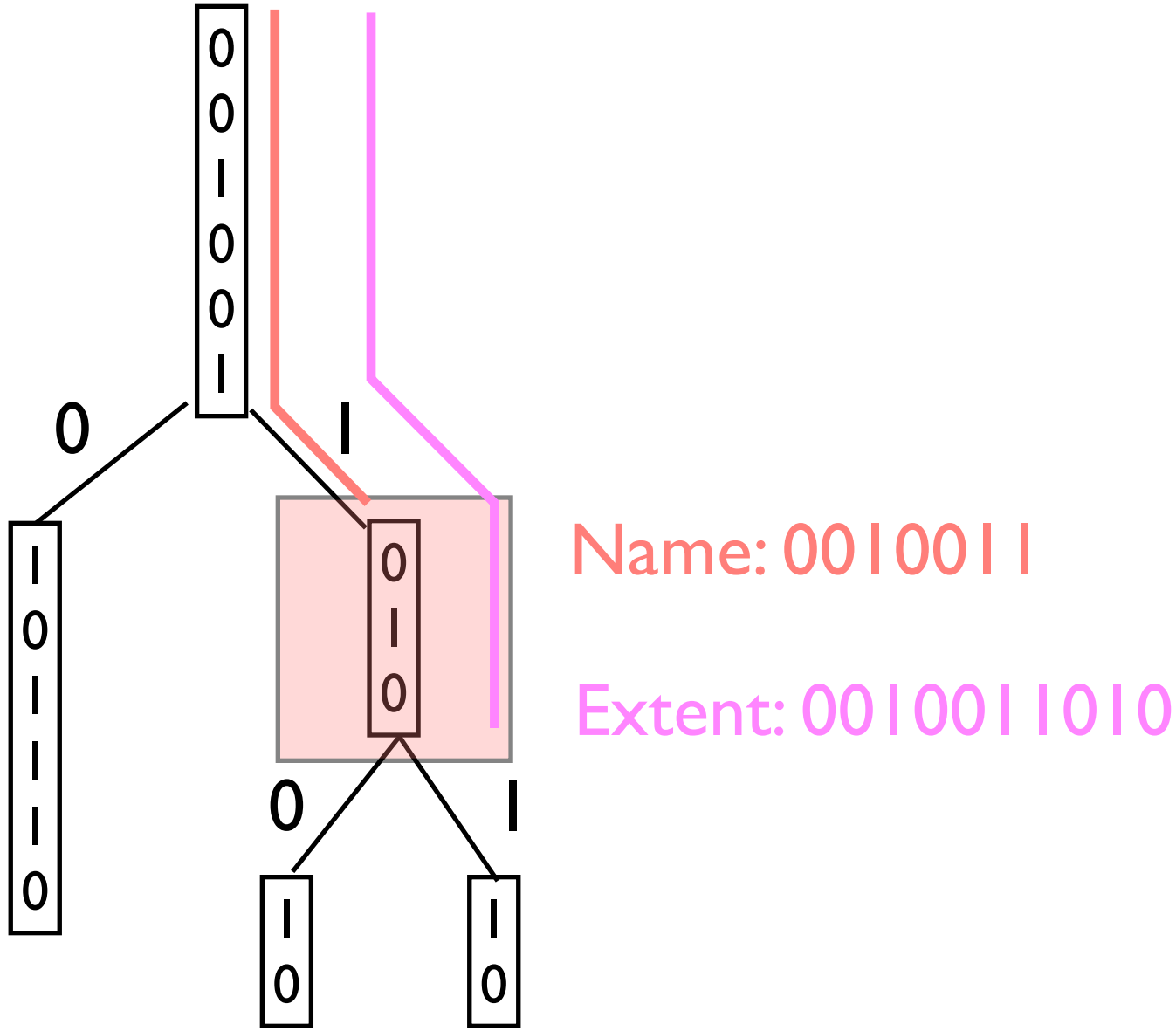
- Call f in $[x..y]$ the *2-fattest number* in $[x..y]$ if f has the largest number of trailing zeroes in its binary representation; equivalently, if $f = a2^b$ with maximum b in $[x..y]$
- Consider the *compacted trie* on S. Call:
  - *name* of a node the path leading to the node
  - *extent* of a node the path *represented* by the node
  - *skip interval* of a node the interval $[l..r)$ of positions, where l is the length of the extent of the parent (ε for the root) and r the length of the extent of the node

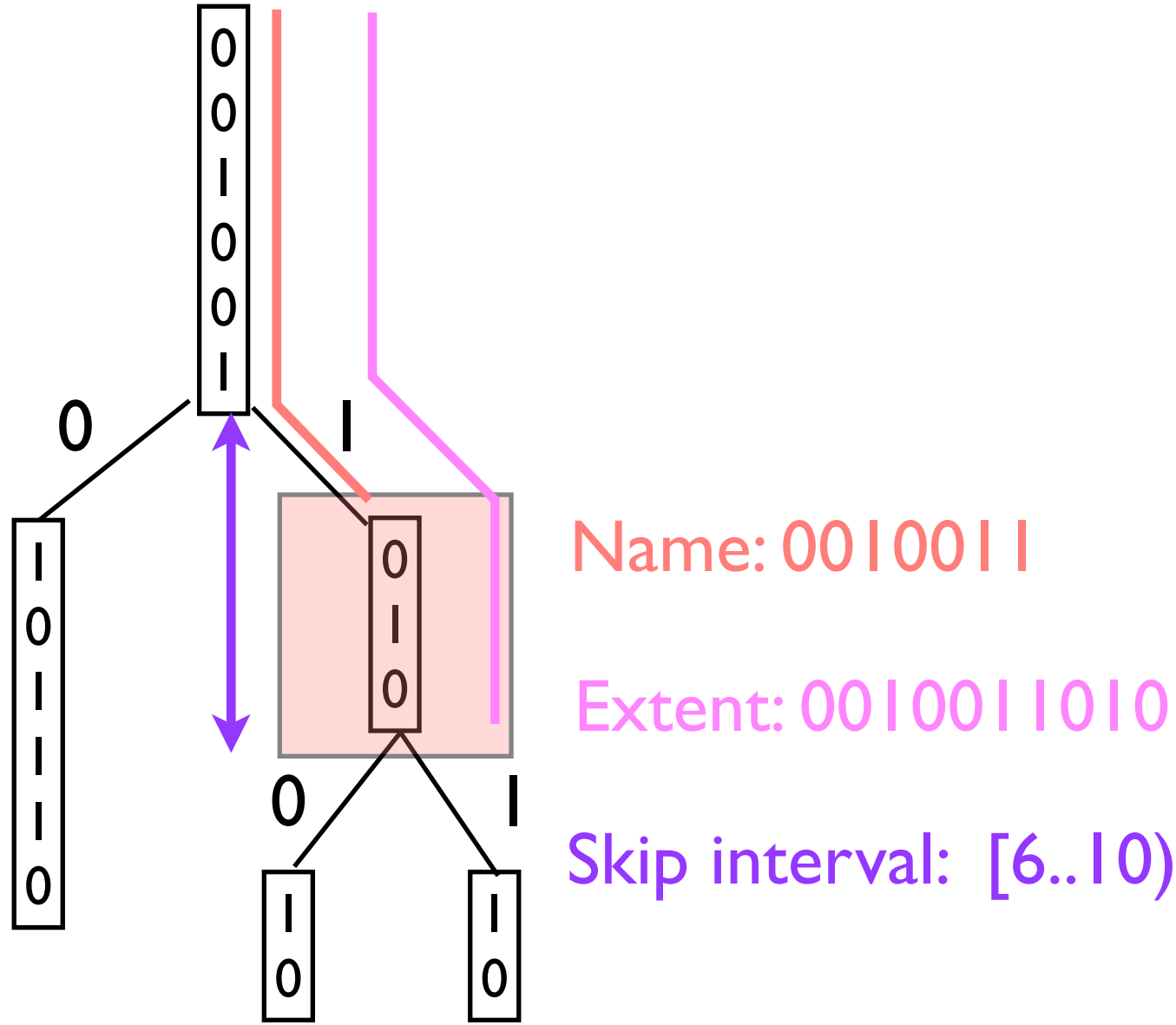

0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
0
1
0
1
0


0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0



Name: 0010011


0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
1
0
1
0
Name: 0010011
Extent: 00100111010


0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
1
0
1
0
Name: 0010011
Extent: 00100111010
Skip interval: [6..10)

The Magic Trick

The Magic Trick

- For a node with extent *e*, we store exactly one mapping from a certain prefix of *e* to *e*

The Magic Trick

- For a node with extent e, we store exactly one mapping from a certain prefix of e to e
- More precisely, if the skip interval of the node is $[l..r)$ we store a *handle*—the prefix of e of length given by the 2-fattest number in $(l..r]$

The Magic Trick

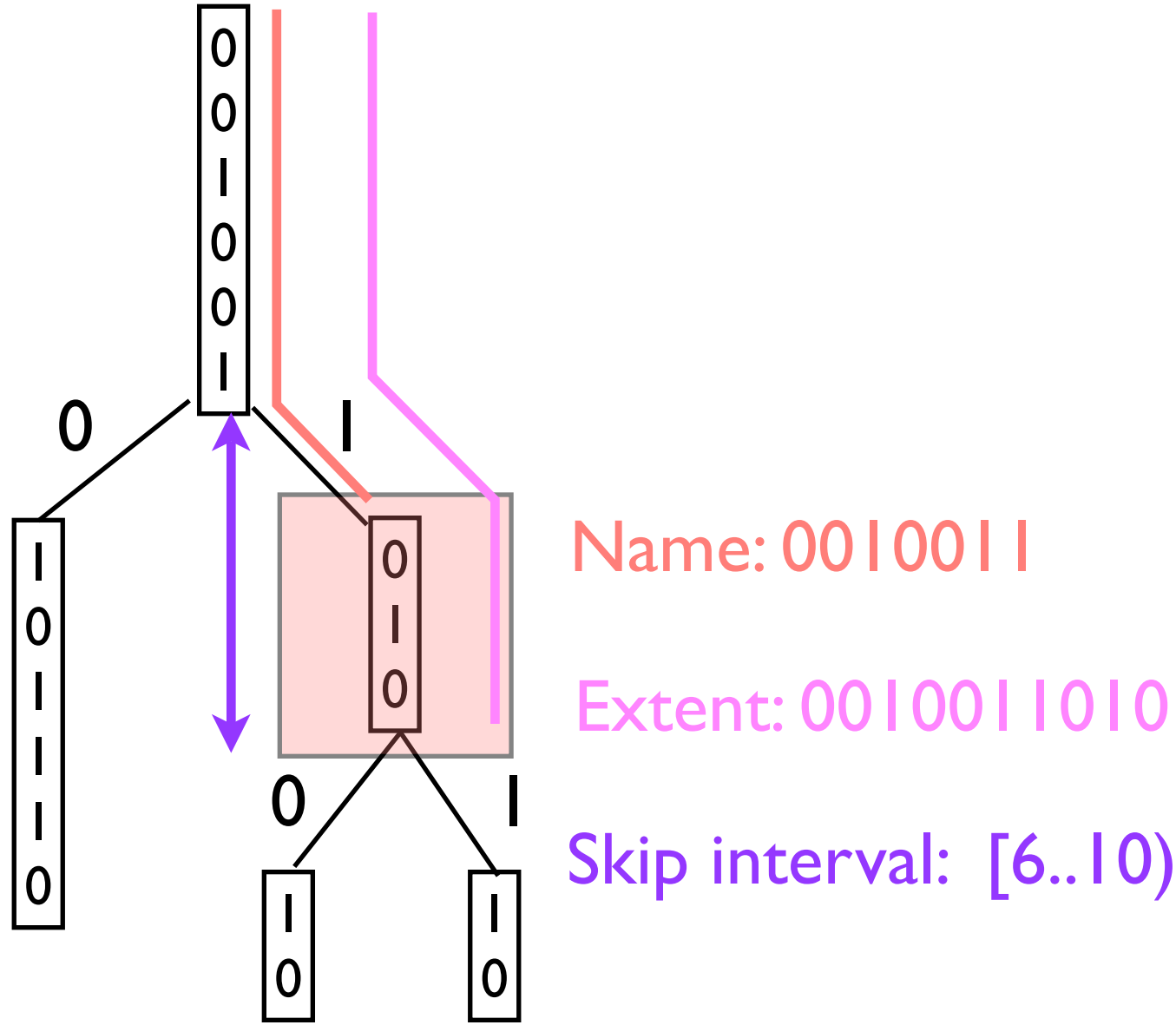
- For a node with extent e, we store exactly one mapping from a certain prefix of *e* to *e*
- More precisely, if the skip interval of the node is [*l*..*r*) we store a *handle*—the prefix of e of length given by the 2-fattest number in (*l*..*r*] ⇐Note the change!

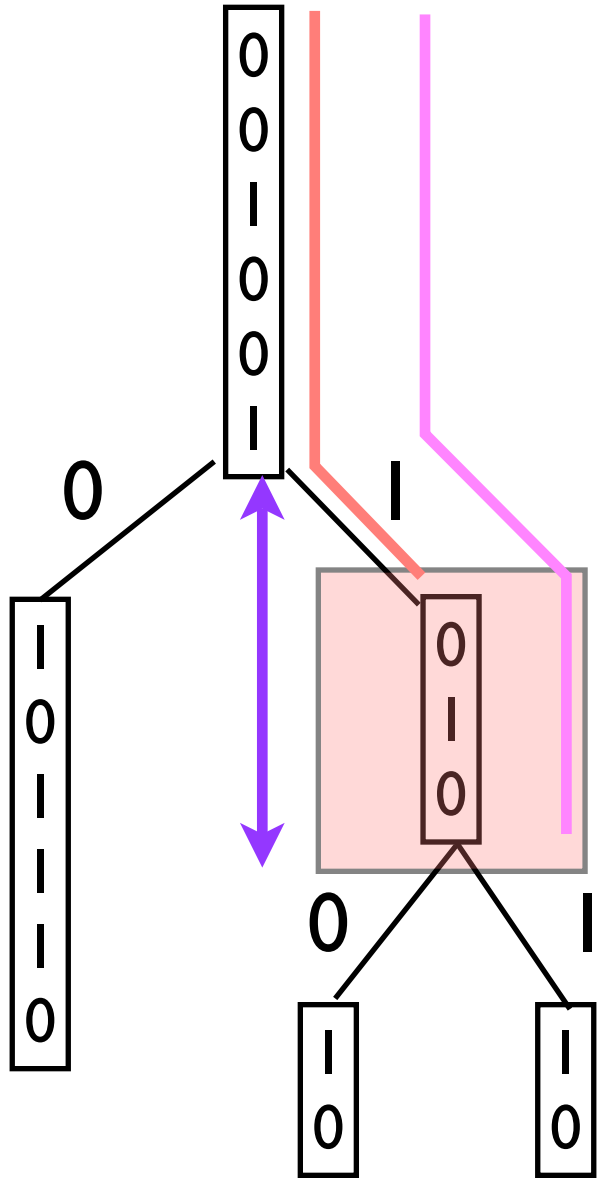
The Magic Trick

- For a node with extent e, we store exactly one mapping from a certain prefix of e to e
- More precisely, if the skip interval of the node is $[l..r)$ we store a *handle*—the prefix of e of length given by the 2-fattest number in $(l..r]$ ⇐Note the change!
- This mapping requires linear (i.e., $O(nw)$) space and it is sufficient to navigate the trie!

The Magic Trick

- For a node with extent e, we store exactly one mapping from a certain prefix of e to e
- More precisely, if the skip interval of the node is $[l..r)$ we store a *handle*—the prefix of e of length given by the 2-fattest number in $(l..r]$ ⇐Note the change!
- This mapping requires linear (i.e., $O(nw)$) space and it is sufficient to navigate the trie!
- All in all, the z-fast trie is entirely specified by the mapping from handles to extent prefixes


0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
1
0
1
0
Name: 0010011
Extent: 00100111010
Skip interval: [6..10)

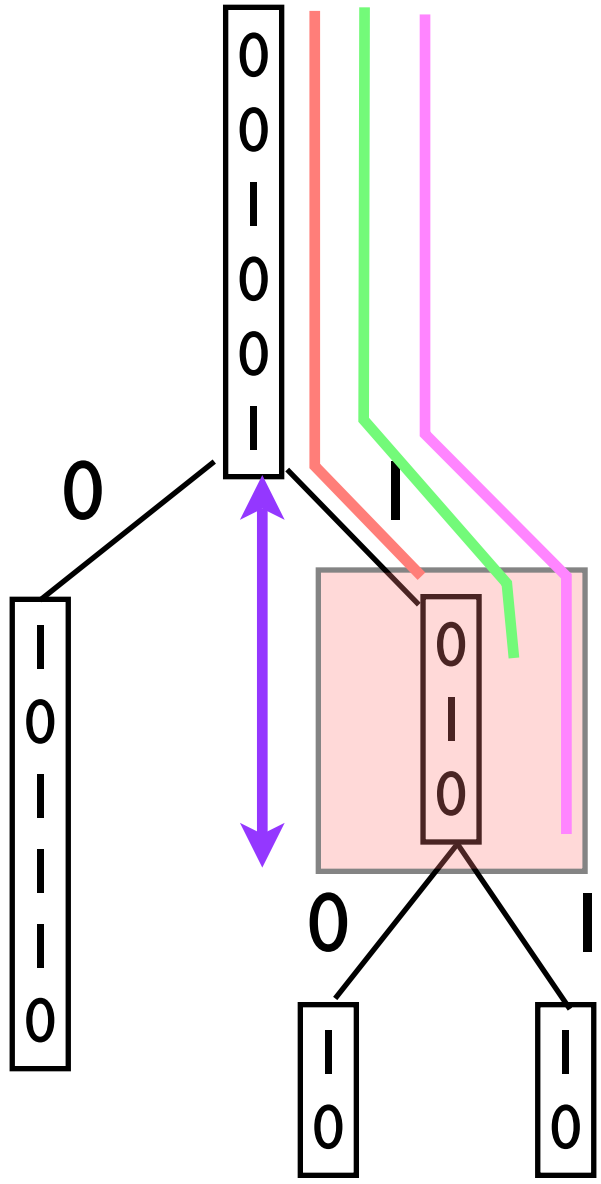


Name: 0010011

Extent: 00100111010

Skip interval: [6..10)

2-fattest is 8



Name: 0010011

Extent: 00100111010

Skip interval: [6..10)

2-fattest is 8

We store the mapping 00100110 → 00100111010

Fat Binary Search

Fat Binary Search

- A *fat binary search* is similar to a binary search, but probes always insist on the 2-fattest number in an interval, rather than on its middle point

Fat Binary Search

- A *fat binary search* is similar to a binary search, but probes always insist on the 2-fattest number in an interval, rather than on its middle point
- In doing a fat binary search on x, our goal is to find *the longest extent e of the trie that is a prefix of x*

Fat Binary Search

- A *fat binary search* is similar to a binary search, but probes always insist on the 2-fattest number in an interval, rather than on its middle point
- In doing a fat binary search on x, our goal is to find *the longest extent e of the trie that is a prefix of x*
- Clearly, $x[0..|e|+1)$ is the name of the exit node (we are adding one bit from x to e)

Fat Binary Search

Notation: substring with indices in the given interval

- A *fat binary search* is similar to a binary search, but probes always insist on the 2-fattest number in an interval, rather than on its middle point
- In doing a fat binary search on x, our goal is to find *the longest extent e of the trie that is a prefix of x*
- Clearly, $x[0..|e|+1)$ is the name of the exit node (we are adding one bit from x to e)

Fat Binary Search

Notation: substring with indices in the given interval

- A *fat binary search* is similar to a binary search, but probes always insist on the 2-fattest number in an interval, rather than on its middle point
- In doing a fat binary search on x, our goal is to find *the longest extent e of the trie that is a prefix of x*
- Clearly, $x[0..|e|+1)$ is the name of the exit node (we are adding one bit from x to e)
- We will not know the exit direction (left/right)

Fat Binary Search

Fat Binary Search

- When given a query string x, we keep track of a current interval of positions $[l..r)$ in which the extent of the parent of the exit node of x might lie

Fat Binary Search

- When given a query string x, we keep track of a current interval of positions $[l..r)$ in which the extent of the parent of the exit node of x might lie
- At each step, we interrogate the trie with the prefix of x of length f—the 2-fattest number in $(l..r)$

Fat Binary Search

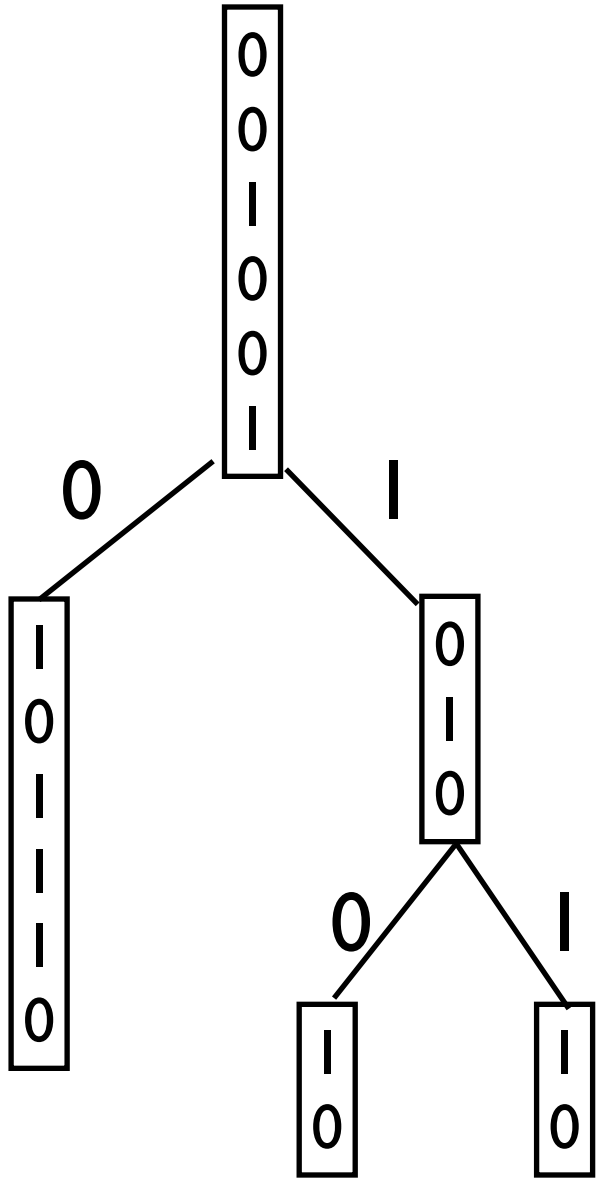
- When given a query string x, we keep track of a current interval of positions $[l..r)$ in which the extent of the parent of the exit node of x might lie
- At each step, we interrogate the trie with the prefix of x of length f—the 2-fattest number in $(l..r)$
- Key property: f is 2-fattest in every subinterval of $(l..r)$ that contains it

Fat Binary Search

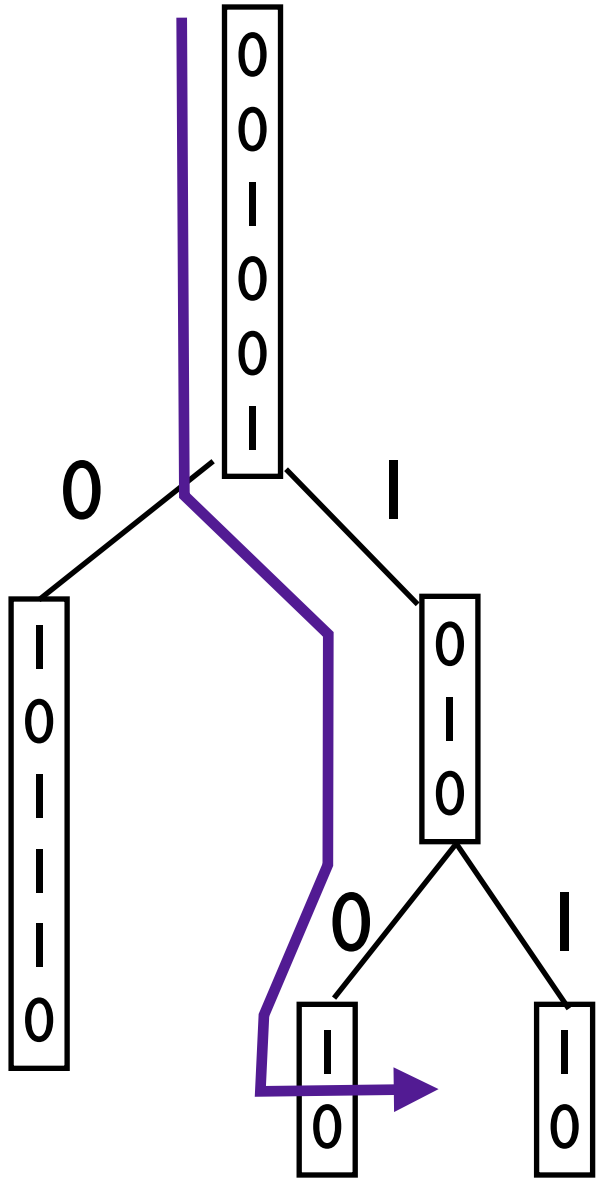
- When given a query string *x*, we keep track of a current interval of positions [*l*..*r*) in which the extent of the parent of the exit node of *x* might lie
- At each step, we interrogate the trie with the prefix of *x* of length *f*—the 2-fattest number in (*l*..*r*)
- Key property: *f* is 2-fattest in every subinterval of (*l*..*r*) that contains it
- Because of this fact, every time we query the z-fast trie with a prefix of *x* we will be using a handle, given that it is a prefix of some string in the trie

Fat Binary Search

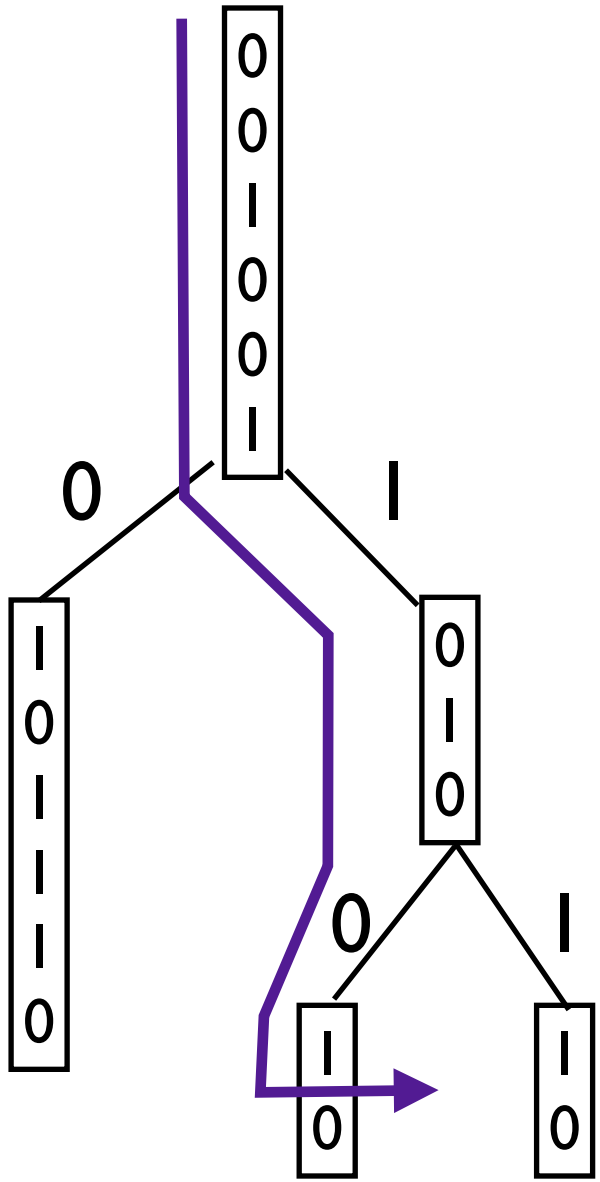
- When given a query string x, we keep track of a current interval of positions $[l..r)$ in which the extent of the parent of the exit node of x might lie
- At each step, we interrogate the trie with the prefix of x of length f—the 2-fattest number in $(l..r)$
- Key property: f is 2-fattest in every subinterval of $(l..r)$ that contains it
- Because of this fact, every time we query the z-fast trie with a prefix of x we will be using a handle, given that it is a prefix of some string in the trie
- If x is a prefix of the resulting string s *and* $s \in (l..r)$, we move to $(|s|..r)$; otherwise, we move to $(l..f)$



x = 0010011010011

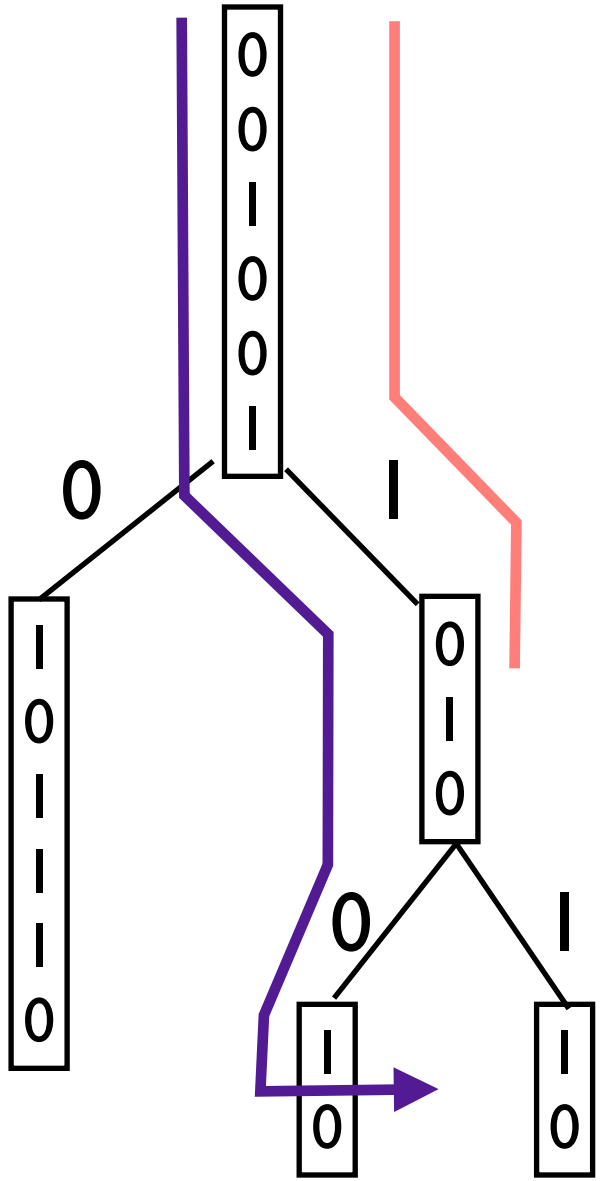


x = 0010011010011



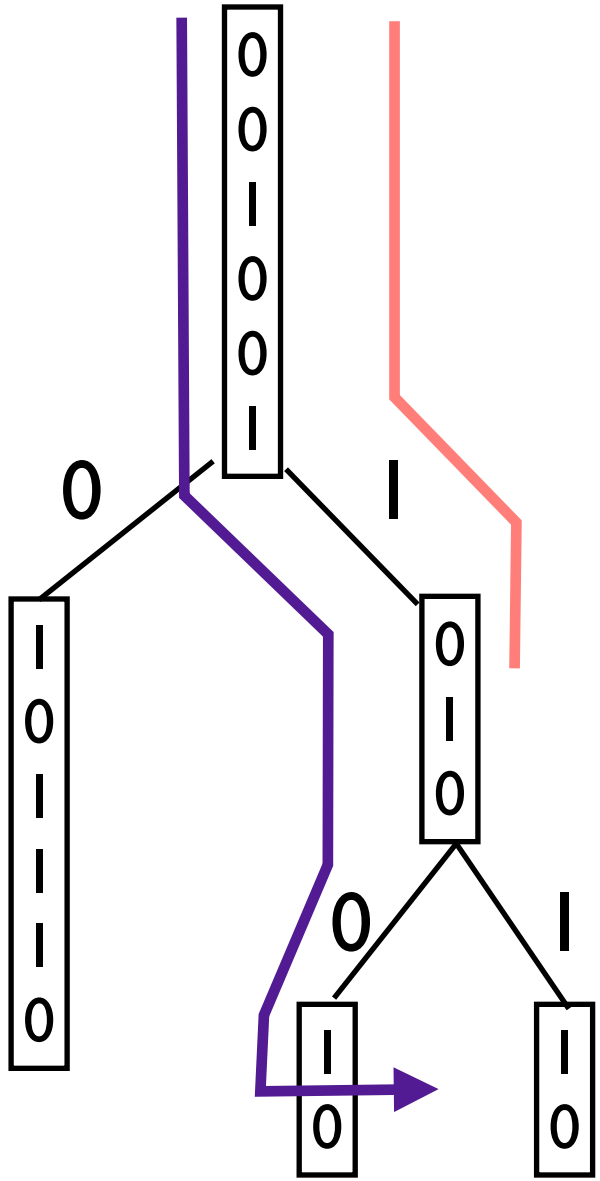
$(l..r) = (0..13) \Rightarrow f = 8$

x = 00100111010011



$(l..r) = (0..13) \Rightarrow f = 8$

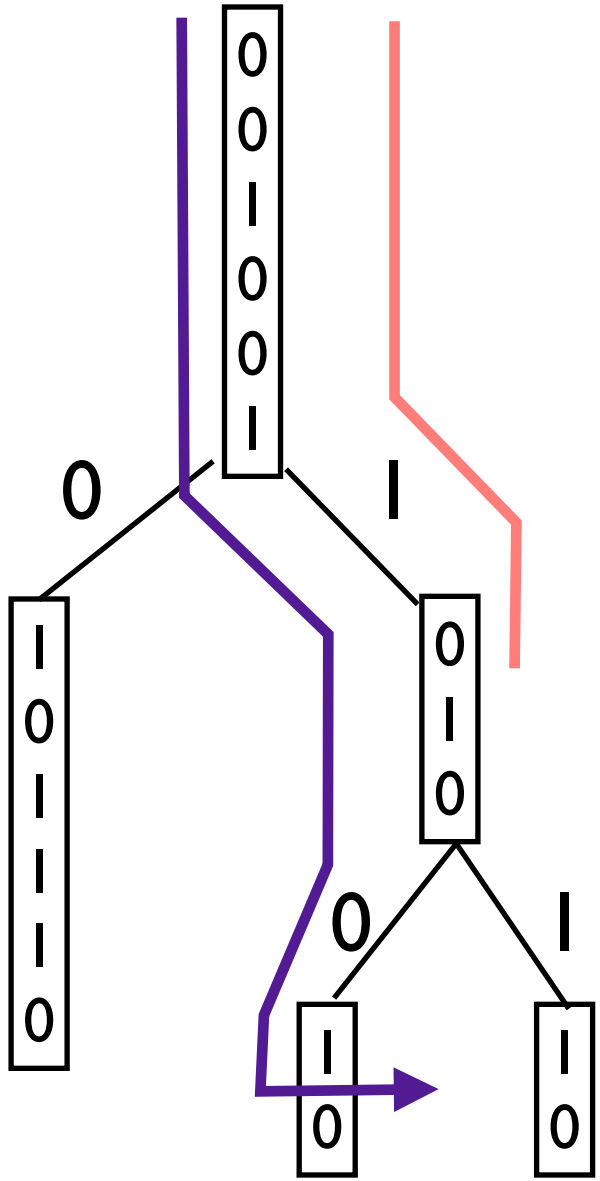
$x = 00100110100111$



$(l..r) = (0..13) \Rightarrow f = 8$

Function returns 00100111010 ≼ x

x = 00100111010011

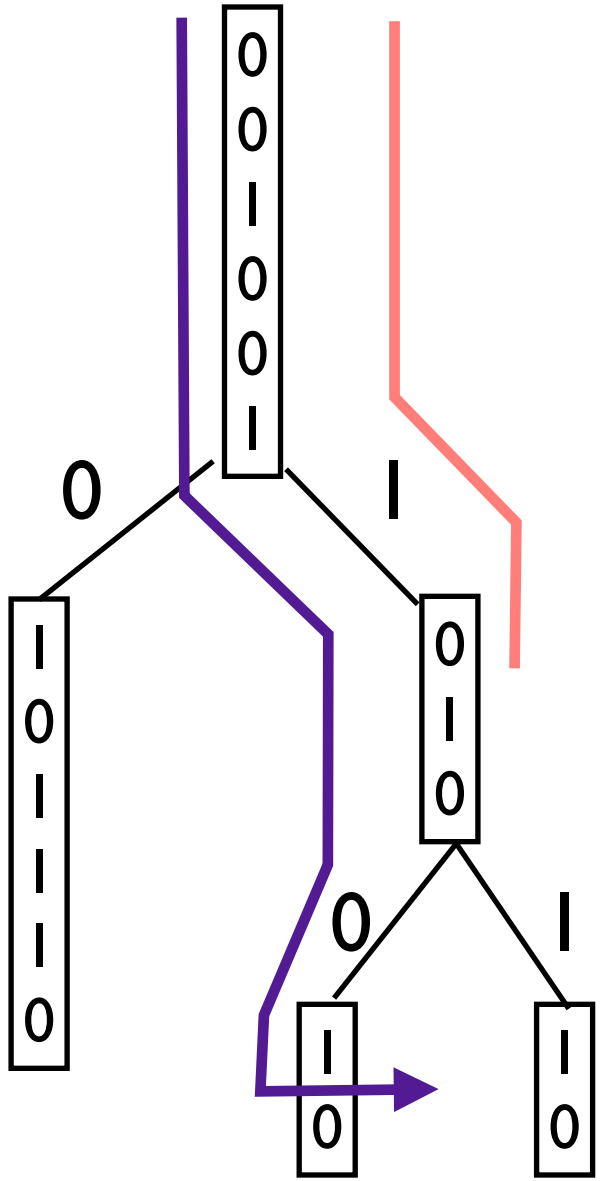


$(l..r) = (0..13) \Rightarrow f = 8$

Function returns 00100111010 $\preccurlyeq x$

$\Rightarrow (l..r) = (10..13) \Rightarrow f = 12$

x = 00100110 10011



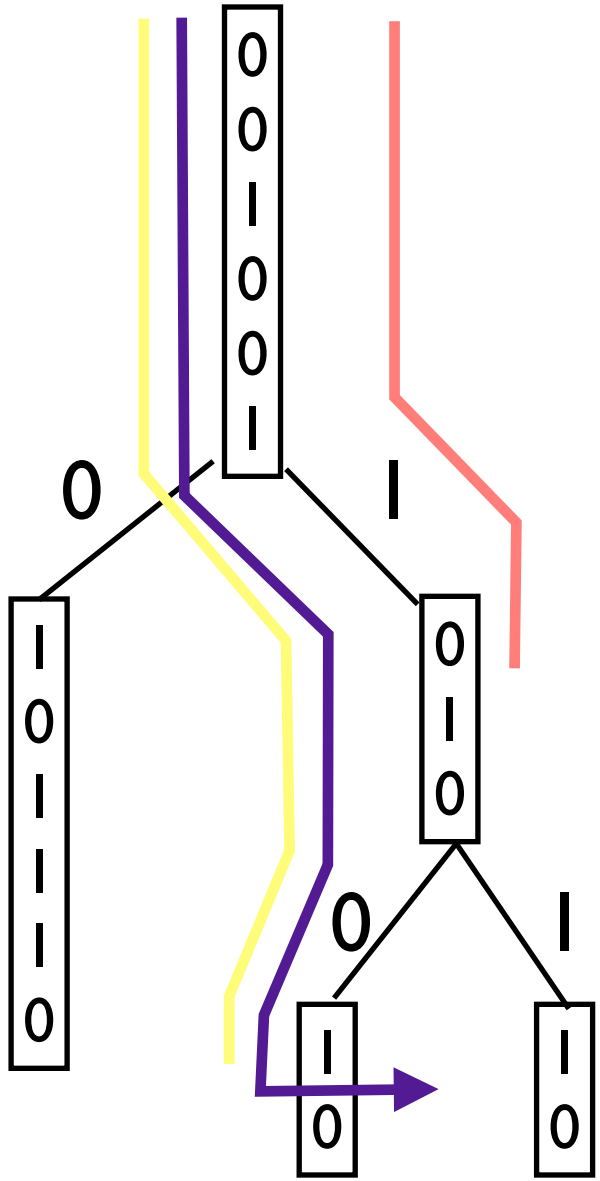
$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $00100111010 \leqslant x$

$\Rightarrow (l..r) = (10..13) \Rightarrow f = 12$

Note: binary search would move to (8..13)!

$x = 00100110\,10011$



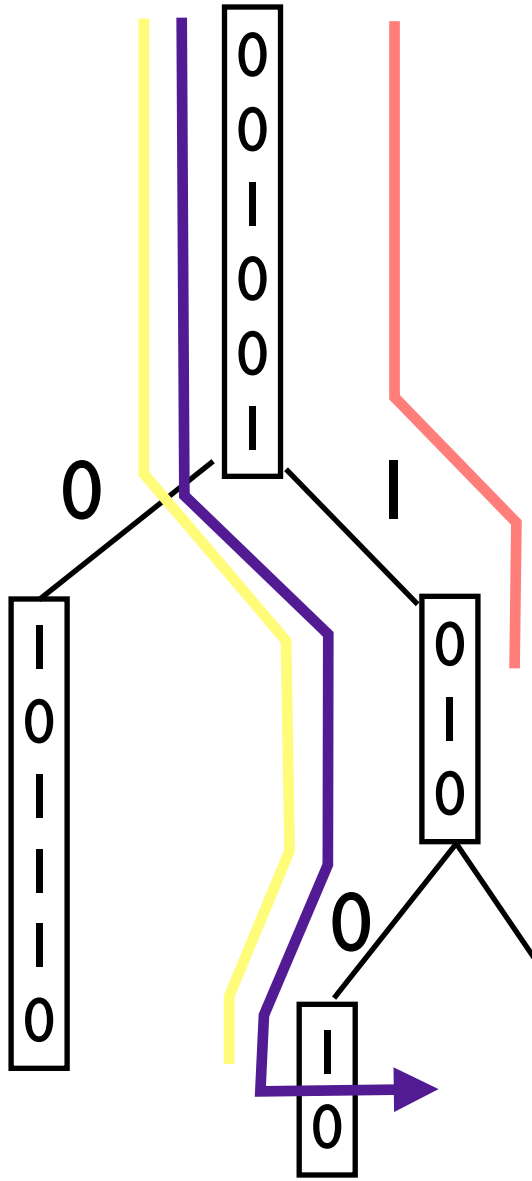
$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $00100011010 \leqslant x$

$\Rightarrow (l..r) = (10..13) \Rightarrow f = 12$

Note: binary search would move to (8..13)!

$x = 00100011010011$



$(l..r) = (0..13) \Rightarrow f = 8$

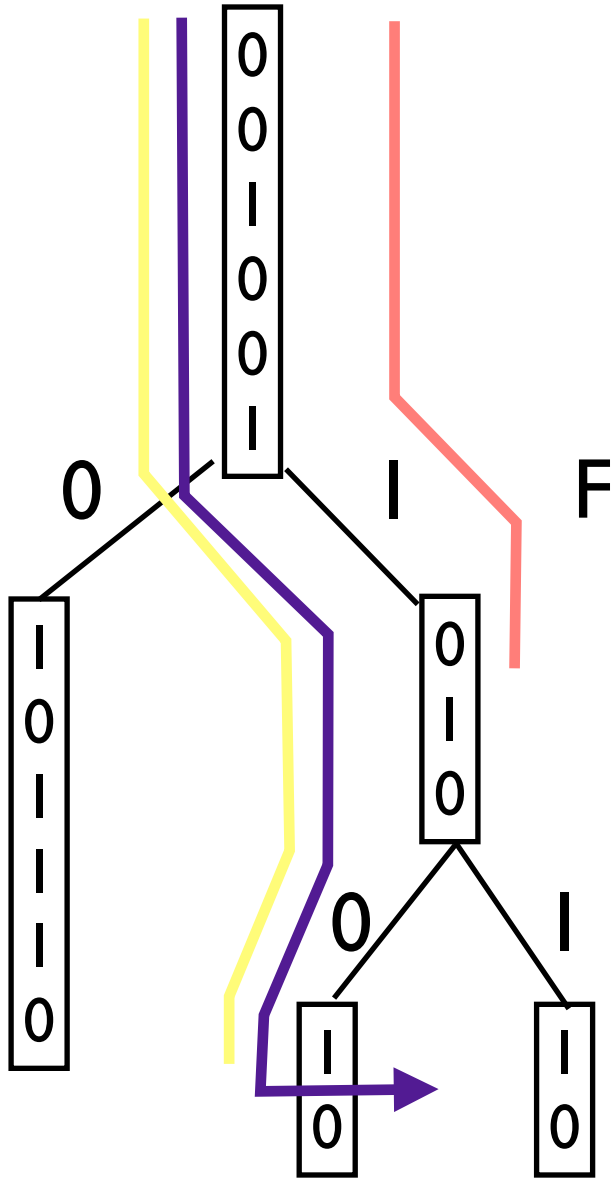
Function returns $0010011010 \leqslant x$

$\Rightarrow (l..r) = (10..13) \Rightarrow f = 12$

Note: binary search would move to (8..13)!

Function returns $00100110100010 \not\leqslant x$

$x = 00100110100111$



$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $0010011010 \preccurlyeq x$

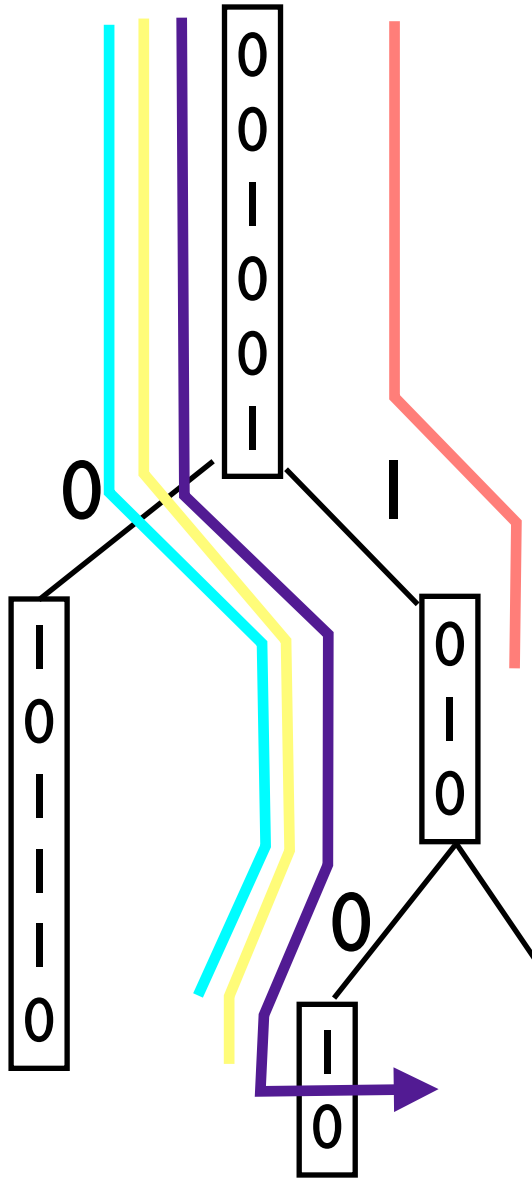
$\Rightarrow (l..r) = (10..13) \Rightarrow f = 12$

Note: binary search would move to (8..13)!

Function returns $00100110010010 \not\preccurlyeq x$

$\Rightarrow (l..r) = (10..12) \Rightarrow f = 11$

x = 00100110100111



$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $0010011010 \preccurlyeq x$

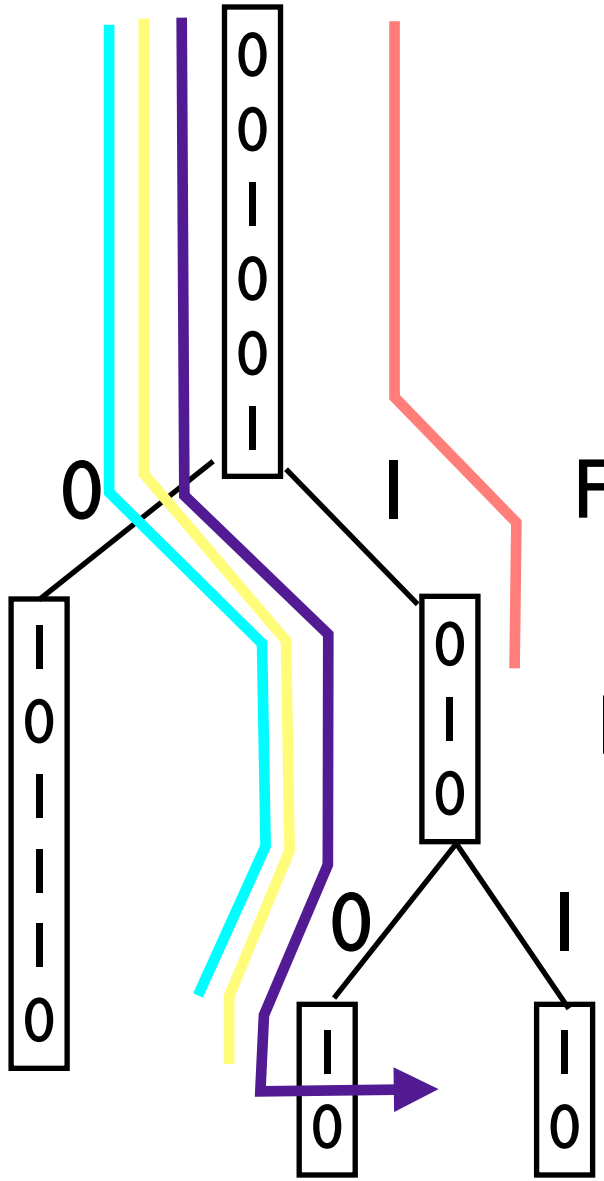
$\Rightarrow (l..r) = (10..13) \Rightarrow f = 12$

Note: binary search would move to (8..13)!

Function returns $00100110100 10 \not\preccurlyeq x$

$\Rightarrow (l..r) = (10..12) \Rightarrow f = 11$

x = 00100110100 11



$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $0010011010 \preccurlyeq x$

$\Rightarrow (l..r) = (10..13) \Rightarrow f = 12$

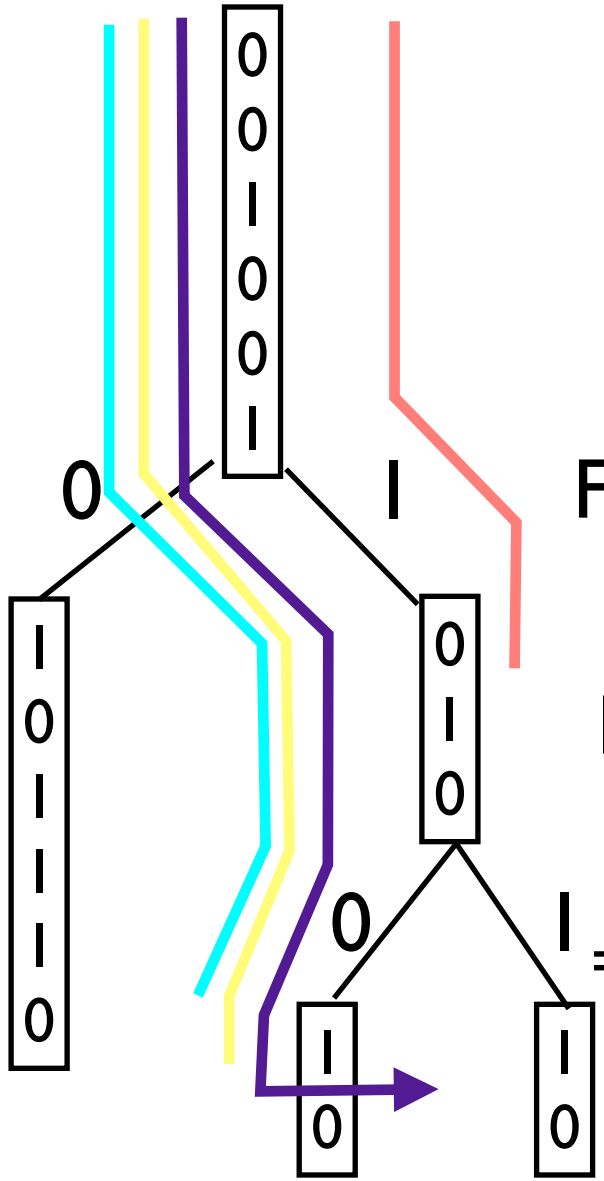
Note: binary search would move to (8..13)!

Function returns $00100110100 10 \not\preccurlyeq x$

$\Rightarrow (l..r) = (10..12) \Rightarrow f = 11$

Function returns a random string s and either $s \not\preccurlyeq x$ or $|s| \leq 10$

$x =$ 00100110100 11



$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $0010011010 \preccurlyeq x$

Note: binary search would move to (8..13)!

$\Rightarrow (l..r) = (10..13) \Rightarrow f = 12$

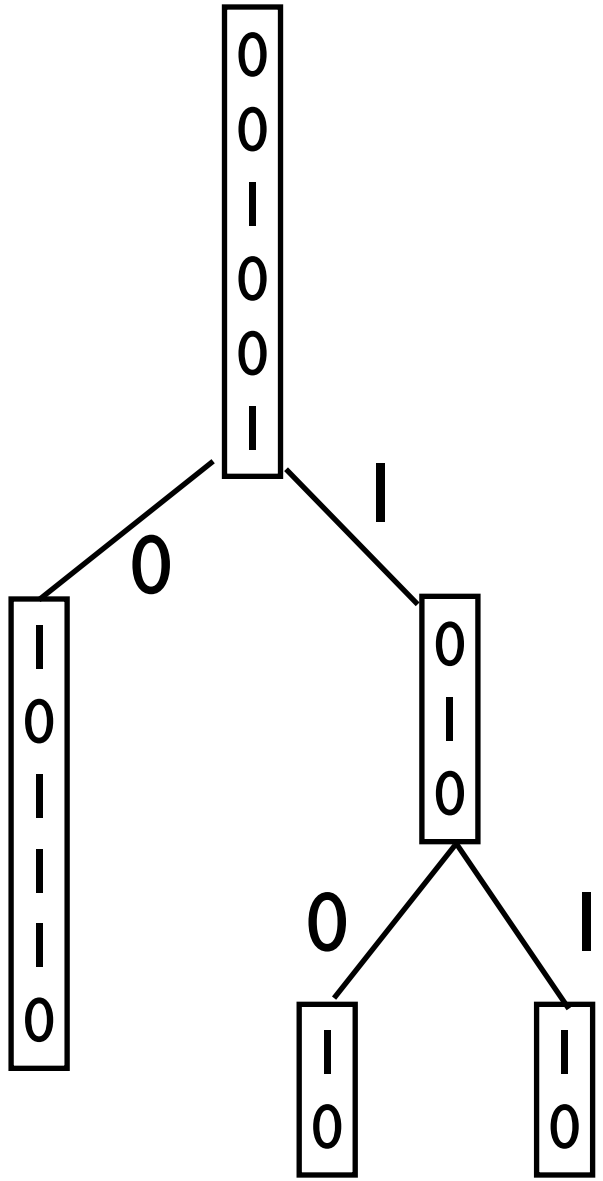
Function returns $00100110100 10 \not\preccurlyeq x$

$\Rightarrow (l..r) = (10..12) \Rightarrow f = 11$

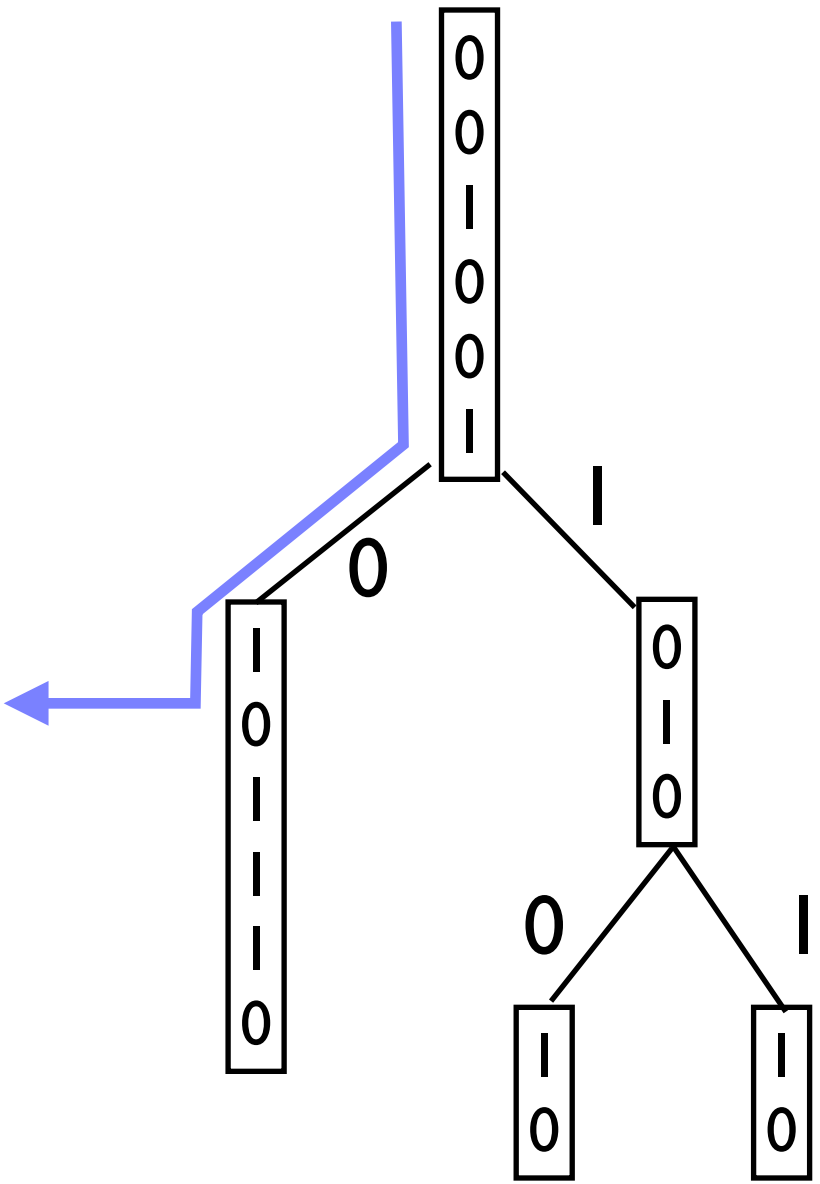
Function returns a random string s and either $s \not\preccurlyeq x$ or $|s| \leq 10$

$\Rightarrow x$ exits at node with name x[0..11)

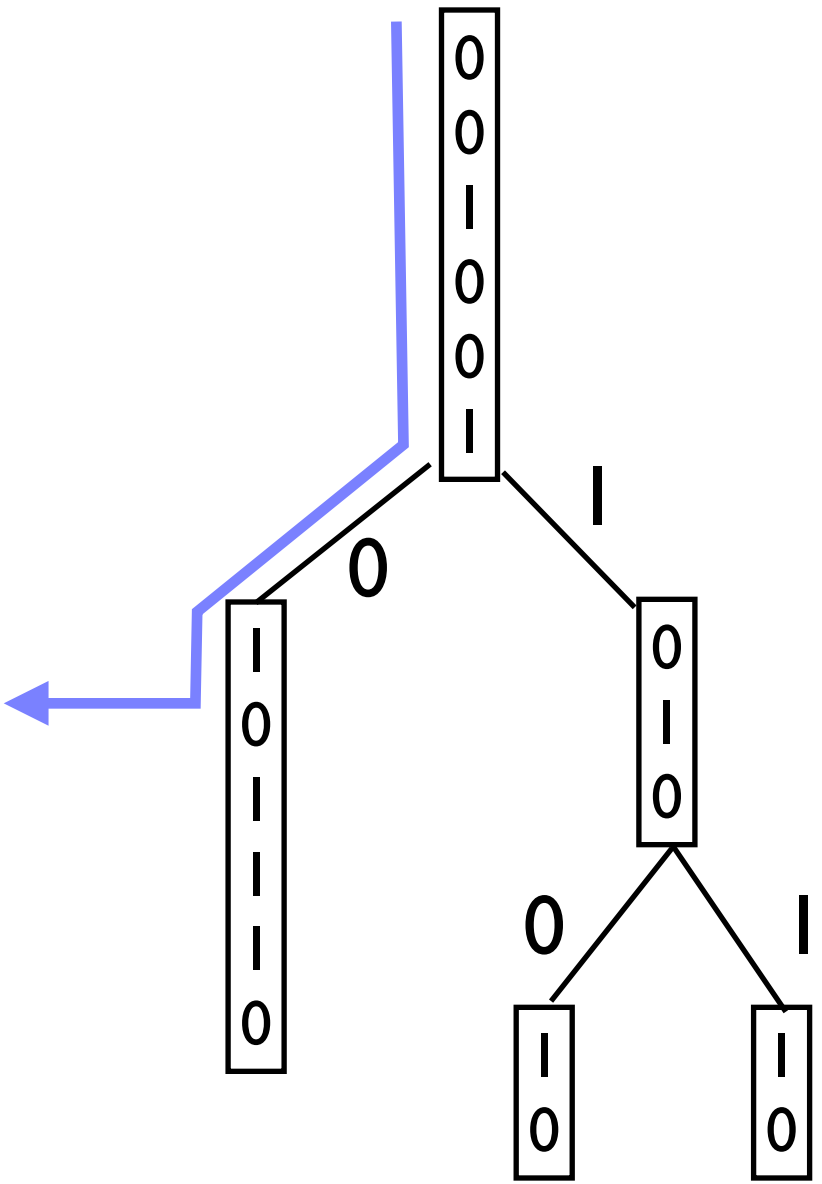
x = 00100110100 11



x = 001001010011 0

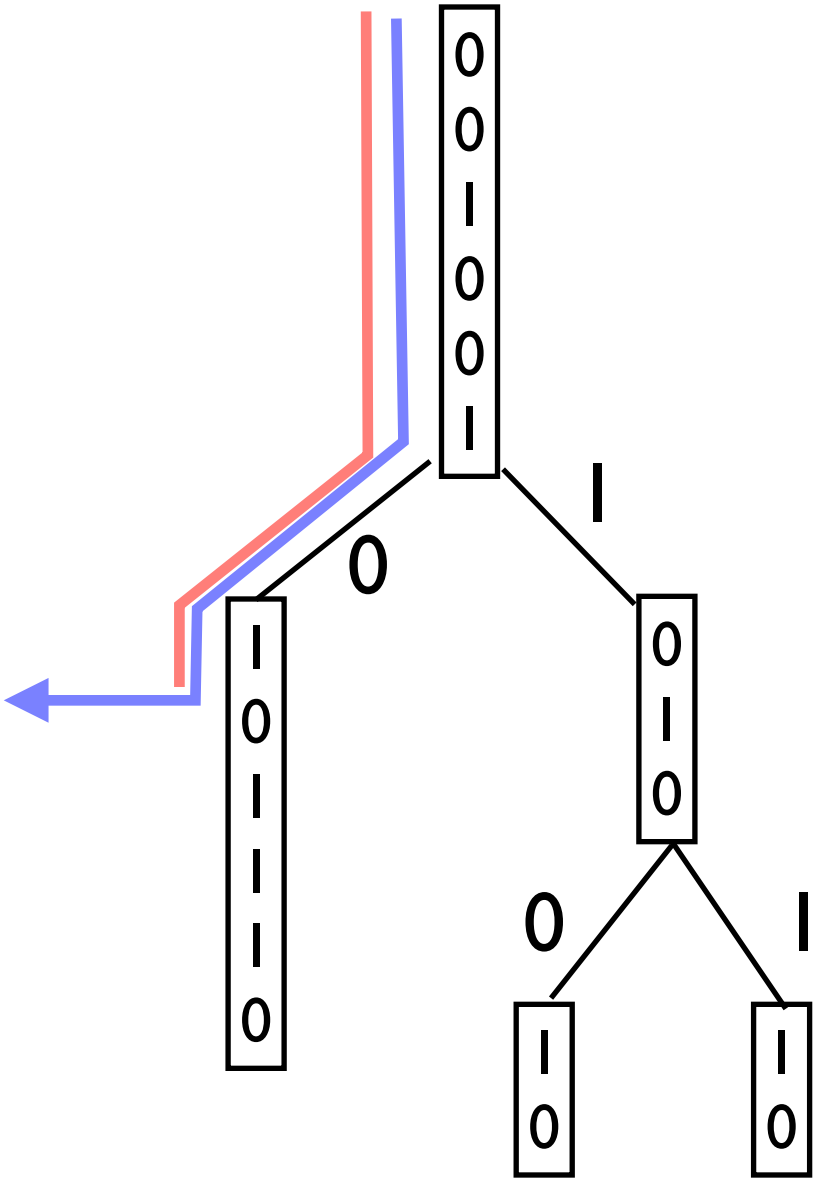


x = 001001010011 0



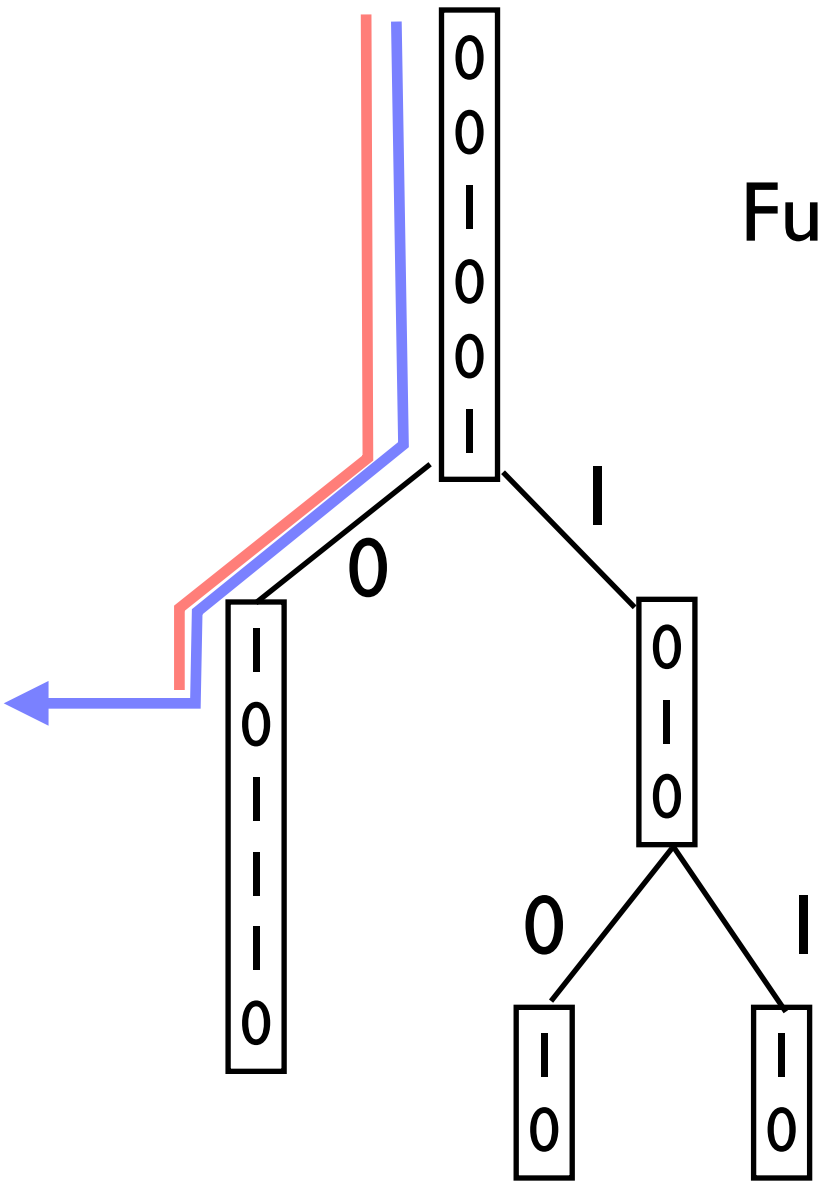
$(l..r) = (0..13) \Rightarrow f = 8$

x = 00100101001110



$(l..r) = (0..13) \Rightarrow f = 8$

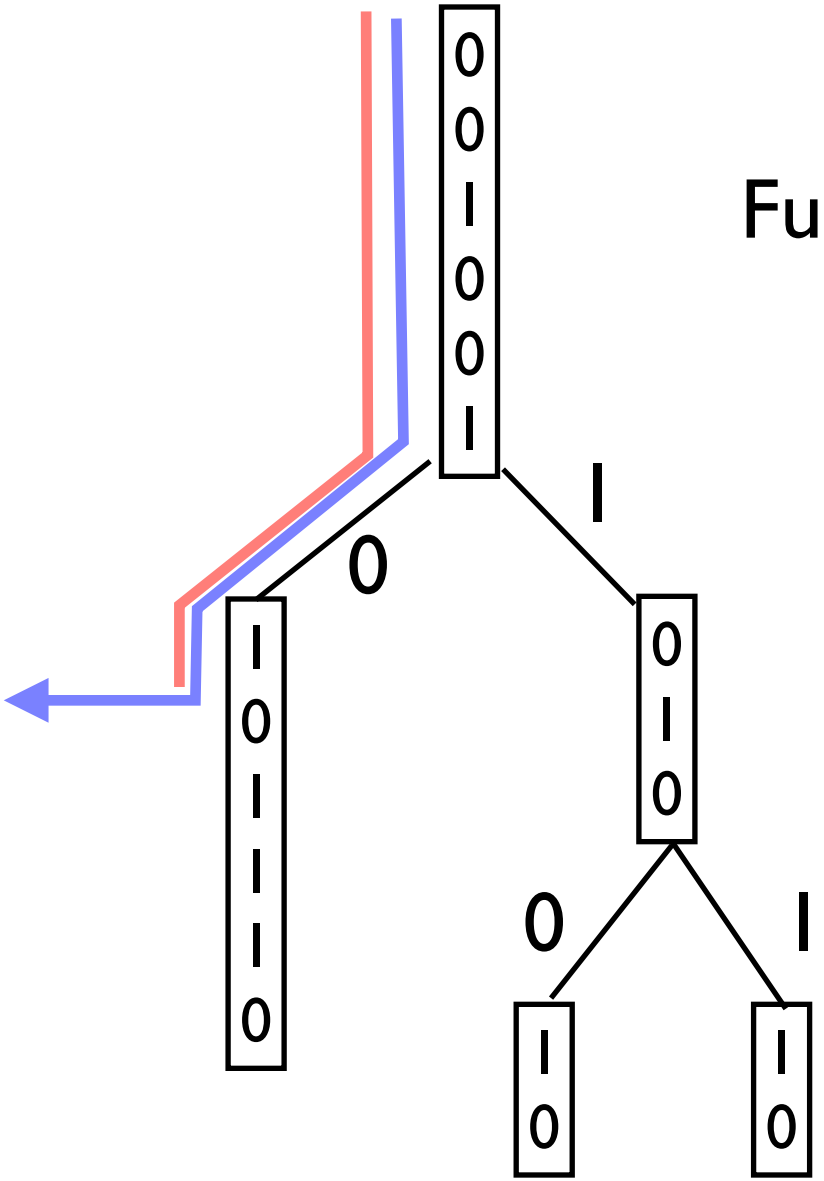
$x = 00100101\,00110$



$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $00100101011110 \nleq x$

$x = 00100101\,00110$

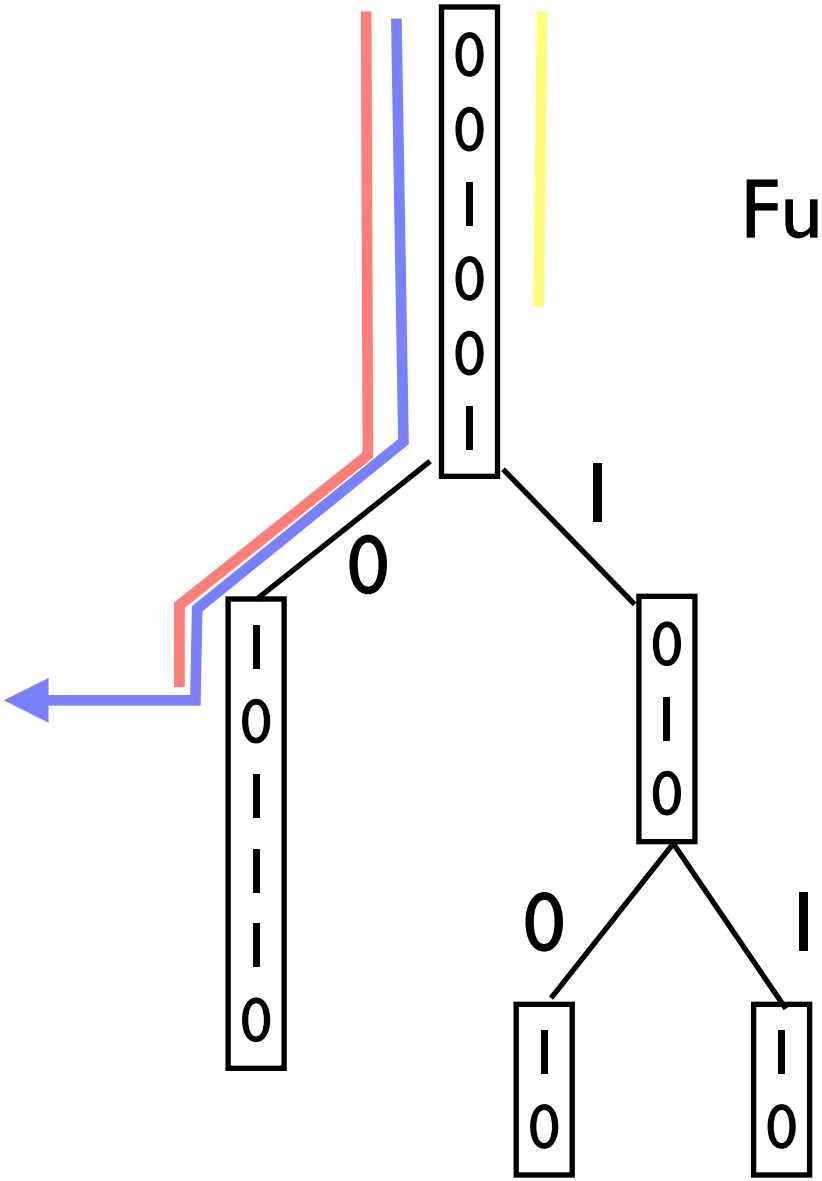


$(l..r) = (0..13) \Rightarrow f = 8$

Function returns 00100101011110 $\not< x$

$\Rightarrow (l..r) = (0..8) \Rightarrow f = 4$

x = 00100101 00110

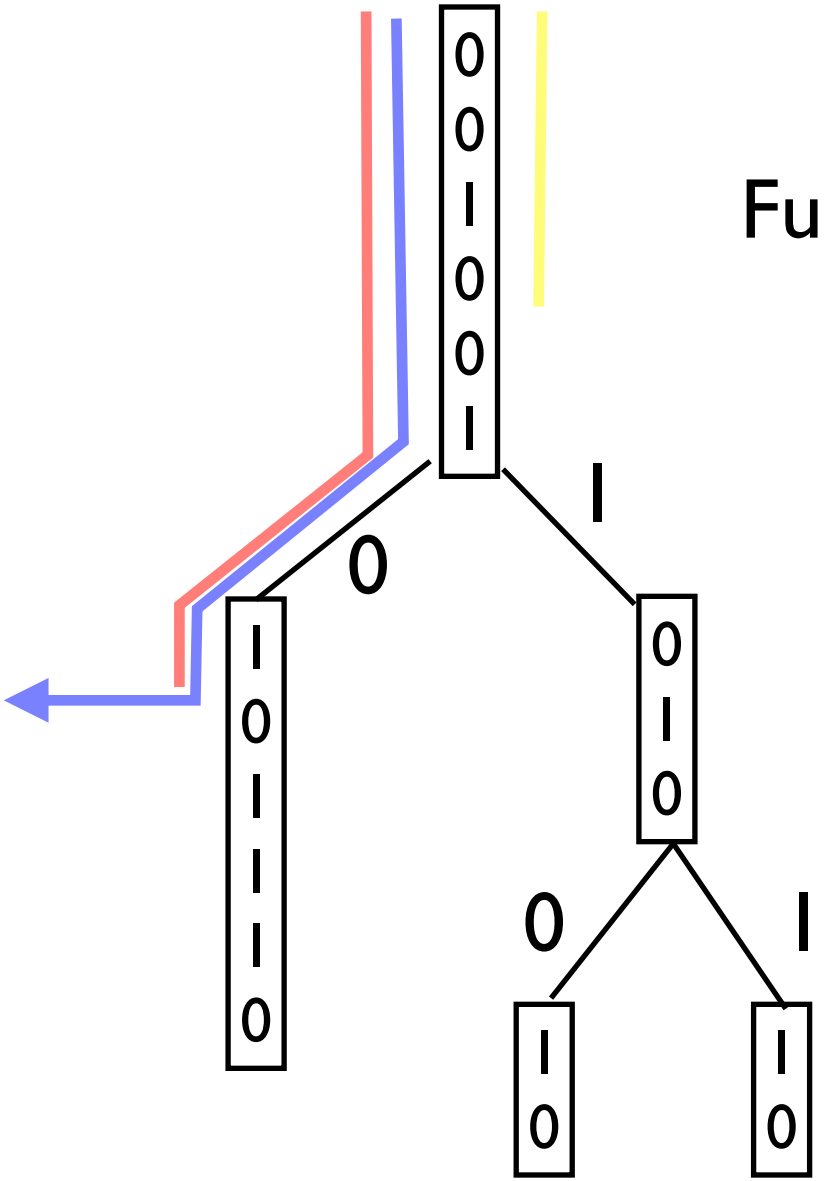


$(l..r) = (0..13) \Rightarrow f = 8$

Function returns 00100101011110 $\nleq x$

$\Rightarrow (l..r) = (0..8) \Rightarrow f = 4$

x = 00100101 00110



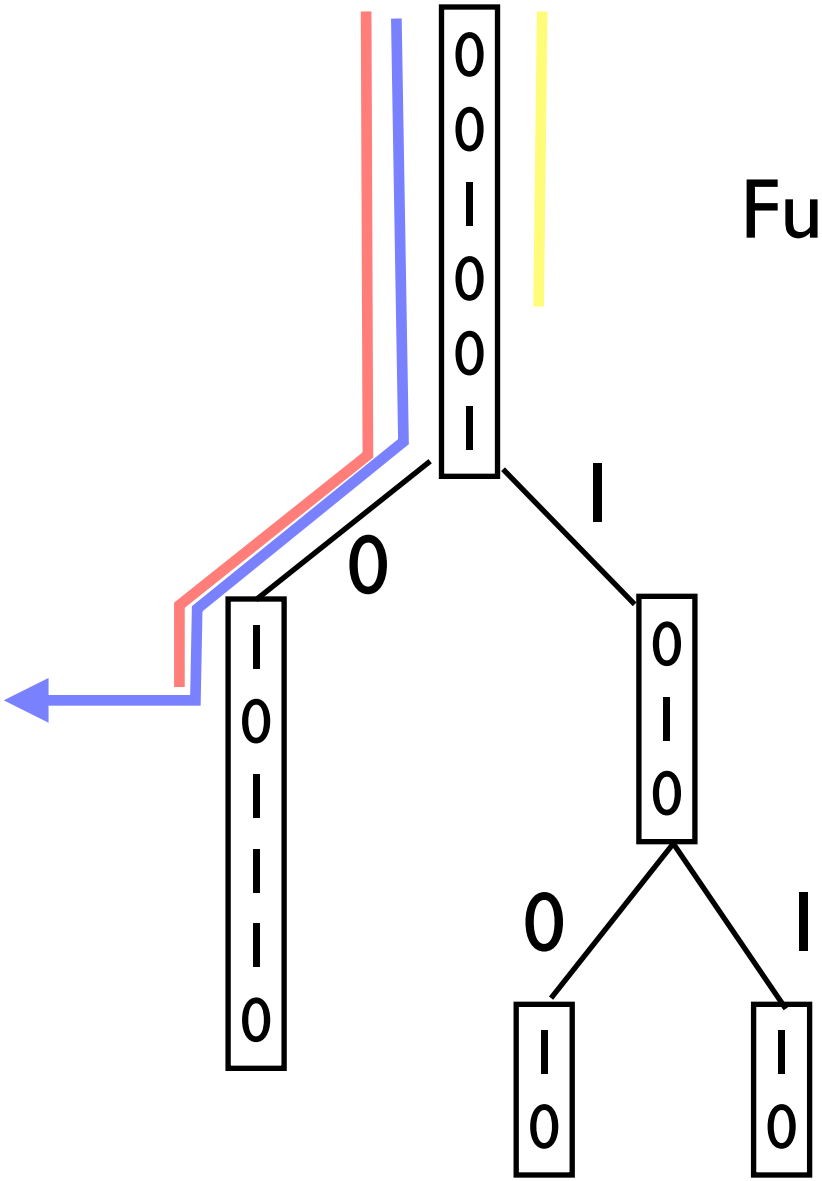
$(l..r) = (0..13) \Rightarrow f = 8$

Function returns 00100101011110 $\not\preccurlyeq x$

$\Rightarrow (l..r) = (0..8) \Rightarrow f = 4$

Function returns 001001 $\preccurlyeq x$

x = 00100101 00110



$(l..r) = (0..13) \Rightarrow f = 8$

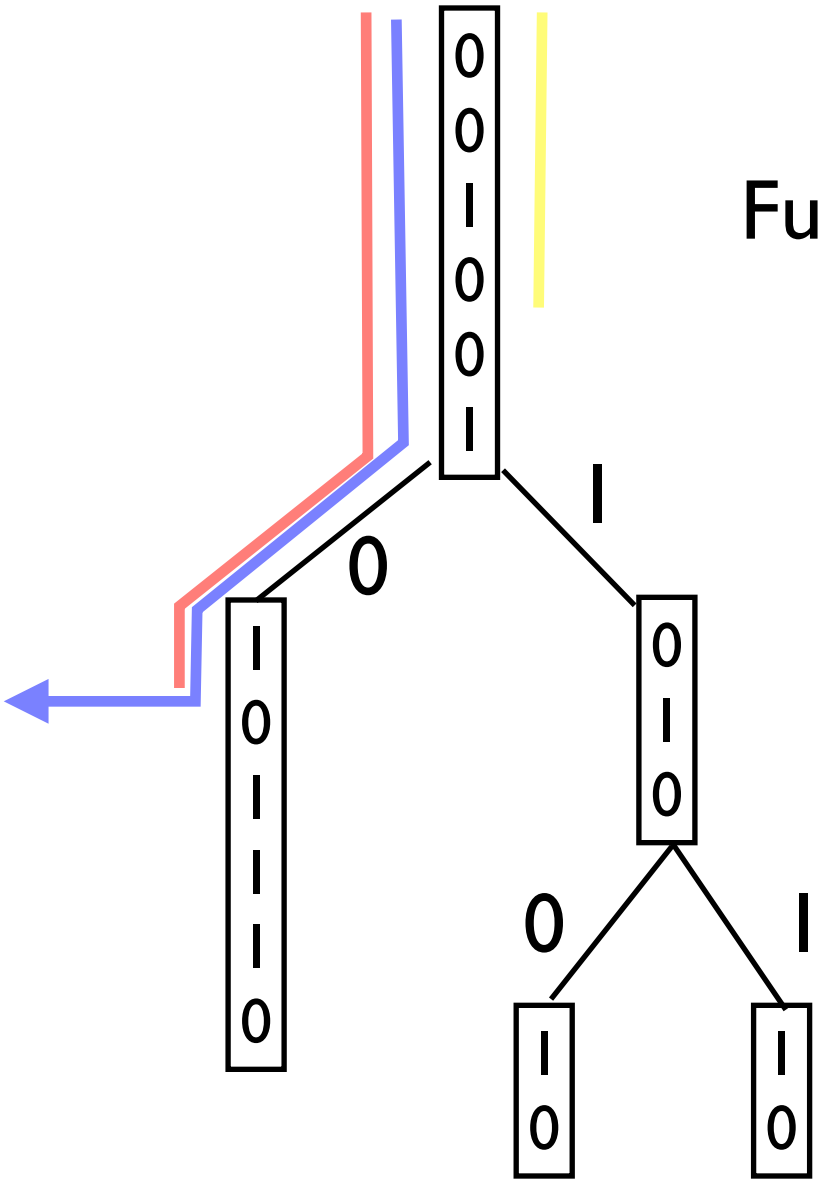
Function returns 00100101011110 $\nleq x$

$\Rightarrow (l..r) = (0..8) \Rightarrow f = 4$

Function returns 001001 $\leqslant x$

$\Rightarrow (l..r) = (6..8) \Rightarrow f = 7$

x = 00100101 00110



$(l..r) = (0..13) \Rightarrow f = 8$

Function returns 00100101011110 $\nleqslant x$

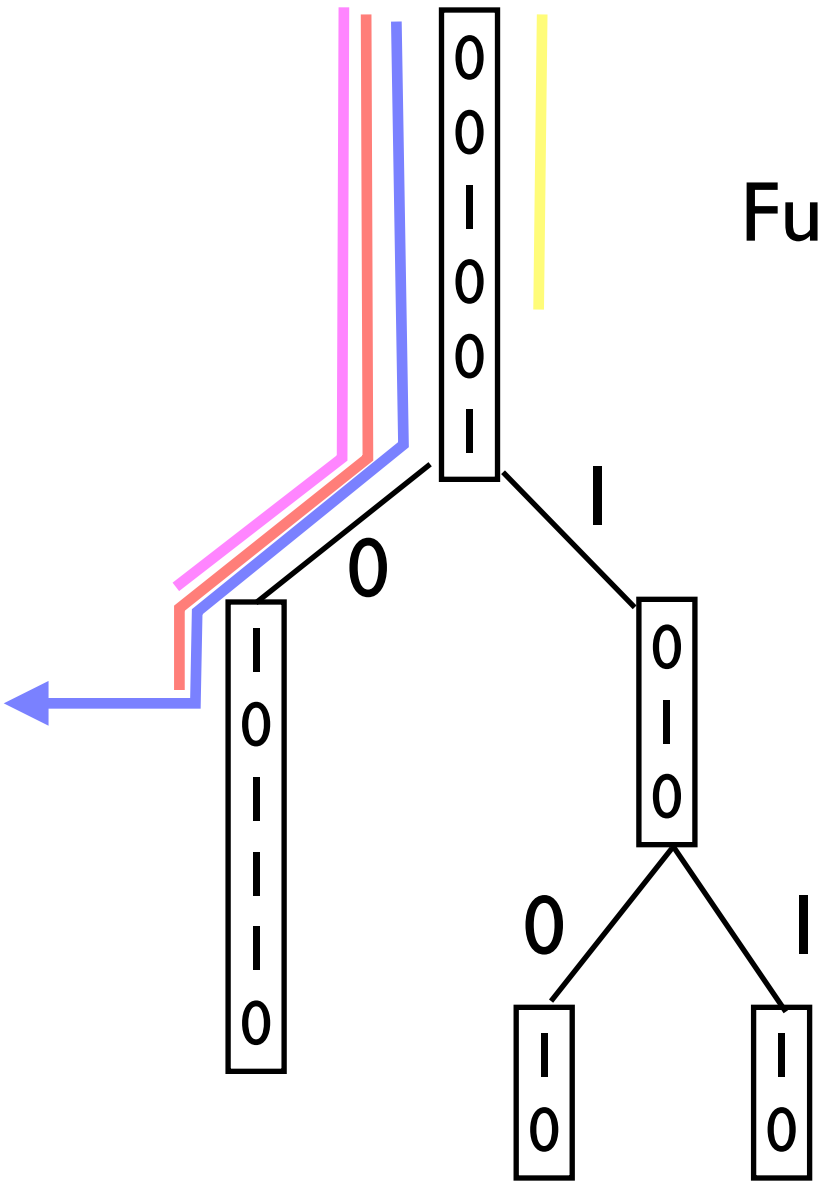
$\Rightarrow (l..r) = (0..8) \Rightarrow f = 4$

Function returns 001001 $\leqslant x$

$\Rightarrow (l..r) = (6..8) \Rightarrow f = 7$

Note: binary search would choose 6!

x = 00100101 00110



$(l..r) = (0..13) \Rightarrow f = 8$

Function returns 00100101011110 $\nleq x$

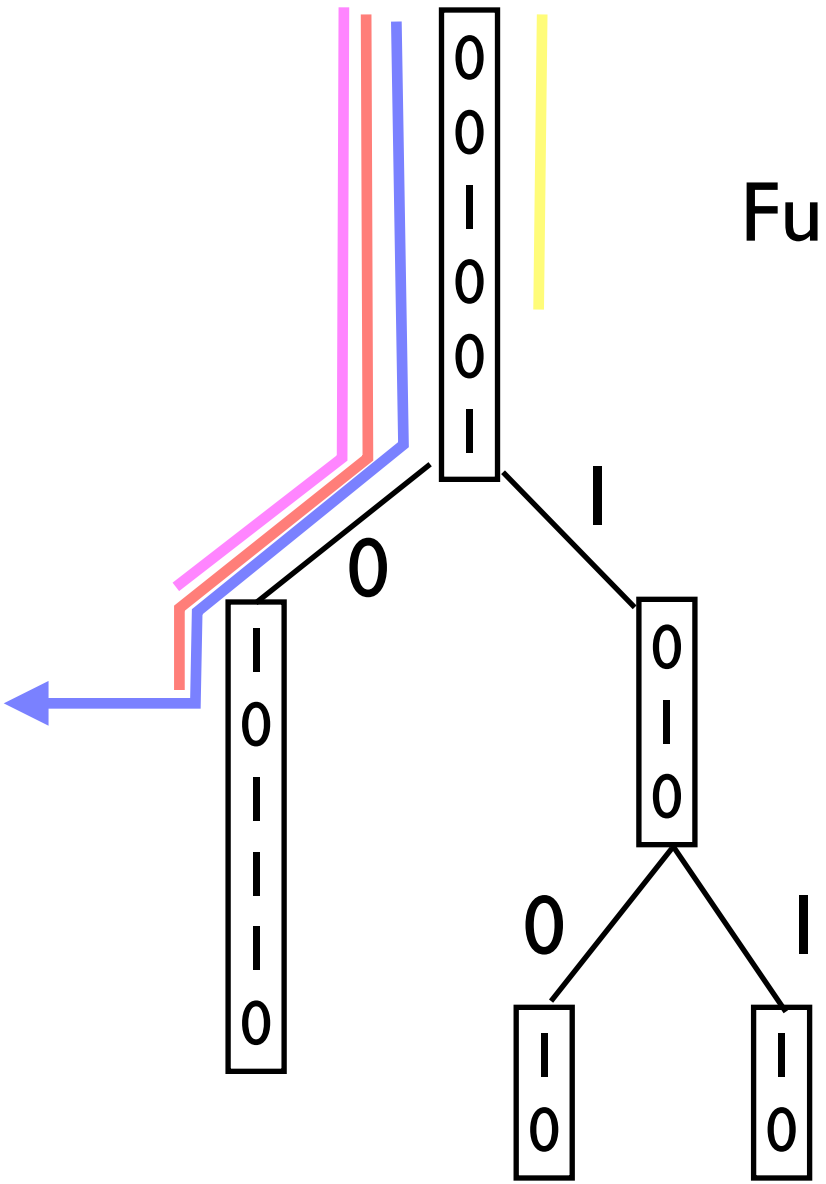
$\Rightarrow (l..r) = (0..8) \Rightarrow f = 4$

Function returns 001001 $\leqslant x$

$\Rightarrow (l..r) = (6..8) \Rightarrow f = 7$

Note: binary search would choose 6!

x = 00100101100110



$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $00100101011110 \not\preccurlyeq x$

$\Rightarrow (l..r) = (0..8) \Rightarrow f = 4$

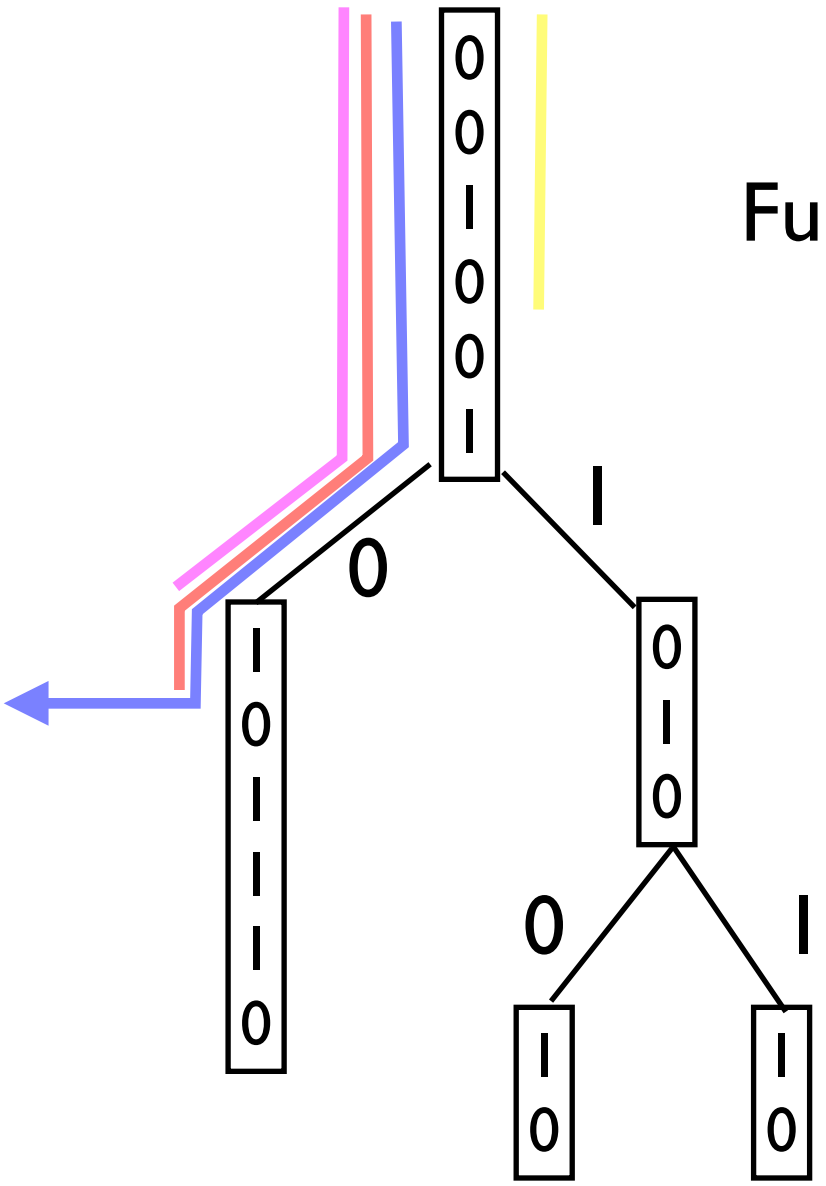
Function returns $001001 \preccurlyeq x$

$\Rightarrow (l..r) = (6..8) \Rightarrow f = 7$

Function returns a random string

Note: binary search would choose 6!

$x = 00100101100110$



$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $00100101011110 \nleq x$

$\Rightarrow (l..r) = (0..8) \Rightarrow f = 4$

Function returns $001001 \leqslant x$

$\Rightarrow (l..r) = (6..8) \Rightarrow f = 7$

Function returns a random string

Note: binary search would choose 6!

$\Rightarrow x$ exits at node x[0..7)

x = 00100101100110

We Don’t Keep The Paths!

We Don’t Keep The Paths!

- The main point of all this is that we will *not* map handles to extents, but handles to log *w* bits *signatures* of extents

We Don’t Keep The Paths!

- The main point of all this is that we will *not* map handles to extents, but handles to log *w* bits *signatures* of extents
- The space occupancy per node is now log *w*!

We Don't Keep The Paths!

- The main point of all this is that we will *not* map handles to extents, but handles to log *w* bits *signatures* of extents
- The space occupancy per node is now log *w*!
- Of course, we might make mistakes; and we don't know in which direction we exit

We Don’t Keep The Paths!

- The main point of all this is that we will *not* map handles to extents, but handles to log *w* bits *signatures* of extents
- The space occupancy per node is now log *w*!
- Of course, we might make mistakes; and we don’t know in which direction we exit
- Moreover, the only information we get about the trie structure is the name of the exit node—not a bucket!

We Don't Keep The Paths!

- The main point of all this is that we will *not* map handles to extents, but handles to log *w* bits *signatures* of extents
- The space occupancy per node is now log *w*!
- Of course, we might make mistakes; and we don't know in which direction we exit
- Moreover, the only information we get about the trie structure is the name of the exit node—not a bucket!
- So how can we use the z-fast trie built on the delimiter set *D* as a distributor?

From Names to Ranks

From Names to Ranks

- We build a set P containing, for each extent e of an internal node, the strings $e0$, $e1$, $e1^+$ (the successor of $e1$)

From Names to Ranks

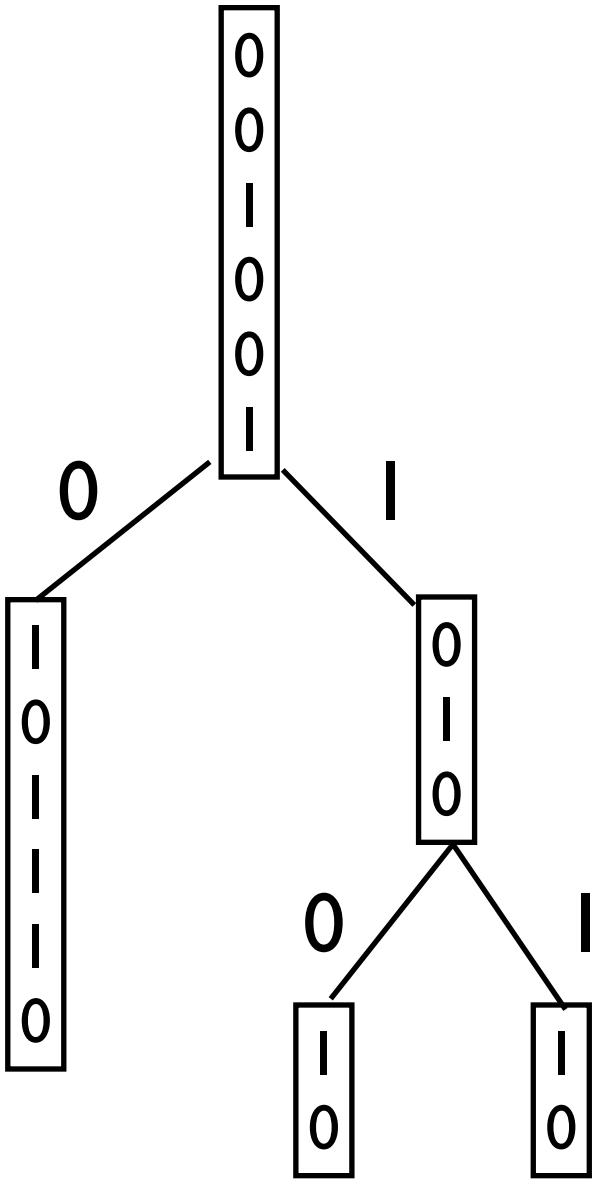
- We build a set P containing, for each extent e of an internal node, the strings $e0$, $e1$, $e1^+$ (the successor of $e1$)
- We use an LCP monotone minimal perfect hash f to map P onto a bit vector containing ones in the positions corresponding to names of leaves

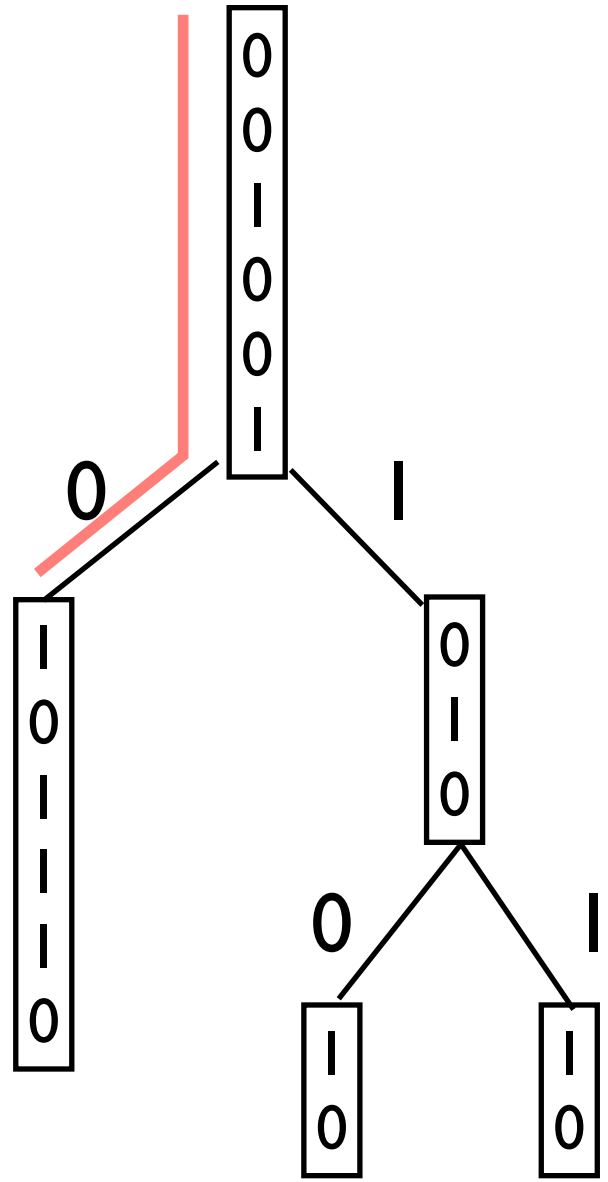
From Names to Ranks

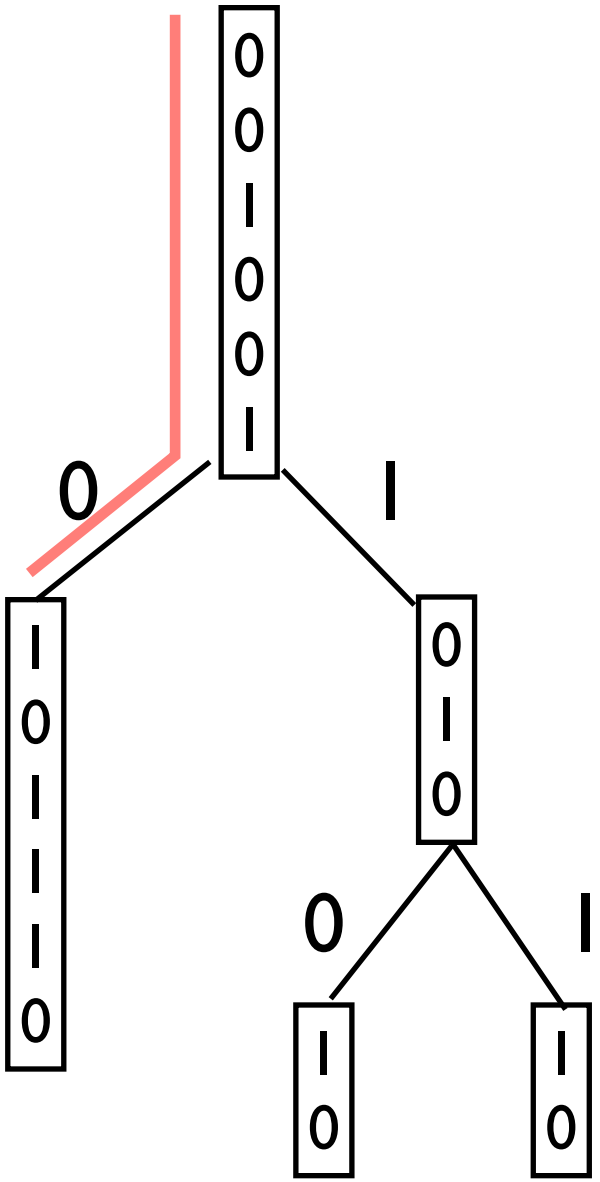
- We build a set P containing, for each extent e of an internal node, the strings $e0$, $e1$, $e1^+$ (the successor of $e1$)
- We use an LCP monotone minimal perfect hash f to map P onto a bit vector containing ones in the positions corresponding to names of leaves
- If the name of an exit node is of the form $e0$, by ranking $f(e0)$ and $f(e1)$ we have the possible indices of the buckets corresponding to x! (analogously for $e1$)

From Names to Ranks

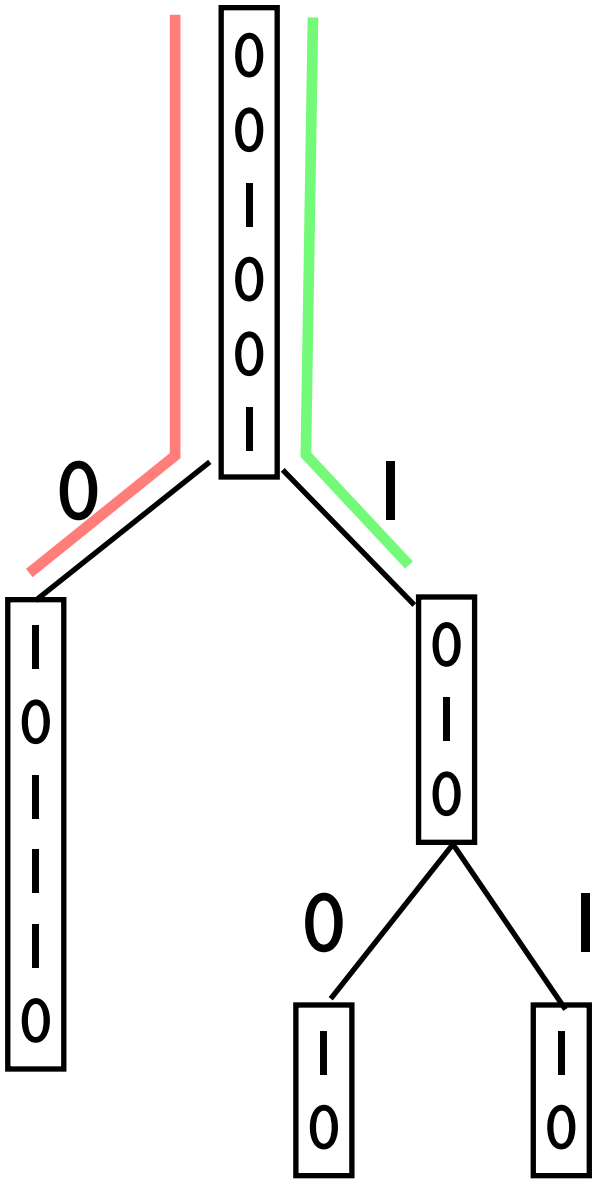
- We build a set P containing, for each extent e of an internal node, the strings $e0$, $e1$, $e1^+$ (the successor of $e1$)
- We use an LCP monotone minimal perfect hash f to map P onto a bit vector containing ones in the positions corresponding to names of leaves
- If the name of an exit node is of the form $e0$, by ranking $f(e0)$ and $f(e1)$ we have the possible indices of the buckets corresponding to x! (analogously for $e1$)


0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
0
1
0
1
0

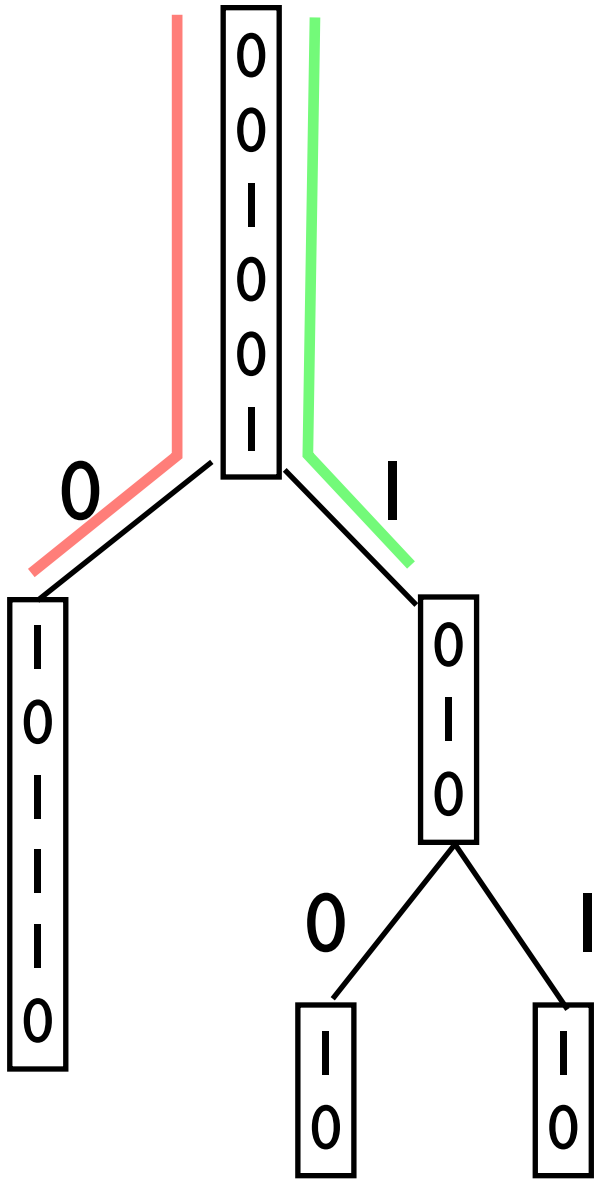

0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
0
1
0
1
0



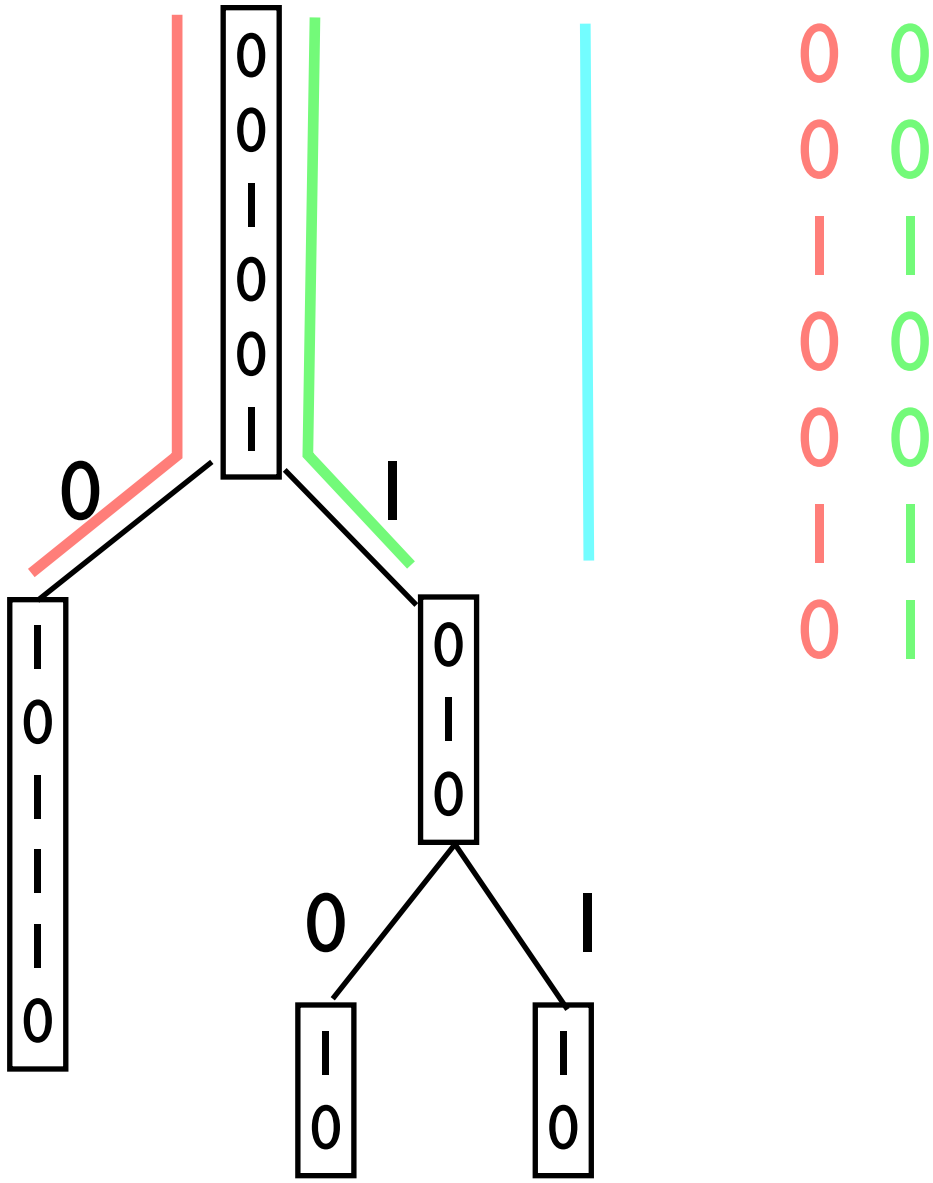
0 0 I 0 0 I 0

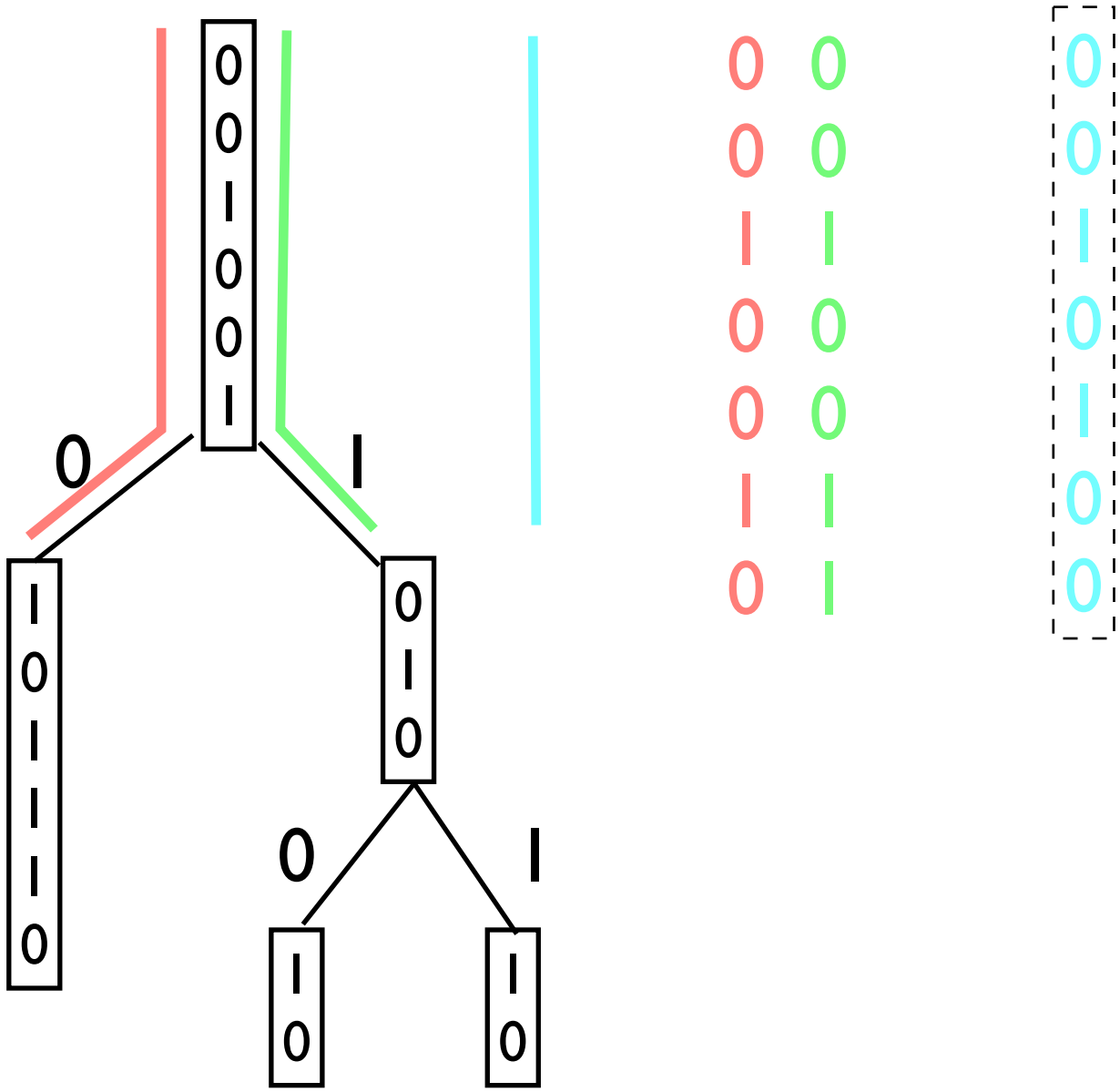


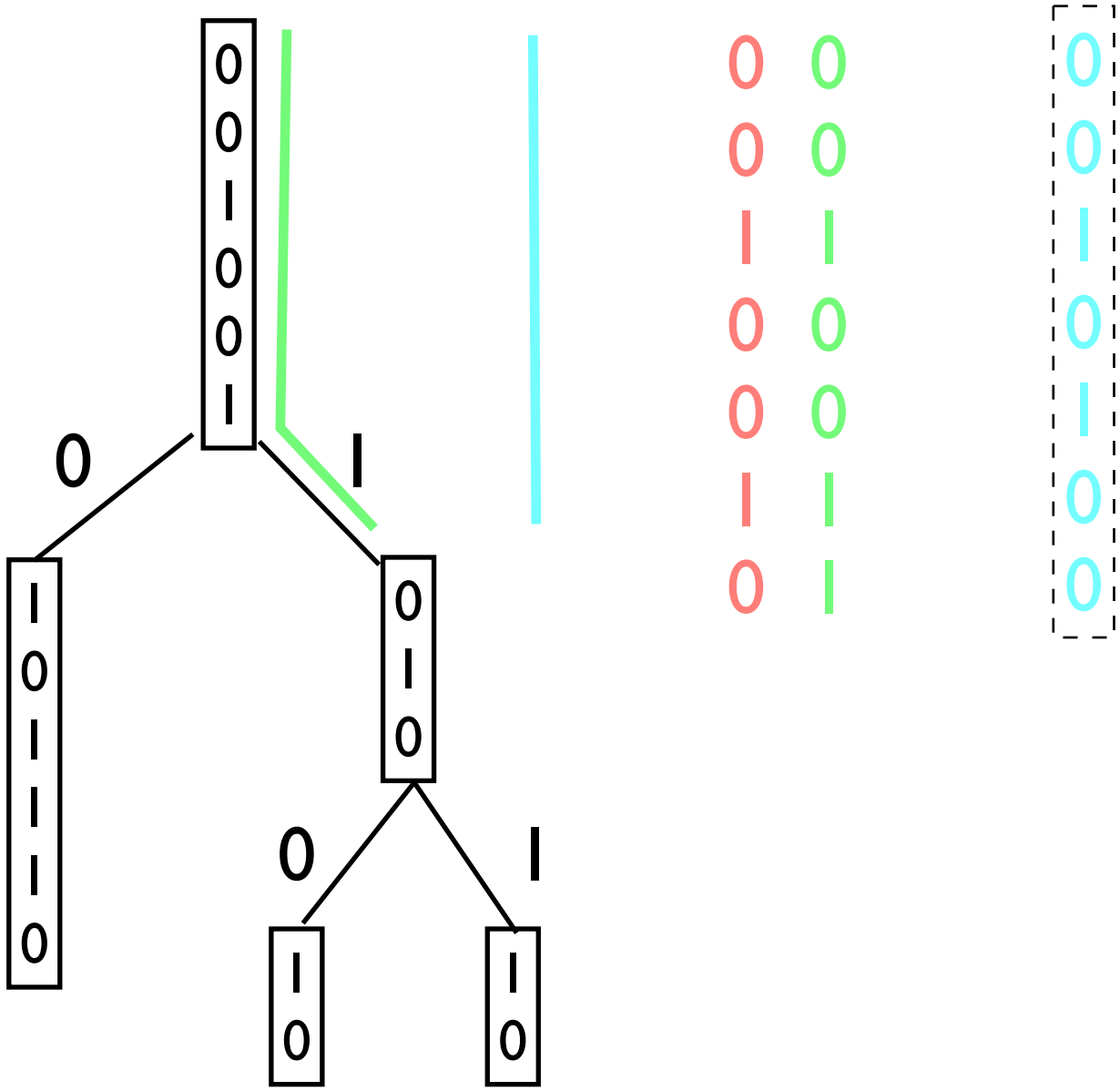
0
0
1
0
0
1
0

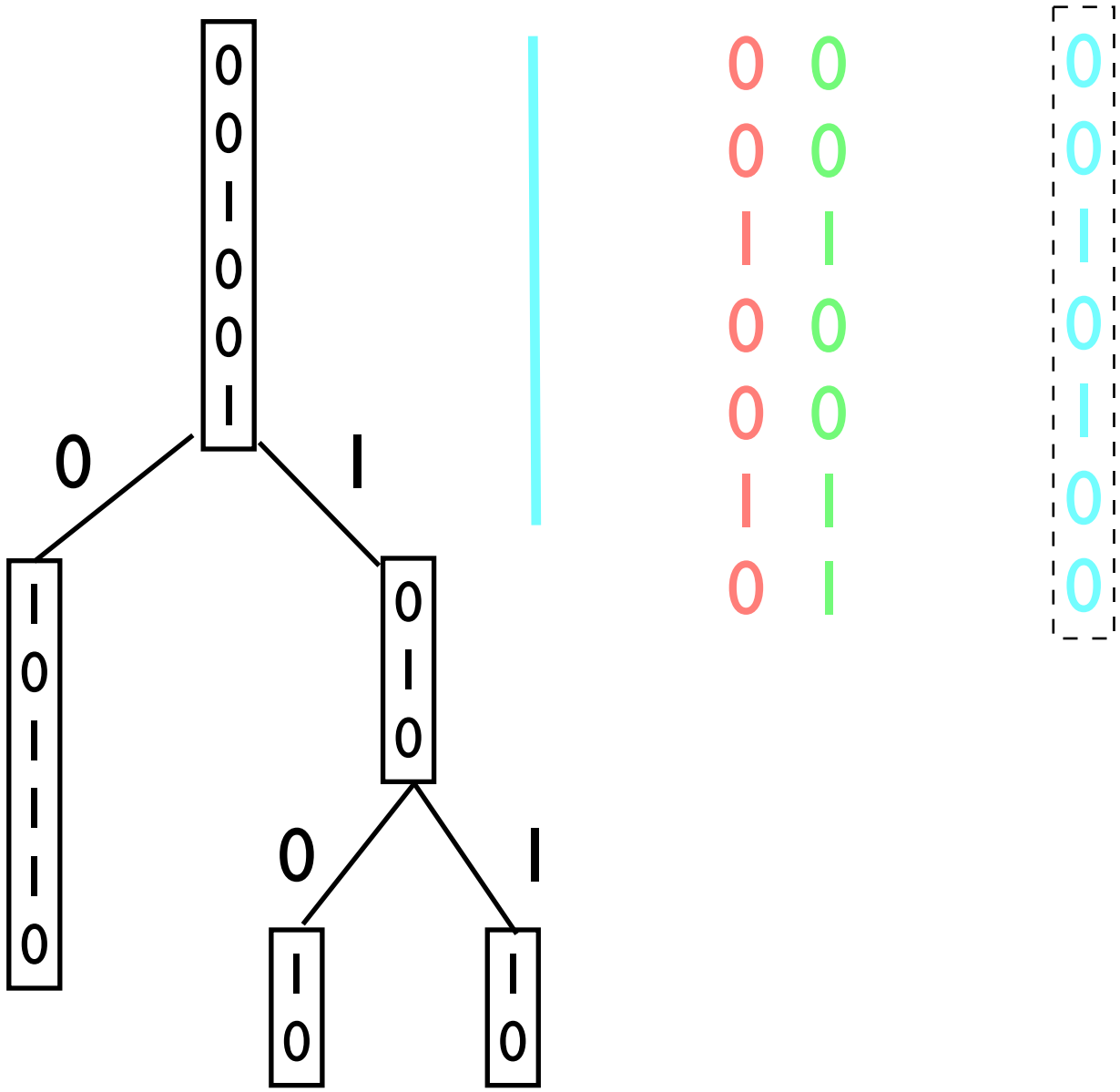


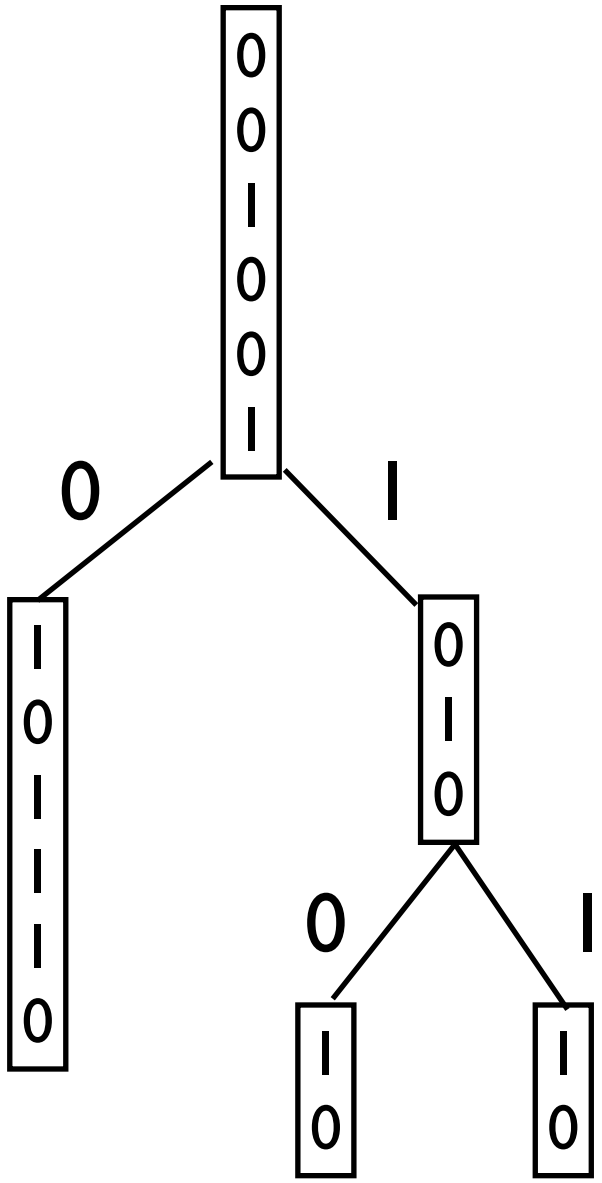
0 0
0 0
1 1
0 0
0 0
1 1
0 1

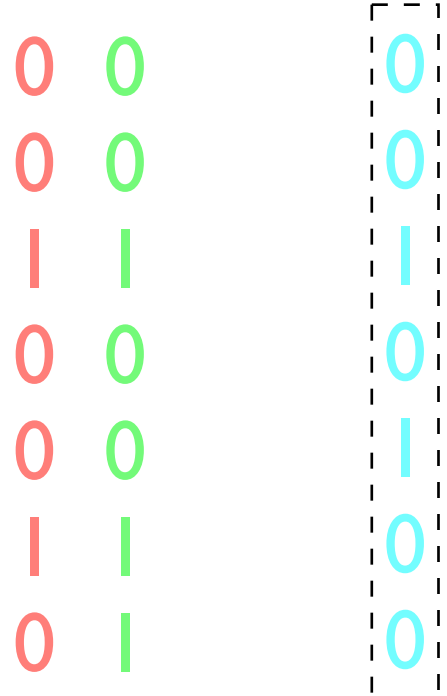

0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
0
1
0
0
1
0
1
0
0
1
0
0
0
1
0
0
1
0
0
1
0
1
0
1

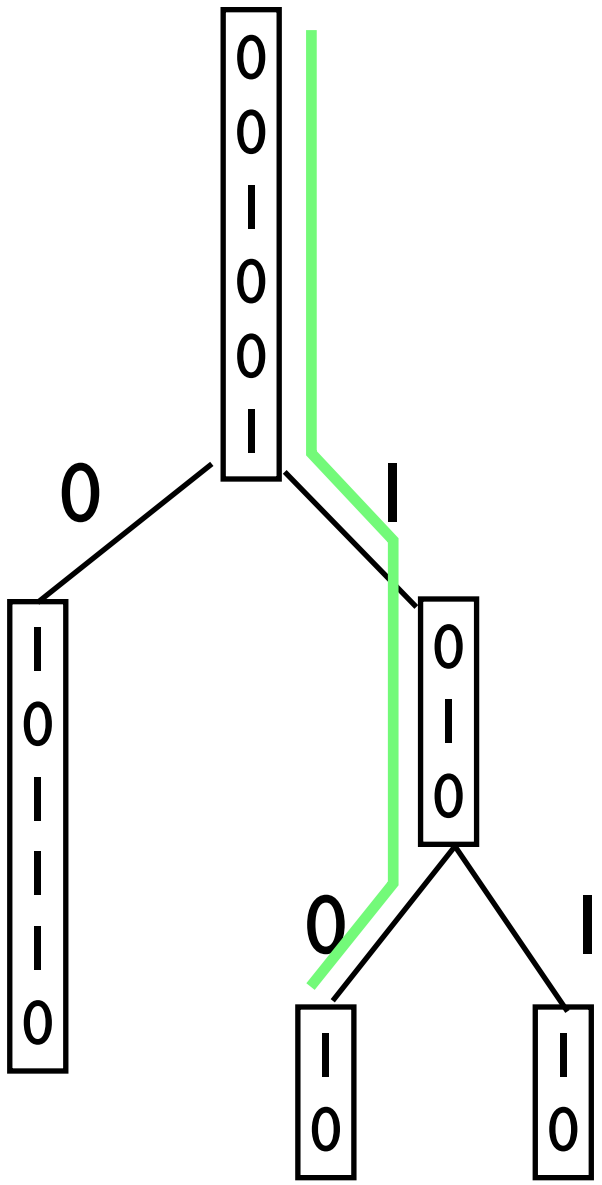

0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
1
0
1
0
0
0
1
0
0
0
1
0
1
0
0
1
0
0
1
0
0
1
0
1
0
0

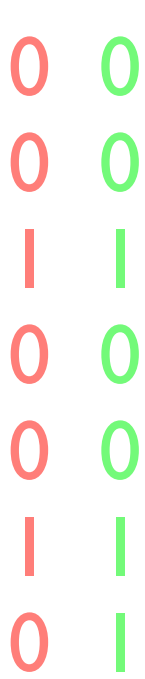

0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0


0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0

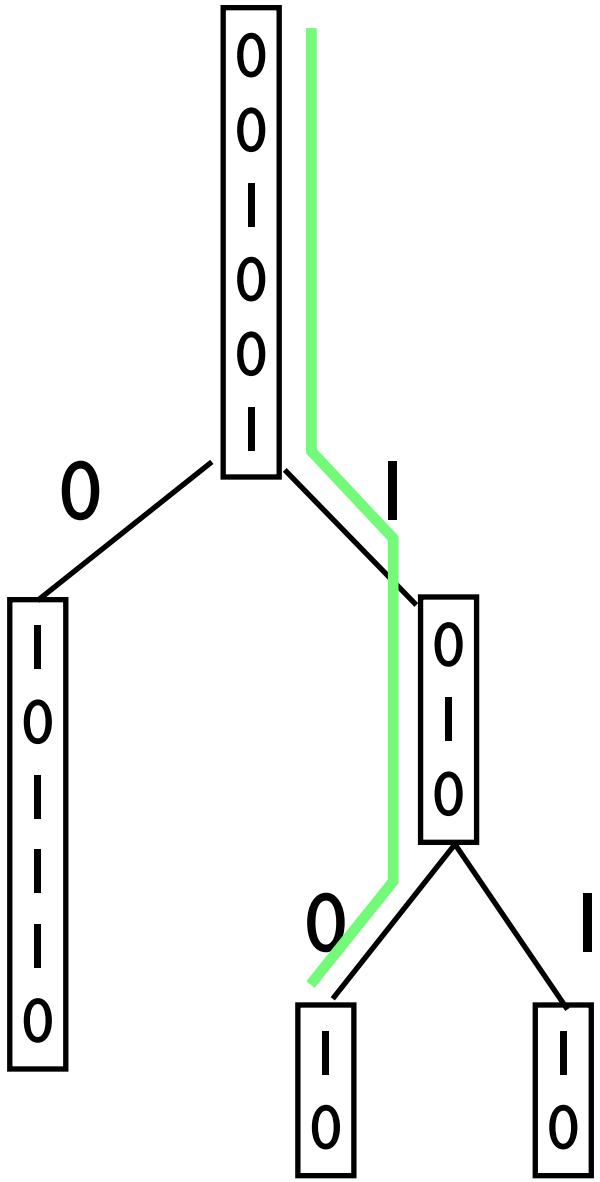

0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0

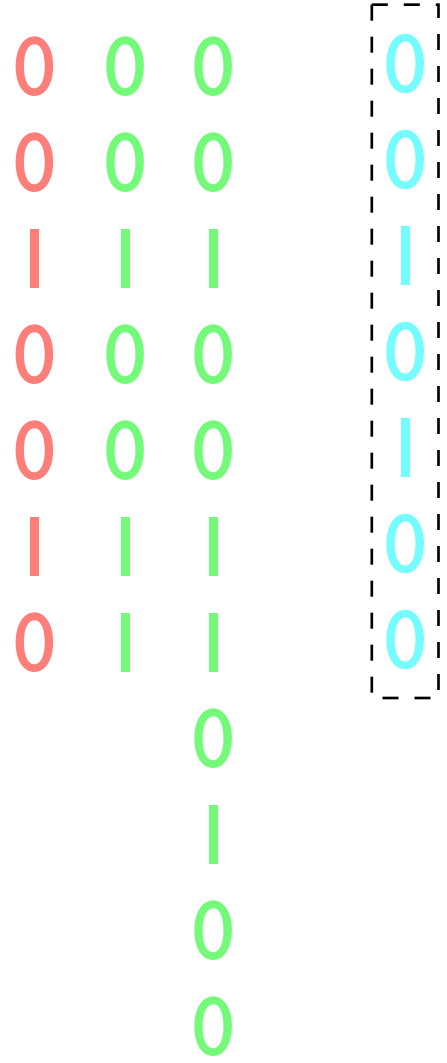

0 0
0 0
1 1
0 0
0 0
1 1
0 1
0 0 1 0 1 0 0

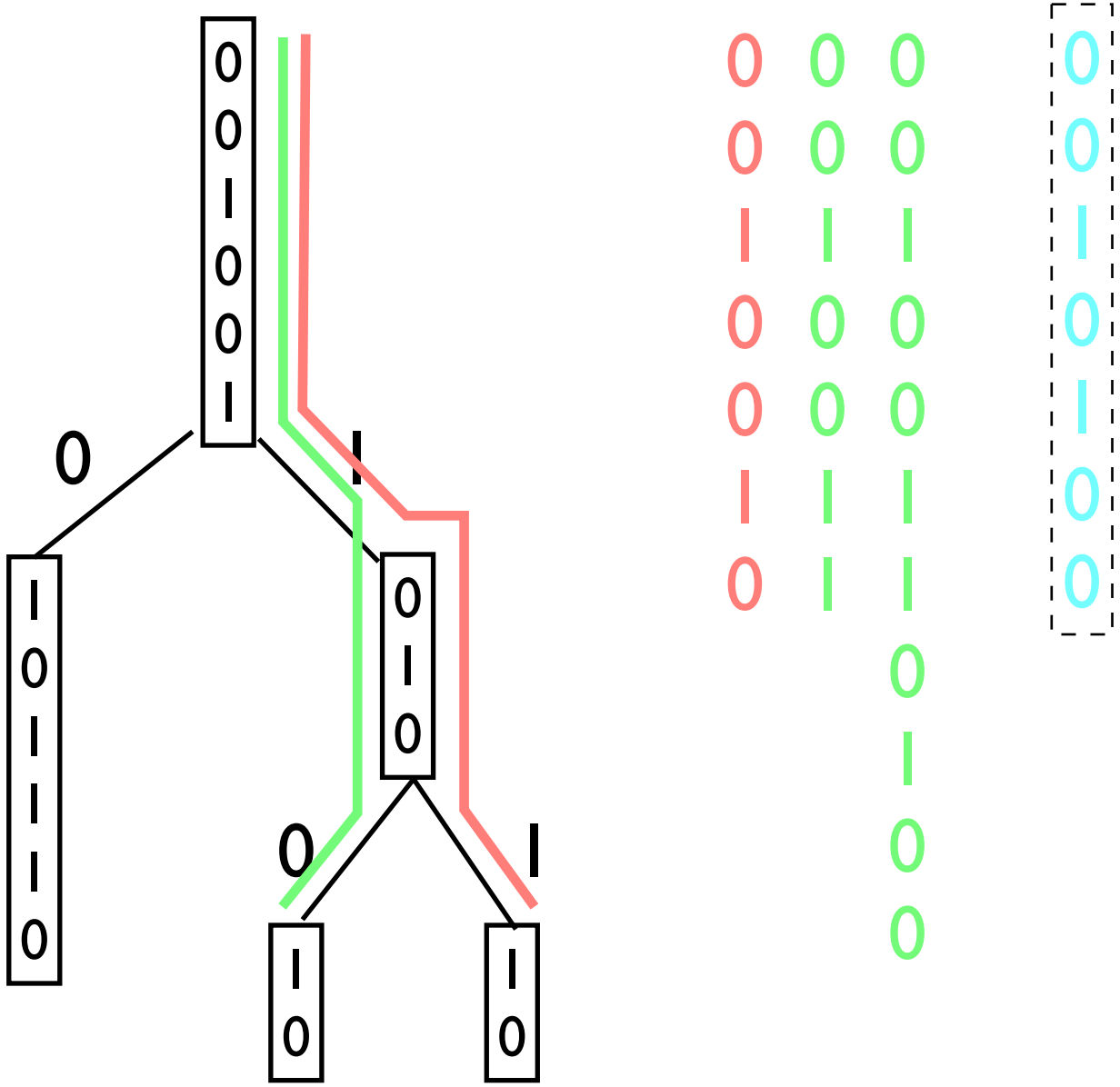

0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0

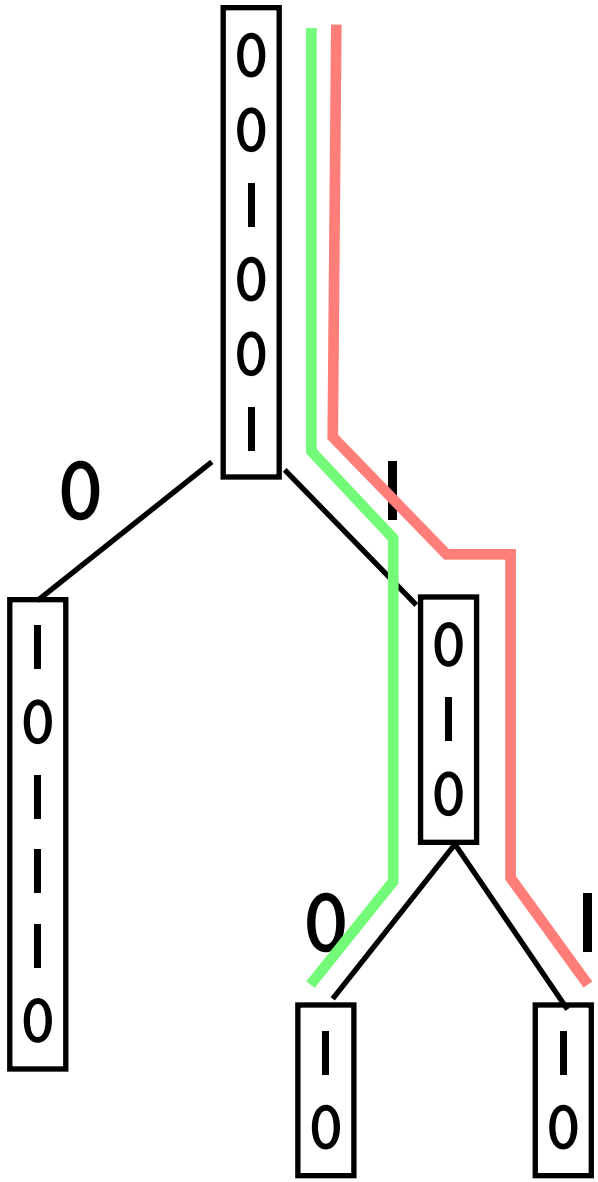

0 0
0 0
1 1
0 0
0 0
1 1
0 1

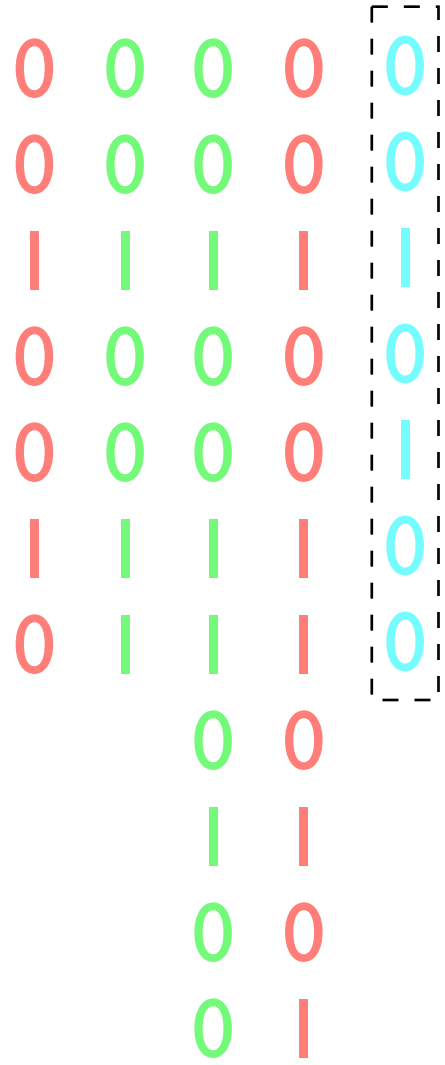

0
0
1
0
1
0
0

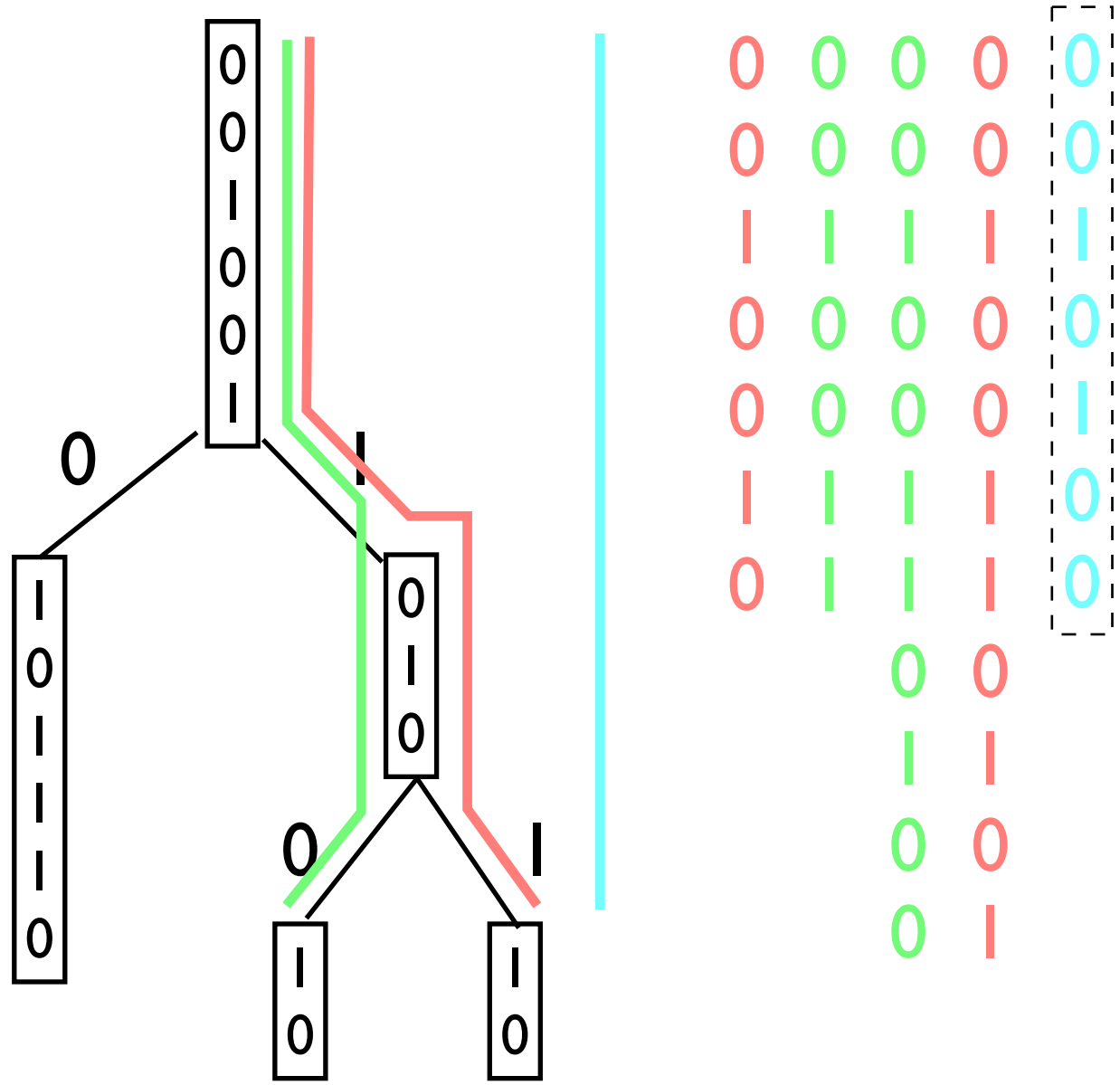

0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
1
0
1
0

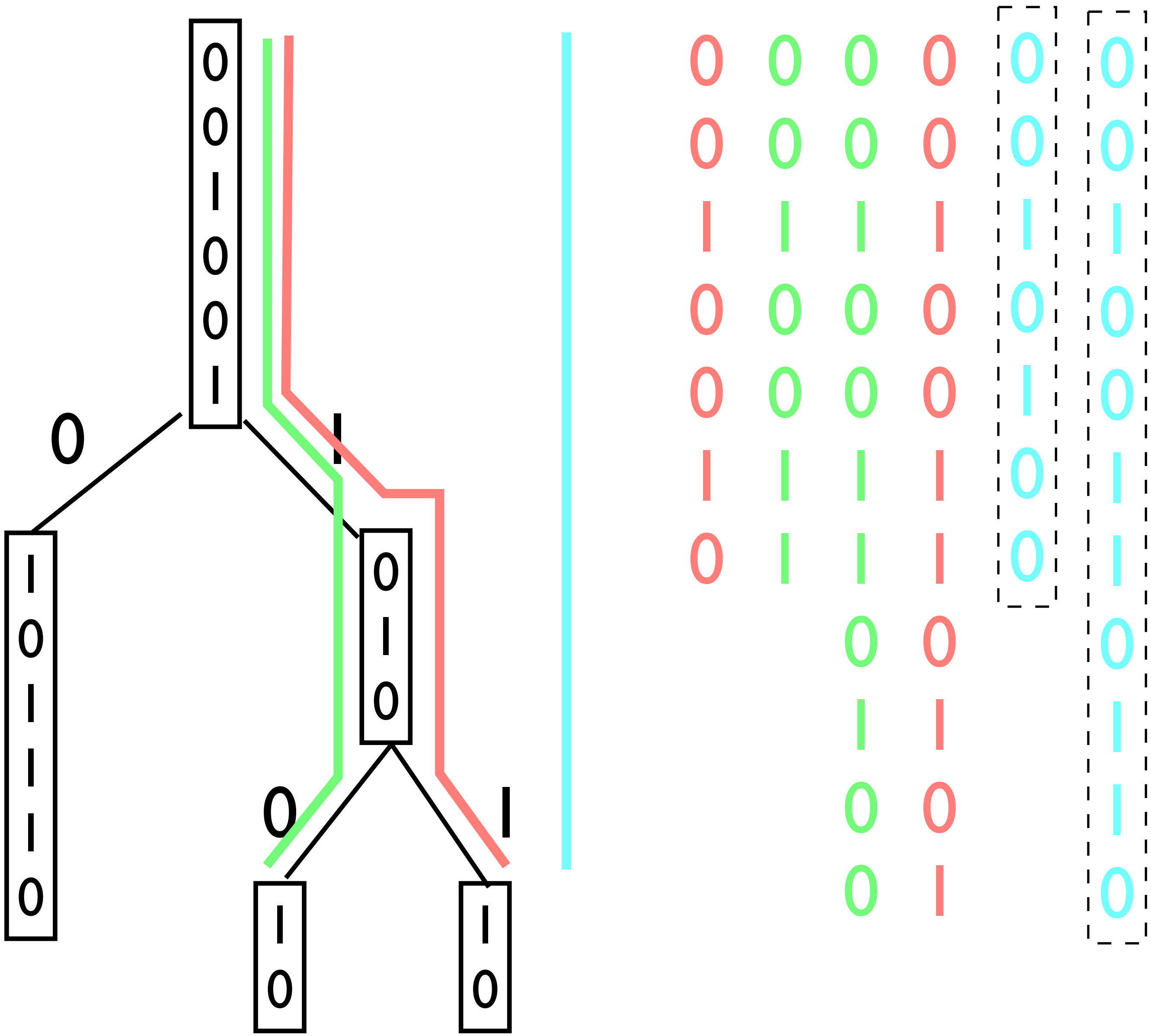

0 0 0
0 0 0
1 1 1
0 0 0
0 0 0
1 1 1
0 1 1
0
1
0
0
0
0
0
1
0
1
0


0
0
1
0
0
1
0
1
0
1
1
0
1
1
1
0
0
1
0
0
1
0
1
0

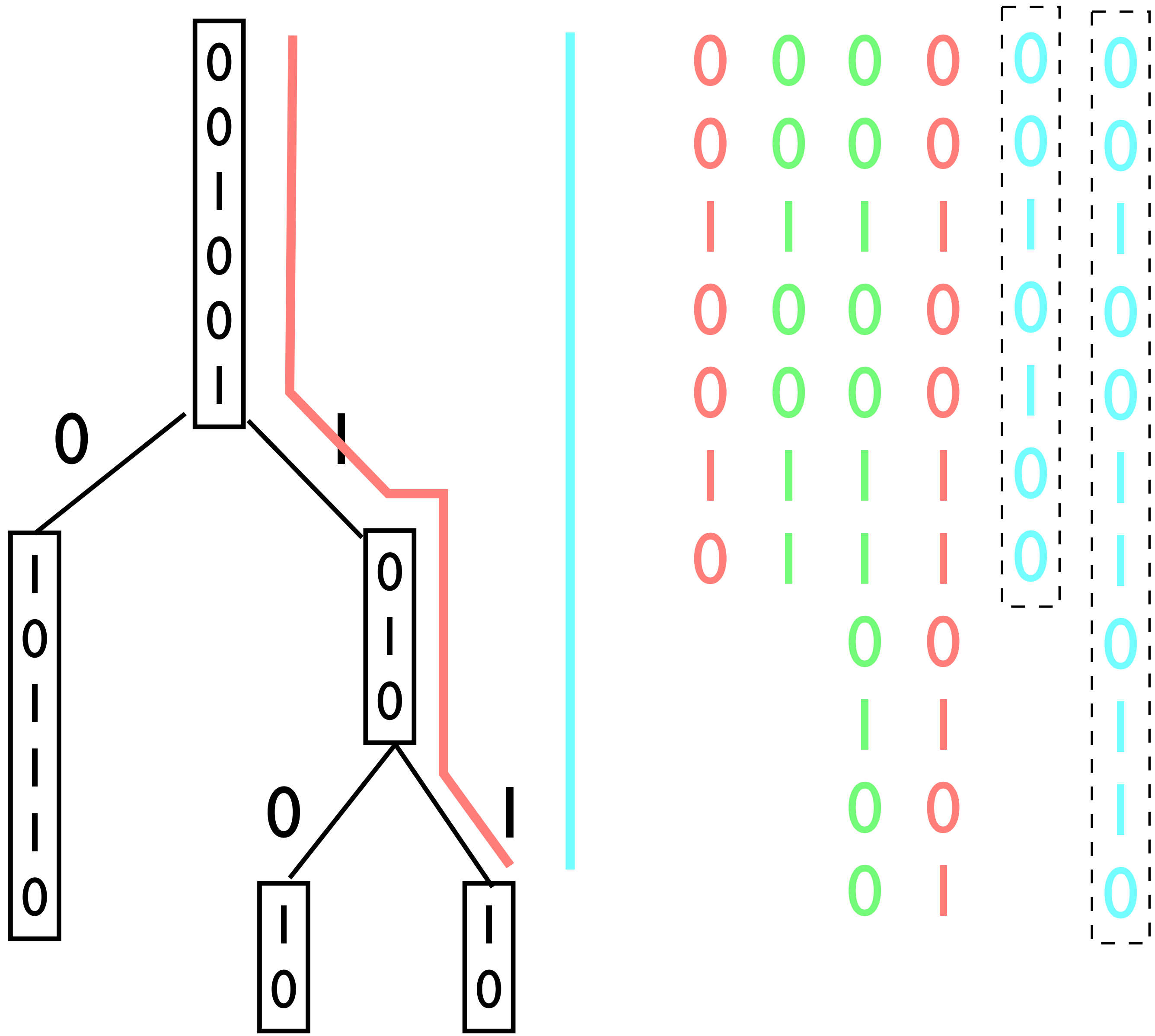

0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
1
0
1
0

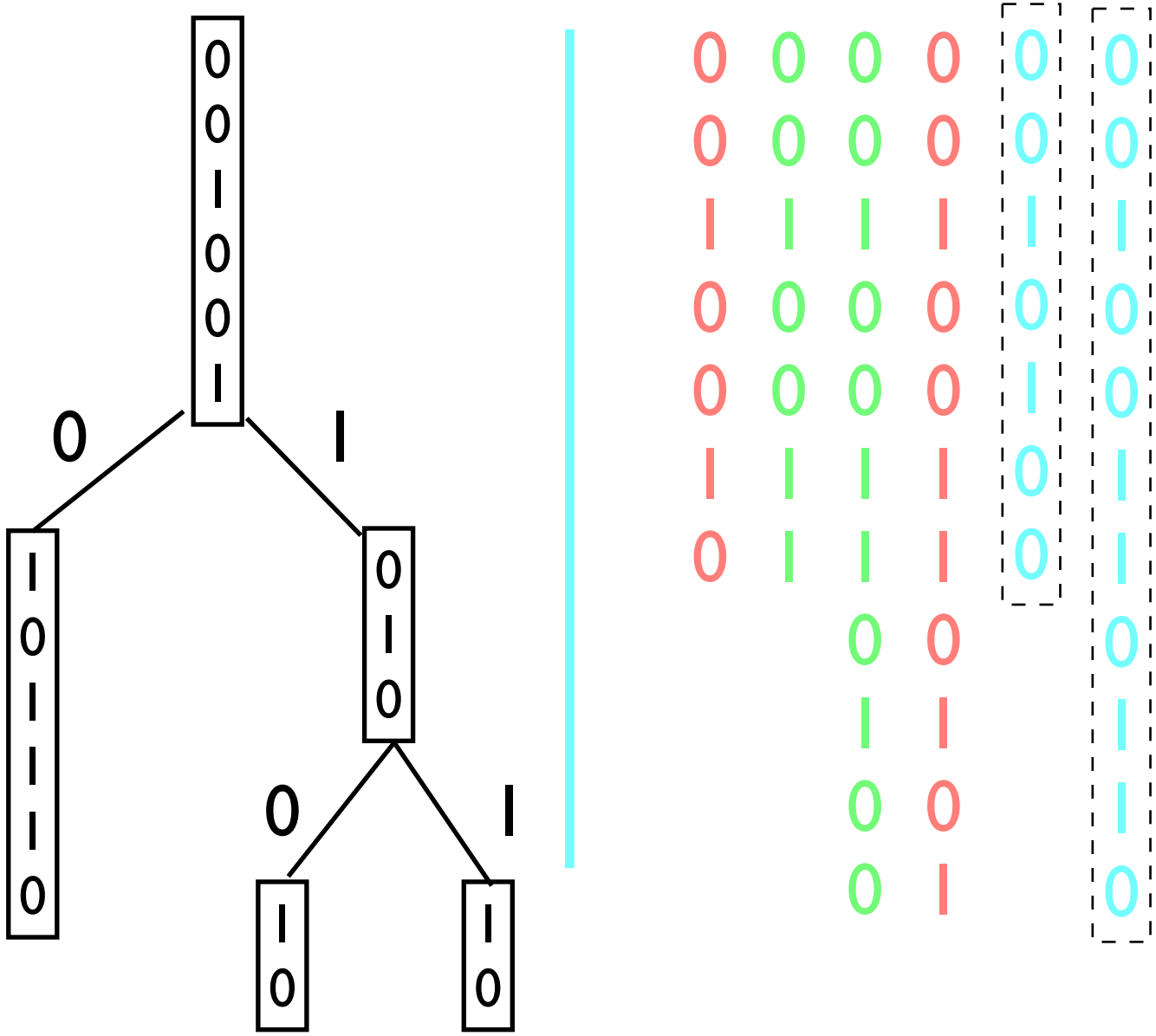


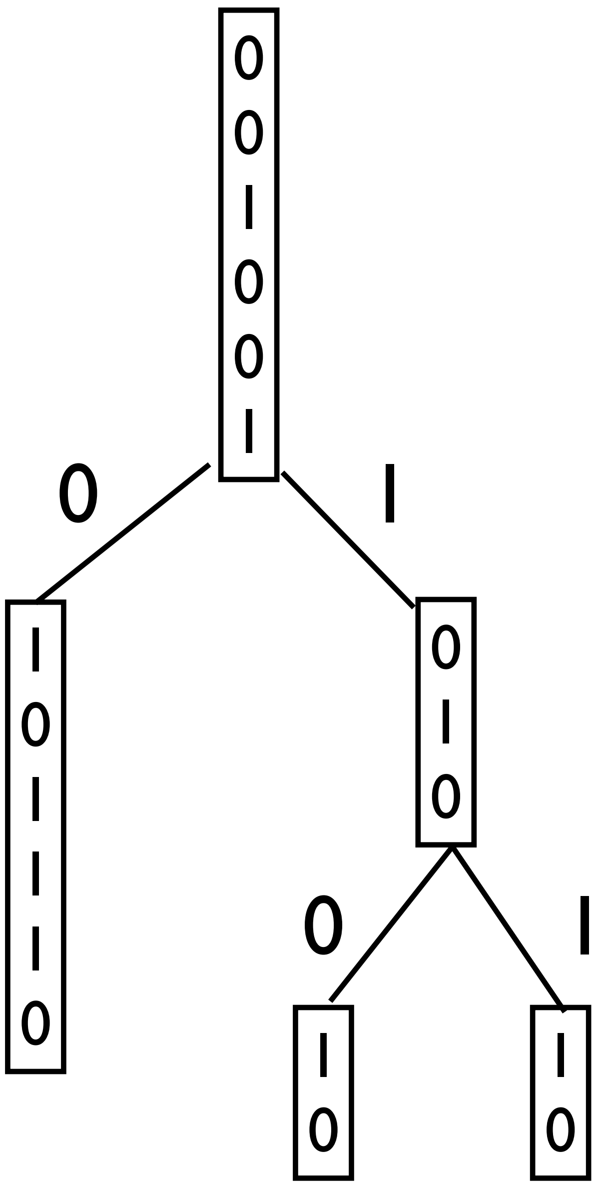

0 0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0



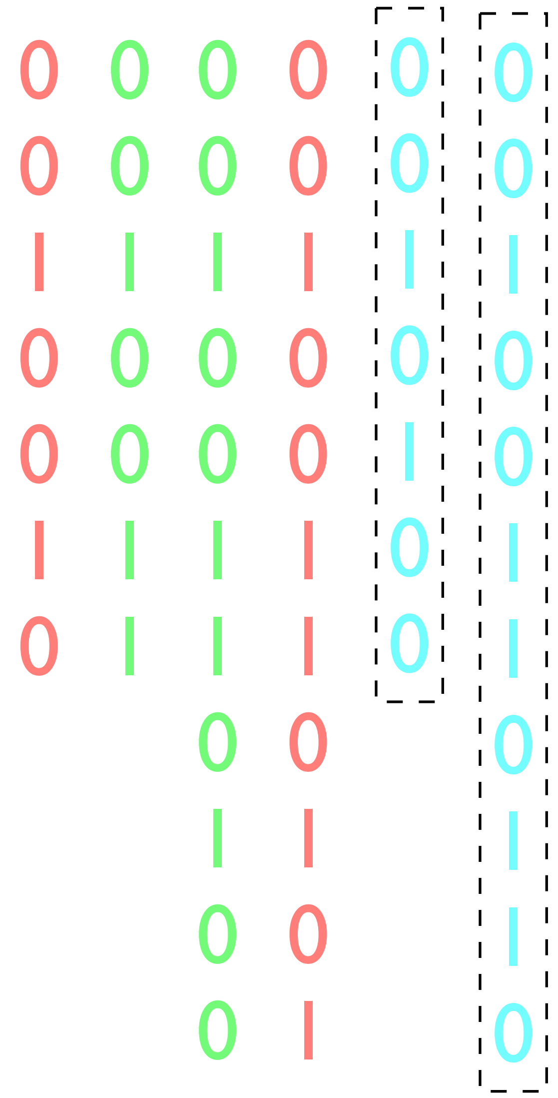
001001
0
1
101110
010
0
1
10
10
0010010
0010011
00100110100
00100110101
0010100
00100110110


001001
0
1
10111
0
010
0
1
10
10
0010010
0010011
00100110100
00100110101
0010100
00100110110

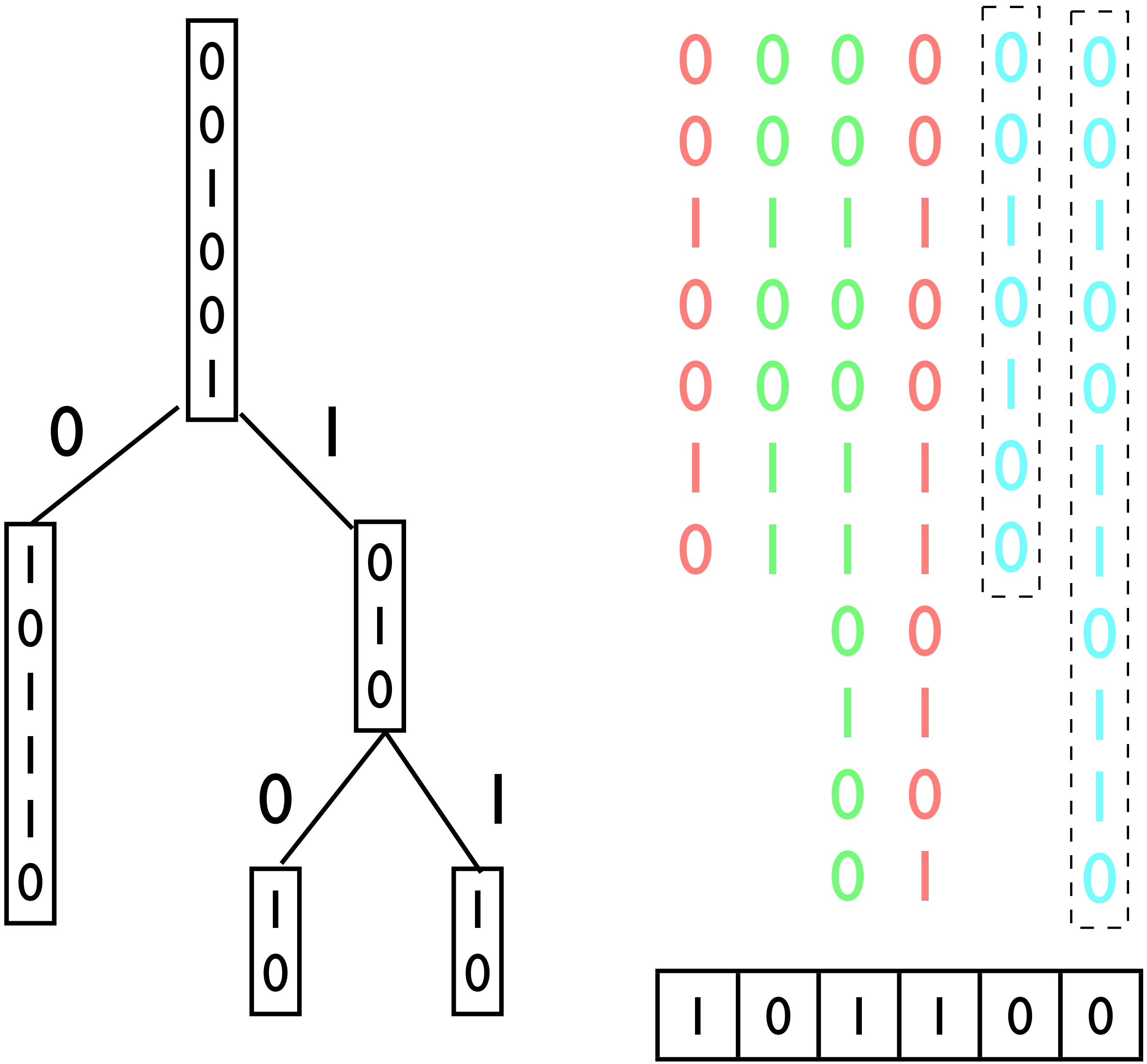

0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1
1 0

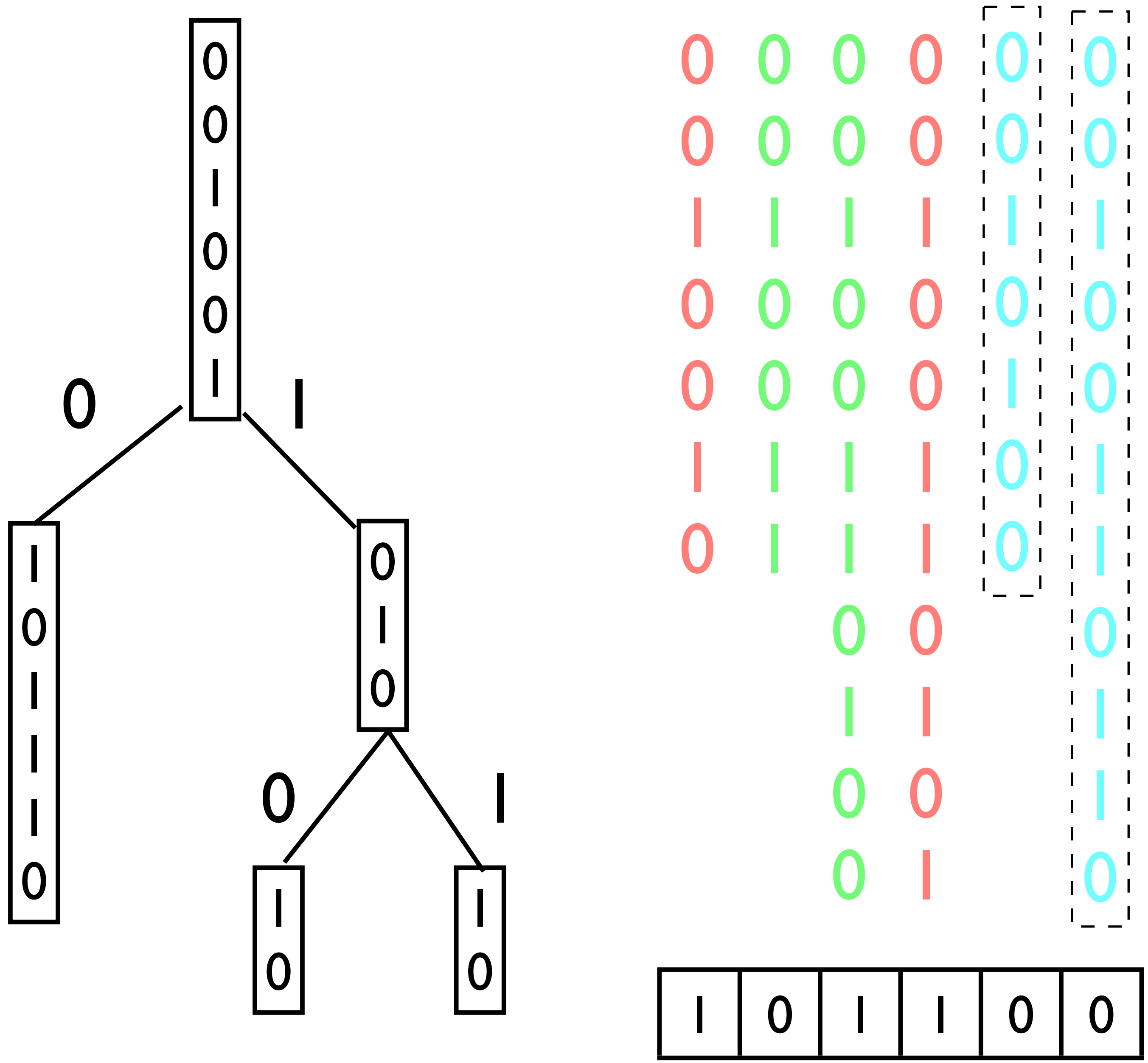


001001
0
1
101110
010
0
1
10
10

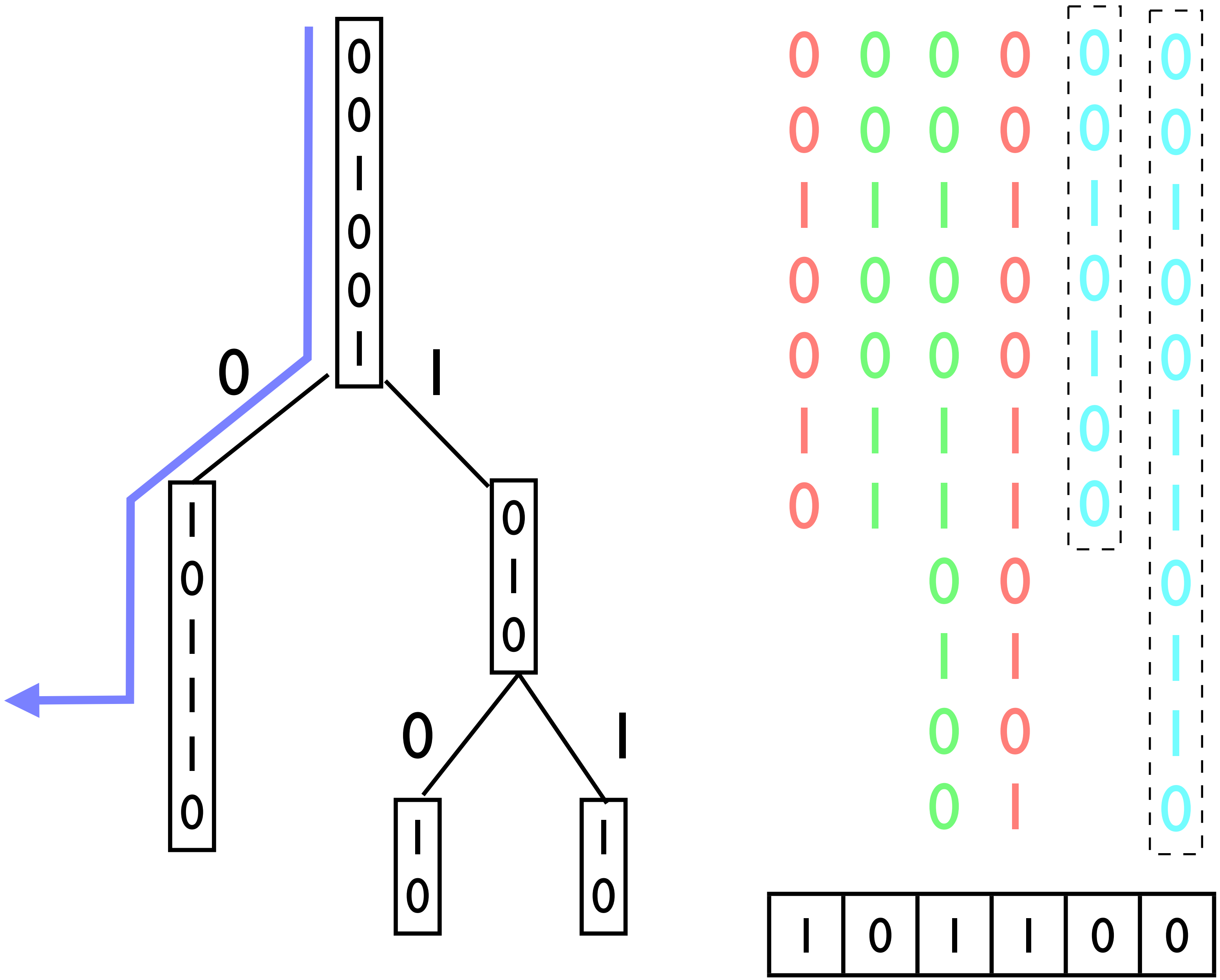


0010010
0010011
00100110100
00100110101
0010100
00100110110

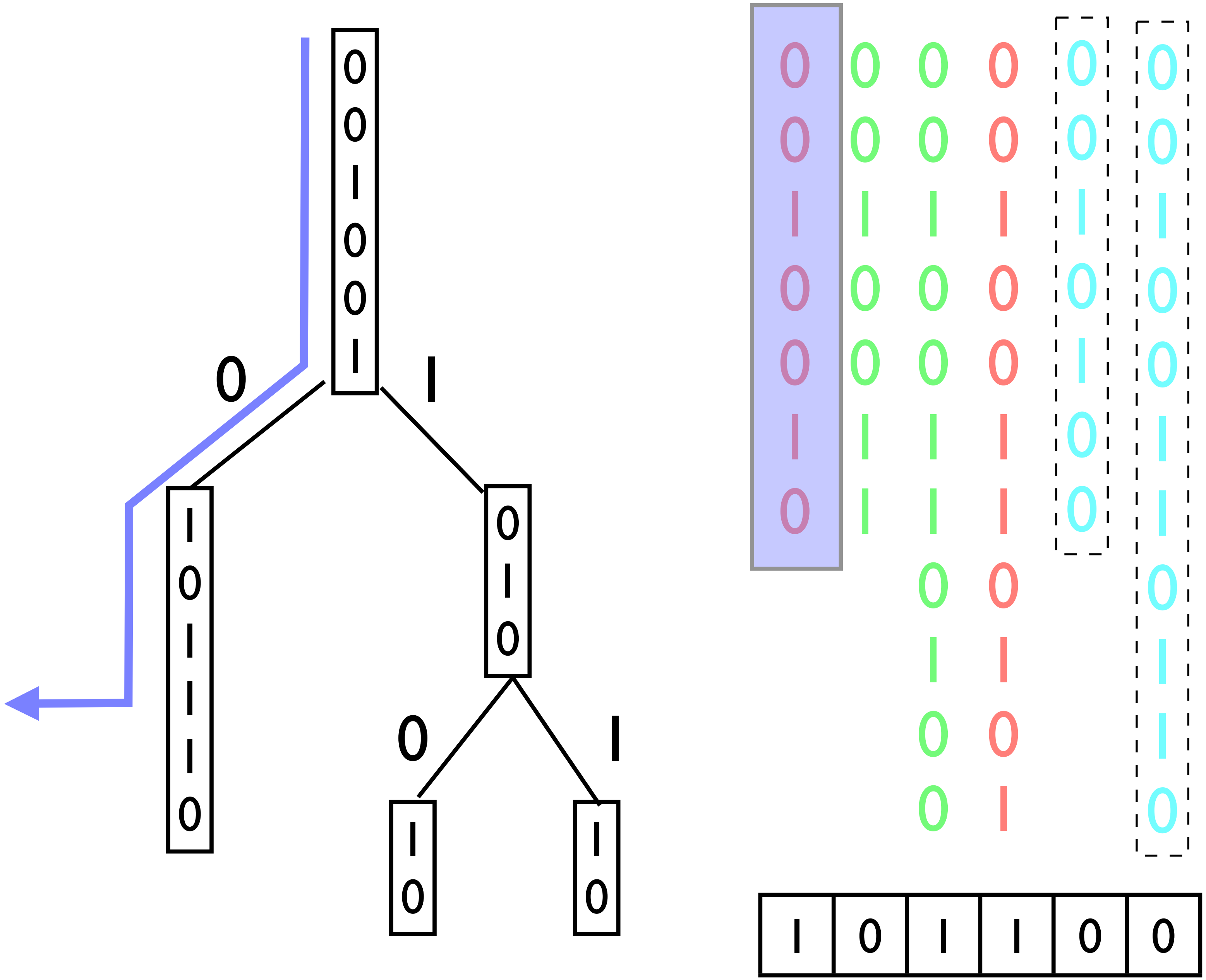

001001
0
1
101110
010
0
1
10
10
0010010
0010011
00100110100
00100110101
0010100
00100110110
1 0 1 1 0 0



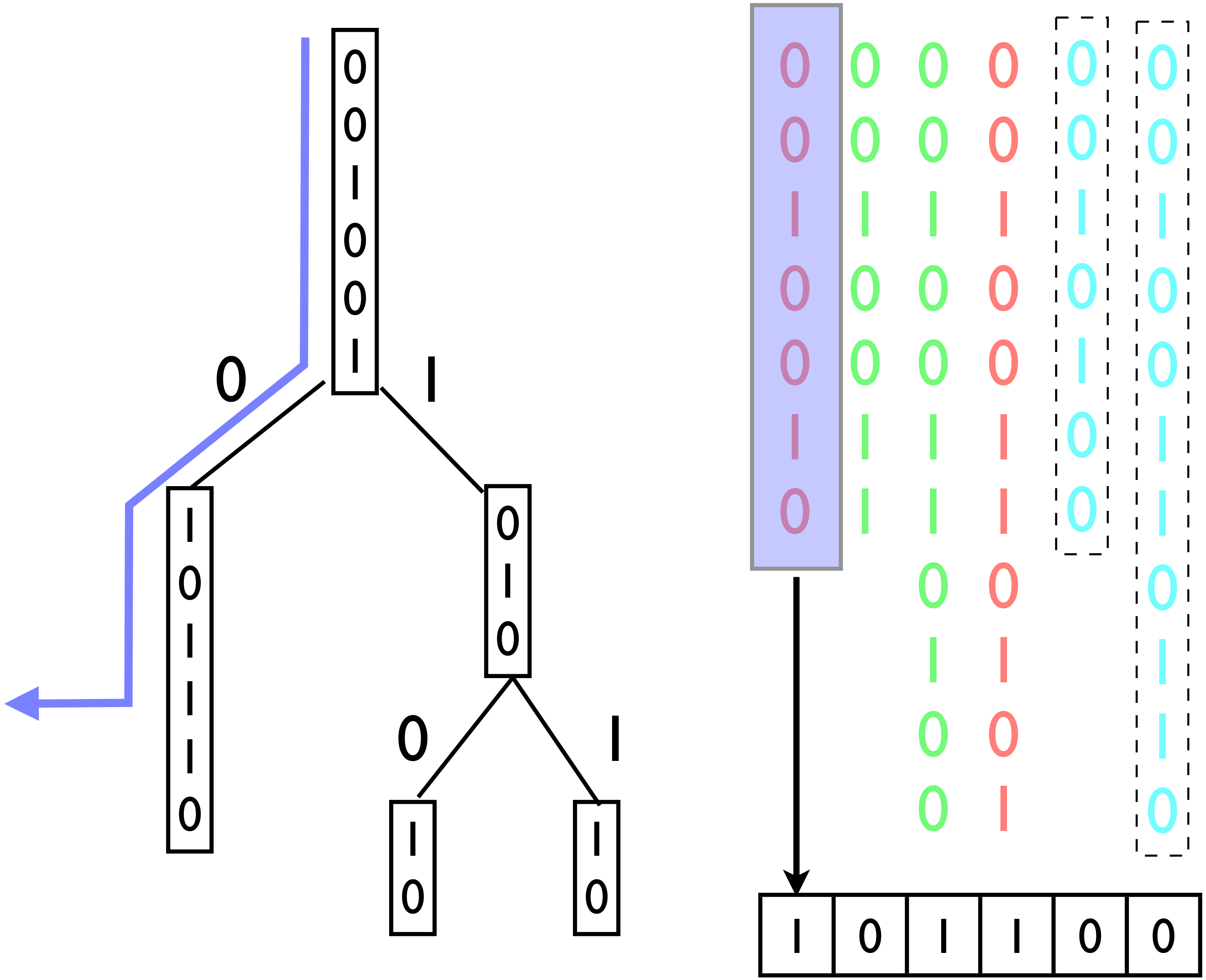
001001
0
1
101110
010
0
1
10
10
0010010
0010011
00100110100
00100110101
0010100
00100110110
1 0 1 1 0 0



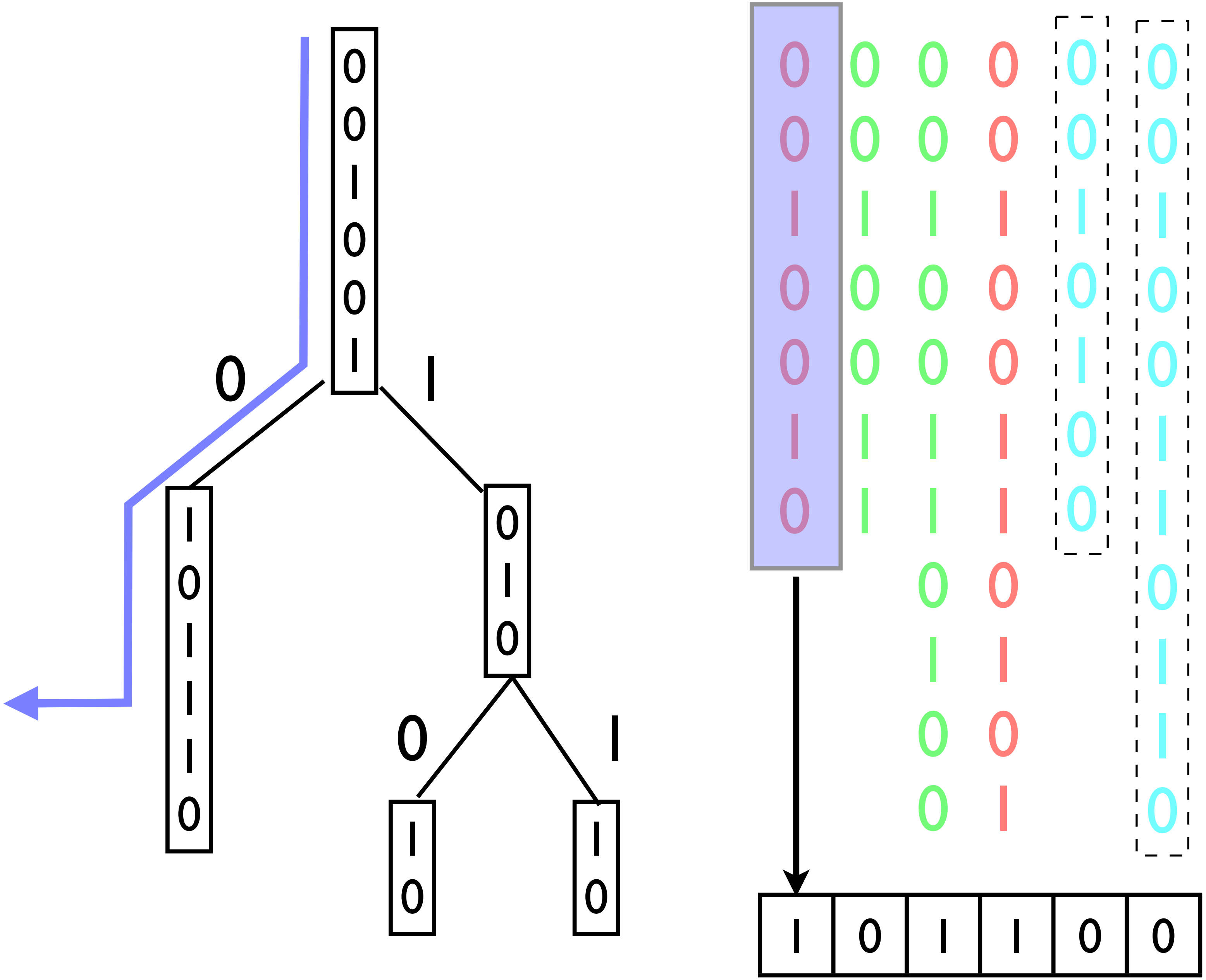
001001
0
1
101110
010
0
1
10
10
0010010
0010011
00100110100
00100110101
0010100
00100110110
1 0 1 1 0 0



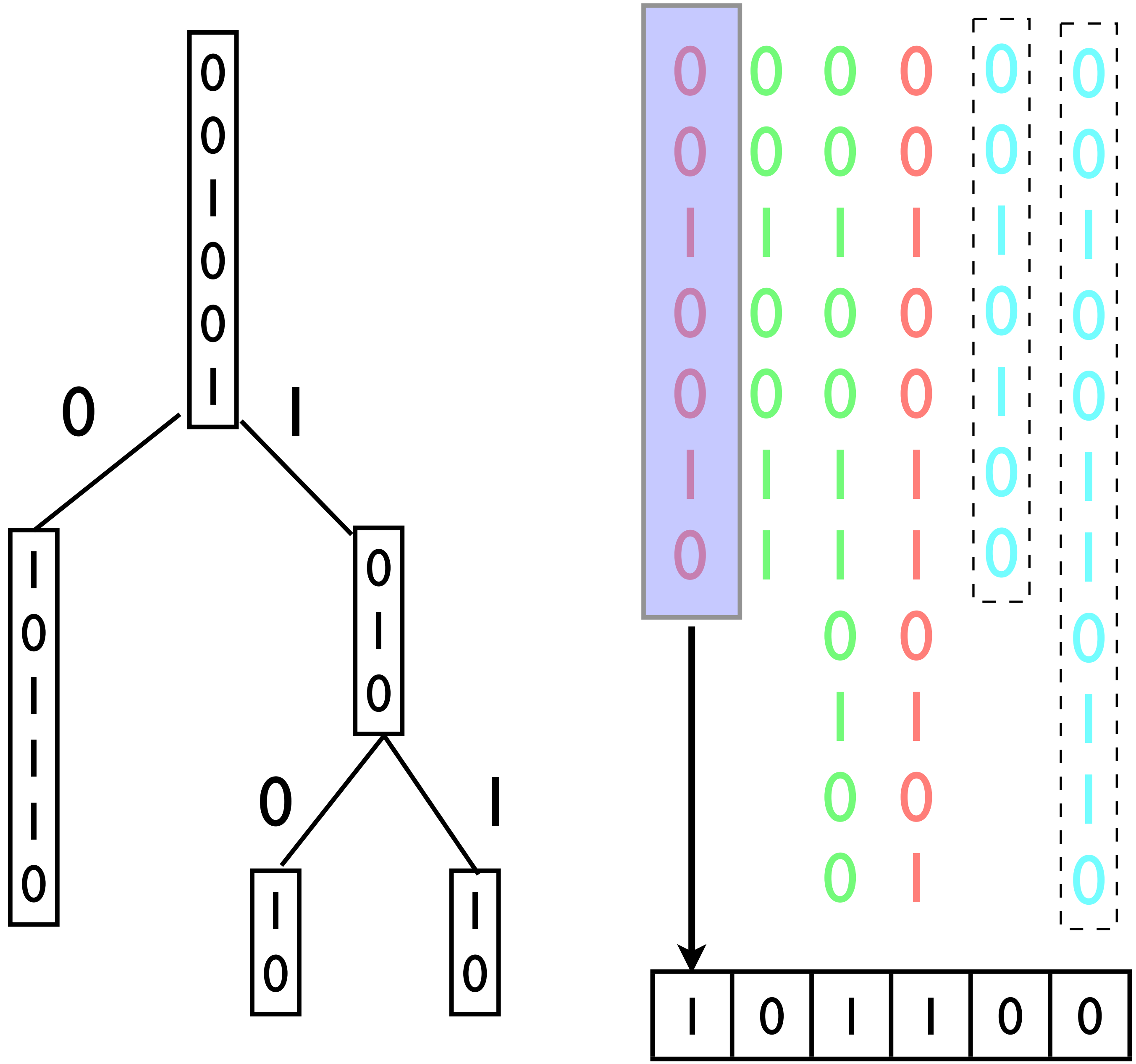
0
1
0
1
1 0 1 1 0 0



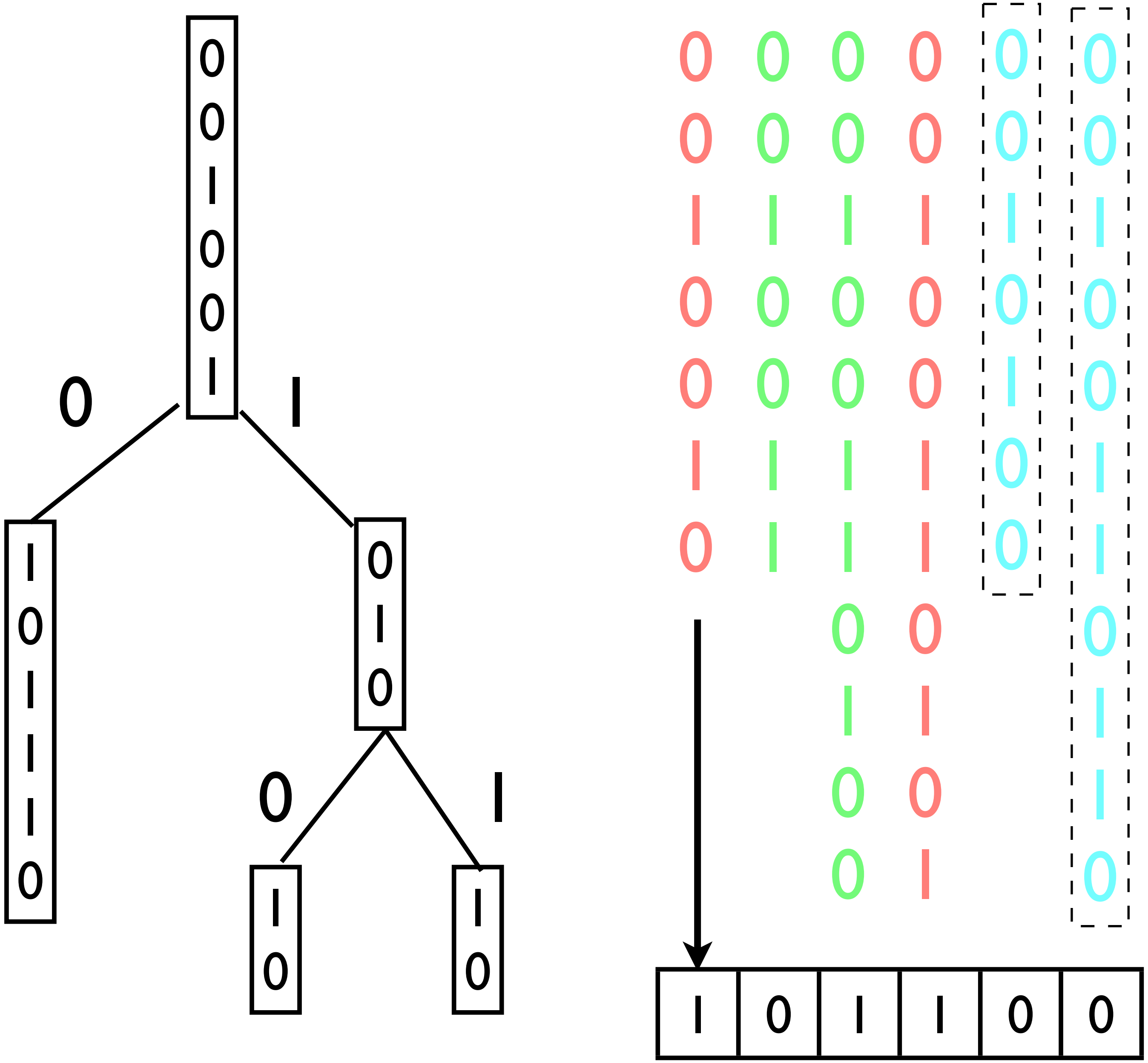
0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0
1 0 1 1 0 0



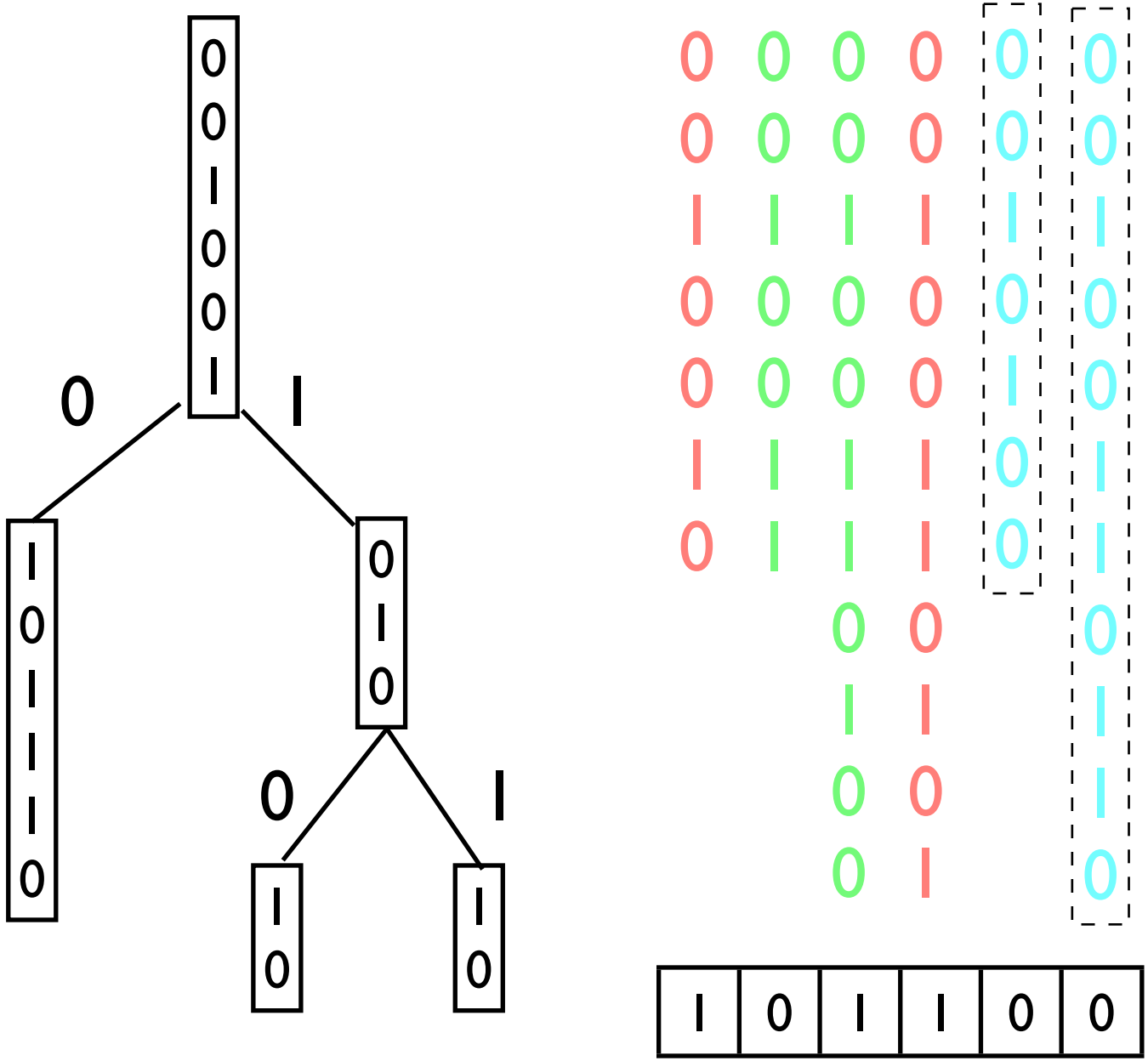
Rank 0 ⇒ bucket 0!



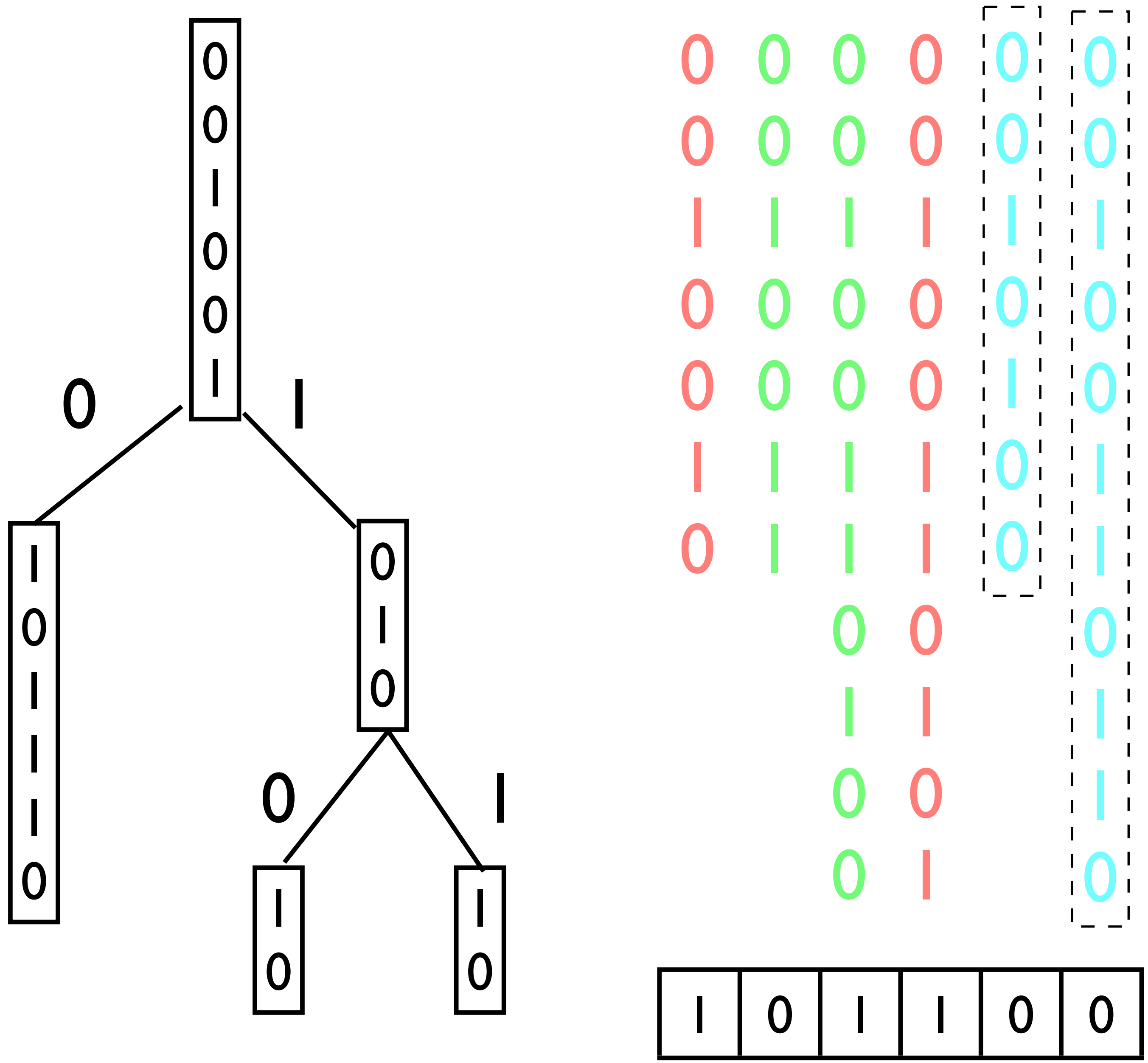
Rank 0 ⇒ bucket 0!

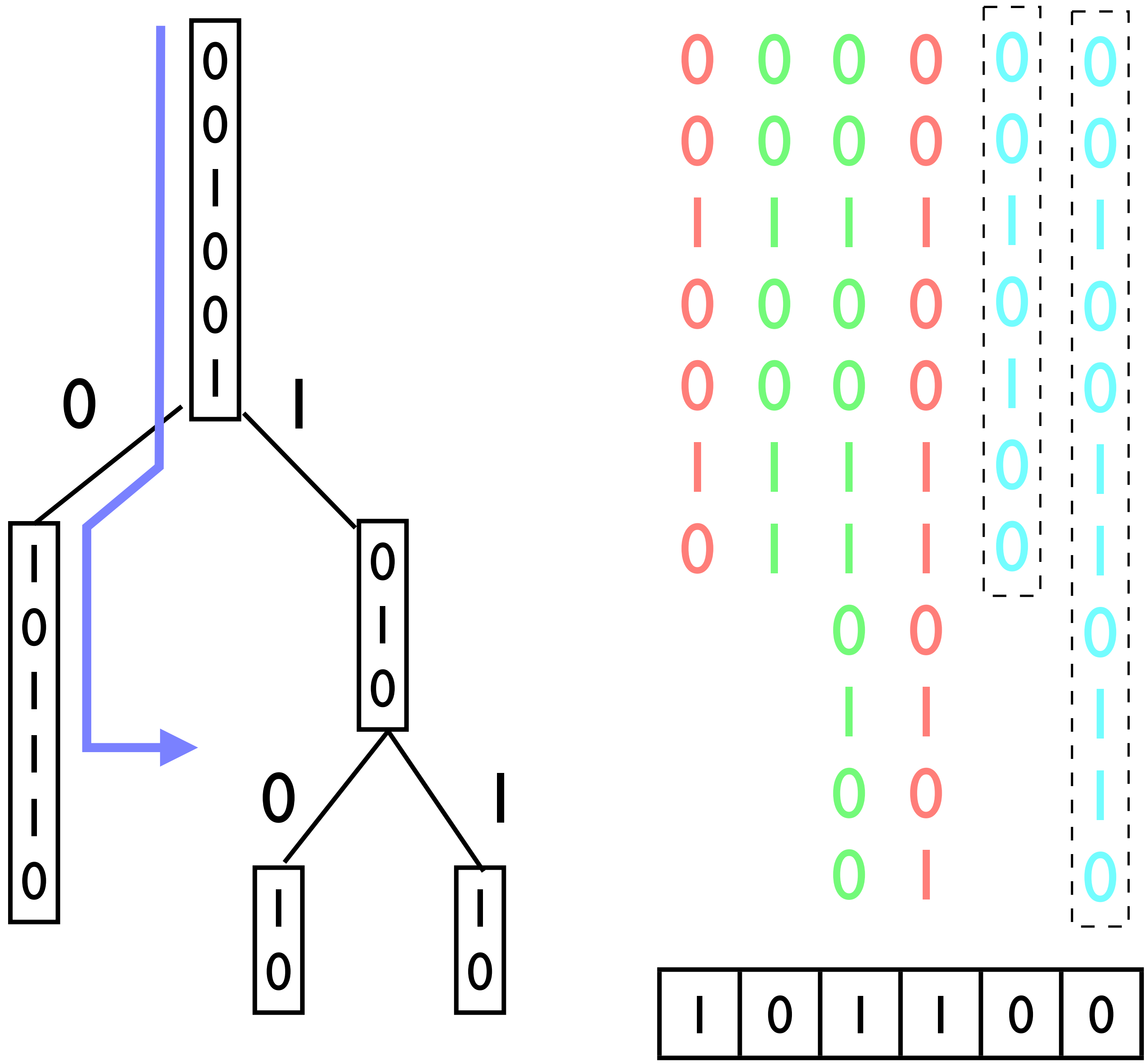


Rank 0 ⇒ bucket 0!

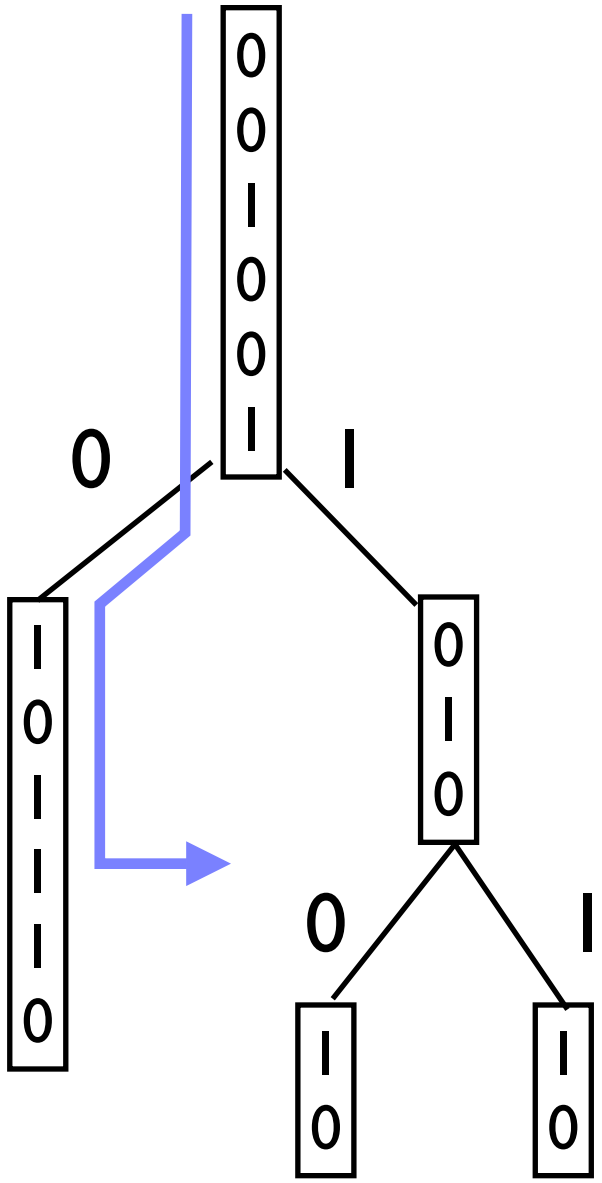


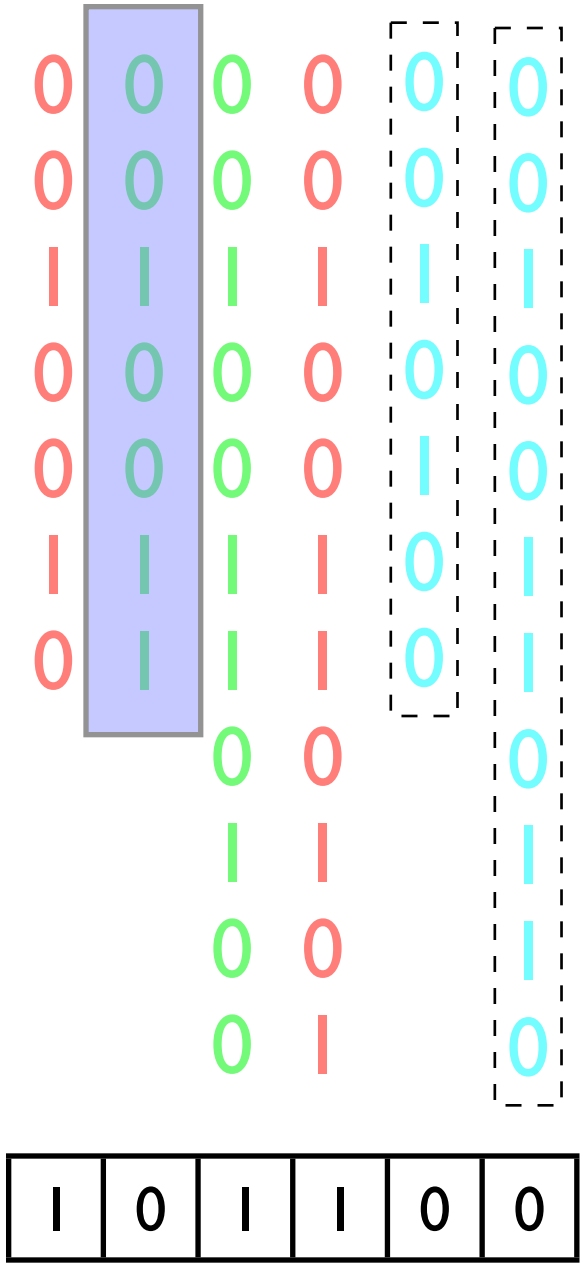
Rank 0 ⇒ bucket 0!

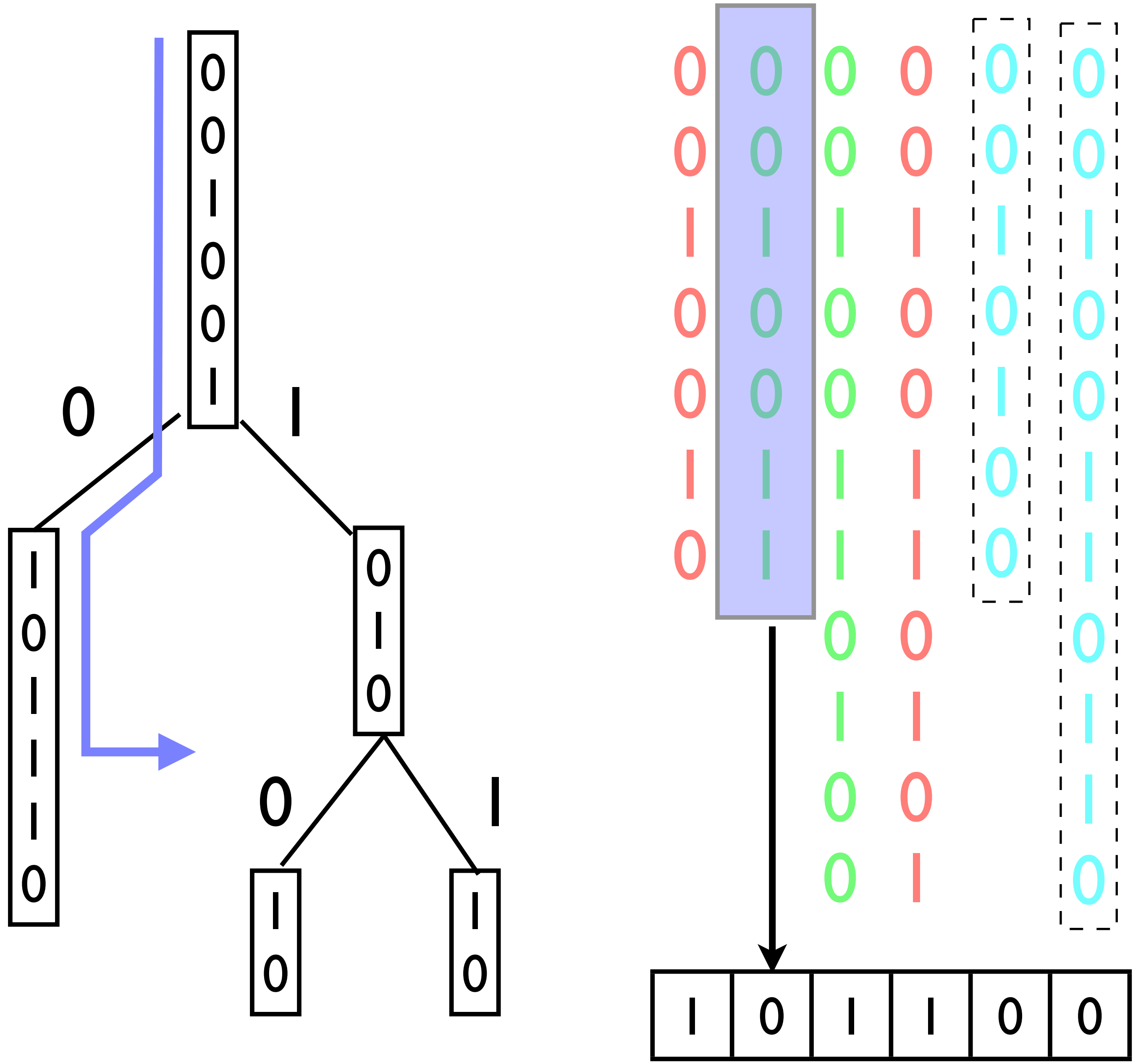

0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0
0 0 1 0 0 1 0
0 0 1 0 0 1 1
0 0 1 0 0 1 1 0 1 0 0
0 0 1 0 0 1 1 0 1 0 1
0 0 1 0 1 0 0
0 0 1 0 0 1 1 0 1 1 0
1 0 1 1 0 0



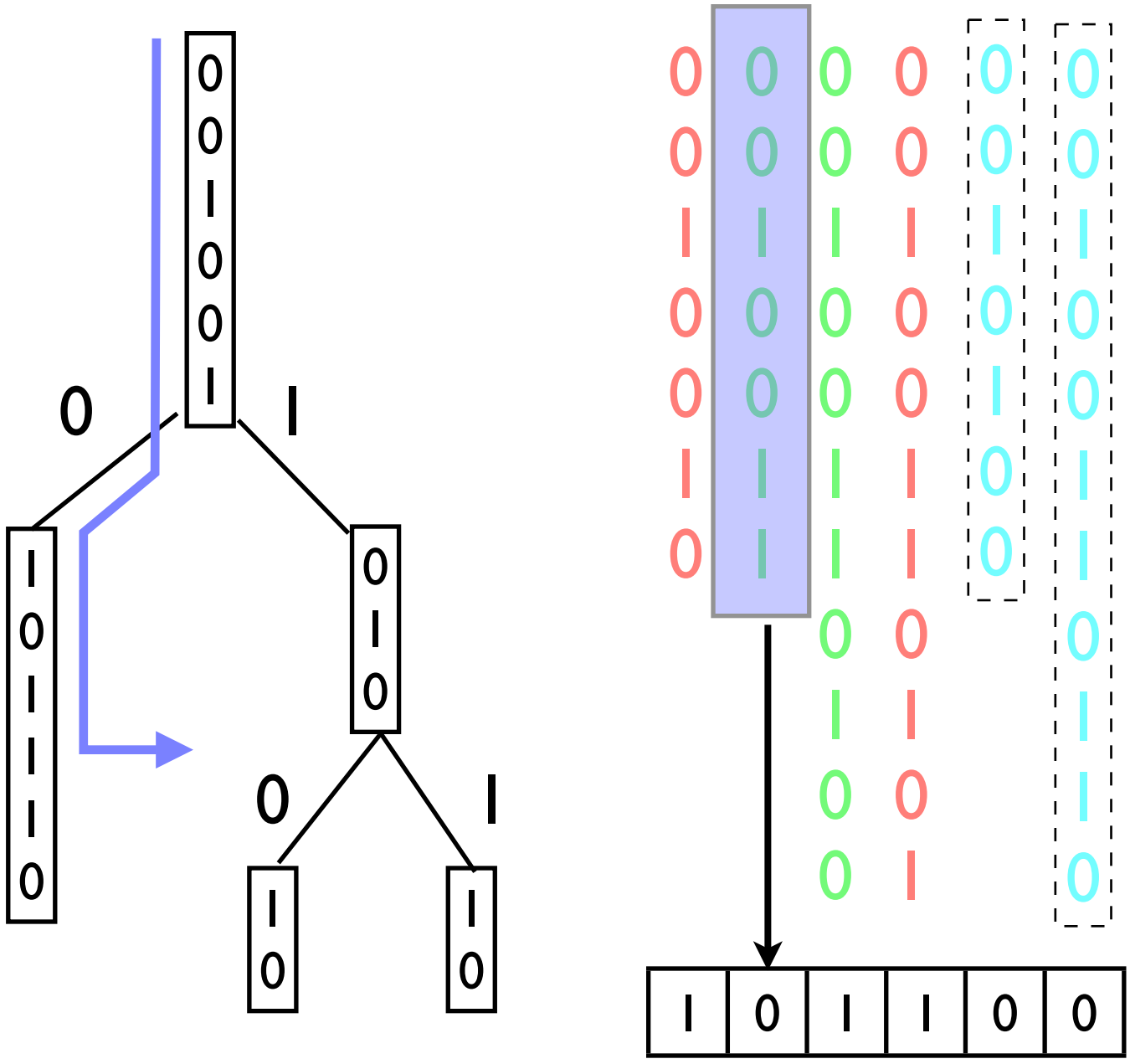
0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0
0 0 1 0 0 1 0
0 0 1 0 0 1 1
0 0 1 0 0 1 1 0 1 0 0
0 0 1 0 0 1 1 0 1 0 1
0 0 1 0 1 0 0
0 0 1 0 0 1 1 0 1 1 0
1 0 1 1 0 0


0
0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
1
0

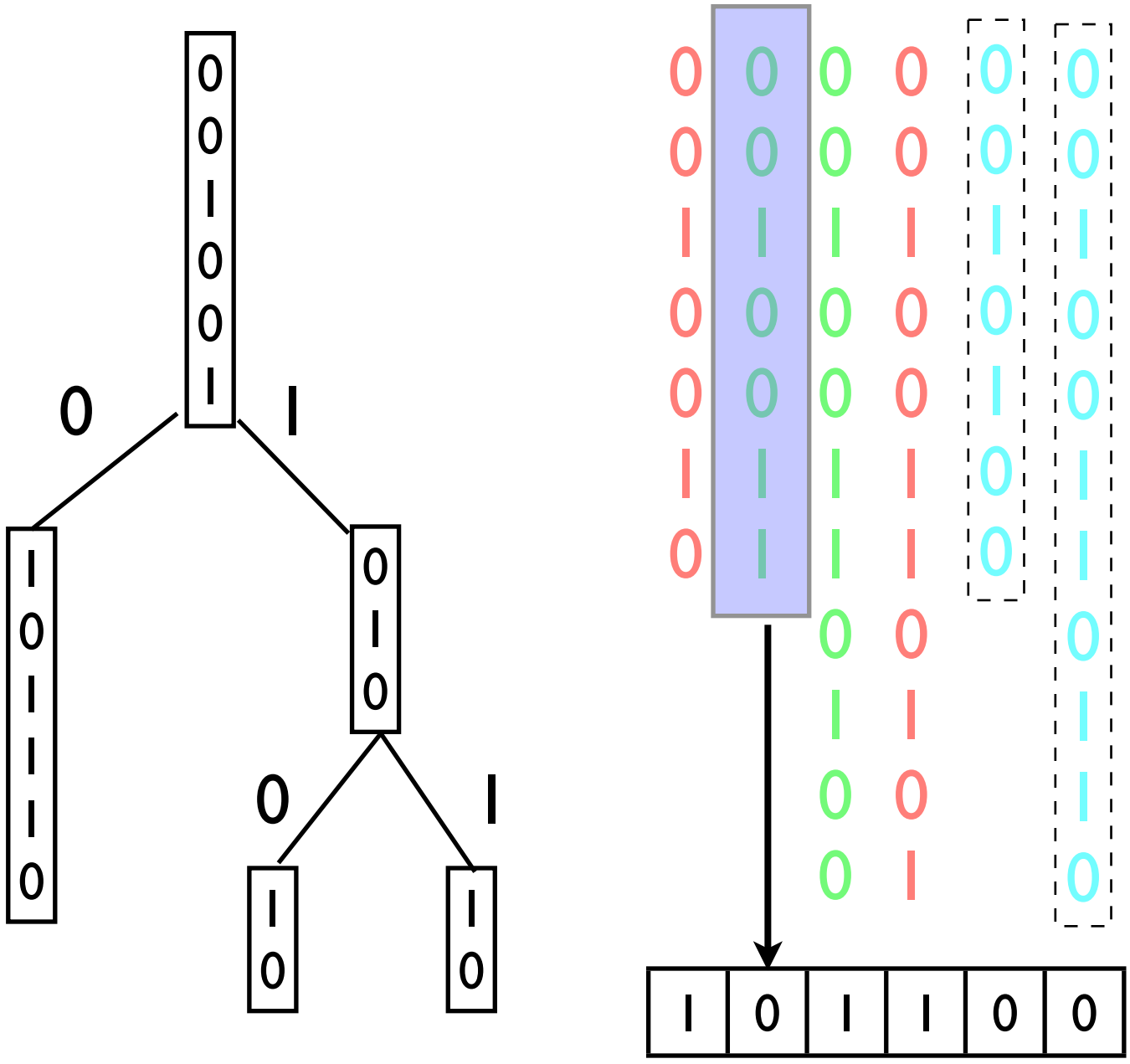

0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
0 1 1 1 0 0
0 0
1 1
0 0
1 1
0 0
1 1
0 0
1 1
0 0
1 1
0 0
1 1
1 0 1 1 0 0



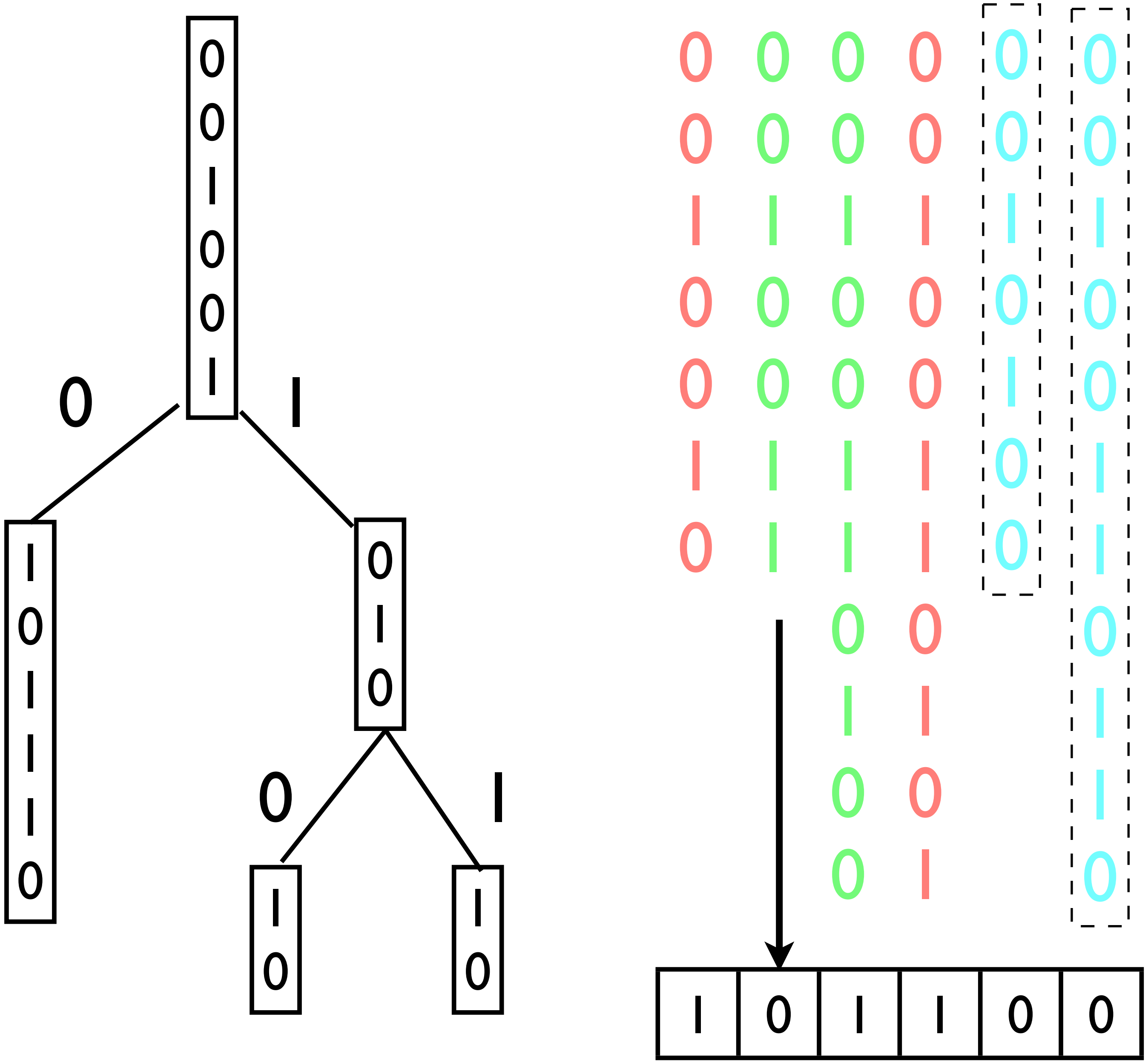
0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0
0 0 1 0 0 1 0
0 0 1 0 0 1 1
0 0 1 0 0 1 1 0 1 0 0
0 0 1 0 0 1 1 0 1 0 1
0 0 1 0 1 0 0
0 0 1 0 0 1 1 0 1 1 0
1 0 1 1 0 0



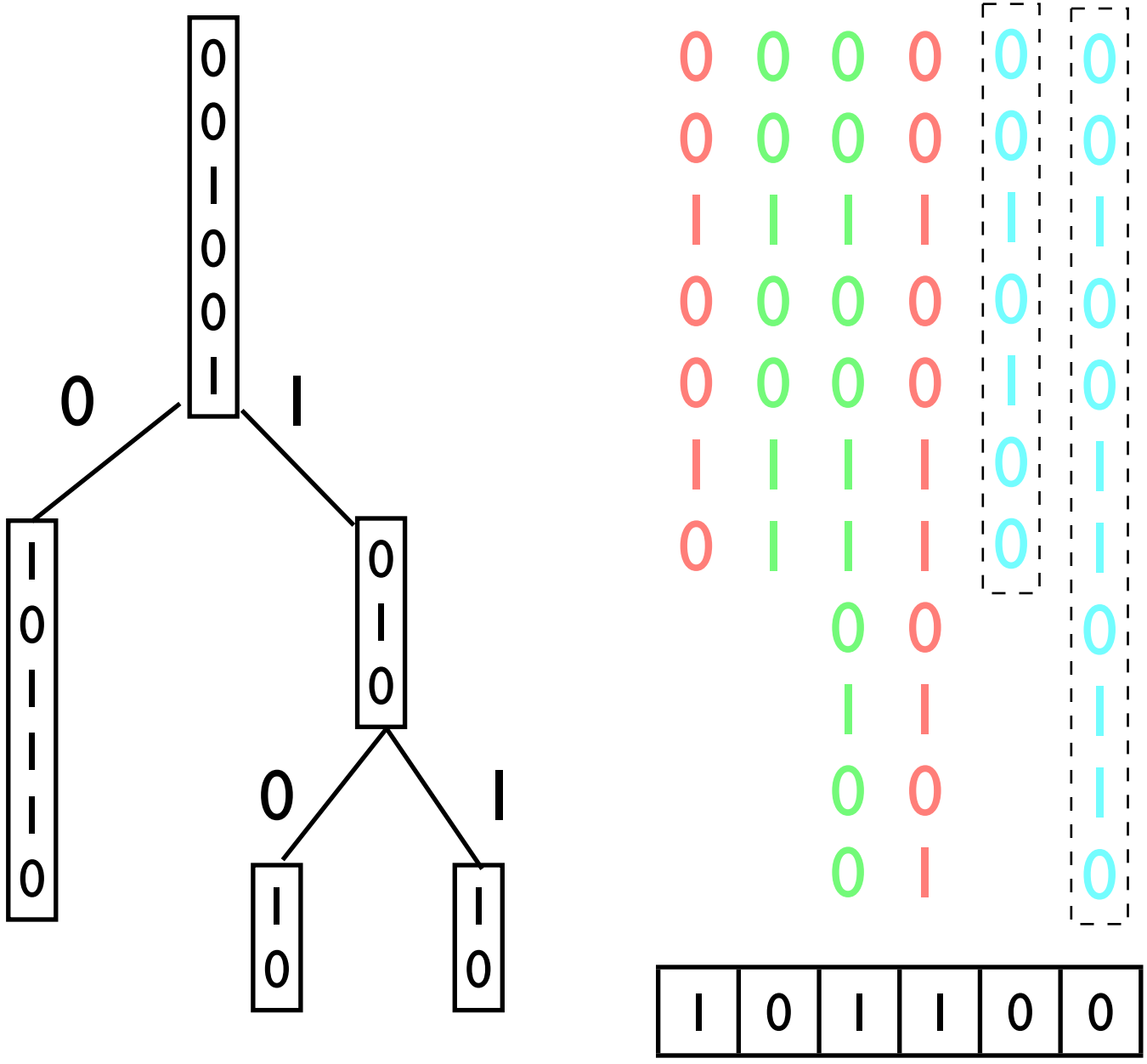
Rank 1 ⇒ bucket 1!

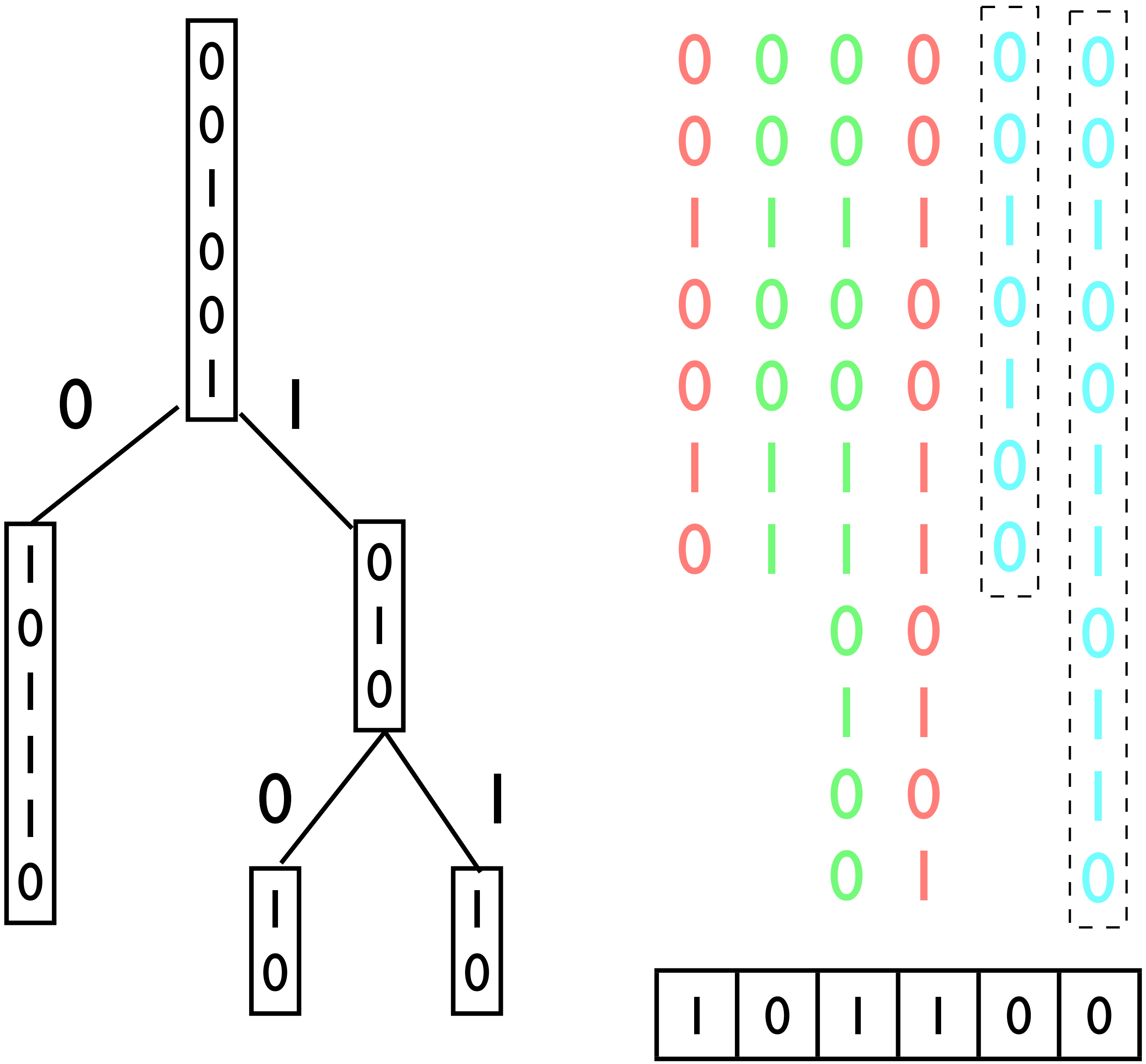


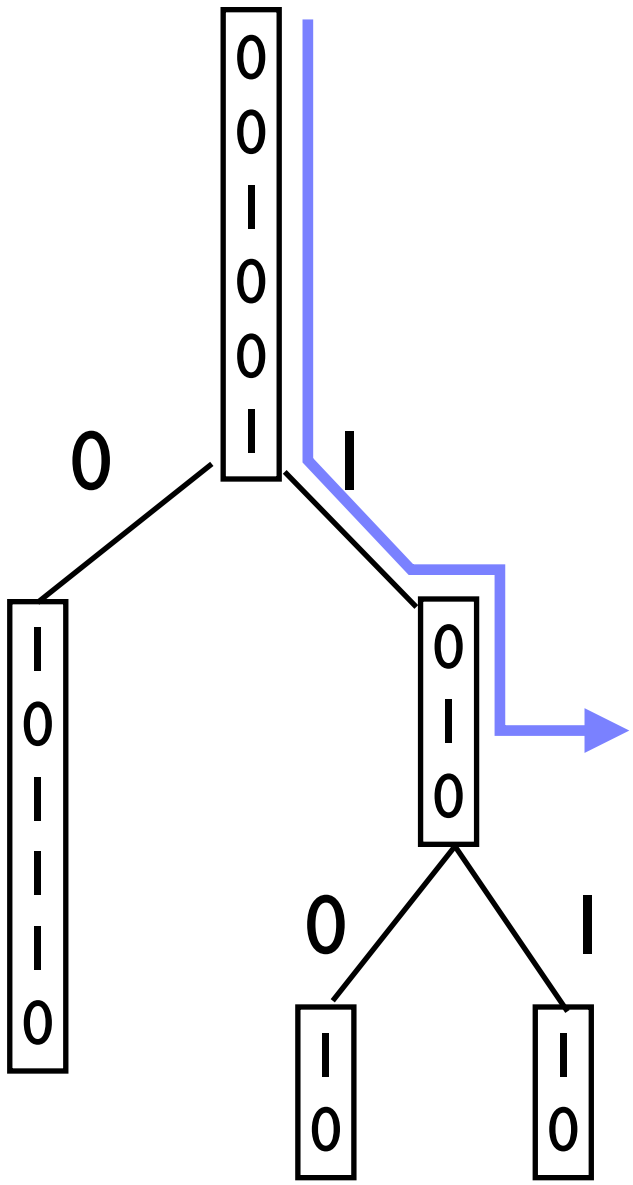
Rank 1 ⇒ bucket 1!

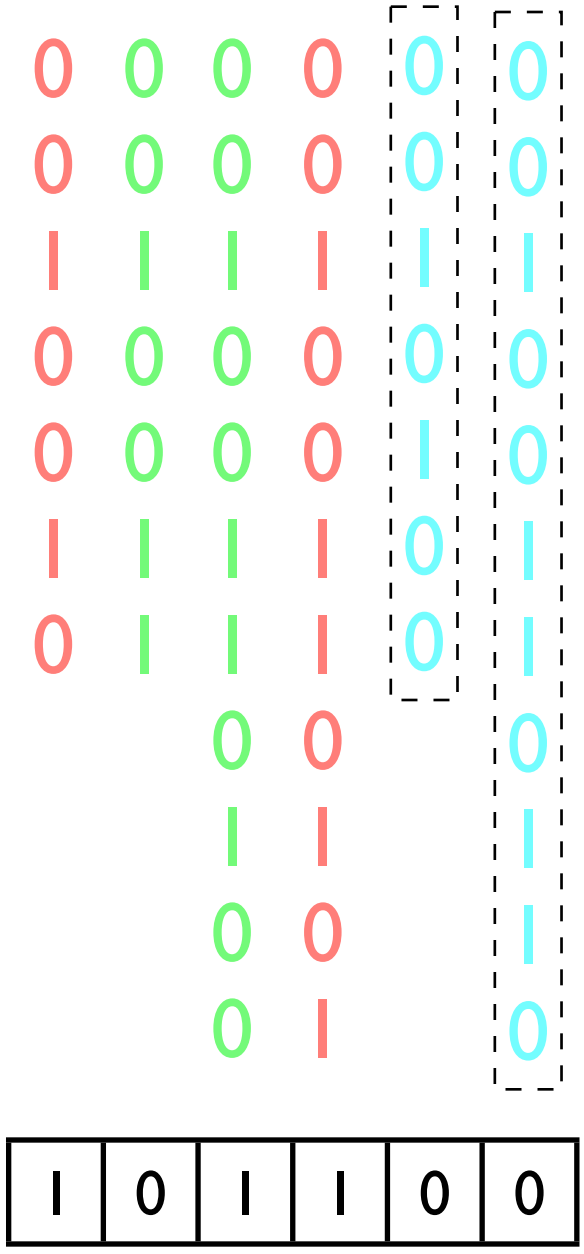


Rank 1 ⇒ bucket 1!

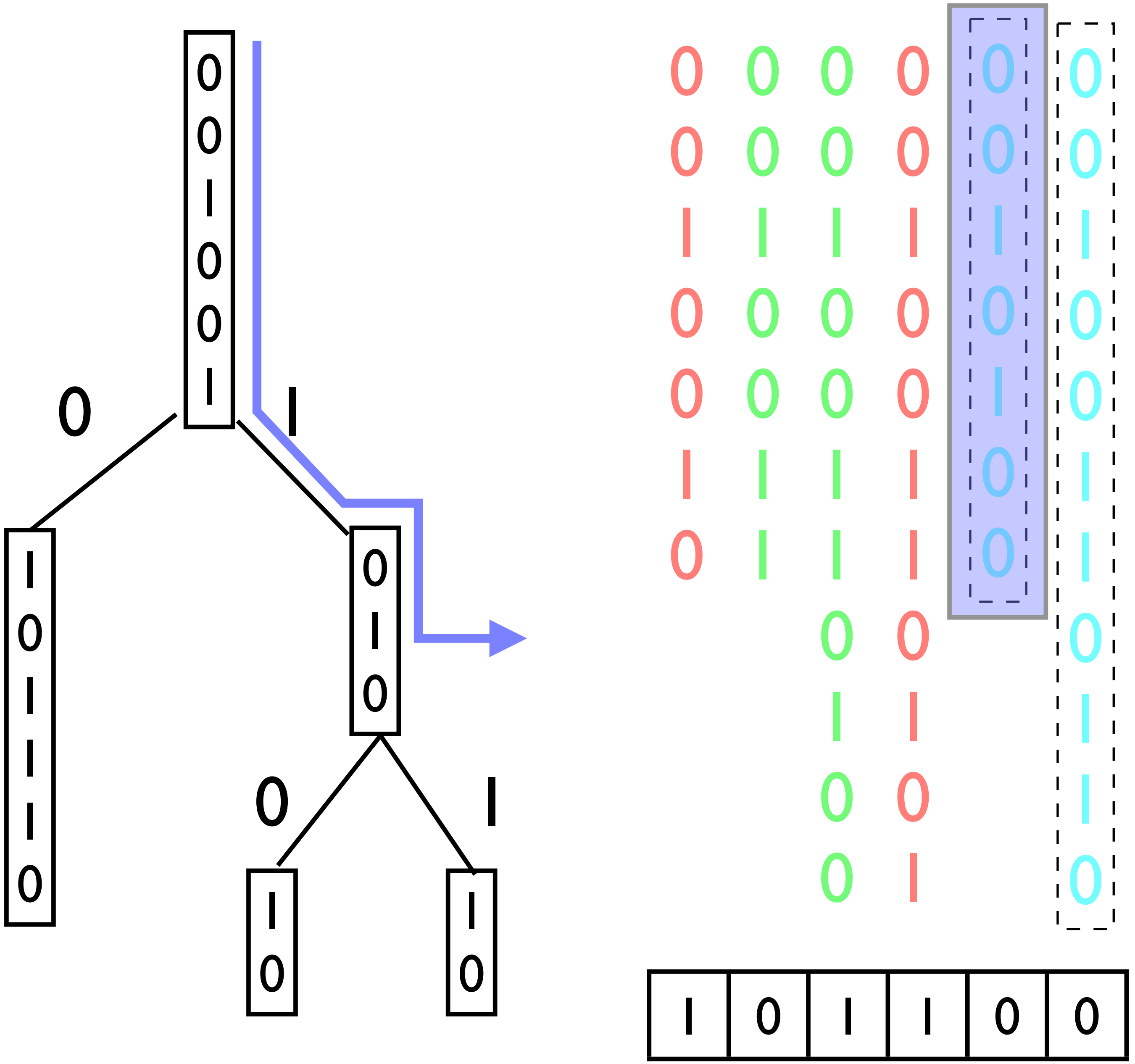

0 0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 1 0
0
1
1 0
1 0
0 0 0 0
0 0 0 0
1 1 1 1
0 0 0 0
0 0 0 0
1 1 1 1
0 1 1 1
0 0
1 1
0 0
0 1
0 0 1 0 1 0
1 0 1 1 0 0
Rank 1 ⇒ bucket 1!

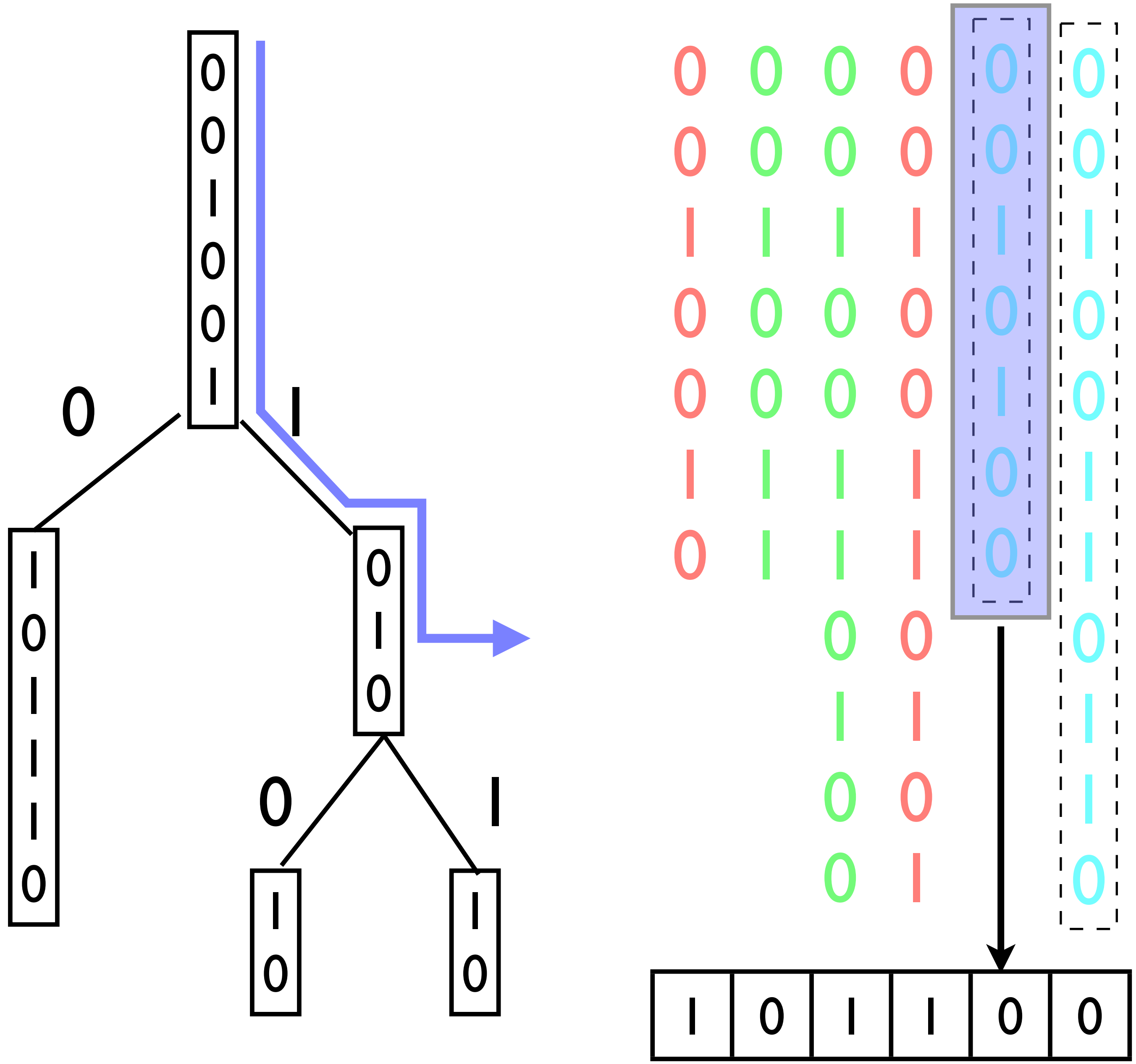

001001
0
1
101110
010
0
1
10
10
0010010
0010011
00100110100
00100110101
0010100
00100110110
1 0 1 1 0 0

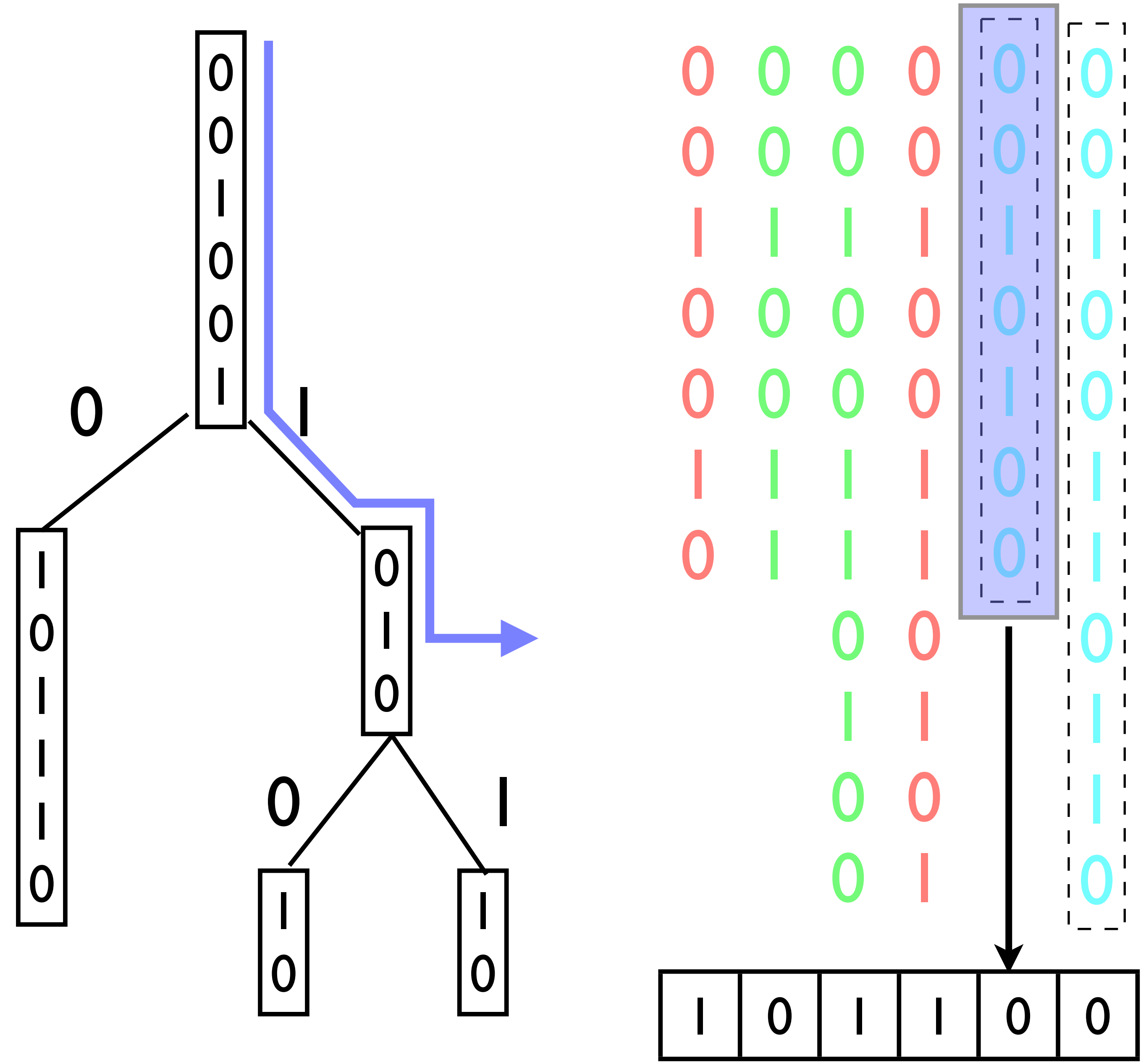

0
0
0
1
0
0
1
0
1
0
1
1
0
0
1
0
0
1
0
1
0


0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
0 1 1 1 0 0
0 1 0 0
1 1
0 0
0 1

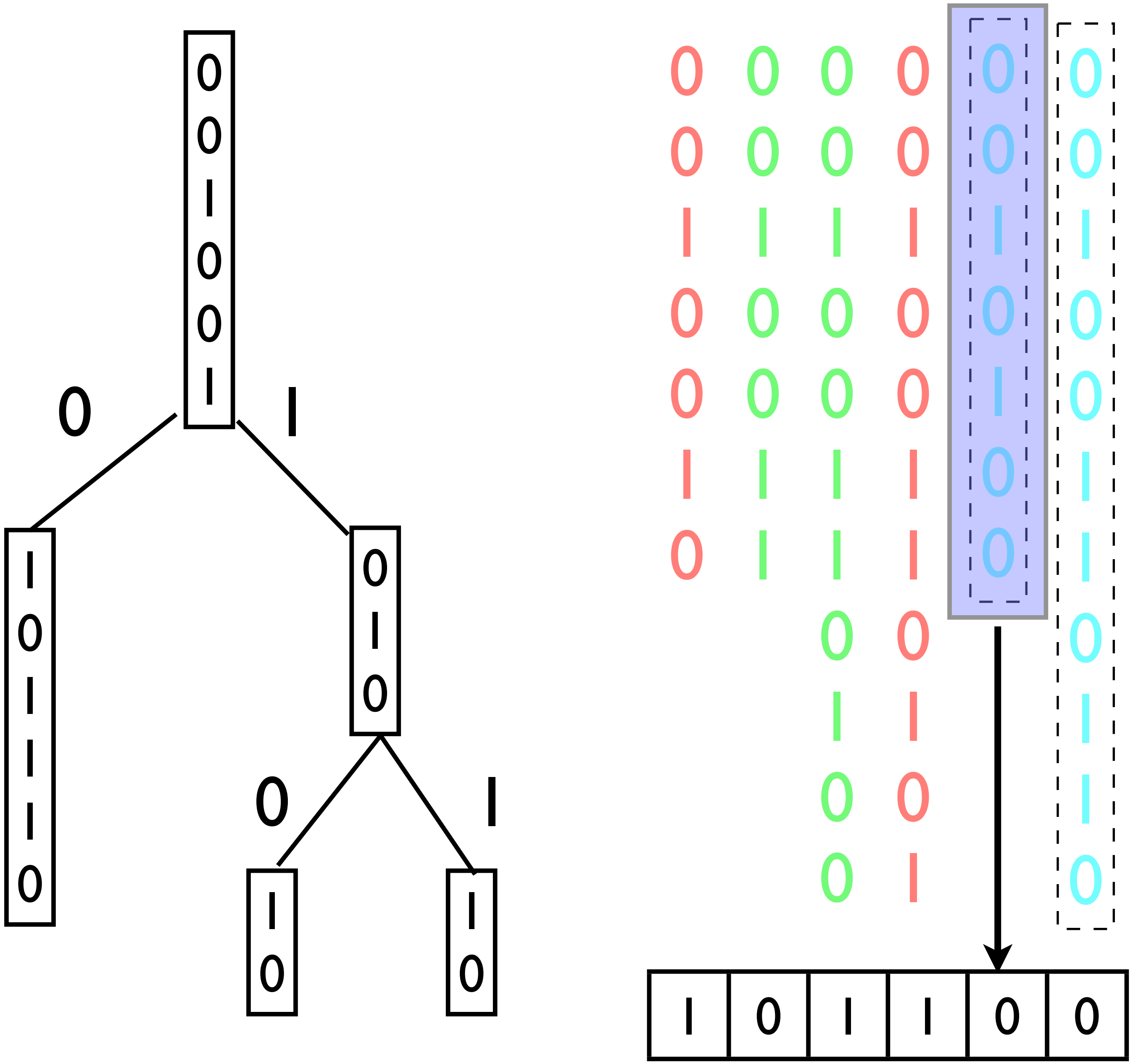
1 0 1 1 0 0


1 0 1 1 0 0

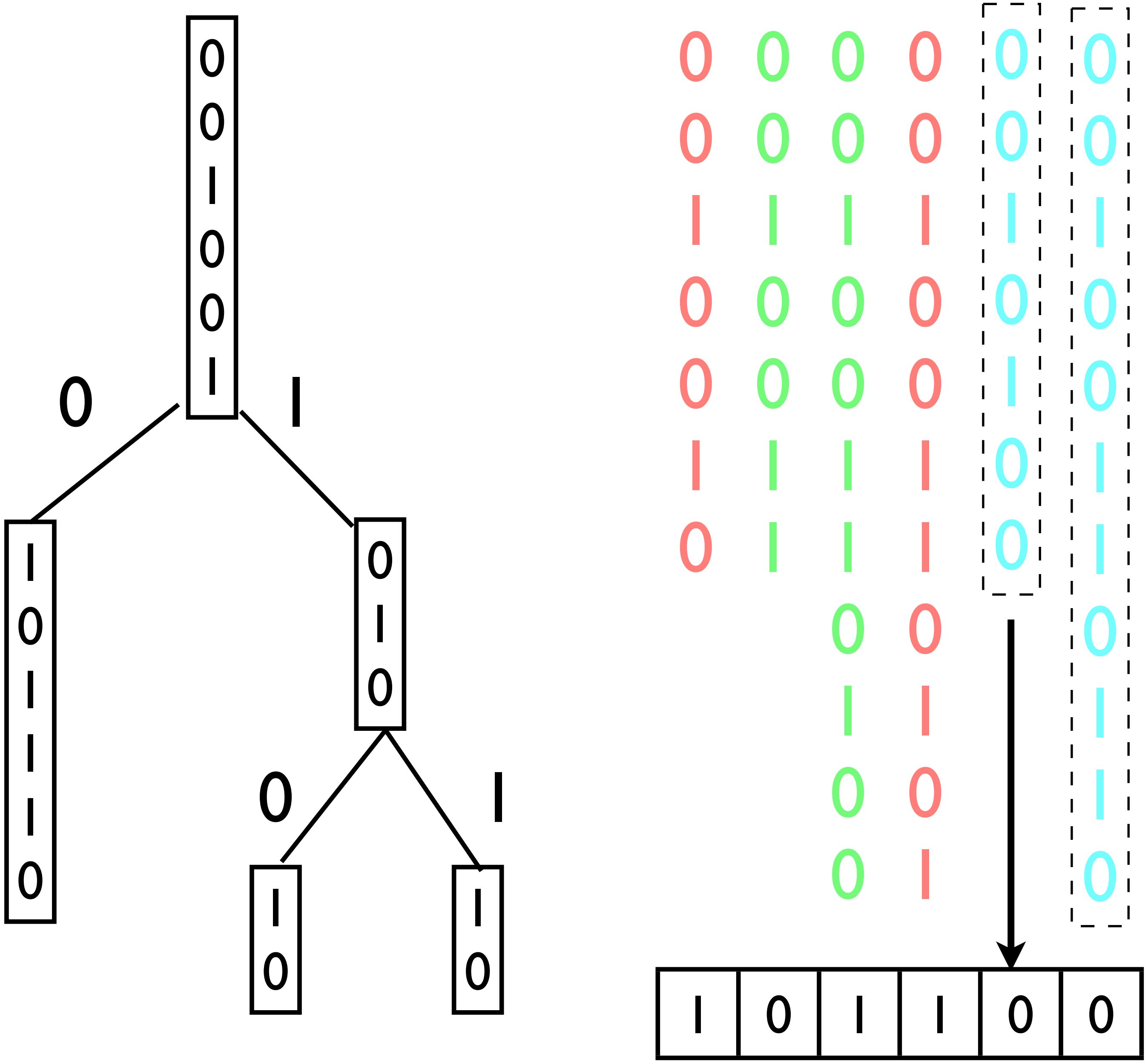

0
1
0
1
1 0 1 1 0 0



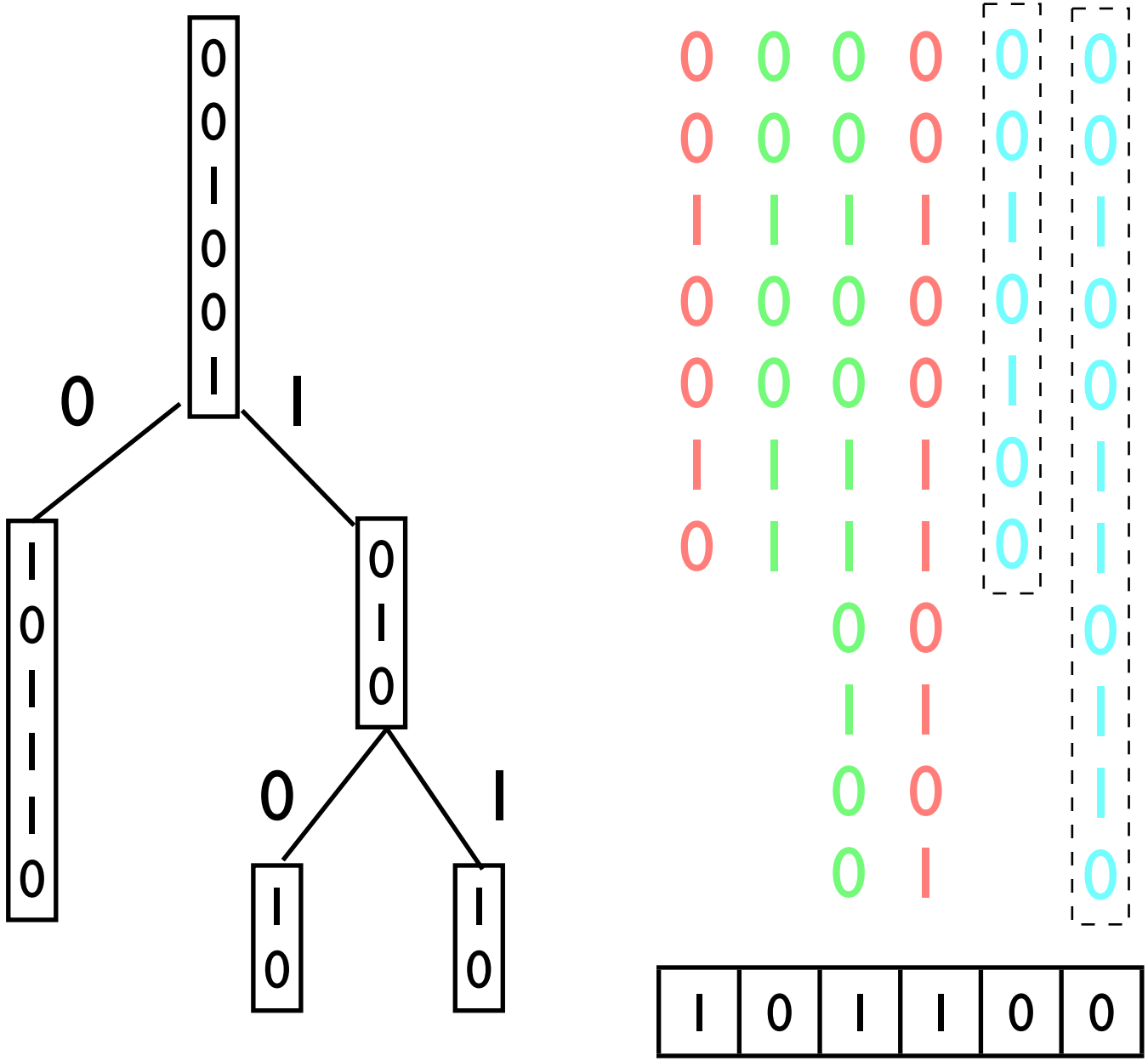
Rank 3 ⇒ bucket 3!



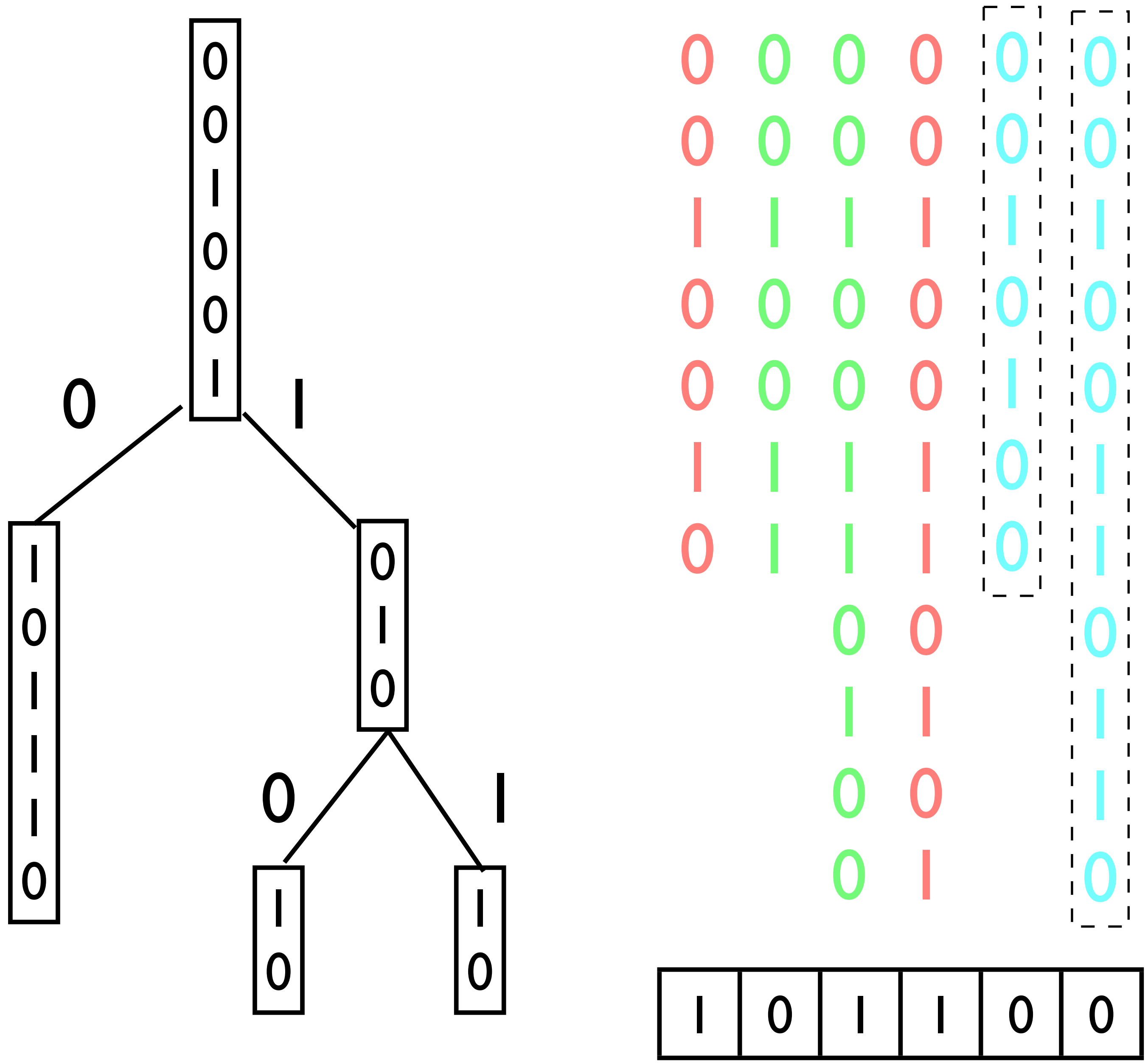
Rank 3 ⇒ bucket 3!

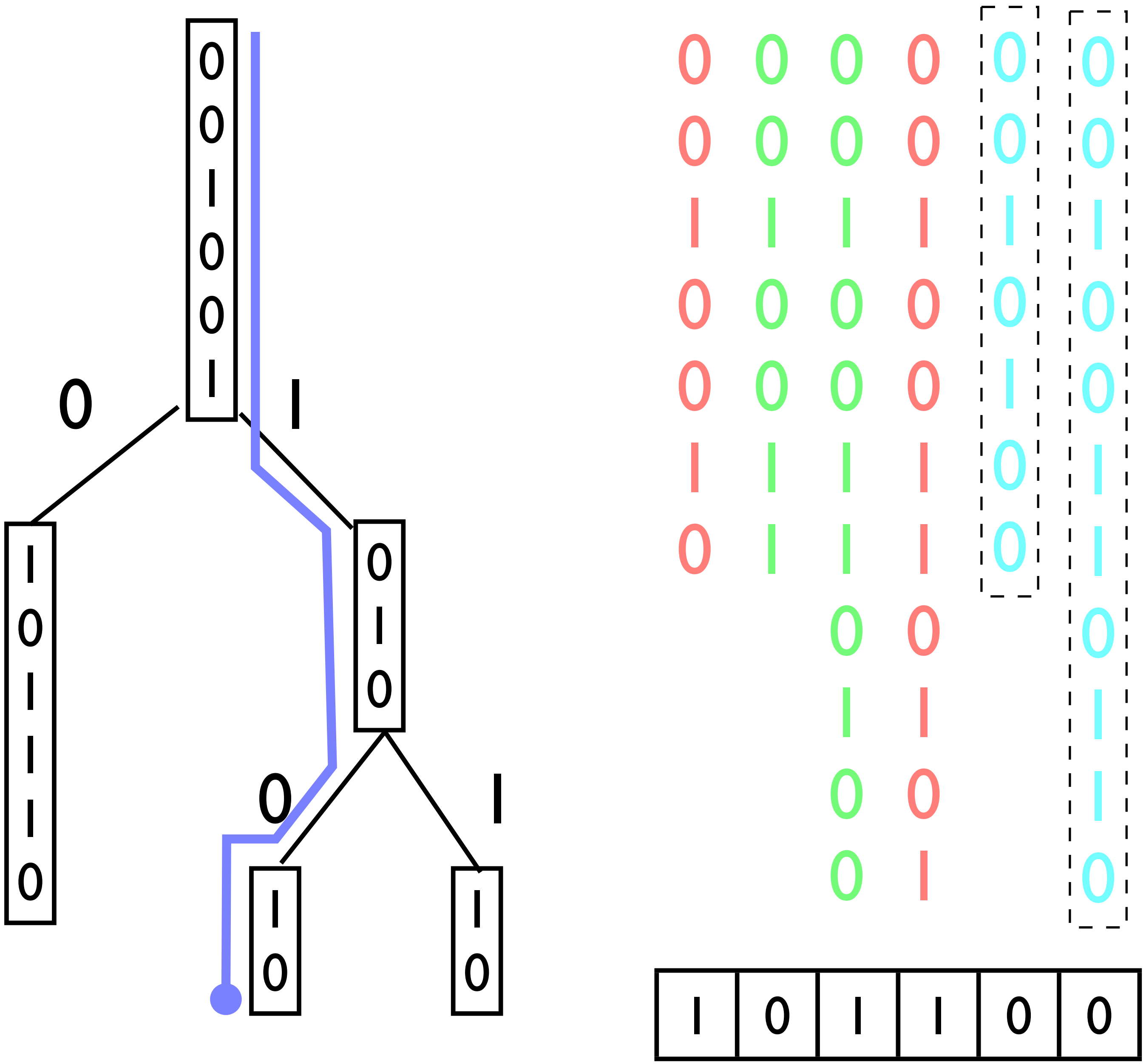


Rank 3 ⇒ bucket 3!

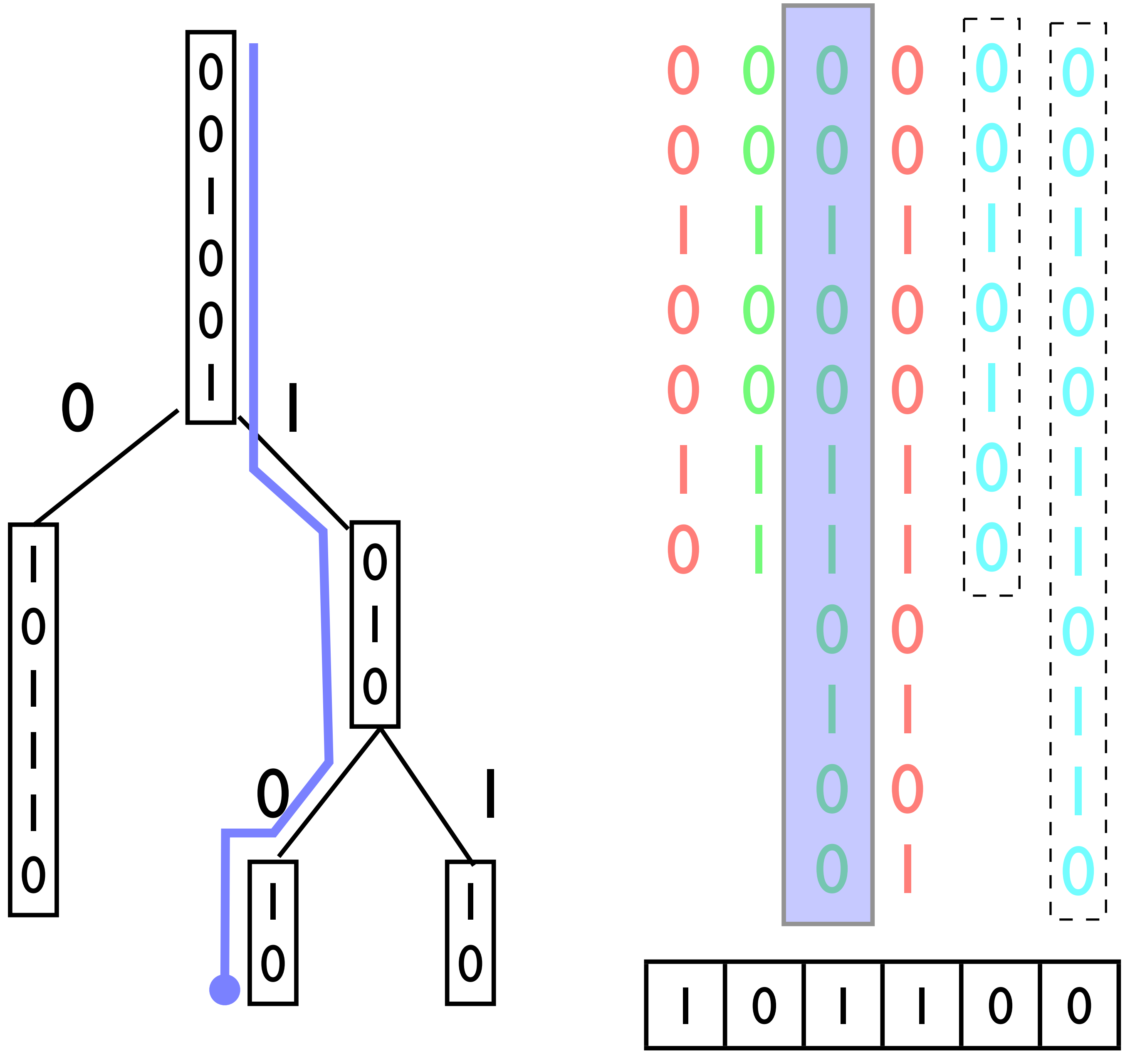


Rank 3 ⇒ bucket 3!

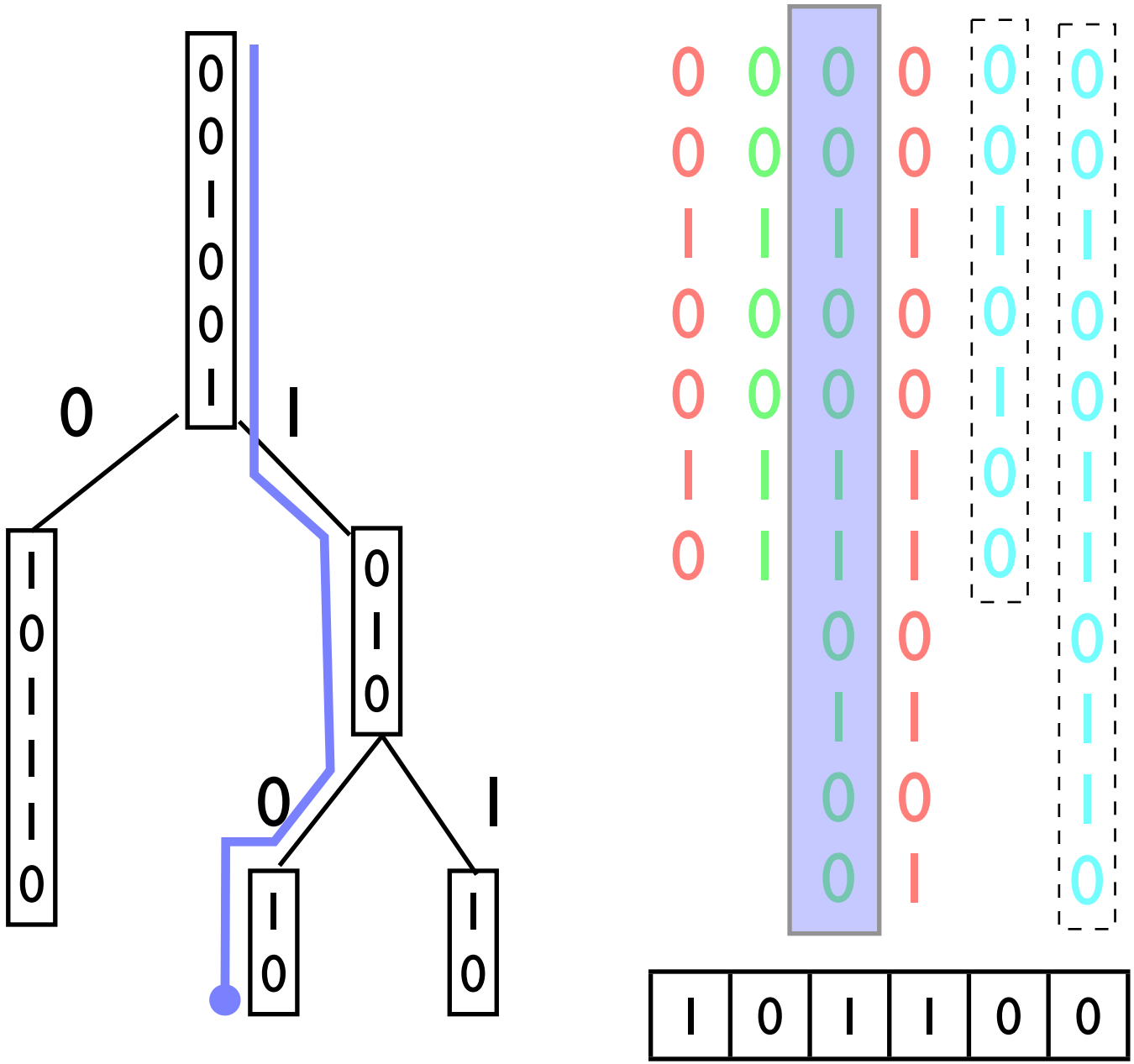

001001
0
1
101110
010
0
1
10
10
0010010
0010011
00100110100
00100110101
0010100
00100110110
1 0 1 1 0 0



001001
0
1
101110
010
0
1
10
10
1 0 1 1 0 0



0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0
0 0 1 0 0 1 0
0 0 1 0 0 1 1
0 0 1 0 0 1 1 0 1 0 0
0 0 1 0 0 1 1 0 1 0 1
0 0 1 0 1 0 0
0 0 1 0 0 1 1 0 1 1 0
1 0 1 1 0 0



Rank 1 ⇒ bucket 1!

Missing Pieces

Missing Pieces

- We still don't know whether we exit on the left or on the right; but since we are just interested in strings in S, we can just store a function $S \to 2$ to record this fact

Missing Pieces

- We still don't know whether we exit on the left or on the right; but since we are just interested in strings in S, we can just store a function $S \to 2$ to record this fact
- We use signatures, not paths, so there will be mistakes: but with log w-sized signatures mistakes are so few that they can be stored in a *dictionary relative to* S in space $O(n)$

Missing Pieces

- We still don't know whether we exit on the left or on the right; but since we are just interested in strings in S, we can just store a function $S \to 2$ to record this fact
- We use signatures, not paths, so there will be mistakes: but with log w-sized signatures mistakes are so few that they can be stored in a *dictionary relative to* S in space $O(n)$
- All in all, by using buckets of size log w we can do monotone minimal perfect hashing in time $O(\log w)$ using $O(n \log \log w)$ bits (i.e., constant cost per key to all practical purposes)

Conclusions

Conclusions

- Monotone minimal perfect hashing is an interesting problem

Conclusions

- Monotone minimal perfect hashing is an interesting problem
- LCP-based hashing and z-fast tries as distributors provide different, but always interesting, time/space tradeoffs

Conclusions

- Monotone minimal perfect hashing is an interesting problem
- LCP-based hashing and z-fast tries as distributors provide different, but always interesting, time/space tradeoffs
- No lower bounds are in sight, except for the trivial ones (those for the non-monotone case)

Conclusions

- Monotone minimal perfect hashing is an interesting problem
- LCP-based hashing and z-fast tries as distributors provide different, but always interesting, time/space tradeoffs
- No lower bounds are in sight, except for the trivial ones (those for the non-monotone case)
- Z-fast tries are the first log *w*-time trie-like structure with (true) linear space occupancy

Conclusions

- Monotone minimal perfect hashing is an interesting problem
- LCP-based hashing and z-fast tries as distributors provide different, but always interesting, time/space tradeoffs
- No lower bounds are in sight, except for the trivial ones (those for the non-monotone case)
- Z-fast tries are the first log *w*-time trie-like structure with (true) linear space occupancy
- Many applications are possible (e.g., Bee-trees, where internal node indexing is made using z-fast tries)

Thanks!

BTW, all this is implemented and working. Please visit

`http://sux4j.dsi.unimi.it/`