SECOND PART
THE MONOTONE CASE

RESULTS FOR GENERAL MPH

RESULTS FOR GENERAL MPH

- General MPH: space lower bound n log e + log log u + O(log n) [Fredman & Komlós 1984] i.e., 1.44 bits/key

RESULTS FOR GENERAL MPH

- General MPH: space lower bound n log e + log log u + O(log n) [Fredman & Komlós 1984] i.e., 1.44 bits/key
- Can be obtained (up to a small additive factor) [Hagerup & Tholey 2004], but the construction is *impractical*

RESULTS FOR GENERAL MPH

- General MPH: space lower bound n log e + log log u + O(log n) [Fredman & Komlós 1984] i.e., 1.44 bits/key
- Can be obtained (up to a small additive factor) [Hagerup & Tholey 2004], but the construction is *impractical*
- Recent constructions [Botelho, Pagh & Ziviani 2007; Belazzougui, Botelho & Dietzfelbinger 2009] get ≈2.5 bits/key (or less)

ORDER-PRESERVING MINIMAL PERFECT HASH FUNCTIONS (OPMPH)

ORDER-PRESERVING MINIMAL PERFECT HASH FUNCTIONS (OPMPH)

- We are given a *known* set $S \subseteq \Omega$, with a *prescribed* order

ORDER-PRESERVING MINIMAL PERFECT HASH FUNCTIONS (OPMPH)

- We are given a *known* set $S \subseteq \Omega$, with a *prescribed* order
- Say $\Omega=[u]$

ORDER-PRESERVING MINIMAL PERFECT HASH FUNCTIONS (OPMPH)

- We are given a *known* set $S \subseteq \Omega$, with a *prescribed* order
- Say $\Omega=[u]$
- We want a function $[u] \rightarrow [n]$ that is an MPH on S *and* respects the order [OPMPH] [Fox et al. 1991]

ORDER-PRESERVING MINIMAL PERFECT HASH FUNCTIONS (OPMPH)

- We are given a *known* set $S \subseteq \Omega$, with a *prescribed* order
- Say $\Omega=[u]$
- We want a function $[u] \rightarrow [n]$ that is an MPH on S *and* respects the order [OPMPH] [Fox et al. 1991]
- There is just *one* such function (intrinsically)

ORDER-PRESERVING MINIMAL PERFECT HASH FUNCTIONS (OPMPH)

- We are given a *known* set $S \subseteq \Omega$, with a *prescribed* order
- Say $\Omega=[u]$
- We want a function $[u] \rightarrow [n]$ that is an MPH on S *and* respects the order [OPMPH] [Fox et al. 1991]
- There is just *one* such function (intrinsically)
- There are n! possibilities, if S is fixed

ORDER-PRESERVING MINIMAL PERFECT HASH FUNCTIONS (OPMPH)

- We are given a *known* set $S \subseteq \Omega$, with a *prescribed* order
- Say $\Omega=[u]$
- We want a function $[u] \rightarrow [n]$ that is an MPH on S *and* respects the order [OPMPH] [Fox et al. 1991]
- There is just *one* such function (intrinsically)
- There are n! possibilities, if S is fixed
  - A space $\Omega(n \log n)$ lower bound gets in the way!

MONOTONE MINIMAL PERFECT HASHING

MONOTONE MINIMAL PERFECT HASHING

- We are given a *known* set $S \subseteq [u]$, with $|S|=n$

MONOTONE MINIMAL PERFECT HASHING

- We are given a *known* set $S \subseteq [u]$, with $|S|=n$
- We want $[u] \rightarrow [n]$ that preserves the lexicographical ordering of S

MONOTONE MINIMAL PERFECT HASHING

- We are given a *known* set $S \subseteq [u]$, with $|S|=n$
- We want $[u] \rightarrow [n]$ that preserves the lexicographical ordering of S
- Reasonable *construction* time & space, little *space*, fast *query* — constant or $O(\log \log u)$

OUR CASE...

OUR CASE...

- MMPH is an intermediate case between MPH and OPMPH

OUR CASE...

- MMPH is an intermediate case between MPH and OPMPH
- Rank elements in S—but you obtain arbitrary results for the other elements! [a very weak indexed dictionary]

OUR CASE...

- MMPH is an intermediate case between MPH and OPMPH
- Rank elements in S—but you obtain arbitrary results for the other elements! [a very weak indexed dictionary]
- The $\Omega(n \log n)$ lower bound does not hold anymore

OUR CASE...

- MMPH is an intermediate case between MPH and OPMPH
- Rank elements in S—but you obtain arbitrary results for the other elements! [a very weak indexed dictionary]
- The $\Omega(n \log n)$ lower bound does not hold anymore
- Here I will present two constructions:

OUR CASE...

- MMPH is an intermediate case between MPH and OPMPH
- Rank elements in S—but you obtain arbitrary results for the other elements! [a very weak indexed dictionary]
- The $\Omega(n \log n)$ lower bound does not hold anymore
- Here I will present two constructions:

	SPACE/KEY	QUERY TIME
LCP	$O(\log \log u)$	$O(1)$
Z-FAST	$O(\log \log \log u)$	$O(\log \log u)$

OUR CASE...

- MMPH is an intermediate case between MPH and OPMPH
- Rank elements in S—but you obtain arbitrary results for the other elements! [a very weak indexed dictionary]
- The $\Omega(n \log n)$ lower bound does not hold anymore
- Here I will present two constructions:

	SPACE/KEY	QUERY TIME
LCP	$O(\log \log u)$	$O(1)$
Z-FAST	$\approx O(1)$	$O(\log \log u)$

MOTIVATIONS

MOTIVATIONS

- A quite common case, in many practical scenarios

MOTIVATIONS

- A quite common case, in many practical scenarios
  - List of terms for (distributed) indices

MOTIVATIONS

- A quite common case, in many practical scenarios
  - List of terms for (distributed) indices
  - List of URLs when you compress a graph

MOTIVATIONS

- A quite common case, in many practical scenarios
  - List of terms for (distributed) indices
  - List of URLs when you compress a graph
- A further step on the line of Yao's classical problem ("Should tables be sorted?"):

MOTIVATIONS

- A quite common case, in many practical scenarios
  - List of terms for (distributed) indices
  - List of URLs when you compress a graph
- A further step on the line of Yao's classical problem ("Should tables be sorted?"):
  - with O(log log u) additional bits, you need just one access to a sorted table to answer rank queries

PREVIOUS WORK

PREVIOUS WORK

- Surprisingly little (general MPH, which is too weak, and indexed dictionaries, which use too much space)

PREVIOUS WORK

- Surprisingly little (general MPH, which is too weak, and indexed dictionaries, which use too much space)
- [Majewski, Wormald, Havas & Czech 1996] give a general 3-hypergraph based technique needing 1.23n log n bits (for OPMPH)

INGREDIENTS

INGREDIENTS

- A *sparse array* $S \rightarrow [2^r]$ can be stored using $nr+o(n)$ bits (constant query time)

INGREDIENTS

Note: many names and implementations available ("sparse functons", "retrieval data structures", "Bloomier filters")

- A *sparse array* $S \to [2^r]$ can be stored using $nr+o(n)$ bits (constant query time)

INGREDIENTS

Note: many names and implementations available ("sparse functons", "retrieval data structures", "Bloomier filters")

- A *sparse array* $S \rightarrow [2^r]$ can be stored using $nr+o(n)$ bits (constant query time)
  - Practical implementation uses 1.23 additional bits/key (a compacted variation of the Majewski et al.)

INGREDIENTS

Note: many names and implementations available ("sparse functons", "retrieval data structures", "Bloomier filters")

- A *sparse array* $S \rightarrow [2^r]$ can be stored using $nr+o(n)$ bits (constant query time)
  - Practical implementation uses 1.23 additional bits/key (a compacted variation of the Majewski et al.)

INGREDIENTS

Note: many names and implementations available ("sparse functons", "retrieval data structures", "Bloomier filters")

- A *sparse array* $S \rightarrow [2^r]$ can be stored using $nr+o(n)$ bits (constant query time)
  - Practical implementation uses 1.23 additional bits/key (a compacted variation of the Majewski et al.)
  - Other recent resuls [Dietzfelbinger & Pagh 2008], [Porat 2008], [Chazelle, Kilian, Rubinfeld, & Tal 2004]

INGREDIENTS

Note: many names and implementations available ("sparse functons", "retrieval data structures", "Bloomier filters")

- A *sparse array* $S \rightarrow [2^r]$ can be stored using $nr+o(n)$ bits (constant query time)
  - Practical implementation uses 1.23 additional bits/key (a compacted variation of the Majewski et al.)
  - Other recent resuls [Dietzfelbinger & Pagh 2008], [Porat 2008], [Chazelle, Kilian, Rubinfeld, & Tal 2004]
- We use static *rank/select* structures

INGREDIENTS

Note: many names and implementations available ("sparse functons", "retrieval data structures", "Bloomier filters")

- A *sparse array* $S \rightarrow [2^r]$ can be stored using $nr+o(n)$ bits (constant query time)
  - Practical implementation uses 1.23 additional bits/key (a compacted variation of the Majewski et al.)
  - Other recent resuls [Dietzfelbinger & Pagh 2008], [Porat 2008], [Chazelle, Kilian, Rubinfeld, & Tal 2004]
- We use static *rank/select* structures
  - You can rank/select on an n-bit vector with $o(n)$ additional bits

BASIC IDEA: BUCKETING

BASIC IDEA: BUCKETING

- *Bucketing*: splitting S into m buckets $B_0, B_1, \ldots, B_{m-1}$

BASIC IDEA: BUCKETING

- *Bucketing*: splitting S into m buckets $B_0, B_1, \ldots, B_{m-1}$
- A monotone distributor $d: S \rightarrow m$

BASIC IDEA: BUCKETING

- *Bucketing*: splitting S into m buckets $B_0, B_1, \ldots, B_{m-1}$
- A monotone distributor $d: S \rightarrow m$
- A MMPH $g_i: B_i \rightarrow |B_i|$

BASIC IDEA: BUCKETING

- *Bucketing*: splitting S into m buckets $B_0, B_1, \ldots, B_{m-1}$
- A monotone distributor $d: S \rightarrow m$
- A MMPH $g_i: B_i \rightarrow |B_i|$
- A function $s: m \rightarrow n$ such that $s(i) = \sum_{i<j} |B_j|$

BASIC IDEA: BUCKETING

- *Bucketing*: splitting S into m buckets $B_0, B_1, \ldots, B_{m-1}$
- A monotone distributor $d: S \rightarrow m$
- A MMPH $g_i: B_i \rightarrow |B_i|$
- A function $s: m \rightarrow n$ such that $s(i) = \sum_{i<j} |B_j|$
- Then, $h(x)=s(d(x))+g_{d(x)}(x)$ is a MMPH function on S

FIRST CONSTRUCTION

LONGEST COMMON PREFIXES

LONGEST COMMON PREFIXES

- Divide S into buckets of *equal size*

LONGEST COMMON PREFIXES

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

LONGEST COMMON PREFIXES

- Divide S into buckets of *equal size*
- For each bucket, compute the LCP
- **Lemma**: all LCPs are distinct (if p is the LCP of a buckets, it contains p0… and p1…)

LONGEST COMMON PREFIXES

- Divide S into buckets of *equal size*
- For each bucket, compute the LCP
- **Lemma**: all LCPs are distinct (if p is the LCP of a buckets, it contains p0… and p1…)
- Identify a bucket by its LCP

LONGEST COMMON PREFIXES

- Divide S into buckets of *equal size*
- For each bucket, compute the LCP
- **Lemma**: all LCPs are distinct (if p is the LCP of a buckets, it contains p0… and p1…)
- Identify a bucket by its LCP
- Compute the LCP of the bucket of a key storing a map from S to w = log u

000100100000000
001001010110 0
001001010111 0
001001100000 0
001001100100 0
001001101001 0
001001101010 0
001001101010 1
001001101011 0
001001111011 0
010010001 0000

```
     0001001000000
     0010010101100
     0010010101110
  ----------------
     0010011000000
     0010011001000
     0010011010010
  ----------------
     0010011010100
     0010011010101
     0010011010110
  ----------------
     0010011110110
     0100100010000
```

```
  000100100000000
  001001010110 0
  001001010111 0
 ---------------
  001001100000000
  001001100100 0
  001001101001 0
 ---------------
  001001101010 0
  001001101010 1
  001001101011 0
 ---------------
  001001111011 0
  010010001 0000
```

```
 000100100000000
 001001010110 0
 001001010111 0
 ---------------
 001001100000000
 001001100100 0
 001001101001 0
 ---------------
 001001101010 0
 001001101010 1
 001001101011 0
 ---------------
 001001111011 0
 010010001 0000
```

00→0

```
 000100100000000
 001001010101100
 001001010111110
 ---------------
 001001100000000
 001001100101000
 001001101001010
 ---------------
 001001101010100
 001001101010101
 001001101010110
 ---------------
 001001111011110
 010010001010000
```

00→0

```
000100100000000
001001010110 0
001001010111 0
```

00→0

```
001001100000000
001001100 01000
001001100 10010
```

00100110→1

```
001001101010 0
001001101010 1
001001101010 1
```

```
001001110110
010010001 0000
```

```
000100100000000
001001010110 0
001001010111 0
```

00→0

```
001001100000000
001001100 01000
001001100 10010
```

00100110→1

```
001001101010 0
001001101010 1
001001101010 1 0
```

```
001001110110
010010001 0000
```

```
000100100000000
001001010110 0
001001010111 0
```

00→0

```
001001100000000
001001100 01000
001001100 10010
```

00100110→1

```
001001101010 00
001001101010 1
001001101010 10
```

001001101010→2

```
001001111011 0
010010001 0000
```

000100100000000
001001010110 0
001001010111 0

00 → 0

001001100000000
001001100 1000
001001100 10010

00100110 → 1

001001101010 0
001001101010 1
001001101010 1 0

0010011101101 → 2

001001111011 0
010010000 10000

000100100000000

001001010110 0

001001010111 0

001001100000000

001001100 1000

001001100 10010

001001101010 0

001001101010 1

001001101010 1 0

001001111011 0

010010000 10000

00→0

00100110→1

001001101010 1→2

0→3

LCP-to-bucket

000100100000 00	00→0
001001010100	
001001010110	
001001100000 00	00100110→1
001001101000	
001001101001 0	
001001101010 00	001001101010→2
001001101010 01	
001001101010 10	
001001111011 0	0→3
010010000100 00	

LCP-to-bucket

2	0001001000000	00→0
2	0010010101100	
2	0010010101110	
8	0010011000000	00100110→1
8	0010011001000	
8	0010011010010	
11	0010011010100	001001101010→2
11	0010011010101	
11	0010011010110	
1	0010011110110	0→3
1	0100100010000	

key-to-LCP-length		LCP-to-bucket
2	0001001000000	00→0
2	0010010101100	
2	0010010101110	
8	0010011000000	00100110→1
8	0010011001000	
8	0010011010010	
11	0010011010100	001001101010→2
11	0010011010101	
11	0010011010110	
1	0010011110110	0→3
1	0100100010000	

key-to-LCP-length			LCP-to-bucket
2	000100100000 0	0	00→0
2	001001010110 0	1	
2	001001010111 0	2	
8	001001100000 0	0	00100110→1
8	001001101000	1	
8	001001101001 0	2	
11	001001101010 0	0	001001101010→2
11	001001101010 1	1	
11	001001101010 1 0	2	
1	001001111011 0	0	0→3
1	010010001000 0	1	

key-to-LCP-length		key-to-offset	LCP-to-bucket
2	0001001000000	0	00→0
2	0010010101100	1	
2	0010010101110	2	
8	0010011000000	0	00100110→1
8	0010011001000	1	
8	0010011010010	2	
11	0010011010100	0	001001101010→2
11	0010011010101	1	
11	0010011010110	2	
1	0010011110110	0	0→3
1	0100100010000	1	

key-to-LCP-length		key-to-offset	LCP-to-bucket
2	000100100000 0	0	00→0
2	001001010110 0	1	
2	001001010111 0	2	
8	001001100000 0	0	00100110→1
8	001001100100 0	1	
8	001001101001 0	2	
11	001001101010 0	0	001001101010→2
11	001001101010 1	1	
11	001001101010 1 0	2	
1	001001111011 0	0	0→3
1	010010001000 0	1	

key-to-LCP-length		key-to-offset	LCP-to-bucket
2	000100100000 0	0	00→0
2	001001010110 0	1	
2	001001010111 0	2	
8	001001100000 0	0	00100110→1
8	001001100100 0	1	
8	001001101001 0	2	
11	001001101010 0	0	00100110101→2
11	001001101010 1	1	
11	001001101010 1 0	2	
1	001001111011 0	0	0→3
1	010010001000 0	1	

key-to-LCP-length		key-to-offset	LCP-to-bucket
2	000100100000 0	0	00→0
2	001001010110 0	1	
2	001001010111 0	2	
8	001001100000 0	0	00100110→1
8	001001100100 0	1	
8	001001100100 10	2	
11	001001101010 0	0	00100110101→2
11	001001101010 1	1	
11	001001101010 10	2	
1	001001111011 0	0	0→3
1	010010001000 0	1	

key-to-LCP-length		key-to-offset	LCP-to-bucket
2	000100100000 0	0	00→0
2	001001010110 0	1	
2	001001010111 0	2	
8	001001100000 0	0	00100110→1
8	001001100100 0	1	
8	001001100100 10	2	
11	001001101010 0	0	001001101010→2
11	001001101010 1	1	
11	001001101010 10	2	
1	001001111011 0	0	0→3
1	010010001000 0	1	

key-to-LCP-length		key-to-offset	LCP-to-bucket
2	000100100000 0	0	00→0
2	001001010110 0	1	
2	001001010111 0	2	
8	001001100000 0	0	00100110→1
8	001001100100 0	1	
8	001001100100 10	2	
11	001001101010 0	0	00100110101→2
11	001001101010 1	1	
11	001001101010 10	2	
1	001001111011 0	0	0→3
1	010010001000 0	1	

$$1 * 3 + 2 = 5$$

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?
- Remember that there are n/log n buckets…

WHAT DO WE GET?

- If buckets are sized as log n?
- Remember that there are n/log n buckets…
- Also: log log u=number of bits required to write a string length

key-to-LCP-length		key-to-offset	LCP-to-bucket
2	000100100000 0	0	00→0
2	001001010110 0	1	
2	001001010111 0	2	
8	001001100000 0	0	00100110→1
8	001001100100 0	1	
8	001001100100 10	2	
11	001001101010 0	0	00100110101→2
11	001001101010 1	1	
11	001001101010 10	2	
1	001001111011 0	0	0→3
1	010010001000 0	1	

$$1 * 3 + 2 = 5$$

key-to-LCP-length		key-to-offset	LCP-to-bucket
2	0001001000000	0	00→0
2	0010010101100	1	
	0010010101110	2	
	0010011000000	0	00100110→1
	0010011001000	1	
8	0010011010010	2	
11	0010011010100	0	001001101010→2
11	0010011010101	1	
11	0010011010110	2	
1	0010011110110	0	0→3
1	0100100010000	1	

Keys: n
Values: log log u bits
O(n log log u)

$$1 * 3 + 2 = 5$$

key-to-LCP-length		key-to-offset	LCP-to-bucket
2	0001001000000	0	00→0
2	0010010101100		
	0010010101110		
	0010011000000		00100110→1
	0010011000100	1	
8	0010011010010	2	
11	0010011010100	0	001001101010→2
11	0010011010101	1	
11	0010011010110	2	
1	0010011110110	0	0→3
1	0100100010000	1	

Keys: n
Values: log log u bits
O(n log log u)

Keys: n
Values: log log n bits
<O(n log log u)

$$1 * 3 + 2 = 5$$

key-to-LCP-length		key-to-offset	LCP-to-bucket
2	0001001000000	0	00→0
2	0010010101100		
	0010010101110		
	0010011000000		00
	0010011001010	1	
8	0010011010010	2	
11	0010011010100	0	0010011010101→2
11	0010011010101	1	
11	0010011010110	2	
1	0010011110110	0	0→3
1	0100100010000	1	

Keys: n
Values: log log u bits
O(n log log u)

Keys: n
Values: log log n bits
<O(n log log u)

Keys: n/log n
Values: log (n/log n) bits
<O(n log log u)

$1 * 3 + 2 = 5$

WHAT DO WE GET?

WHAT DO WE GET?

- All in all, if buckets are sized as log n...

WHAT DO WE GET?

- All in all, if buckets are sized as log n...
  - ...we get constant-time monotone minimal perfect hashing in O(n log w)=O(n log log u) bits

SECOND CONSTRUCTION

CAN WE DO BETTER?

CAN WE DO BETTER?

- Not *absolutely* better — but a different space/time tradeoff (using a“fast trie”) with O(log log w) space and query time O(log w)

CAN WE DO BETTER?

w=log u

- Not *absolutely* better — but a different space/time tradeoff (using a“fast trie”) with O(log log w) space and query time O(log w)

CAN WE DO BETTER?

w=log u

- Not *absolutely* better — but a different space/time tradeoff (using a“fast trie”) with O(log log w) space and query time O(log w)
- A delimiter set D—the set containing, for each bucket, the last string

CAN WE DO BETTER?

w=log u

- Not *absolutely* better — but a different space/time tradeoff (using a"fast trie") with O(log log w) space and query time O(log w)
- A delimiter set D—the set containing, for each bucket, the last string
- A trie on D would be fine, but too big and too slow!

CAN WE DO BETTER?

w=log u

- Not *absolutely* better — but a different space/time tradeoff (using a "fast trie") with O(log log w) space and query time O(log w)
- A delimiter set D—the set containing, for each bucket, the last string
- A trie on D would be fine, but too big and too slow!
- Use a *z-fast* trie on D as a distributor!

CAN WE DO BETTER?

w=log u

- Not *absolutely* better — but a different space/time tradeoff (using a"fast trie") with O(log log w) space and query time O(log w)
- A delimiter set D—the set containing, for each bucket, the last string
- A trie on D would be fine, but too big and too slow!
- Use a *z-fast* trie on D as a distributor!
- The structure is of independent interest

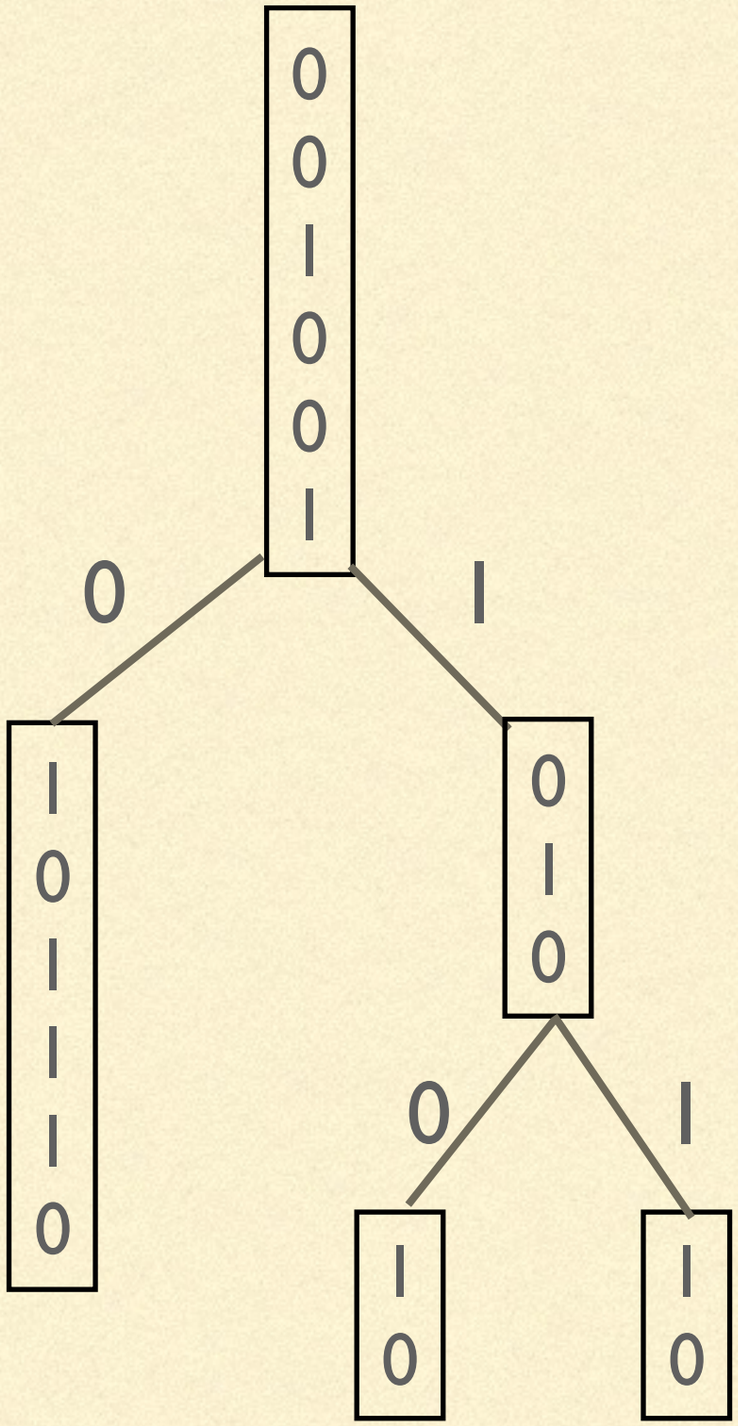
000100100000000
001001010110 0
001001010111 0
001001100000 0
001001100100 0
001001101001 0
001001101010 0
001001101010 1
001001101011 0
001001111011 0
010010001 0000

```
     0001001000000
     0010010101100
     0010010101110
  ----------------
     0010011000000
     0010011001000
     0010011010010
  ----------------
     0010011010100
     0010011010101
     0010011010110
  ----------------
     0010011110110
     0100100010000
```

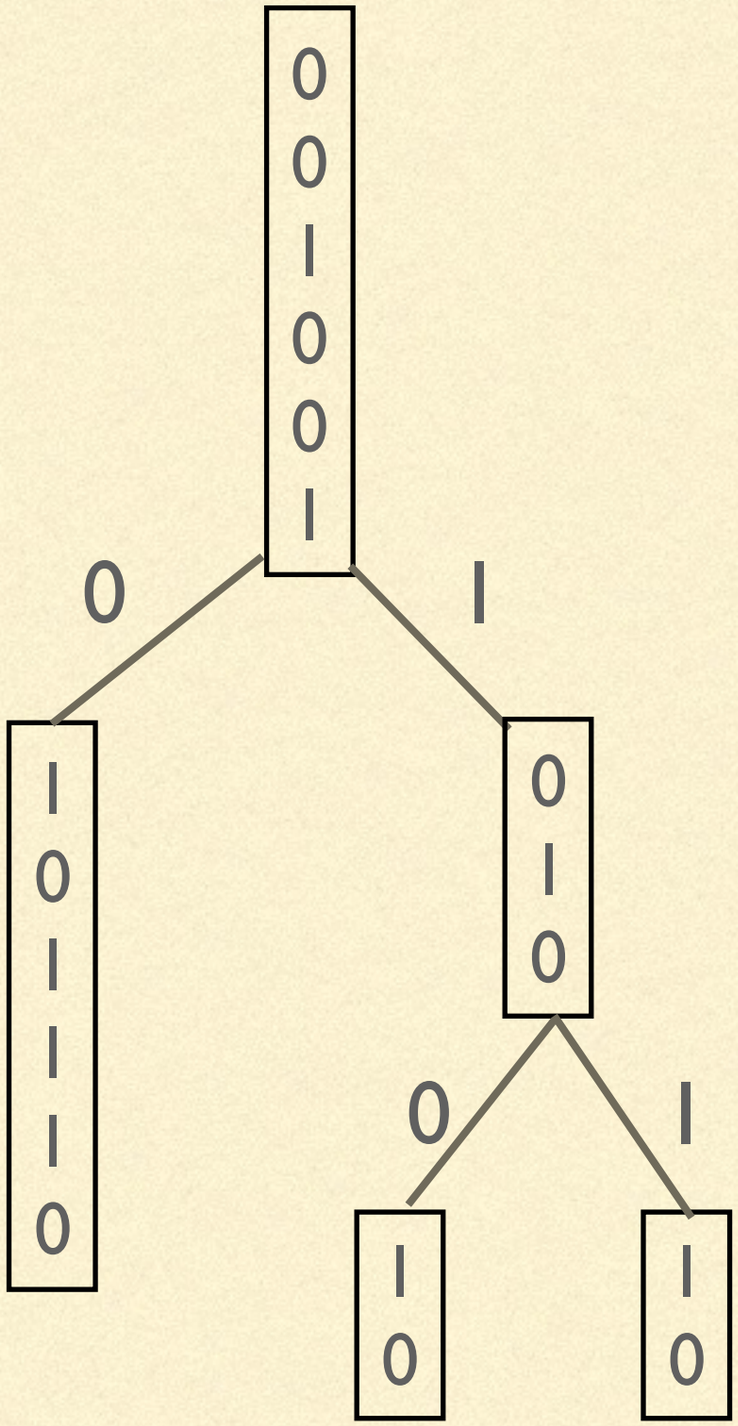
```
  0001001000000
  0010010101100
  0010010101110
---------------
  0010011000000
  0010011001000
  0010011010010
---------------
  0010011010100
  0010011010101
  0010011010110
---------------
  0010011110110
  0100100010000
```

```
  0001001000000
  0010010101100
  0010010101110
---------------
  0010011000000
  0010011001000
  0010011010010
---------------
  0010011010100
  0010011010101
  0010011010110
---------------
  0010011110110
  0100100010000
```

```
  0001001000000
  0010010101100
  0010010101110
---------------
  0010011000000
  0010011001000
  0010011010010
---------------
  0010011010100
  0010011010101
  0010011010110
---------------
  0010011110110
  0100100010000
```

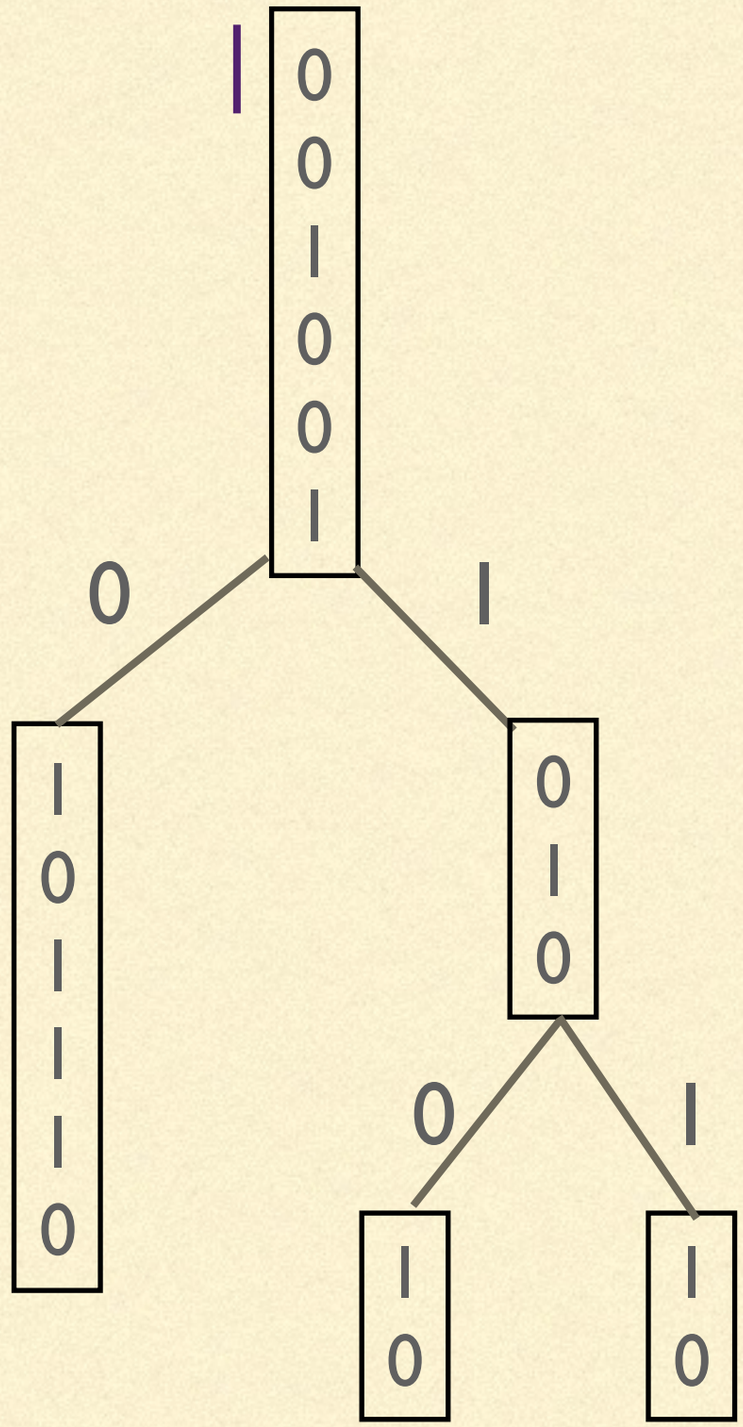


Exit node & exit direction



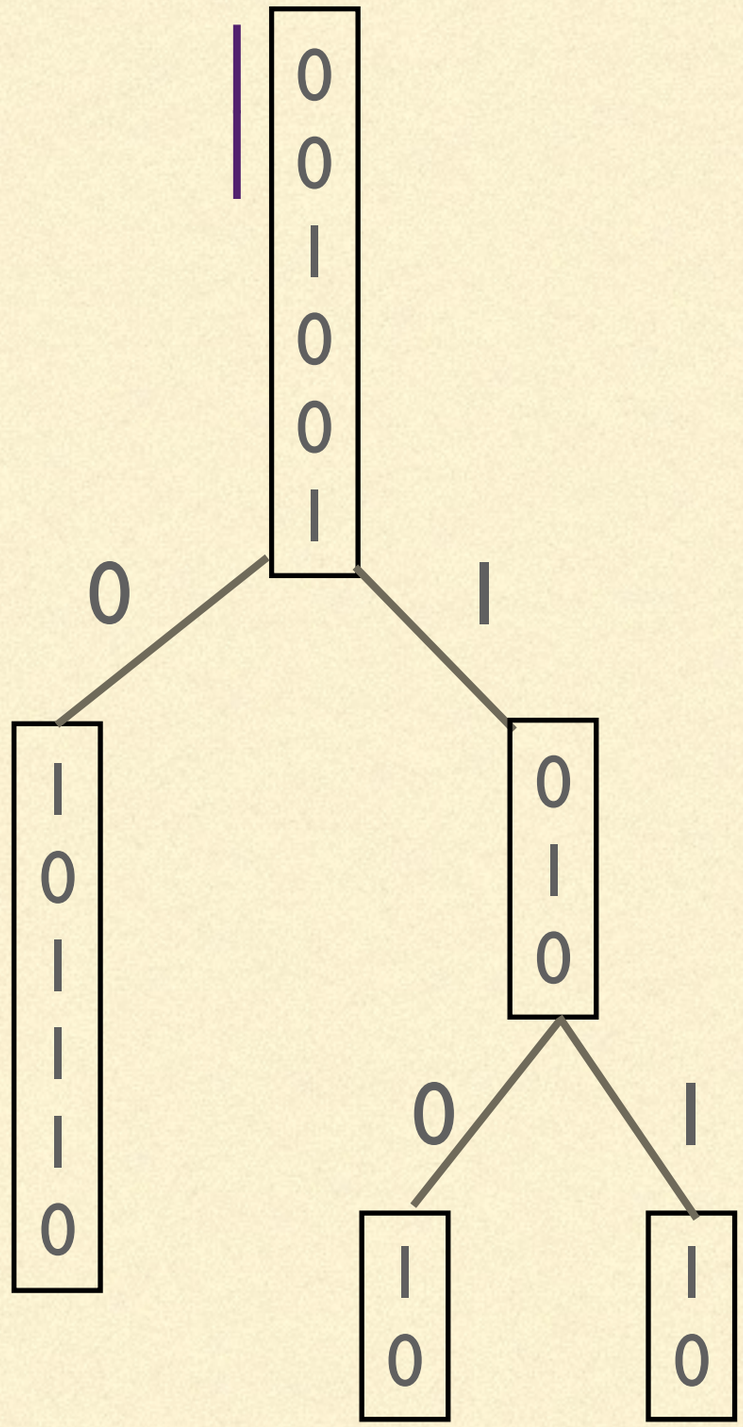
Exit node & exit direction

x = 001001101001 1



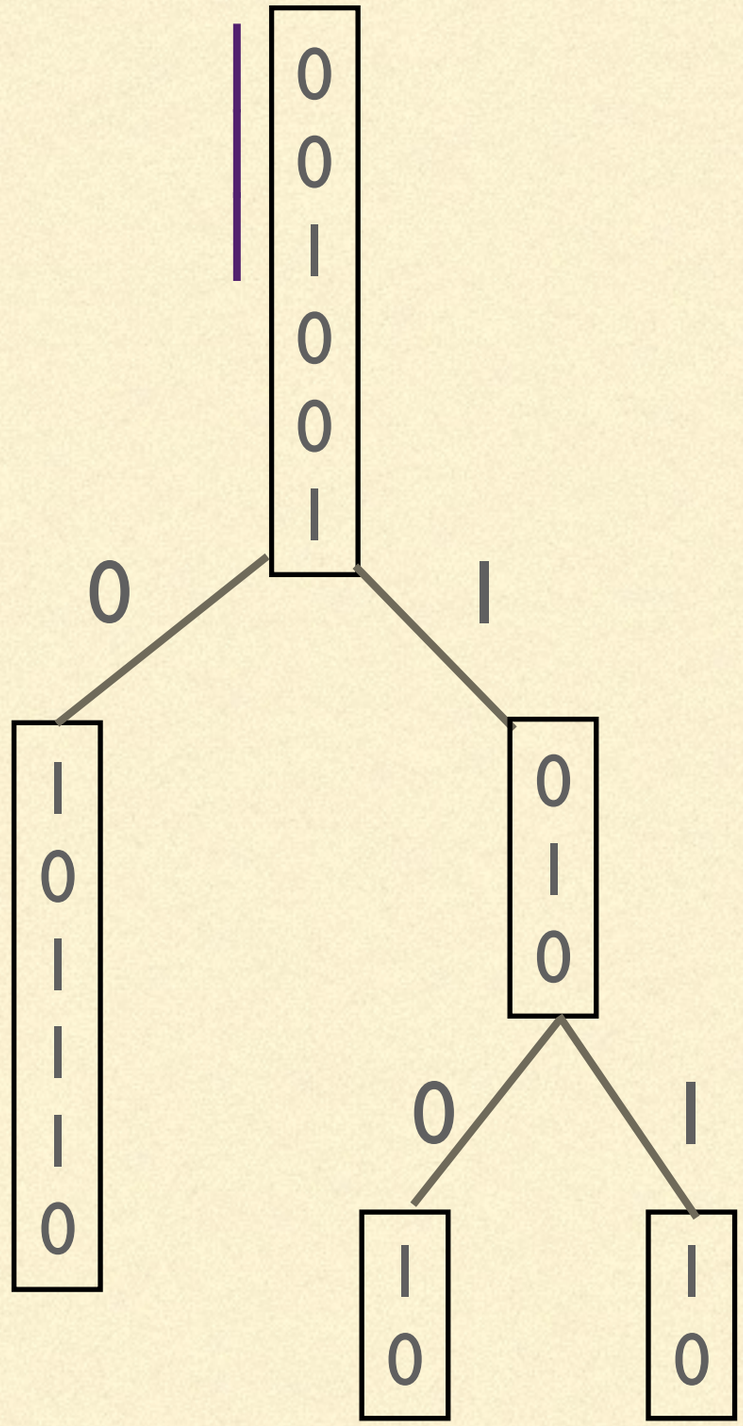
Exit node & exit direction

x = 0010011010011



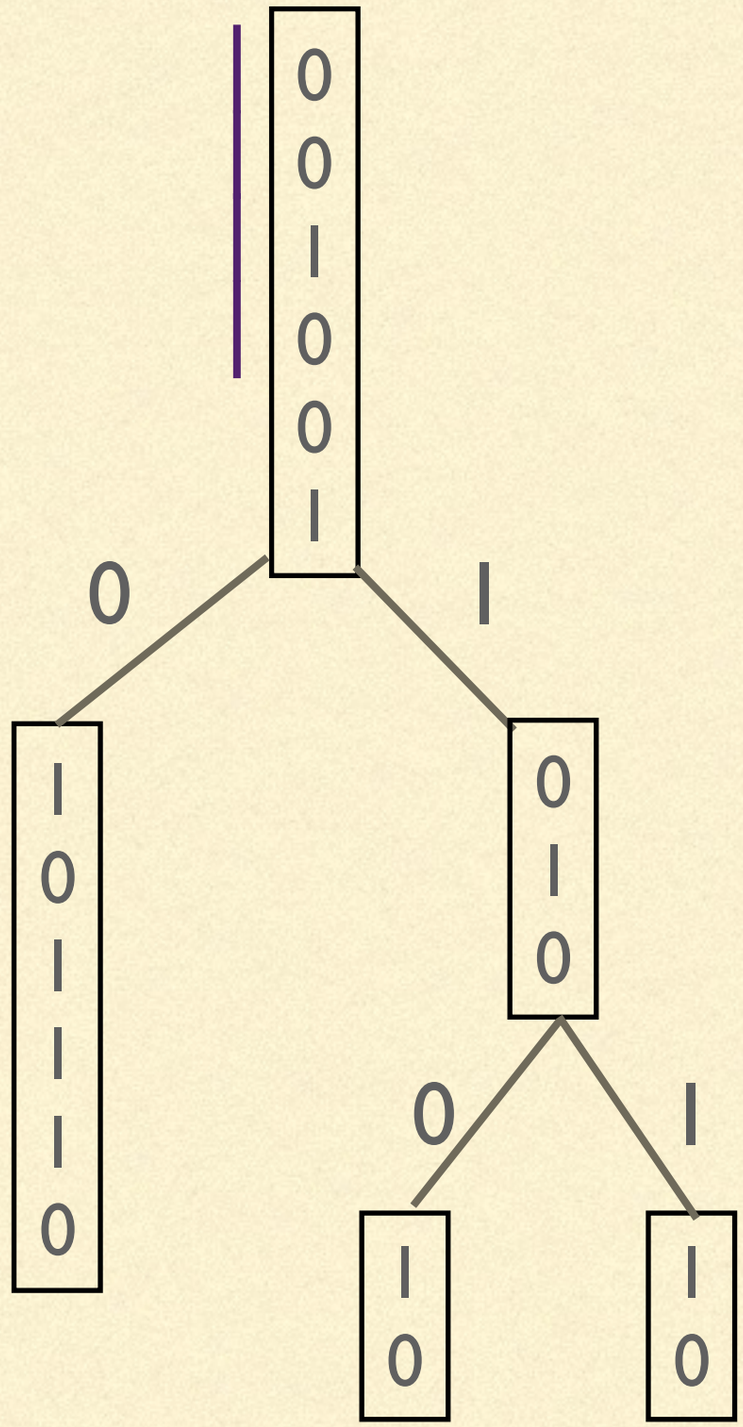
Exit node & exit direction

x = 001001101001 1



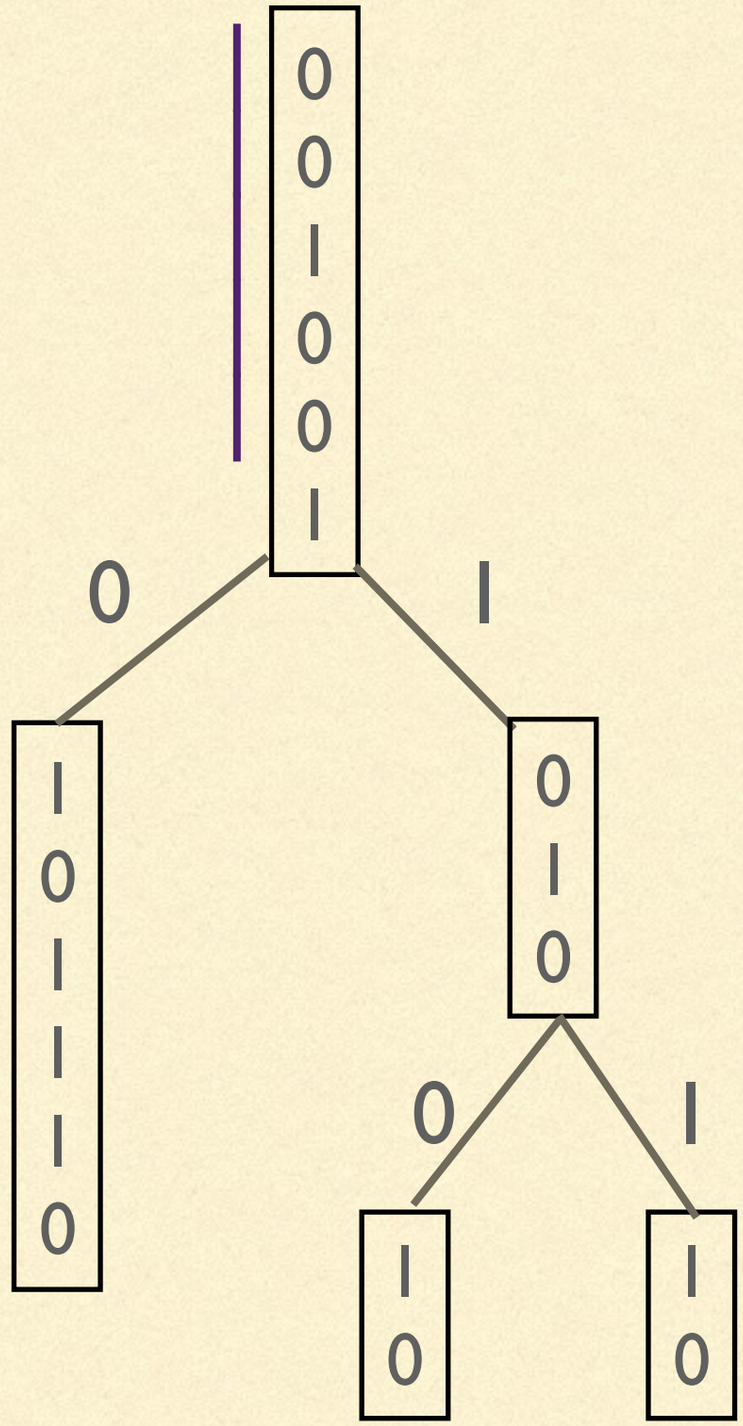
Exit node & exit direction

x = 001001101001 1



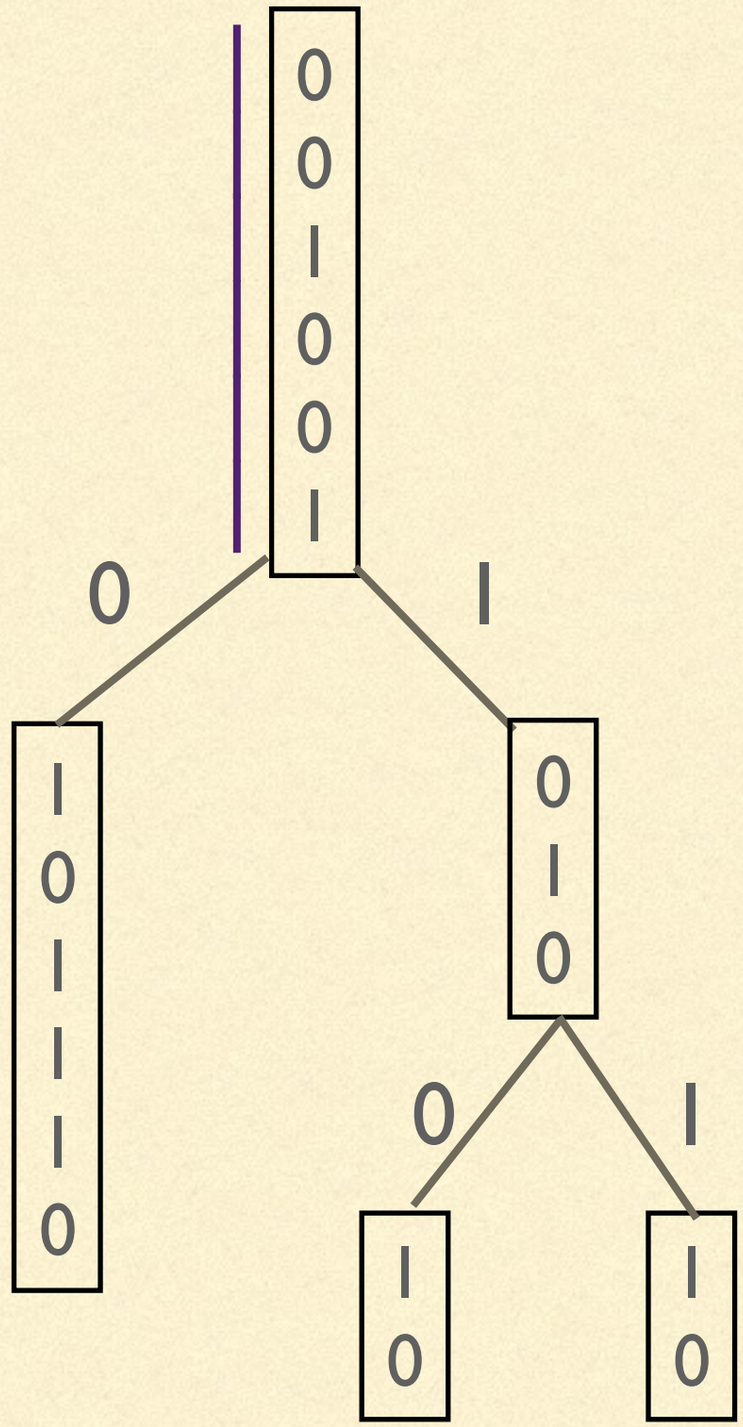
Exit node & exit direction

x = 0010011010011



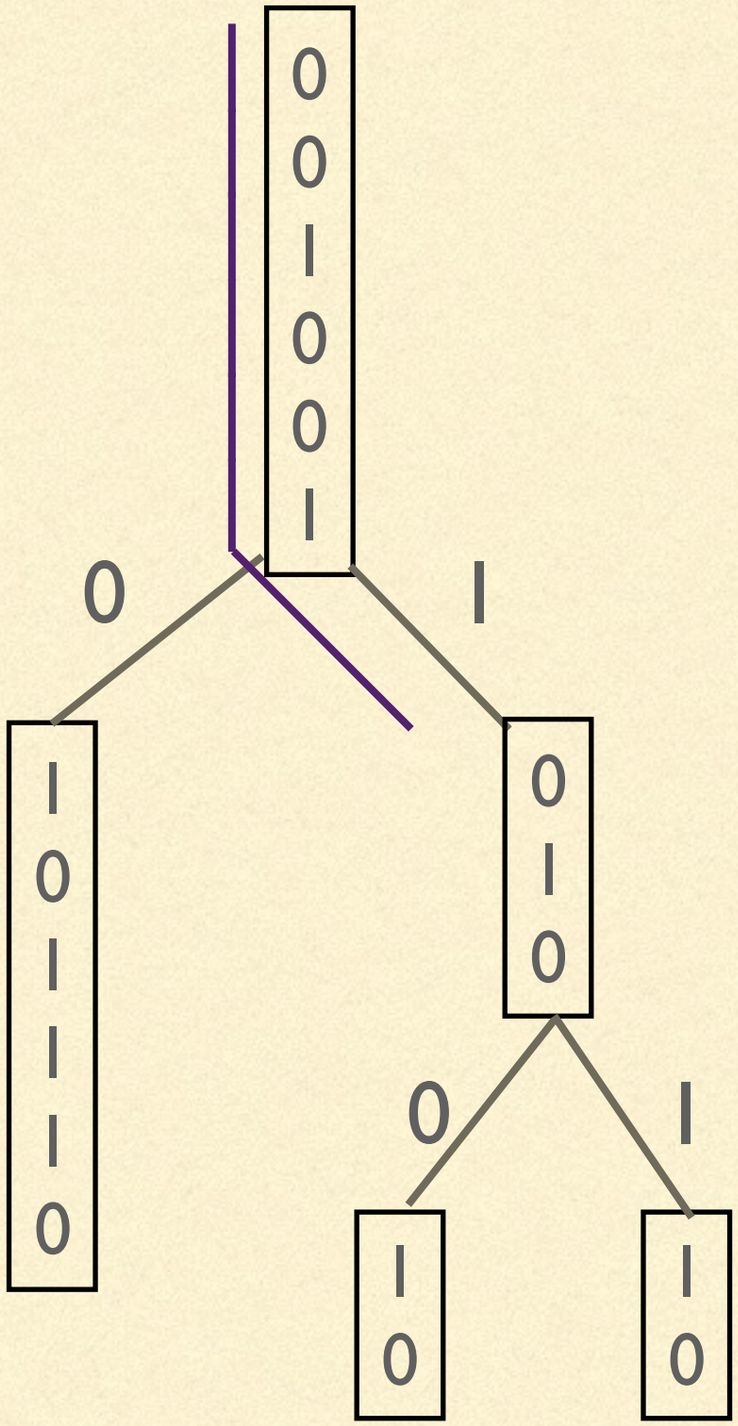
Exit node & exit direction

x = 00100110100 11



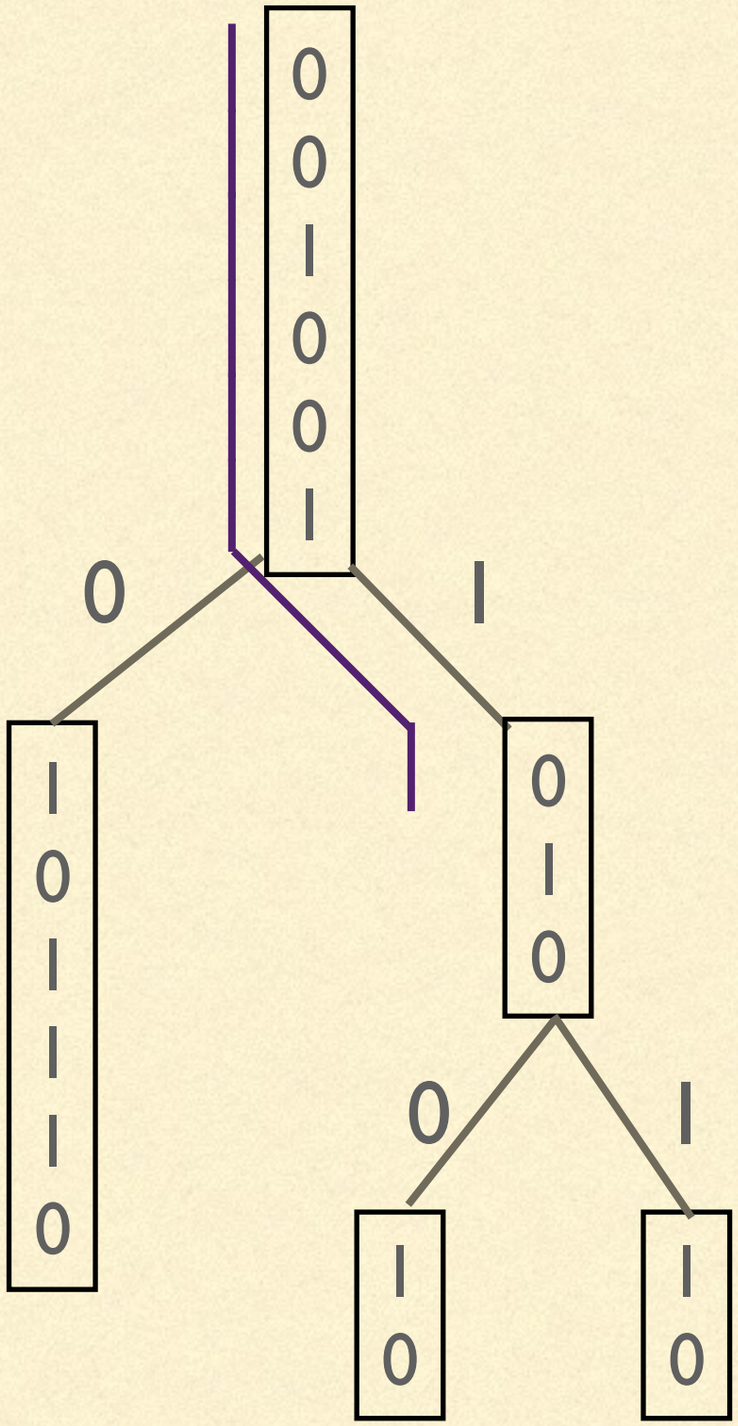
Exit node & exit direction

x = 001001101001 1



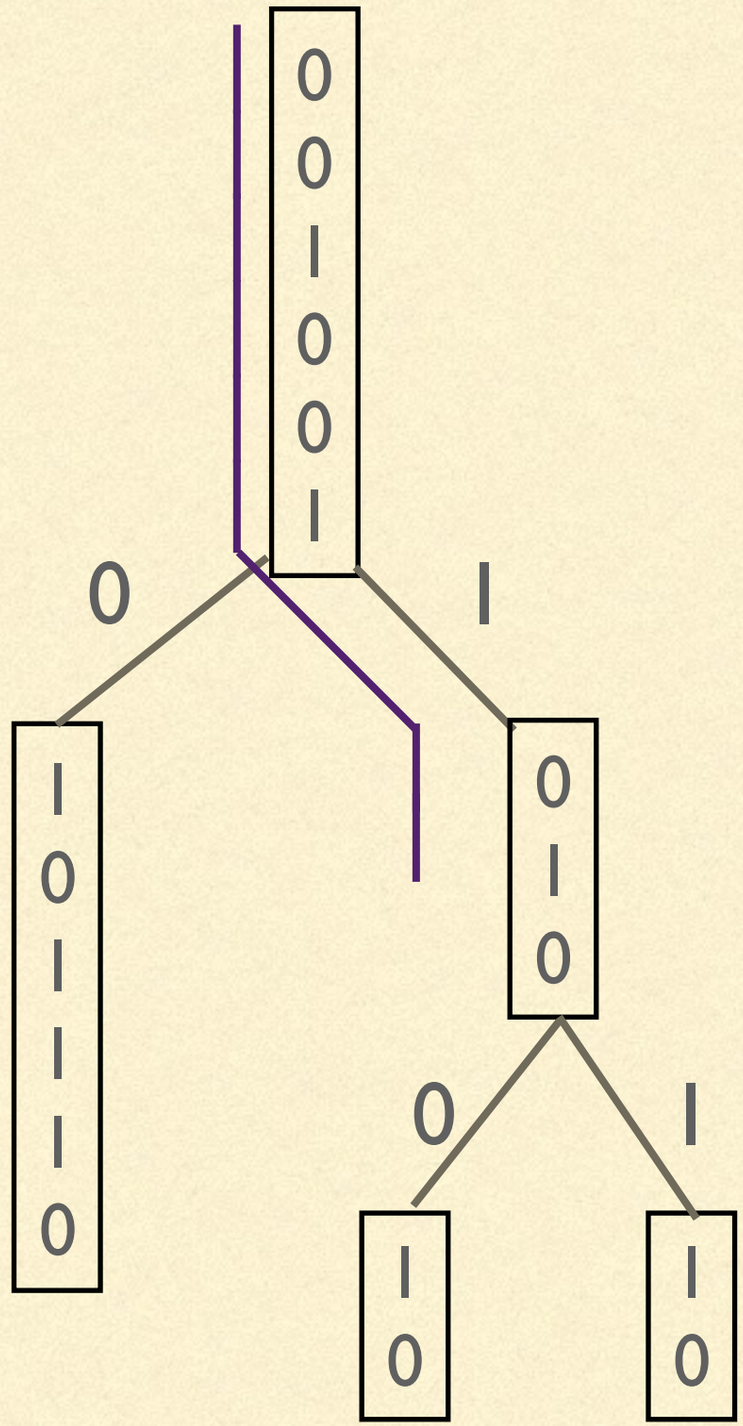
Exit node & exit direction

x = 001001101001 1



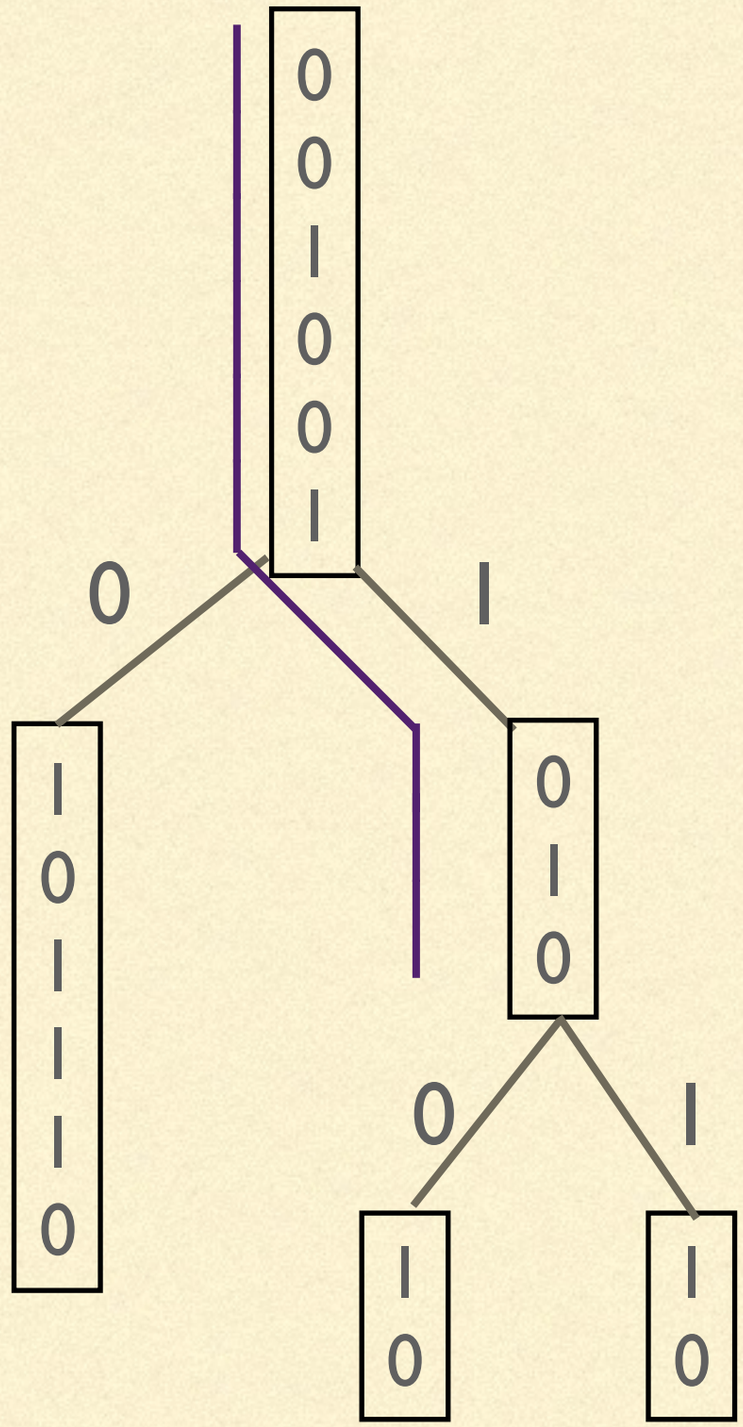
Exit node & exit direction

x = 001001101001 1



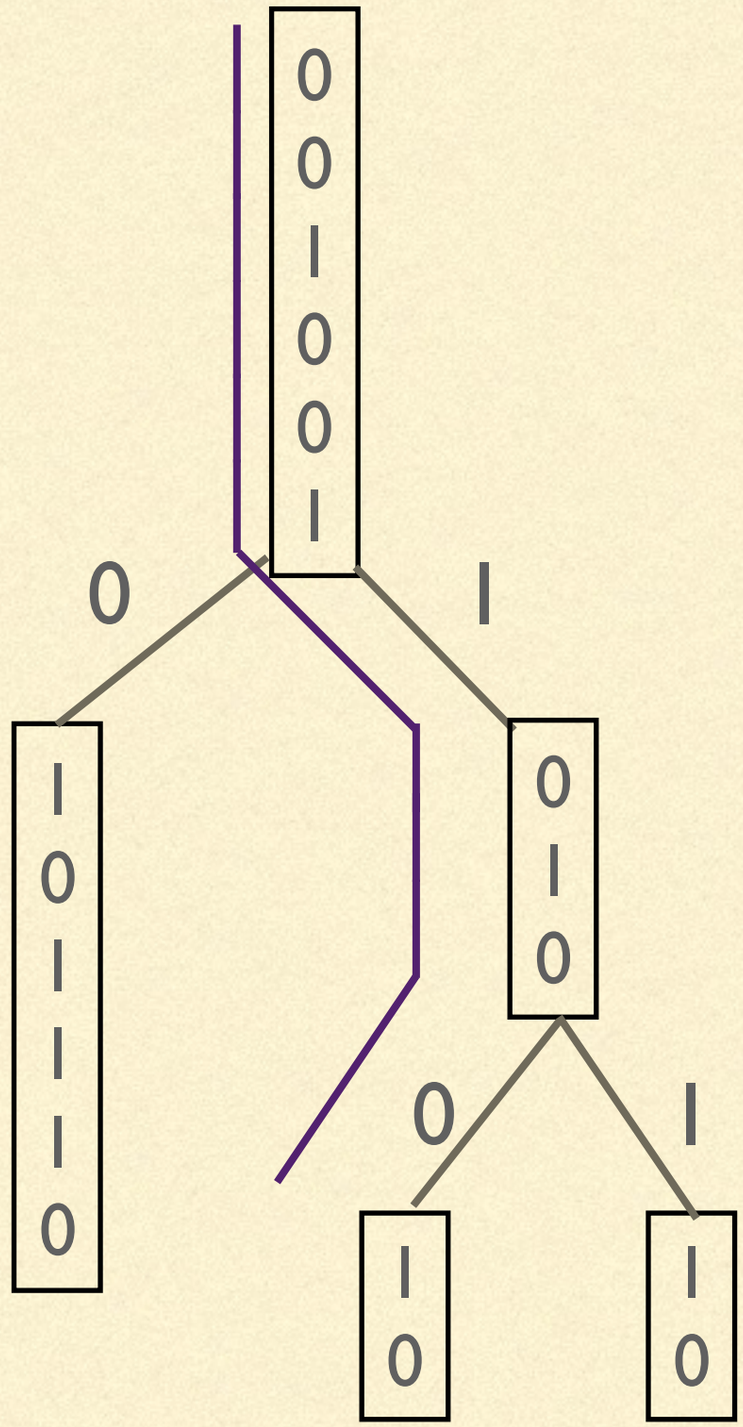
Exit node & exit direction

x = 001001101 0011



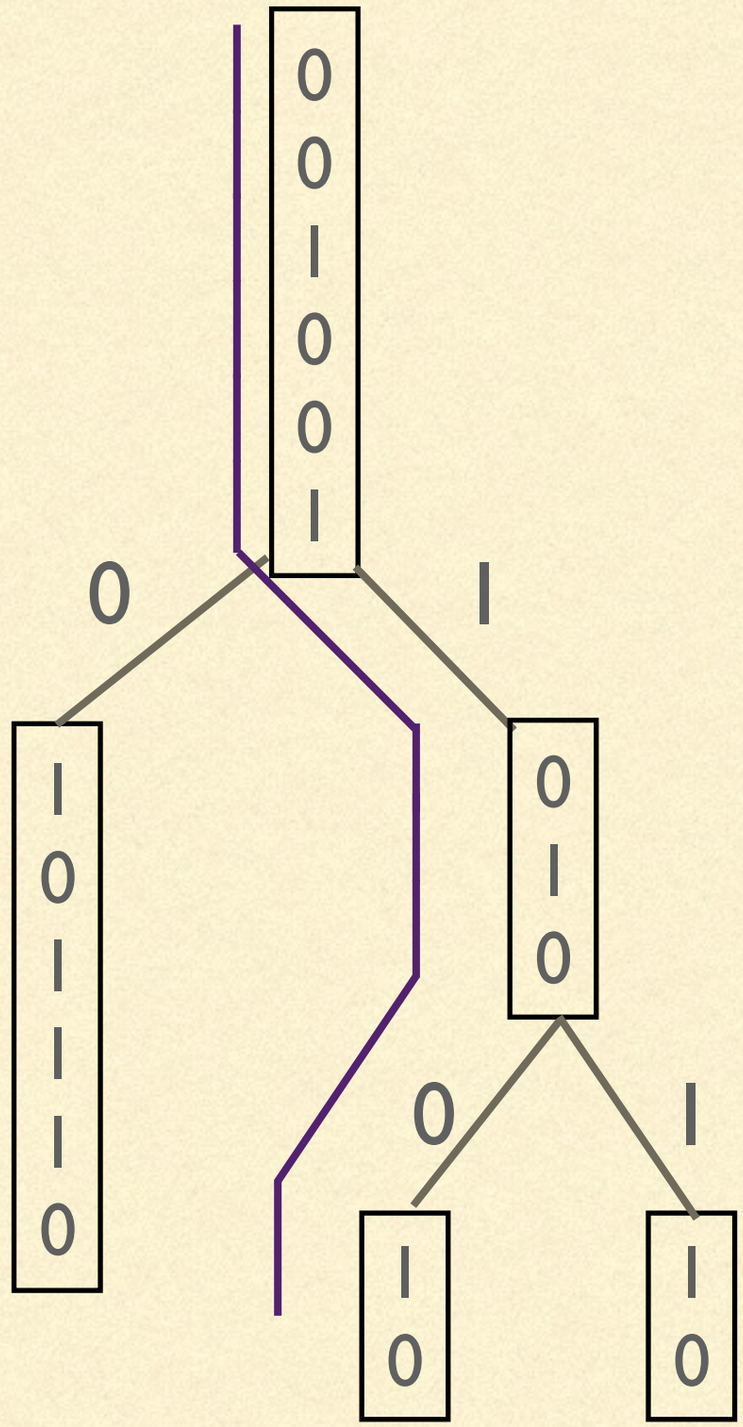
Exit node & exit direction

x = 001001101001011



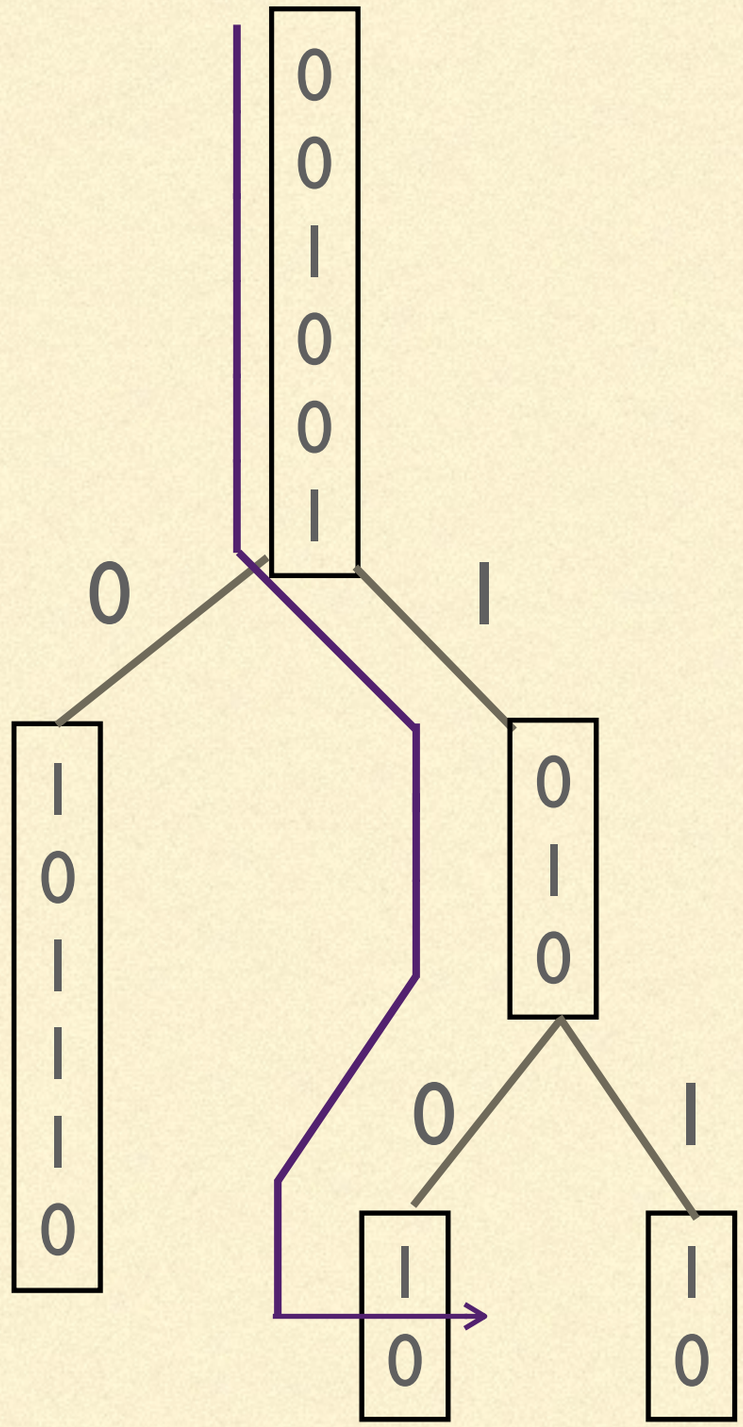
Exit node & exit direction

x = 001001101001 1



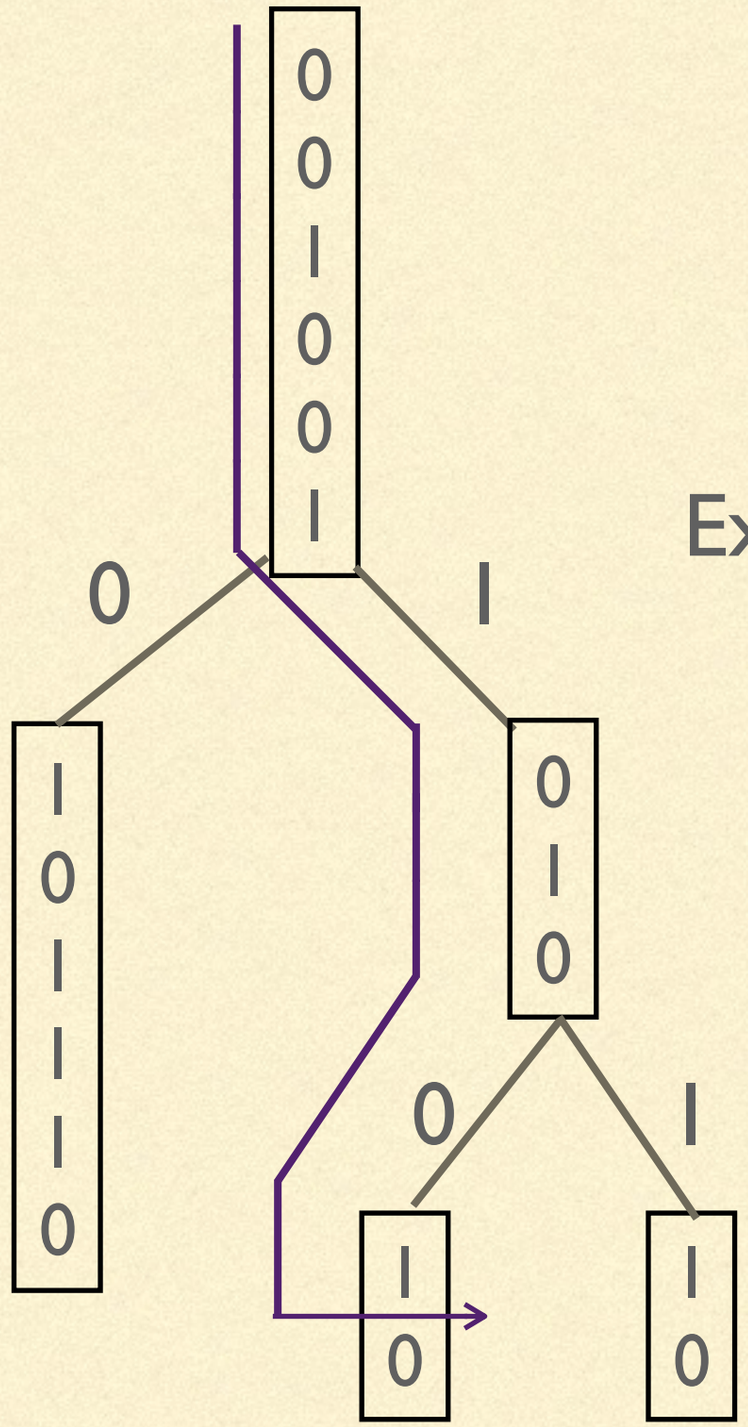
Exit node & exit direction

x = 001001101001 1



Exit node & exit direction

x = 0010011010011

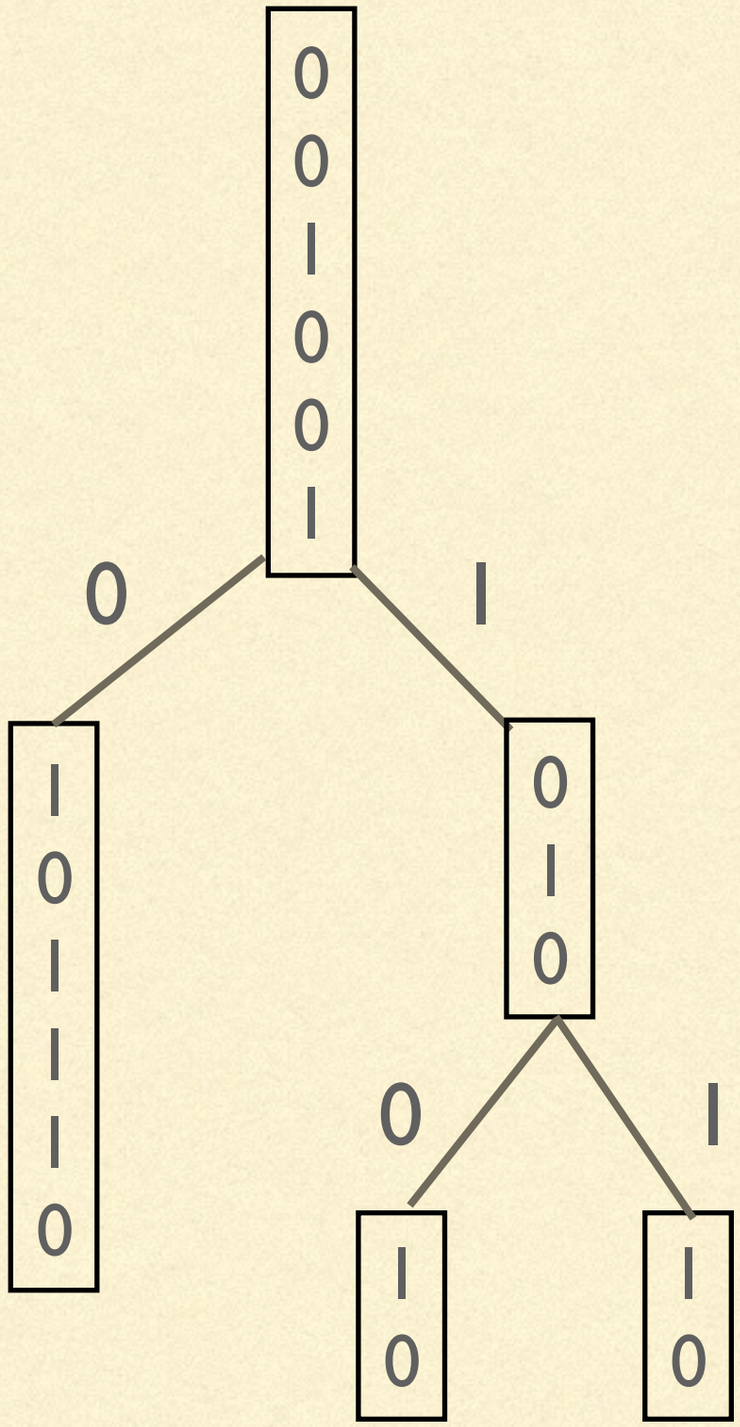


Exit node & exit direction

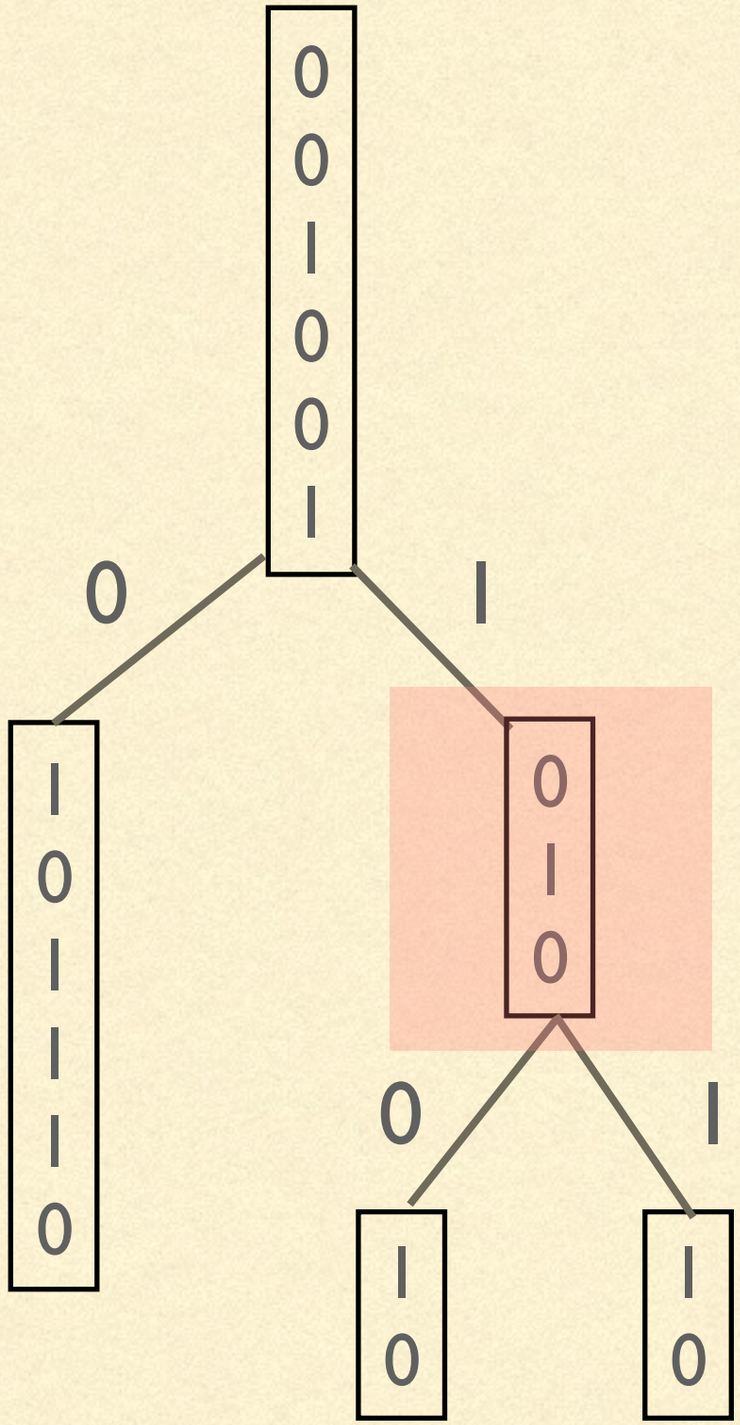
Exit between leaf #1 and #2 ⇒ bucket #2

x = 0010011010011

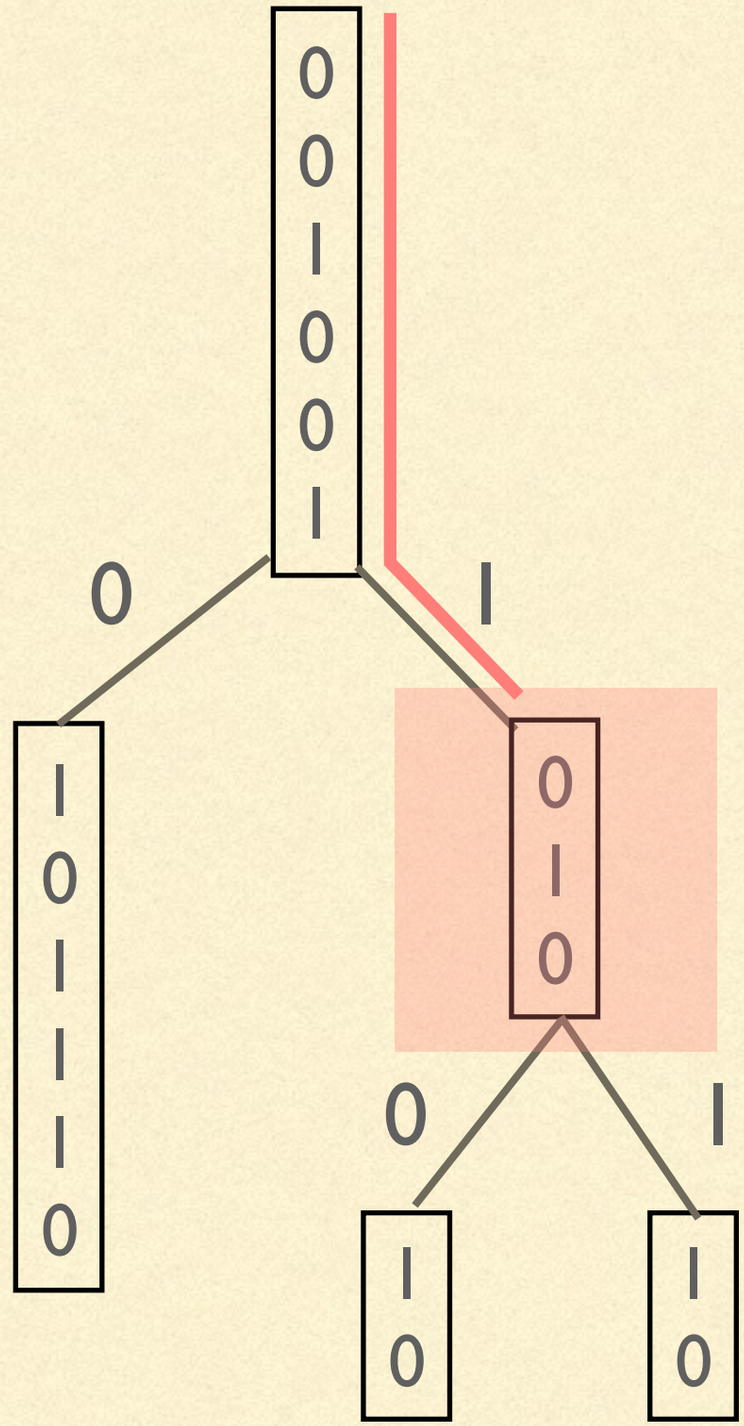
Nomenclature



Nomenclature

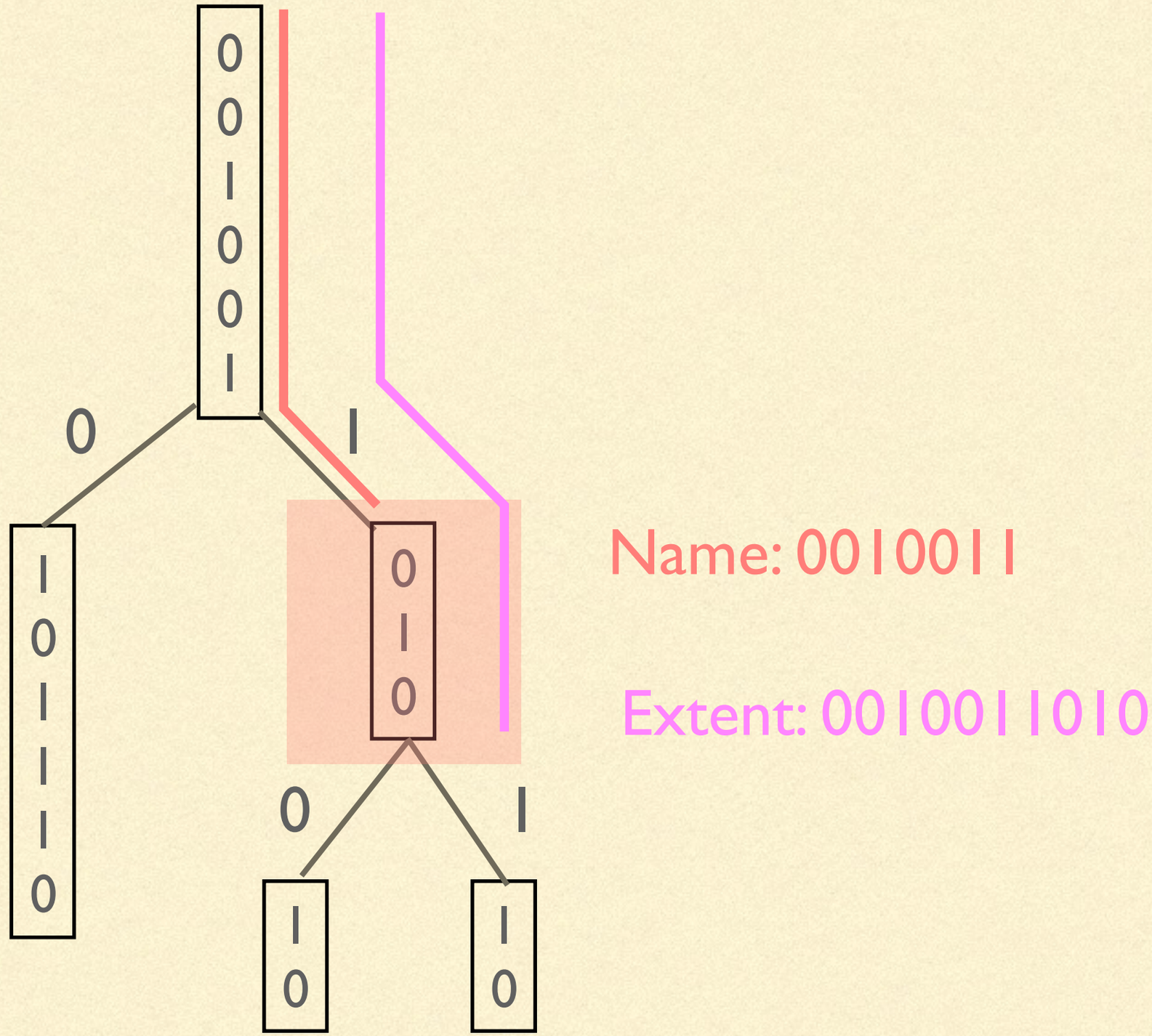


Nomenclature

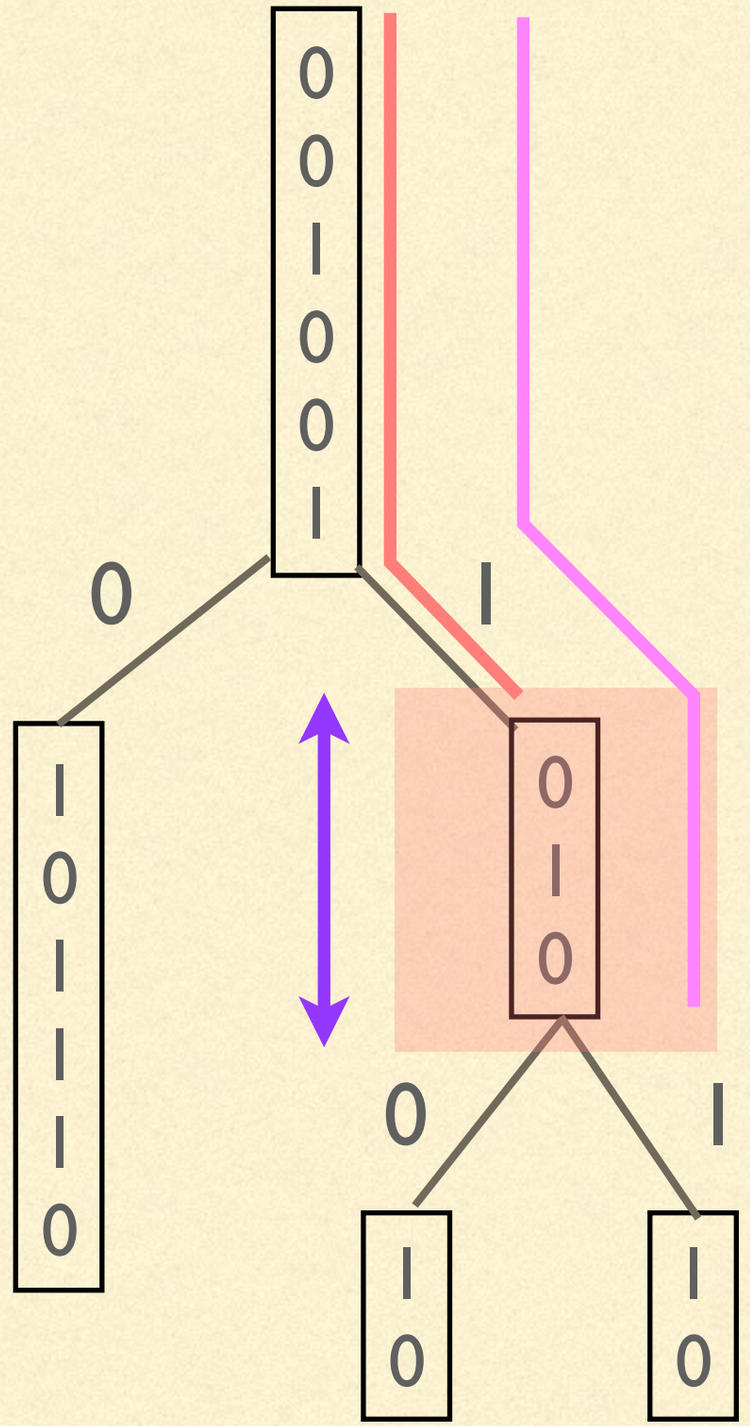


Name: 0010011

Nomenclature



Nomenclature



Name: 0010011

Extent: 0010011010

Skip interval: {7,…,10}

2-FATTEST NUMBERS

2-FATTEST NUMBERS

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

2-FATTEST NUMBERS

- *Definition:* Call z in [x..y] the *2-fattest* number in [x..y] if z has the largest number of trailing zeroes; equivalently, if $z = a2^b$ with maximum b
- Consider the compacted trie on the set D of delimiters

2-FATTEST NUMBERS

- *Definition:* Call z in [x..y] the *2-fattest* number in [x..y] if z has the largest number of trailing zeroes; equivalently, if $z = a2^b$ with maximum b
- Consider the compacted trie on the set D of delimiters
- Purpose: find the exit node & direction!

2-FATTEST NUMBERS

- *Definition:* Call z in [x..y] the *2-fattest* number in [x..y] if z has the largest number of trailing zeroes; equivalently, if $z = a2^b$ with maximum b
- Consider the compacted trie on the set D of delimiters
- Purpose: find the exit node & direction!
- Equivalently: given x, find the *longest extent* e of the trie that is a prefix of x

2-FATTEST NUMBERS

- *Definition:* Call z in [x..y] the *2-fattest* number in [x..y] if z has the largest number of trailing zeroes; equivalently, if $z = a2^b$ with maximum b
- Consider the compacted trie on the set D of delimiters
- Purpose: find the exit node & direction!
- Equivalently: given x, find the *longest extent* e of the trie that is a prefix of x
- x[1…|e|+1] is the name of the exit node

Z-FAST TRIE

Z-FAST TRIE

- For every node, we store (*handle*="suitable" prefix of extent) ↦ extent

Z-FAST TRIE

- For every node, we store (*handle*="suitable" prefix of extent) ↦ extent
- More precisely: use the prefix of e of length given by the 2-fattest number in the skip interval

Z-FAST TRIE

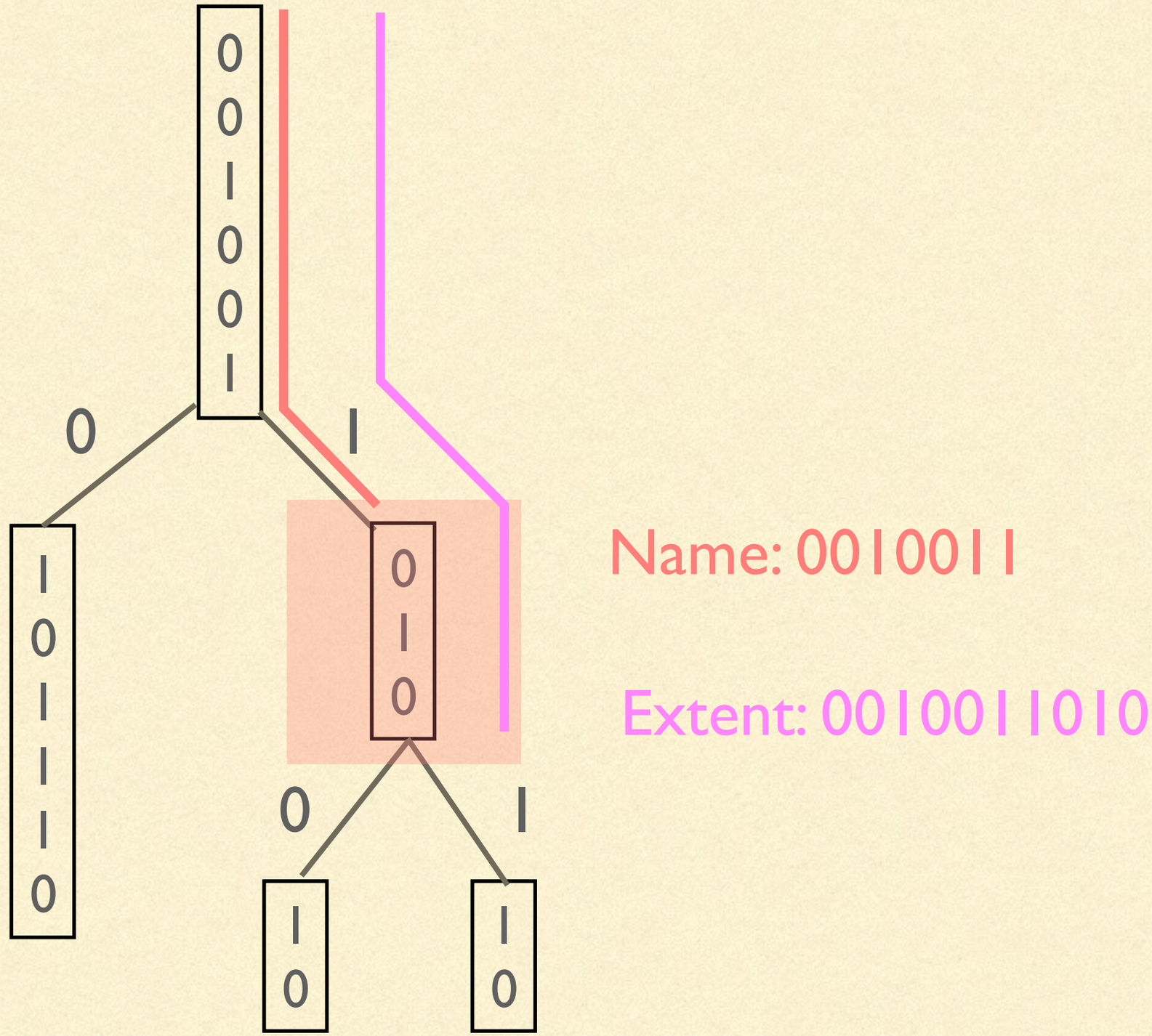
- For every node, we store (*handle*="suitable" prefix of extent) ↦ extent
- More precisely: use the prefix of e of length given by the 2-fattest number in the skip interval
- The z-fast trie is entirely specified by the mapping

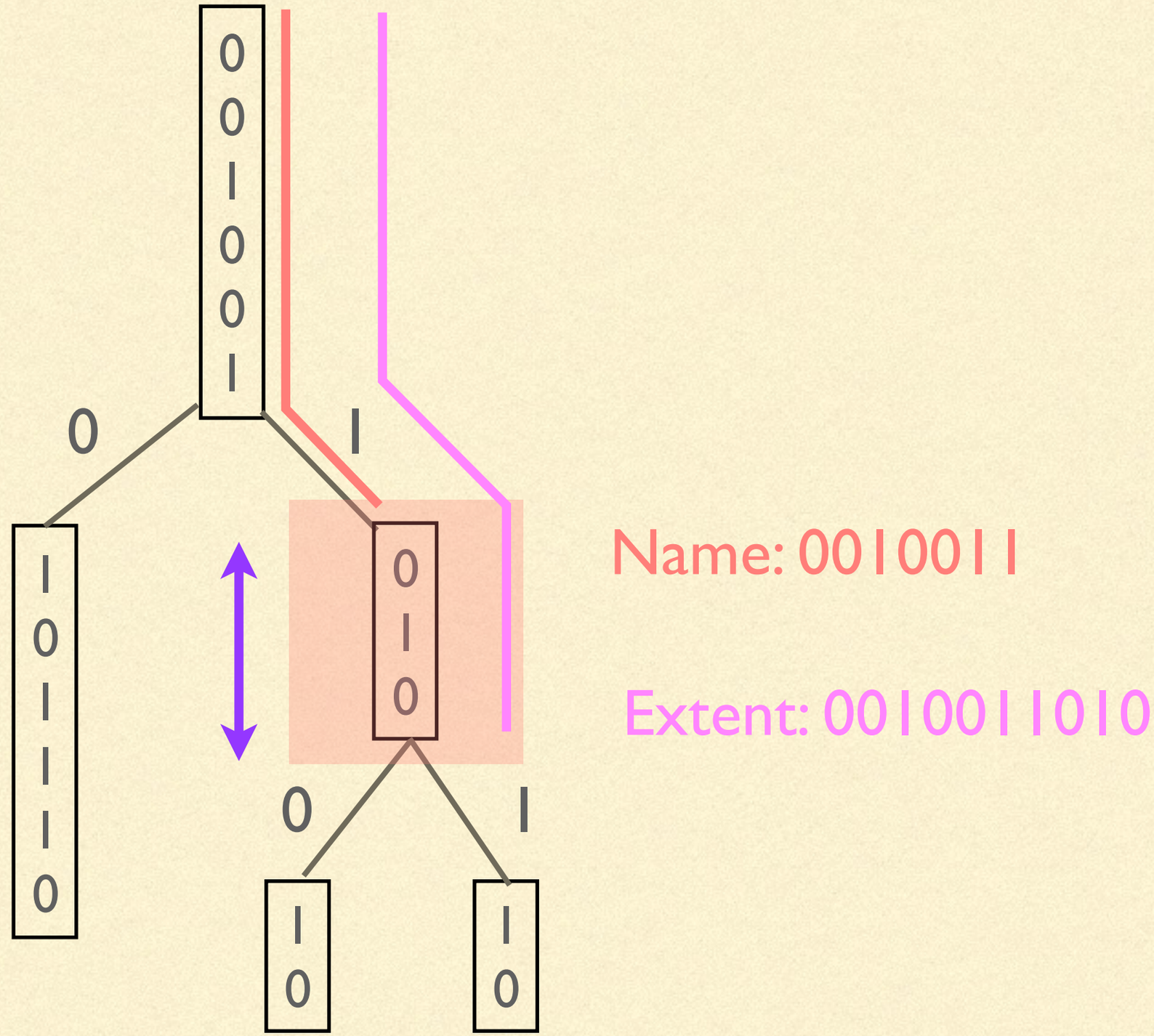
  handles ↦ extent

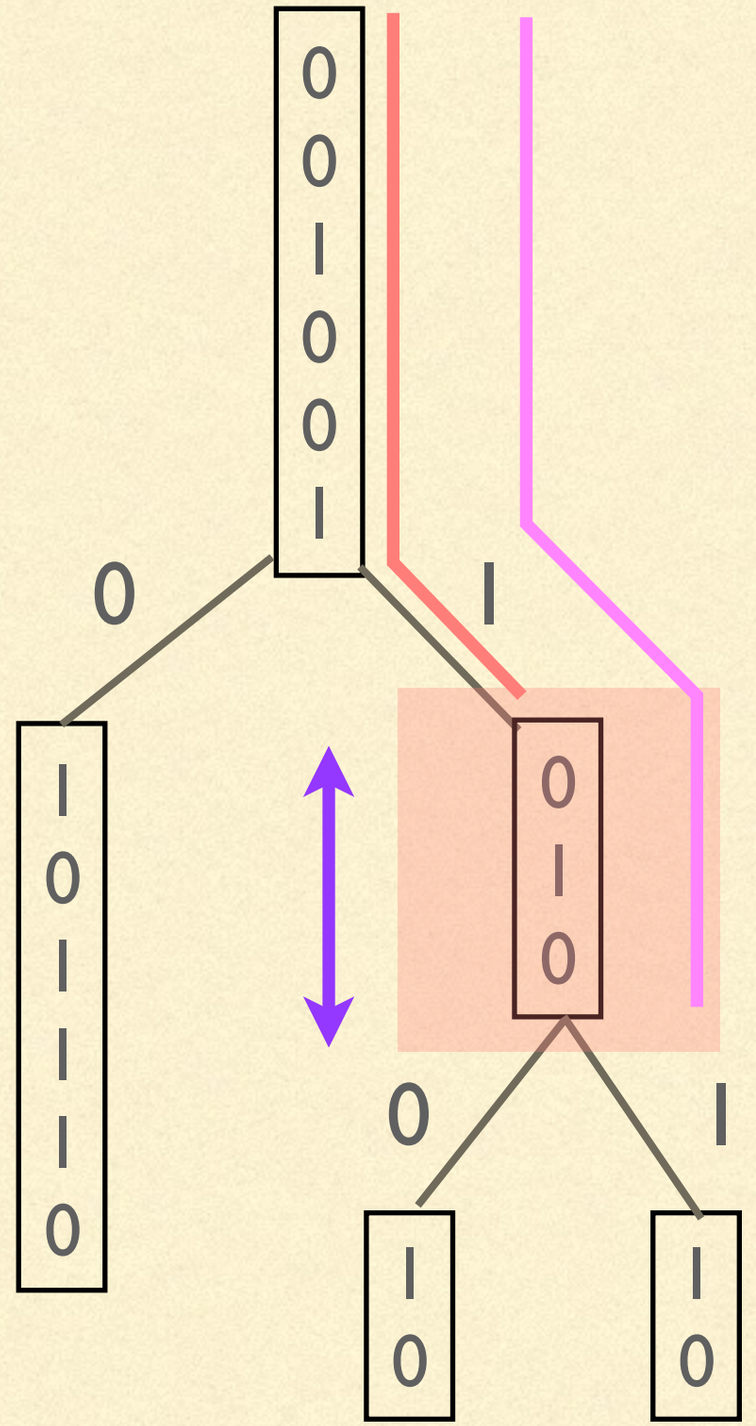
Z-FAST TRIE

- For every node, we store (*handle*="suitable" prefix of extent) ↦ extent
- More precisely: use the prefix of e of length given by the 2-fattest number in the skip interval
- The z-fast trie is entirely specified by the mapping

  handles ↦ extent
- We assume for the moment that the map is *signed*


0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
1
0
0
1
0
Name: 0010011
Extent: 0010011010

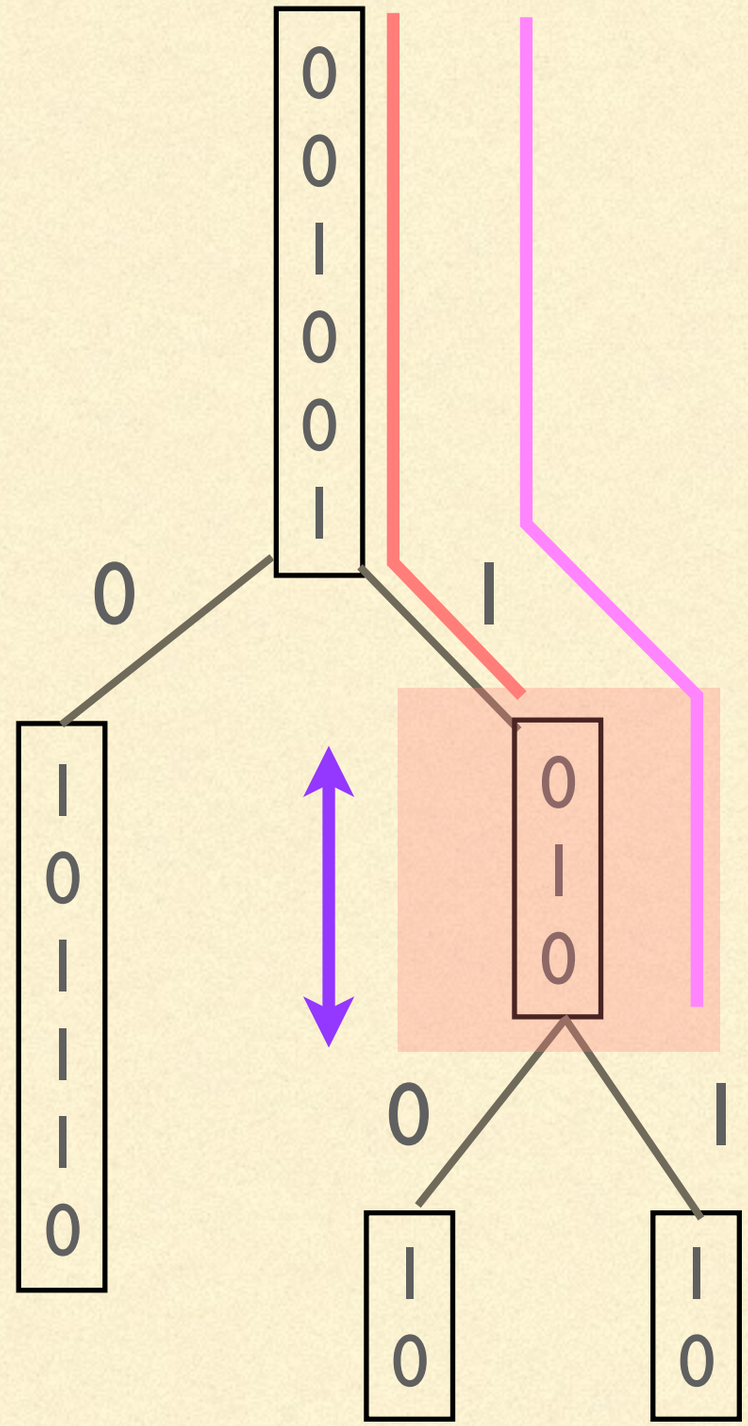

0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
1
0
1
0
Name: 0010011
Extent: 0010011010



Name: 0010011

Extent: 0010011010

Skip interval: {7,…,10}

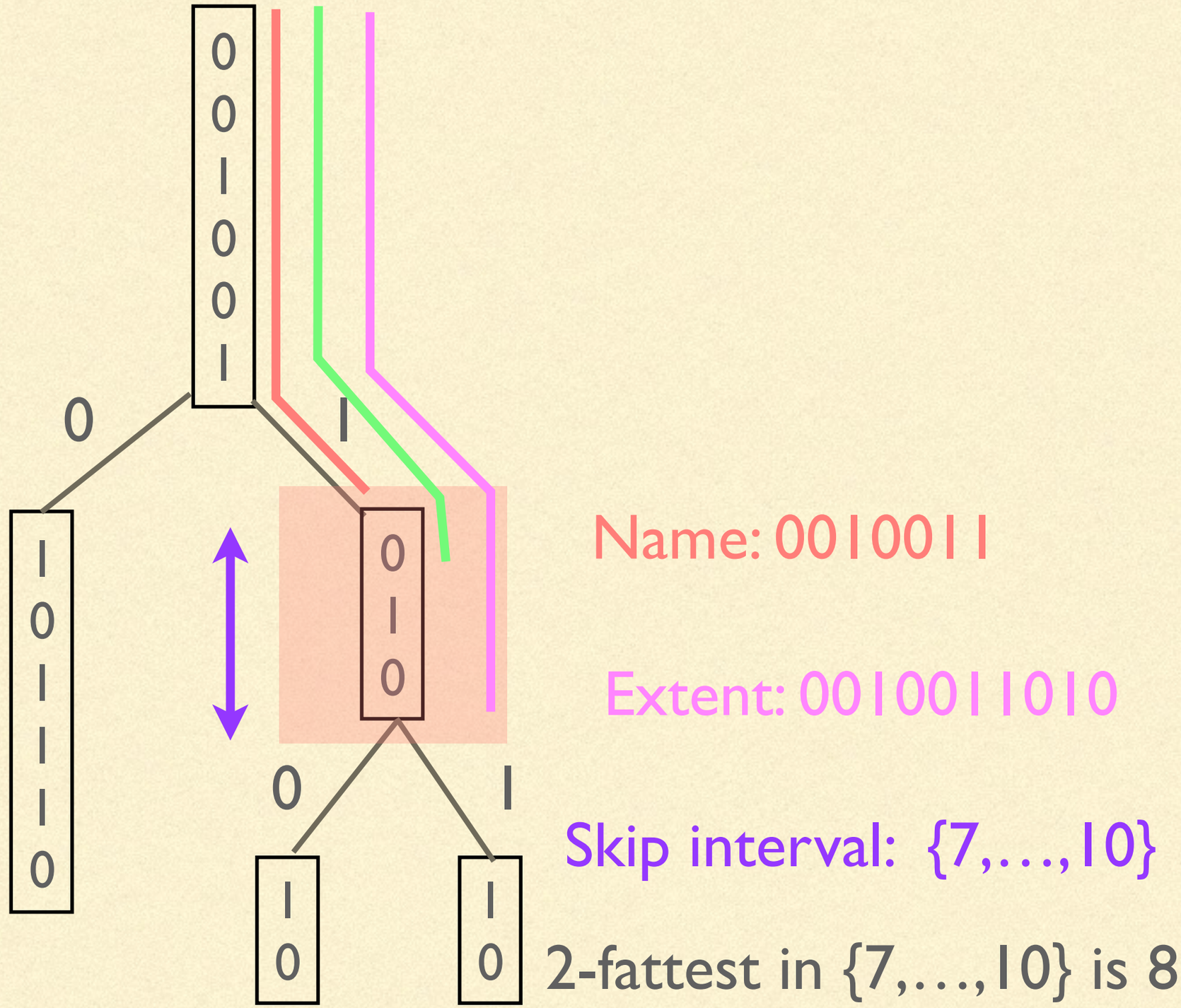


Name: 0010011

Extent: 0010011010

Skip interval: {7,…,10}

2-fattest in {7,…,10} is 8



We store the pair 00100110 → 0010011010

FAT BINARY SEARCH

FAT BINARY SEARCH

- *Fat binary search:* 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

- *Fat binary search:* similar to a binary search, but probes always insist on the 2-fattest number in an interval, rather than on its middle point
- We will not know the exit direction (left/right)

FAT BINARY SEARCH

FAT BINARY SEARCH

- Given x, we keep track of a current interval (l..r) in which the extent of the parent of the exit node of x might lie

FAT BINARY SEARCH

- Given x, we keep track of a current interval (l..r) in which the extent of the parent of the exit node of x might lie
- We query the trie with the prefix of x of length f = the 2-fattest number in (l..r)

FAT BINARY SEARCH

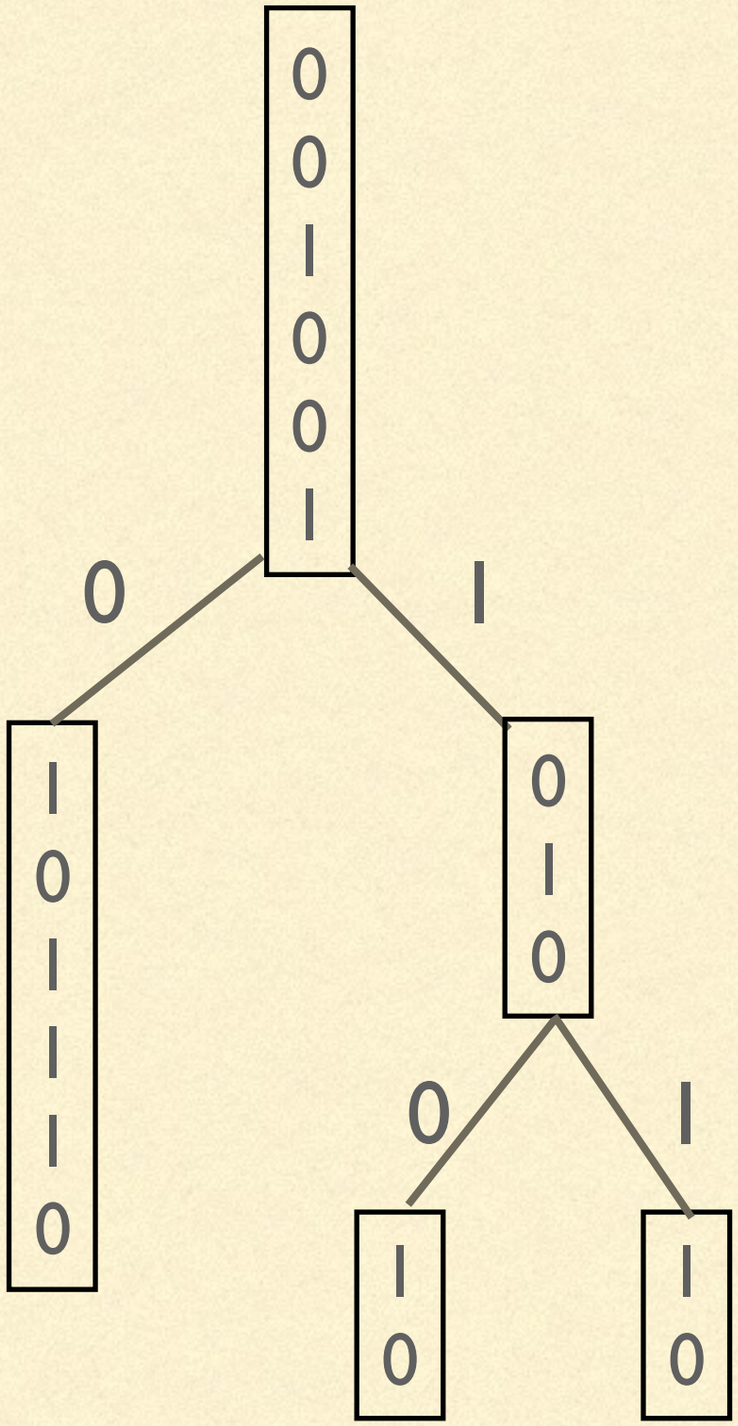
- Given x, we keep track of a current interval (l..r) in which the extent of the parent of the exit node of x might lie
- We query the trie with the prefix of x of length f = the 2-fattest number in (l..r)
- *Property*: f is 2-fattest in every subinterval of (l..r) that contains it

FAT BINARY SEARCH

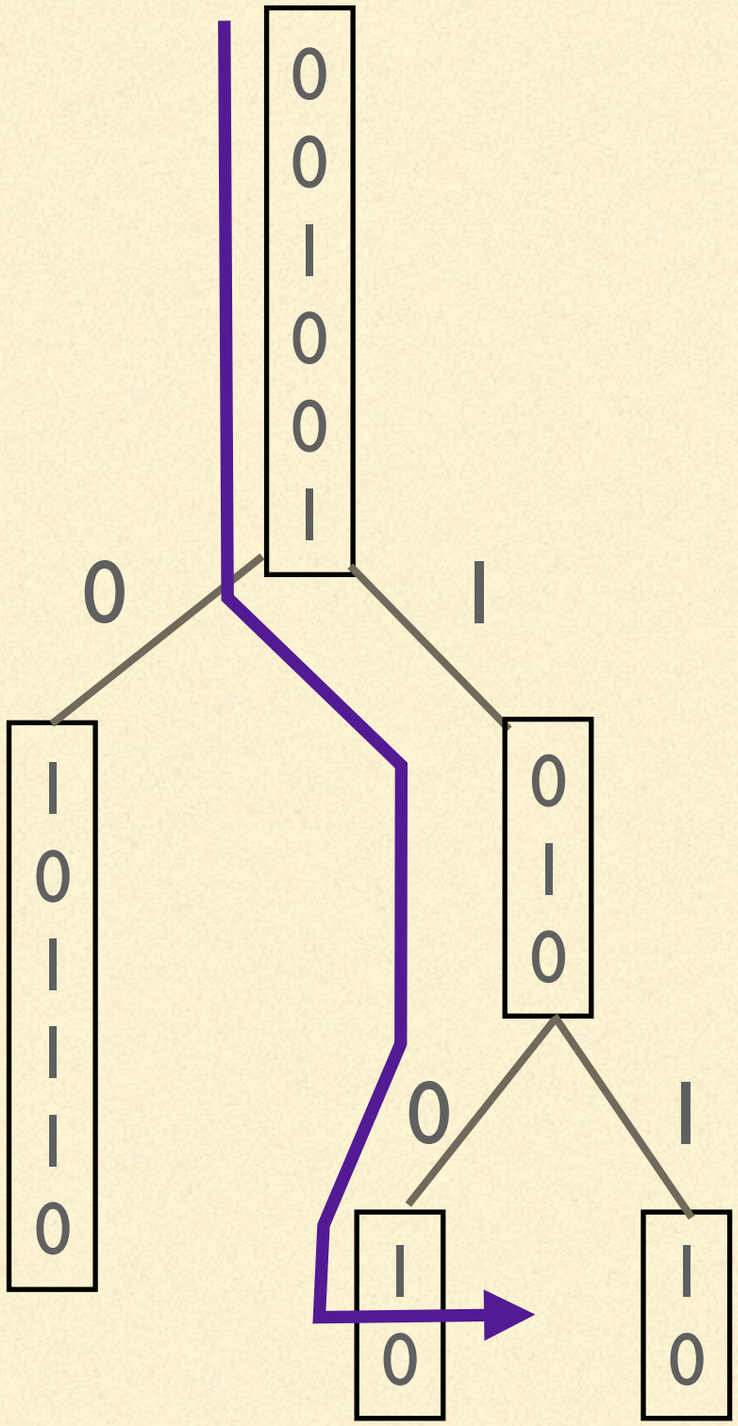
- Given x, we keep track of a current interval (l..r) in which the extent of the parent of the exit node of x might lie
- We query the trie with the prefix of x of length f = the 2-fattest number in (l..r)
- *Property*: f is 2-fattest in every subinterval of (l..r) that contains it
- ⇒ Every time we will be using a handle, given that it is a prefix of some string in the trie

FAT BINARY SEARCH

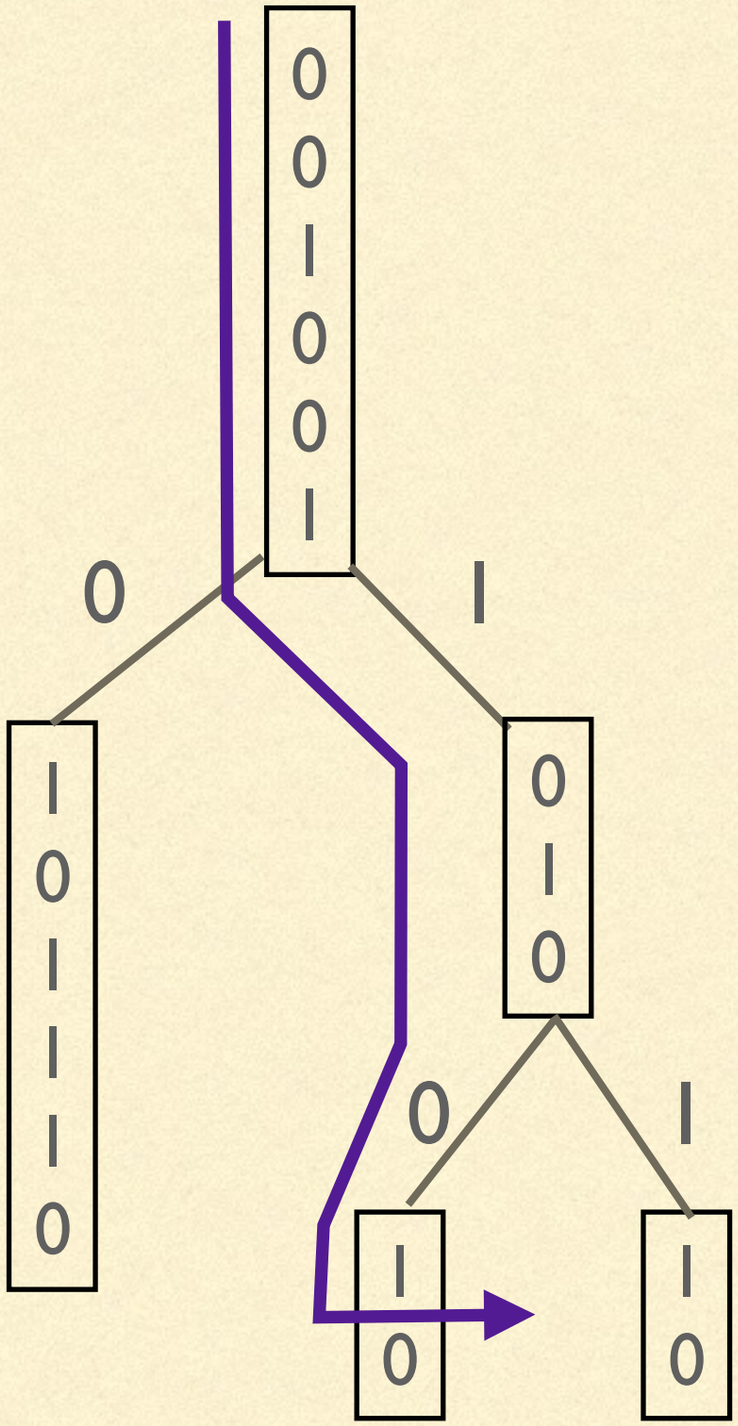
- Given x, we keep track of a current interval (l..r) in which the extent of the parent of the exit node of x might lie
- We query the trie with the prefix of x of length f = the 2-fattest number in (l..r)
- *Property*: f is 2-fattest in every subinterval of (l..r) that contains it
- ⇒ Every time we will be using a handle, given that it is a prefix of some string in the trie
- If the resulting string s is a prefix of x, we move to (|s|..r); otherwise, we move to (l..f)



x = 001001101001 1

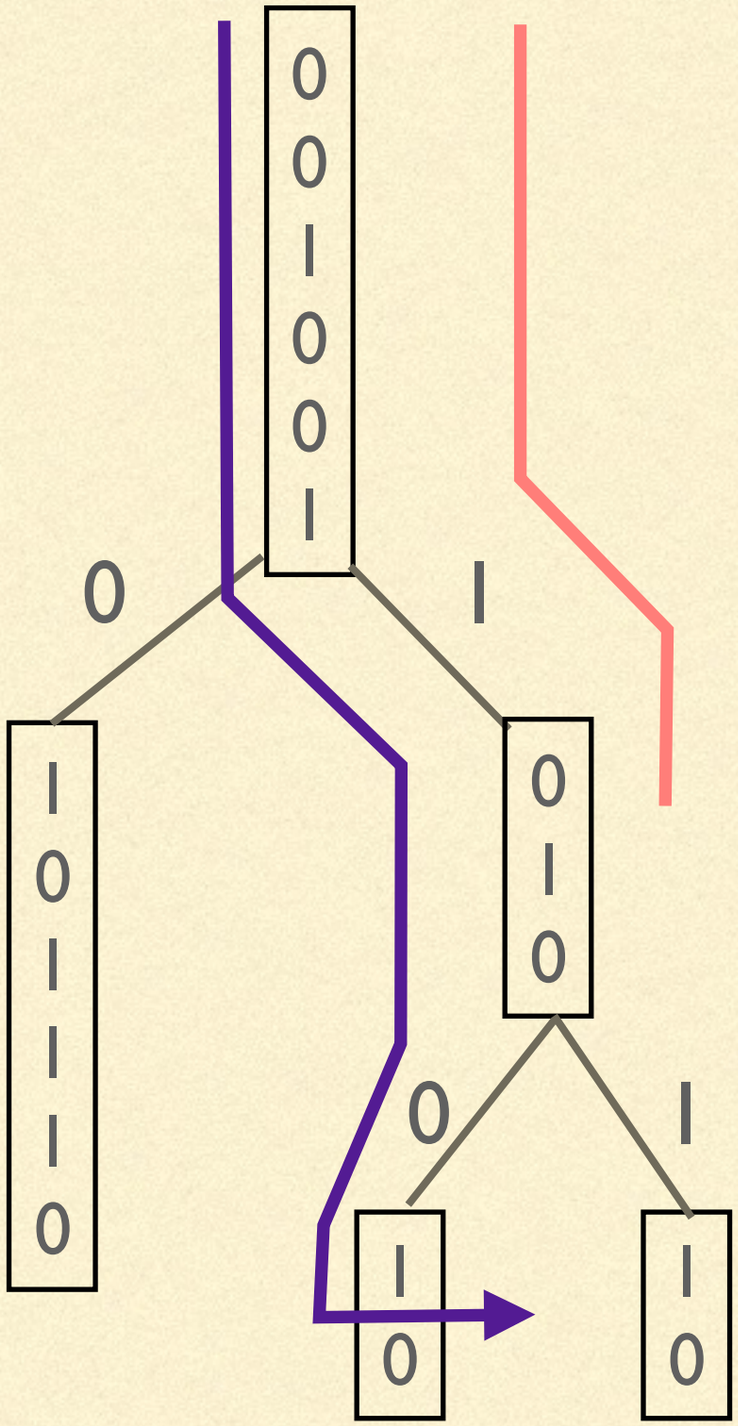


x = 001001101001 1



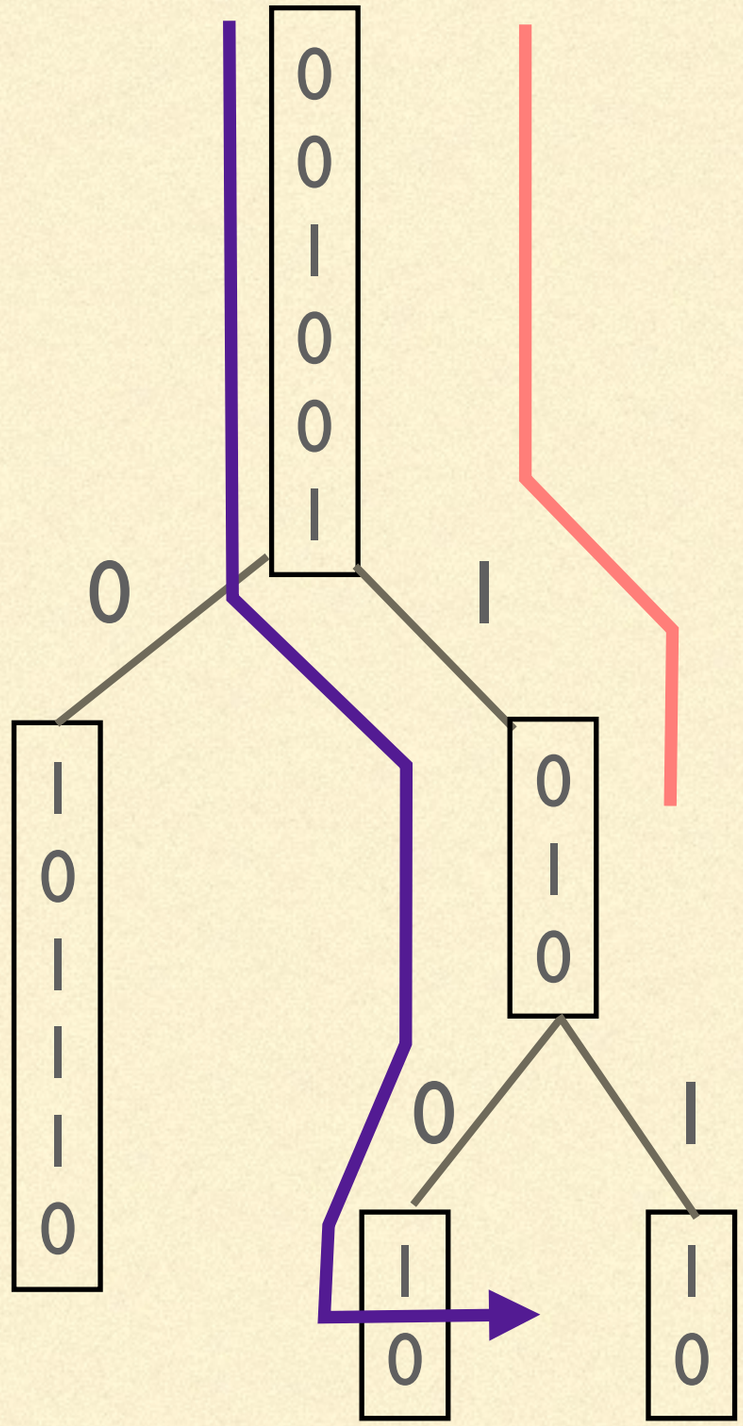
$(l..r) = (0..13) \Rightarrow f = 8$

$x = 0010011010011$



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

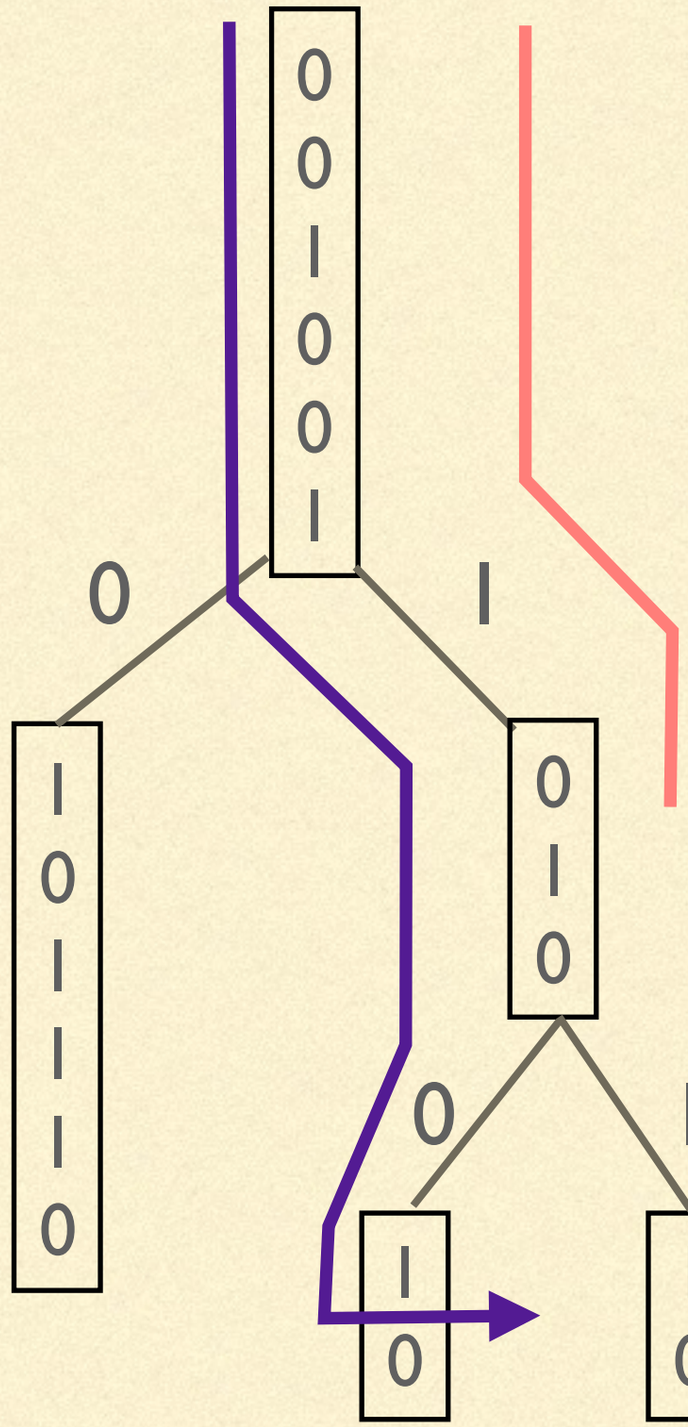
$x = 001001101 0011$



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

Function returns $0010011010 \preccurlyeq x$

$x = 0010011010011$

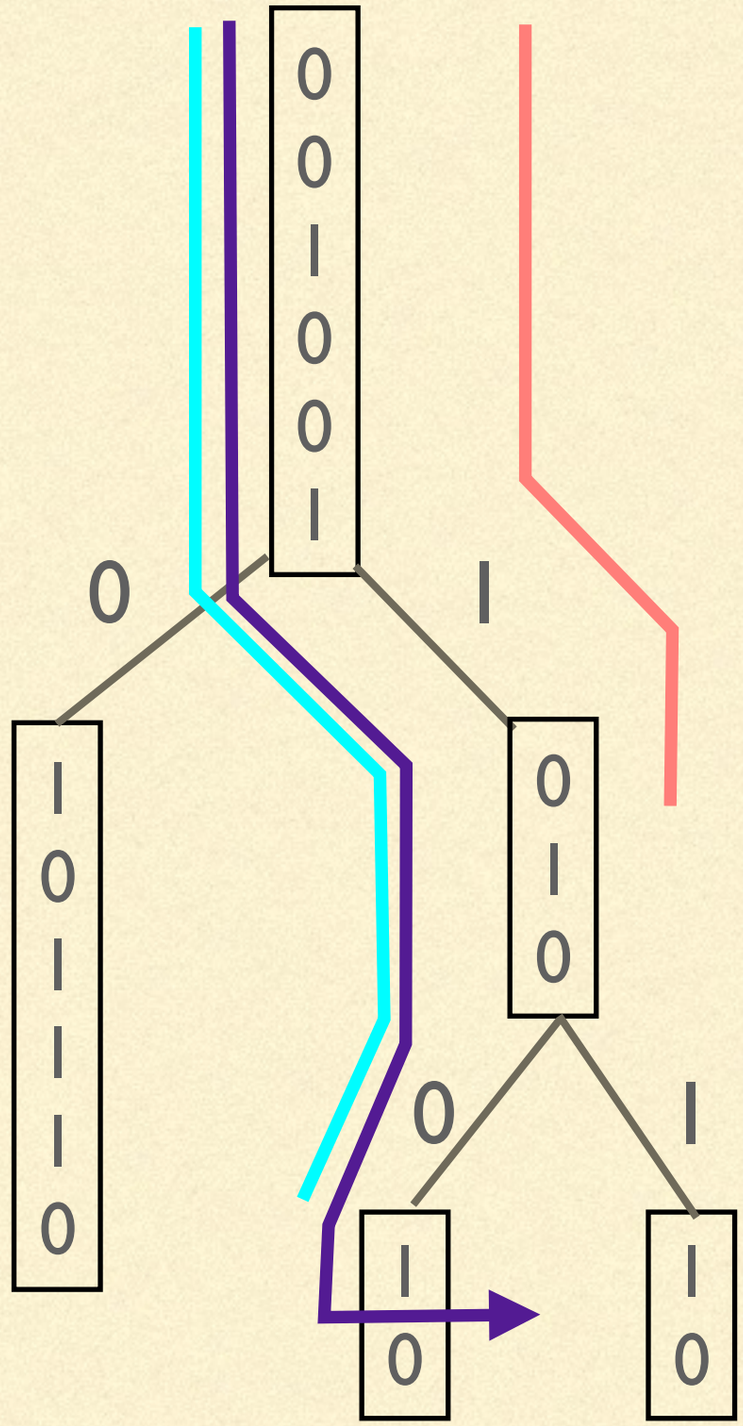


$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $0010011010 \preccurlyeq x$

$\Rightarrow (l..r) = (8..13) \Rightarrow f = 12$

$x = 0010011010011$

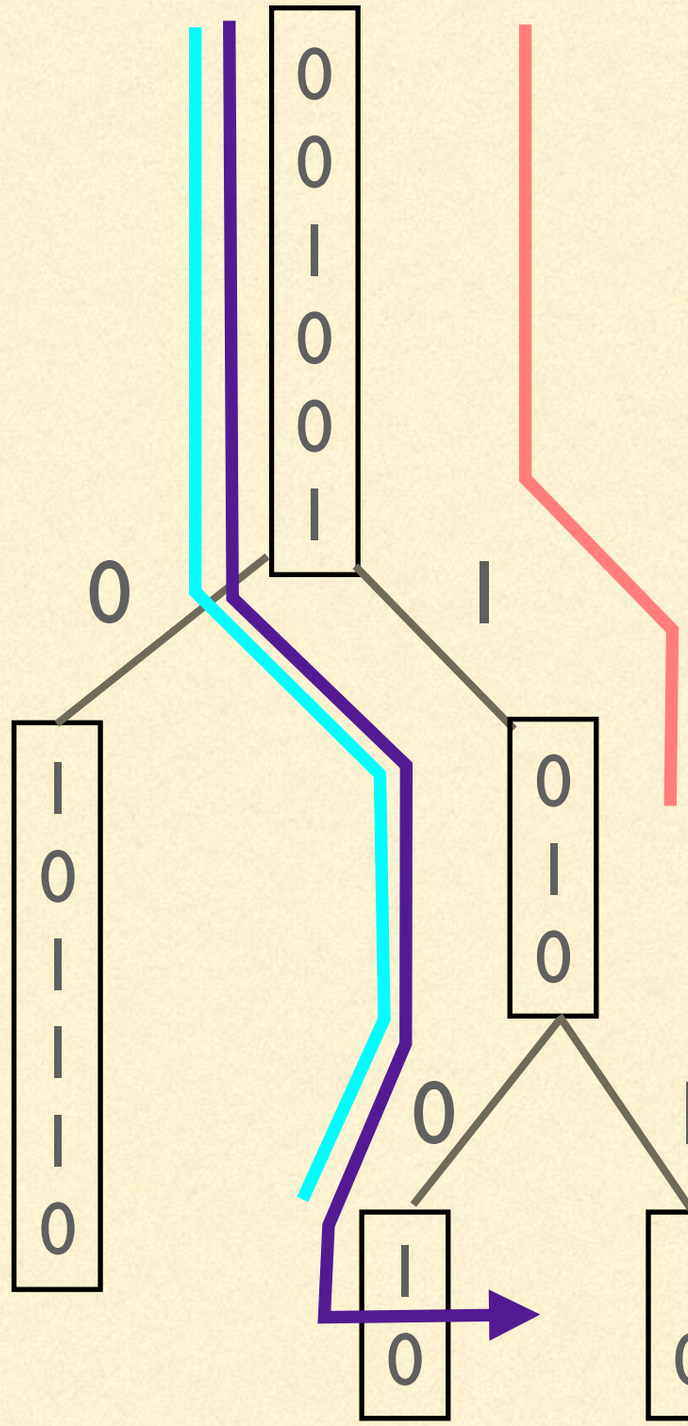


$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $0010011010 \preccurlyeq x$

$\Rightarrow (l..r) = (8..13) \Rightarrow f = 12$

$x = 0010011010011$



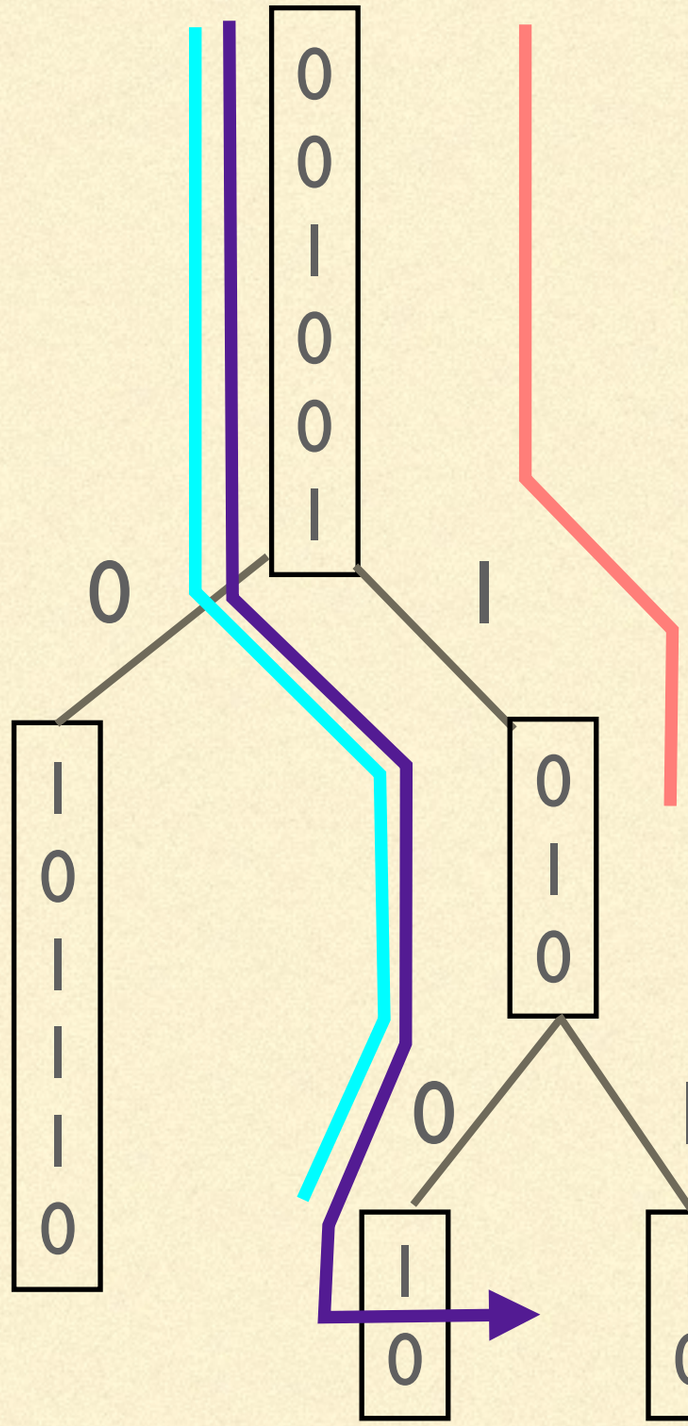
$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $0010011010 \preccurlyeq x$

$\Rightarrow (l..r) = (8..13) \Rightarrow f = 12$

Function returns $0010011010010 \not\preccurlyeq x$

$x = 0010011010011$



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

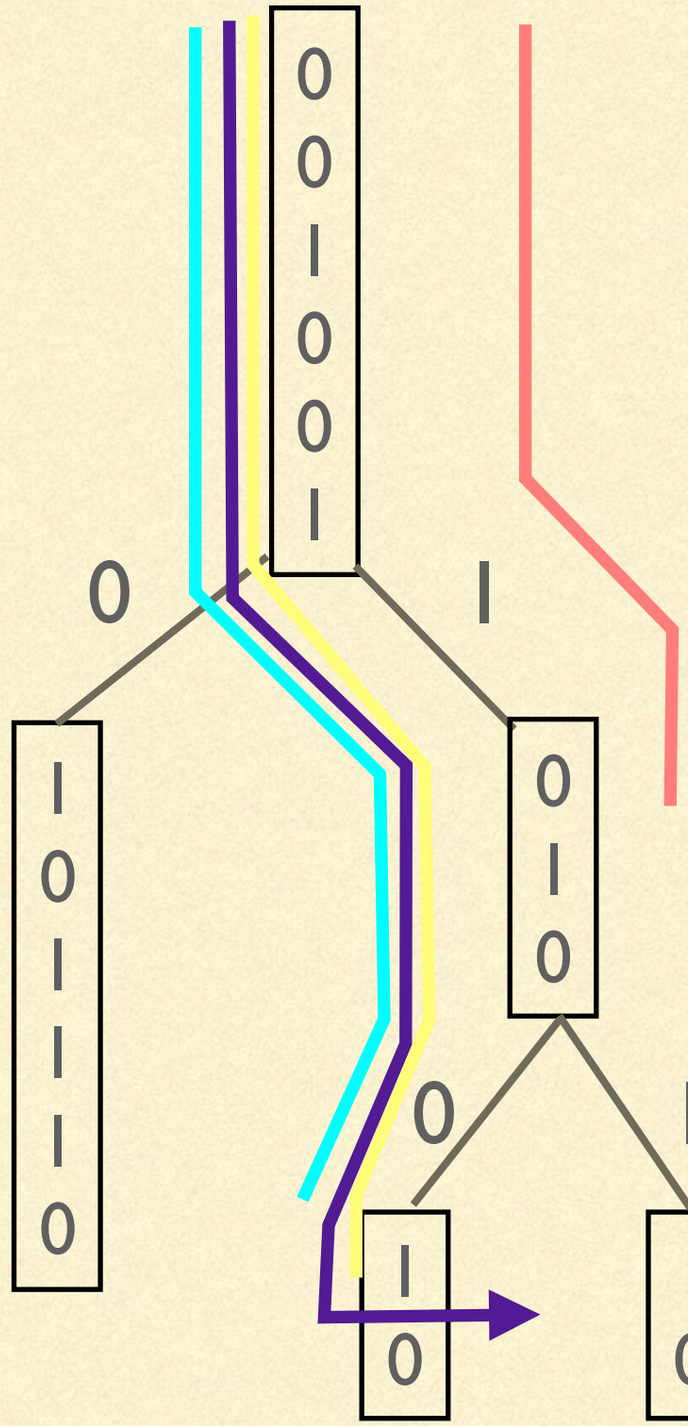
Function returns $0010011010 \preccurlyeq x$

$\Rightarrow (l..r) = (8..13) \Rightarrow f = 12$

Function returns $00100110010010 \not\preccurlyeq x$

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

$x = 00100110100 11$



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

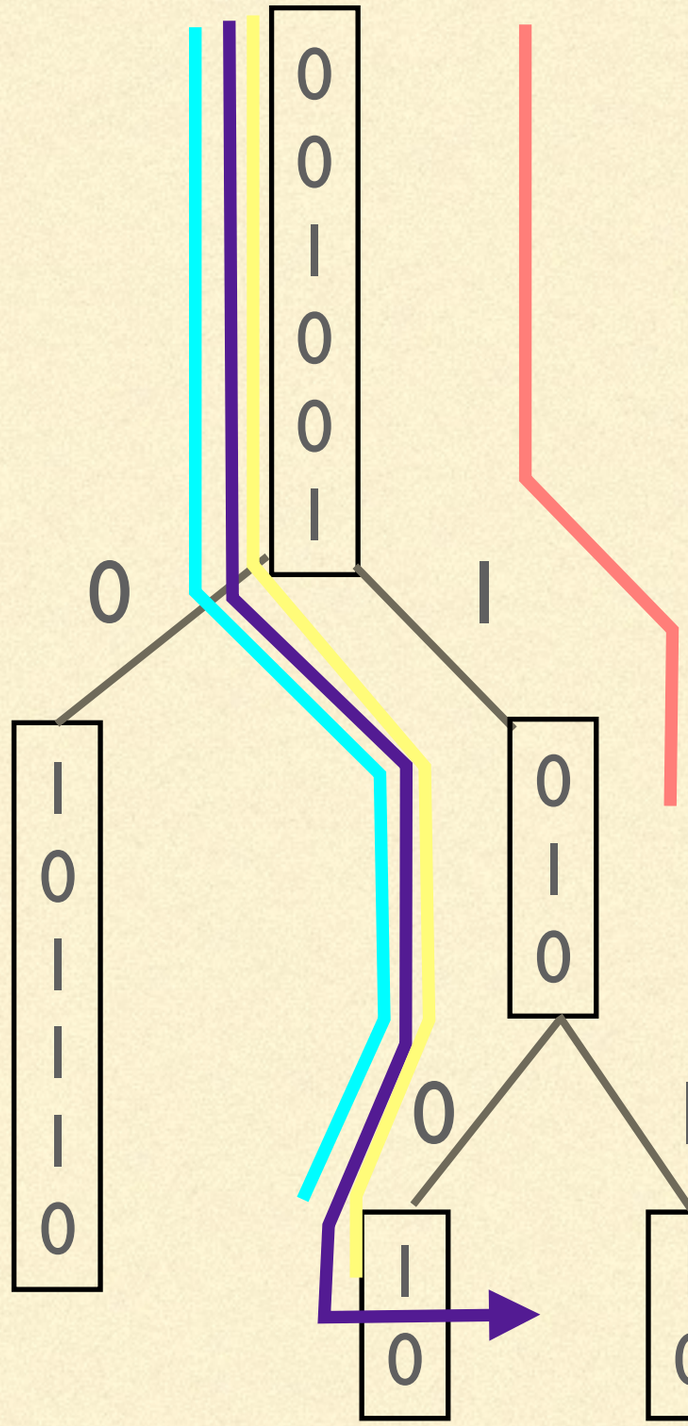
Function returns $0010011010 \preccurlyeq x$

$\Rightarrow (l..r) = (8..13) \Rightarrow f = 12$

Function returns $00100110010010 \not\preccurlyeq x$

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

$x = 0010011010011$



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

Function returns $0010011010 \preccurlyeq x$

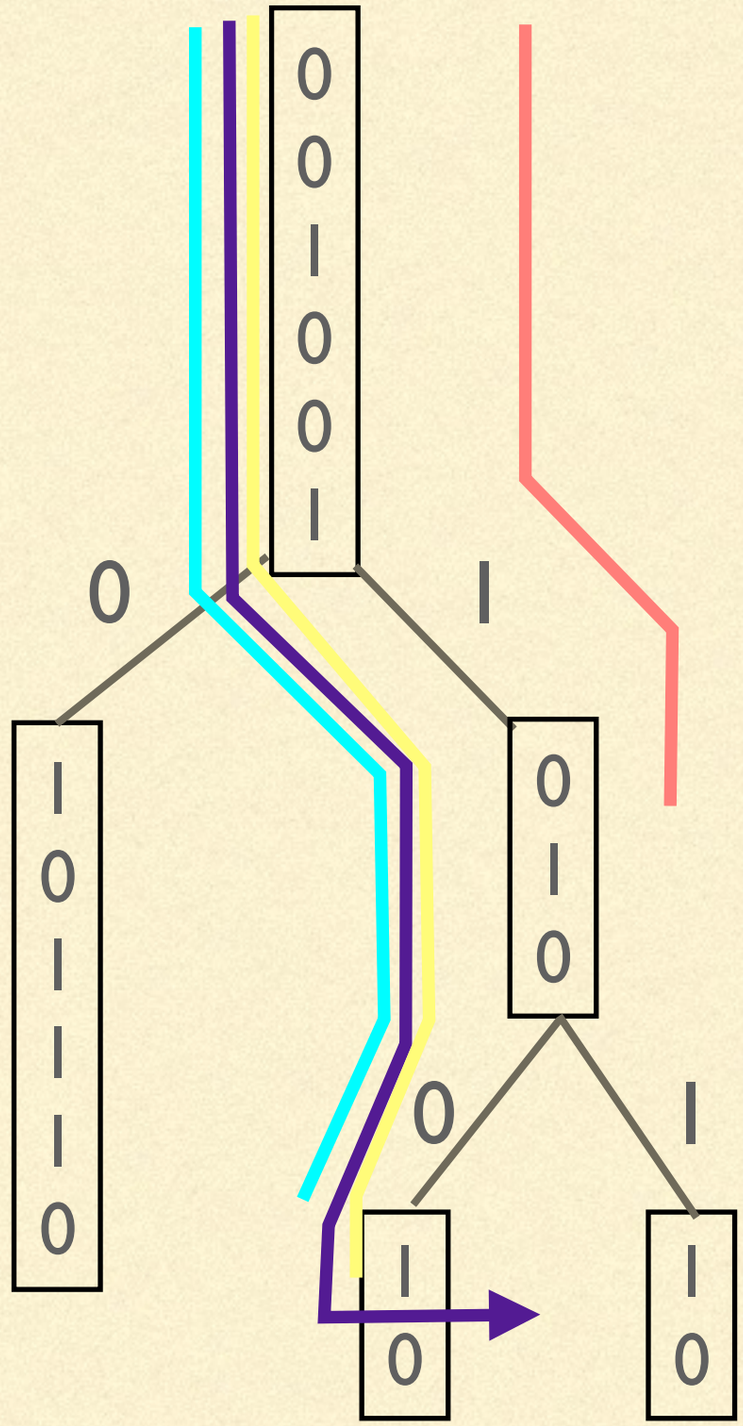
$\Rightarrow (l..r) = (8..13) \Rightarrow f = 12$

Function returns $0010011010010 \not\preccurlyeq x$

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

Function returns $\perp$

$x = 0010011010011$



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

Function returns $0010011010 \preccurlyeq x$

$\Rightarrow (l..r) = (8..13) \Rightarrow f = 12$

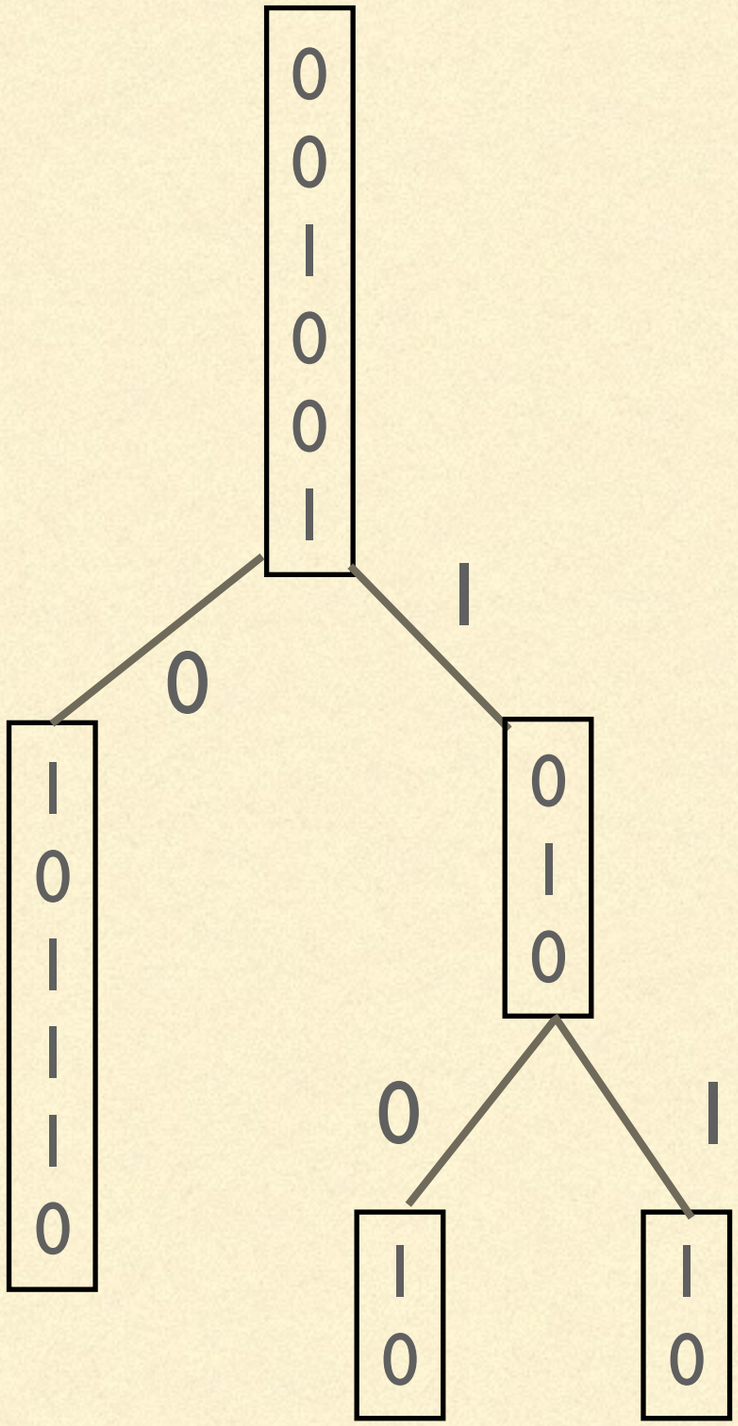
Function returns $0010011010010 \not\preccurlyeq x$

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

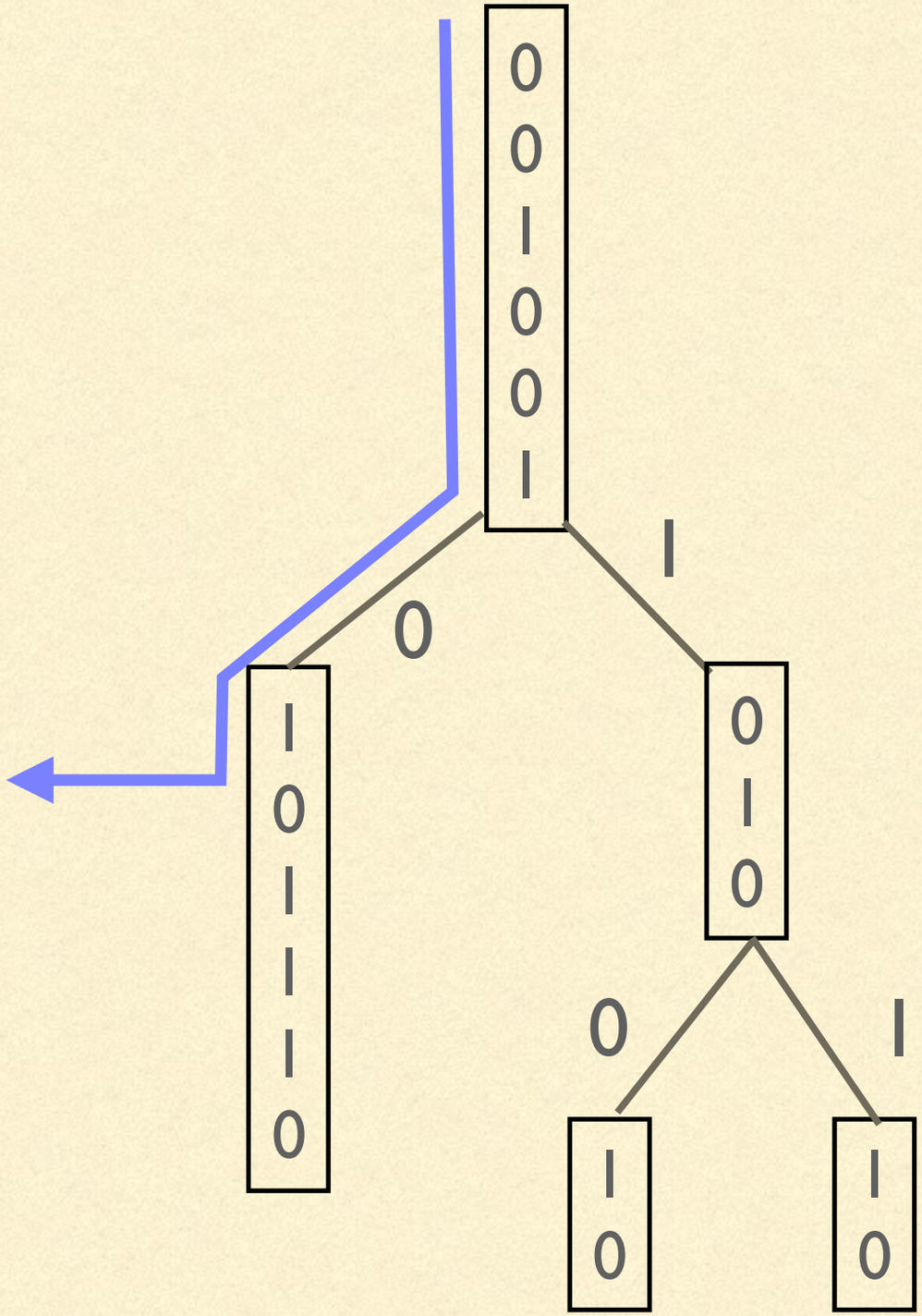
Function returns $\perp$

$\Rightarrow$ x exits at node with name x[1,…,12]

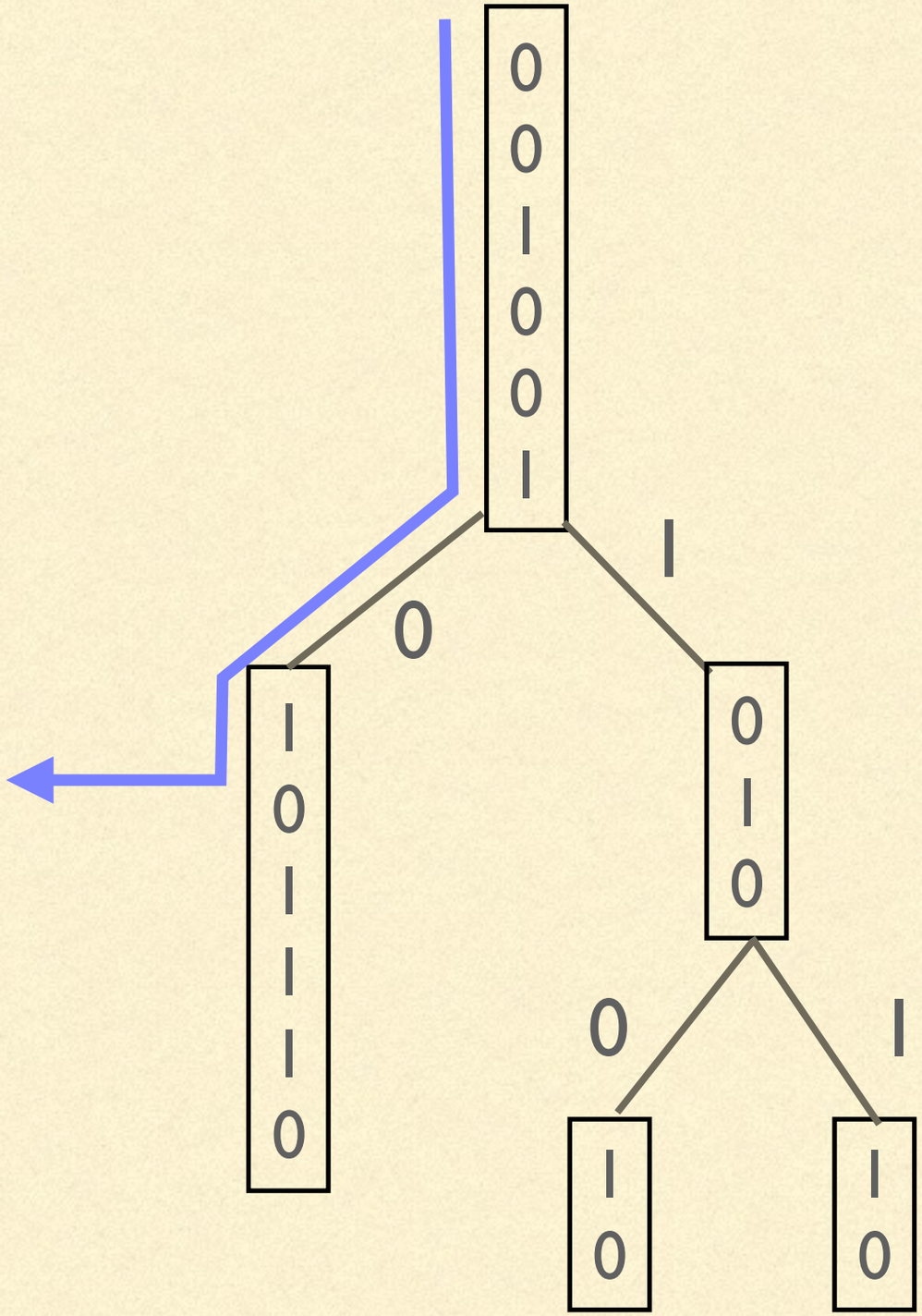
x = 0010011010011



x = 001001010011 0

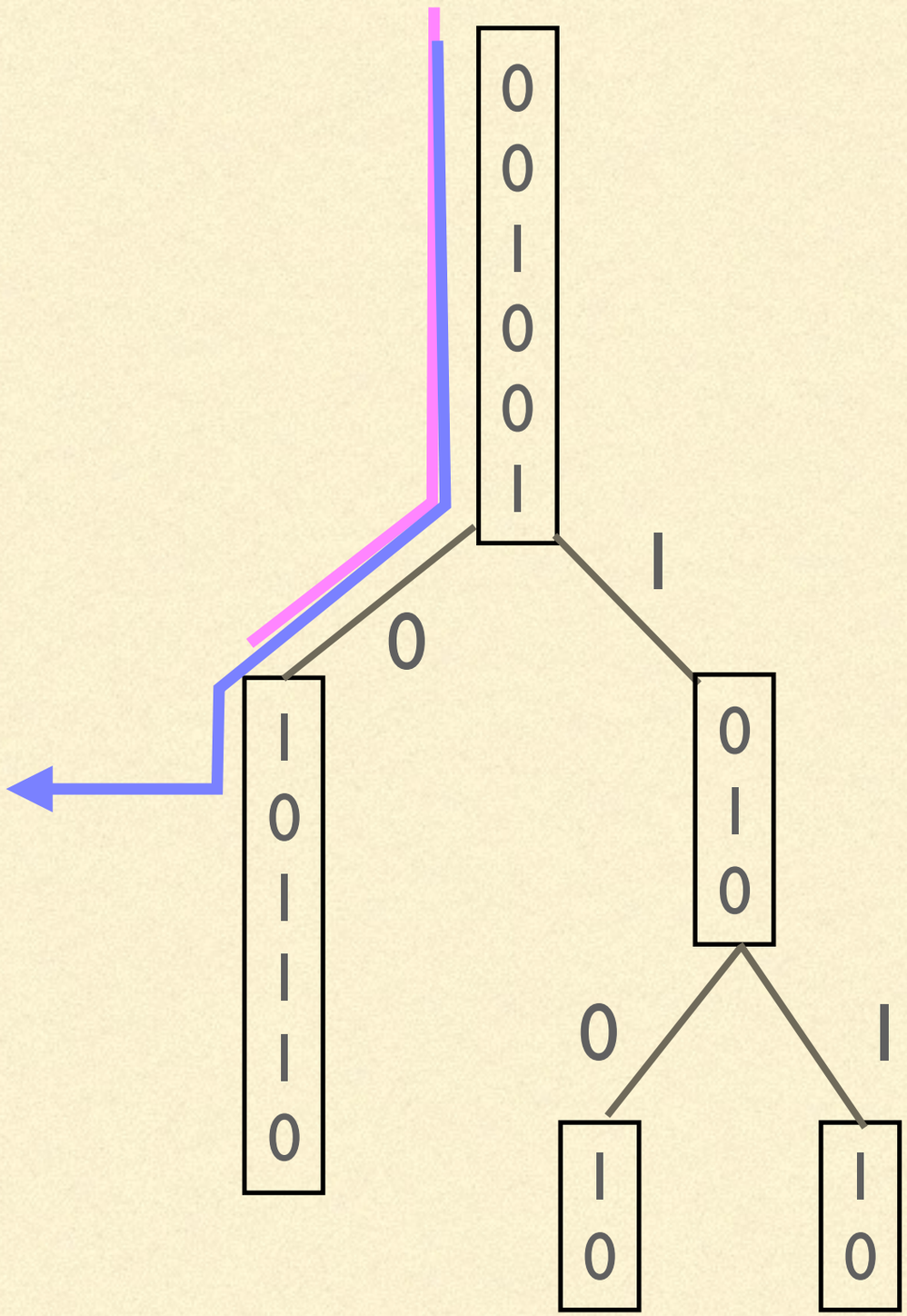


x = 001001010011 0



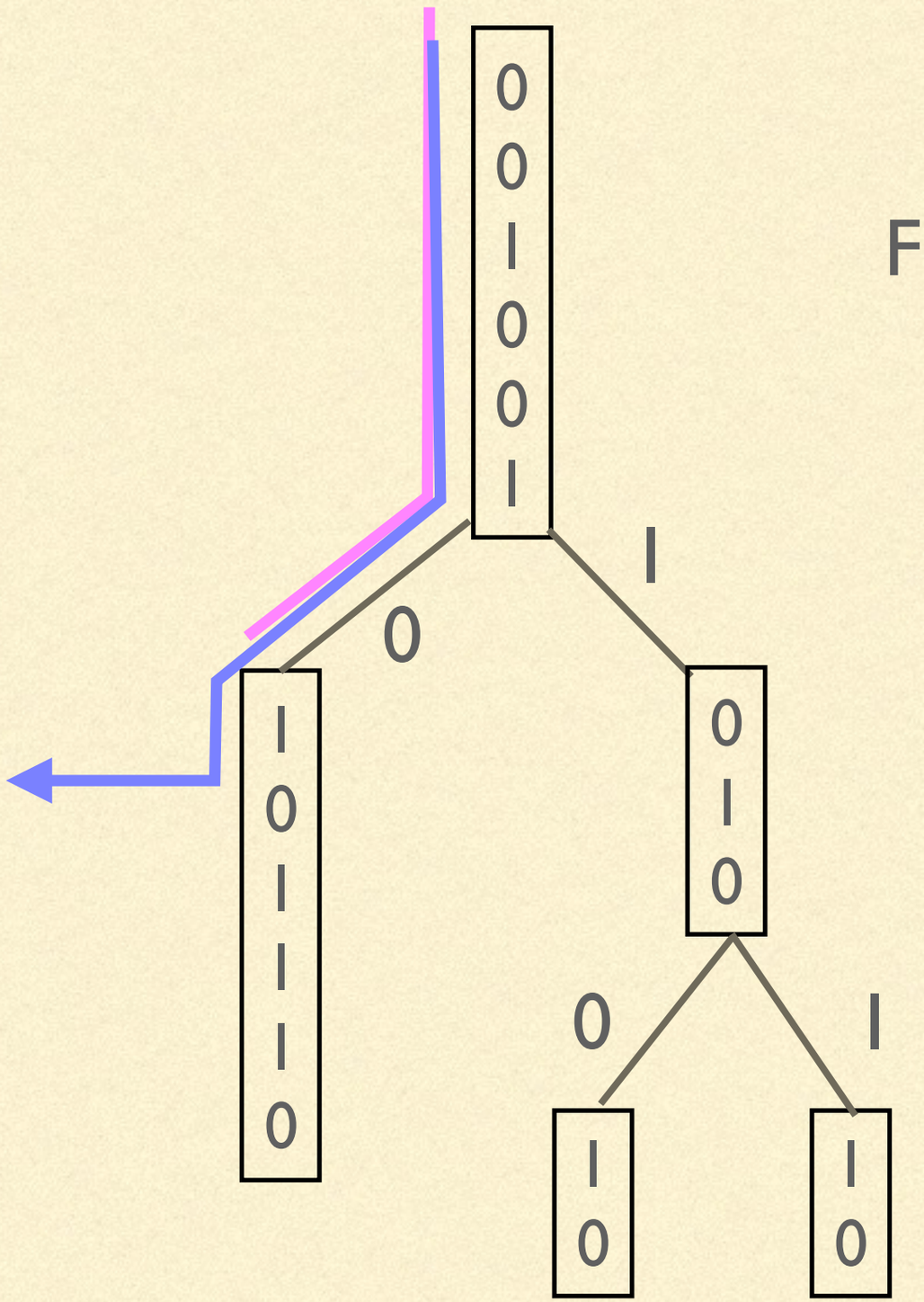
$(l..r) = (0..13) \Rightarrow f = 8$

$x = 0010010100110$



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

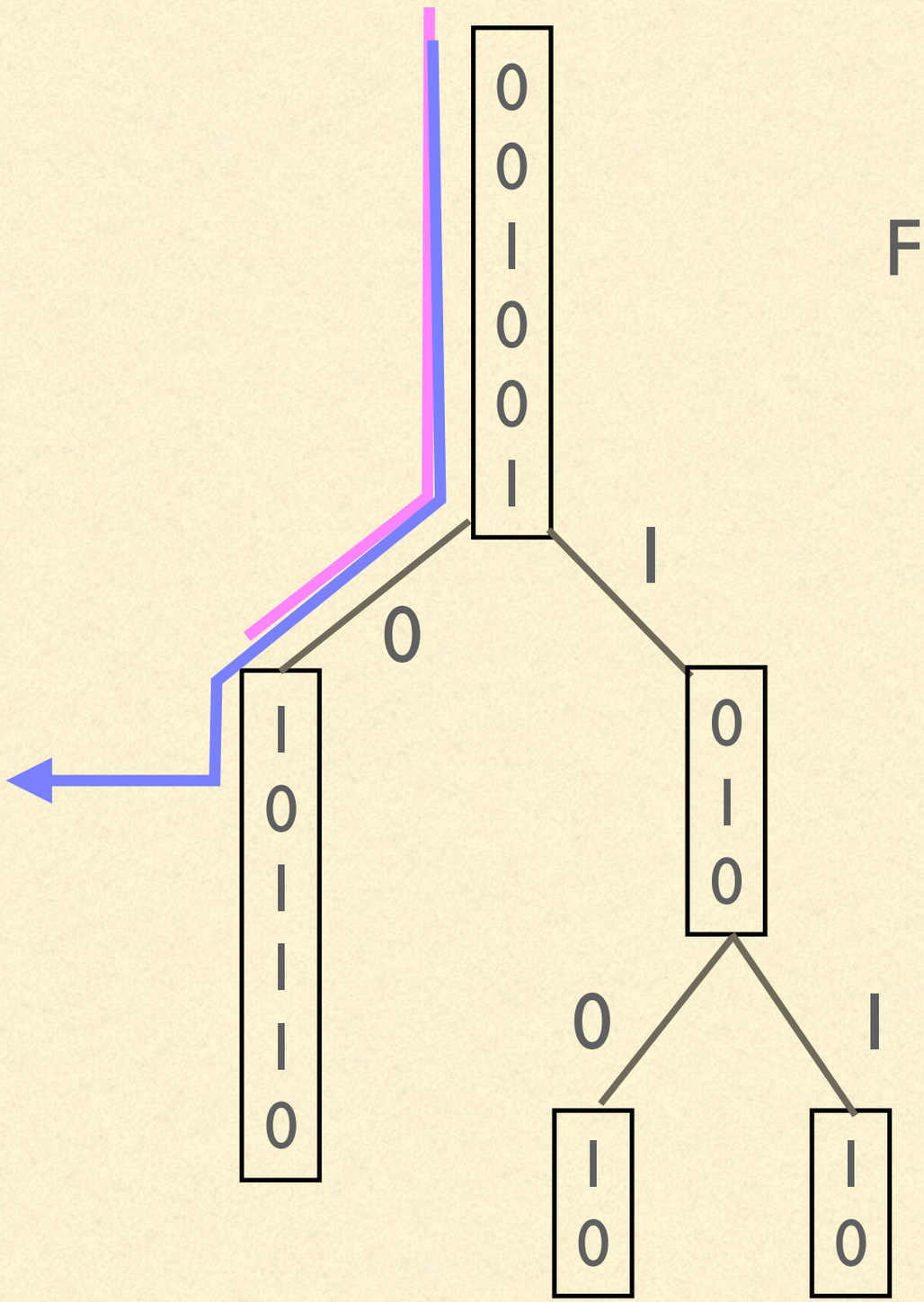
$x = 001001010011 0$



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

Function returns $0010010101110 \nleq x$

$x = 0010010100110$

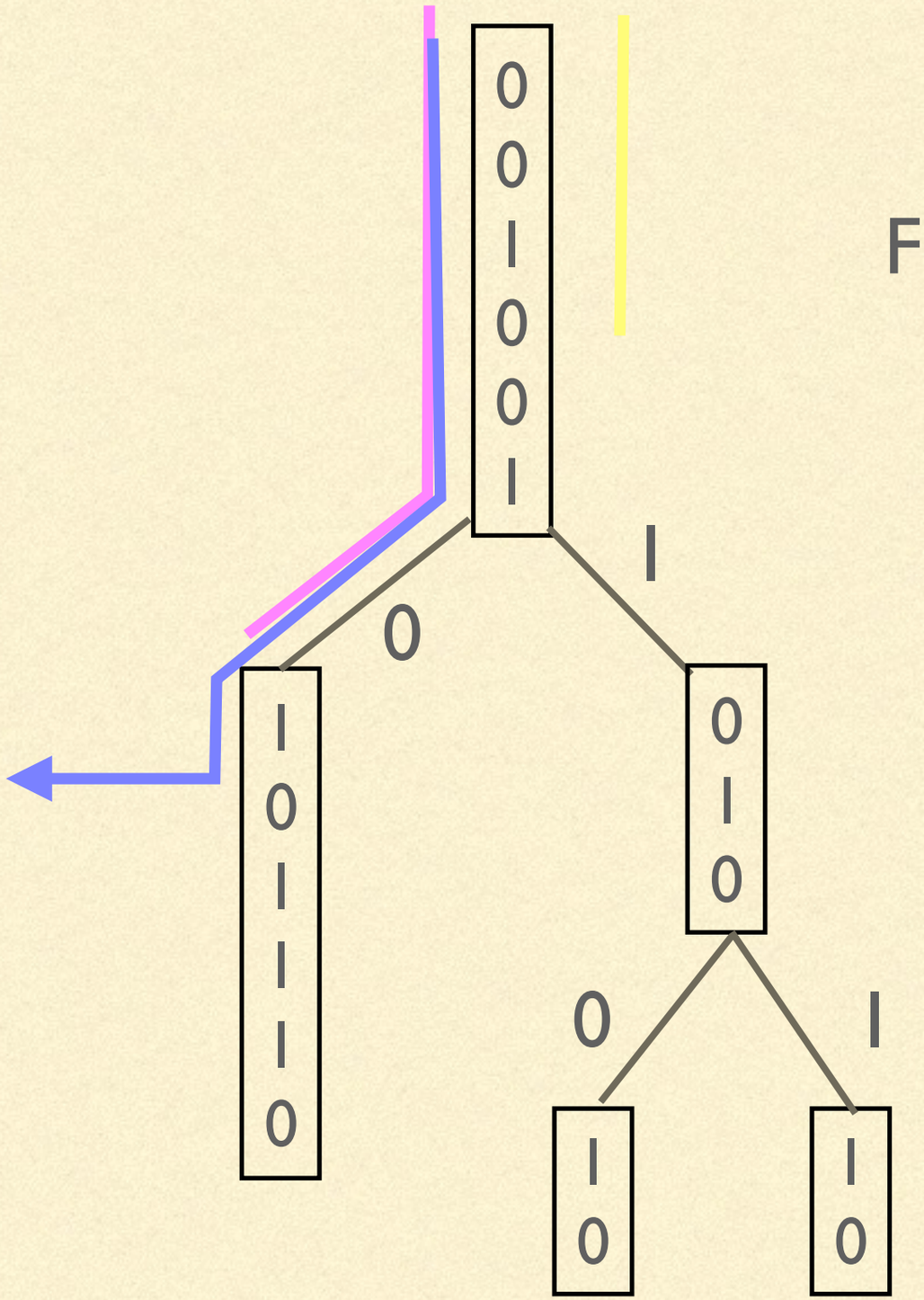


$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $0010010101110 \nleq x$

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

$x = 001001010011 0$

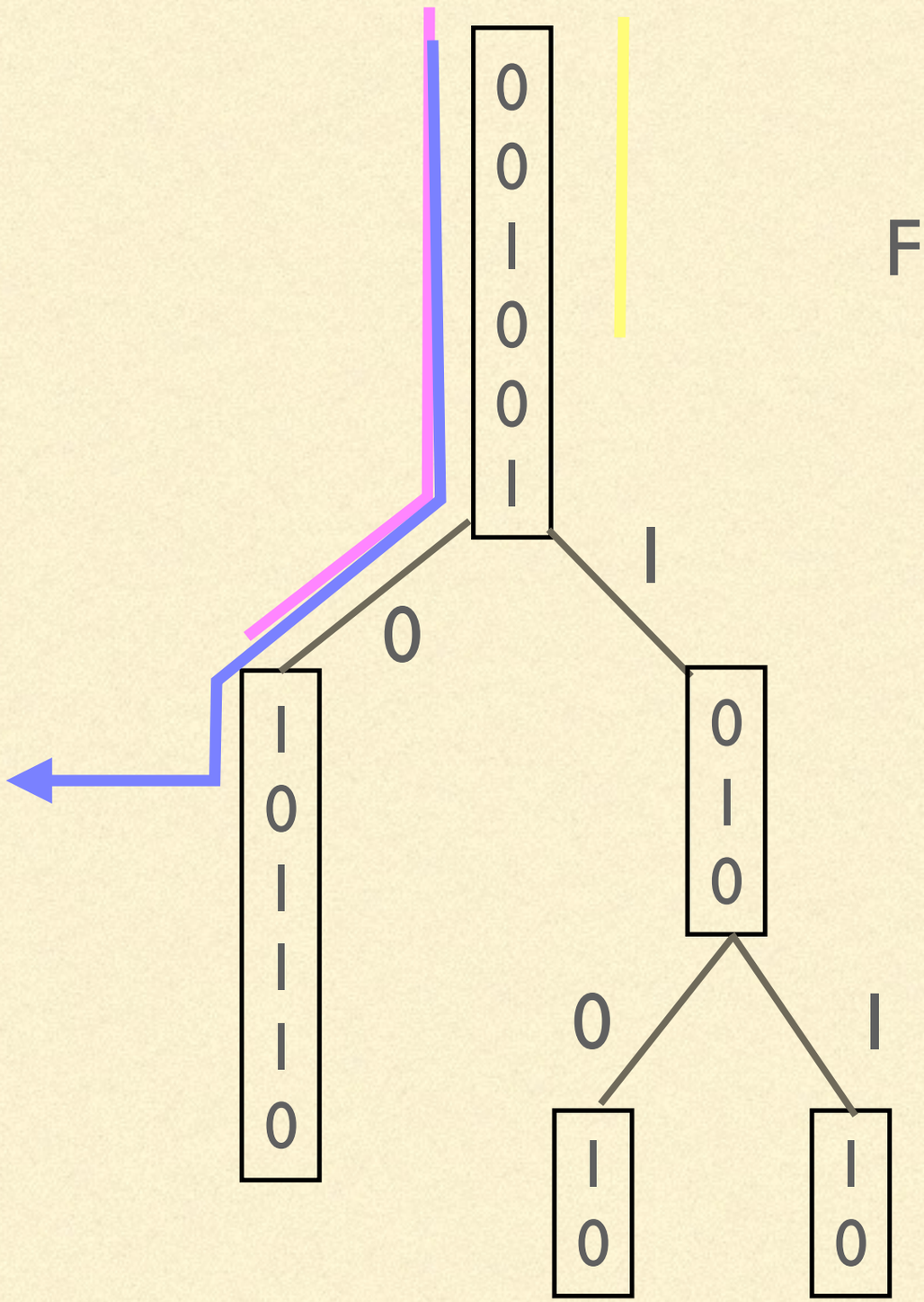


$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $0010010101110 \nleq x$

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

$x = 0010010100110$



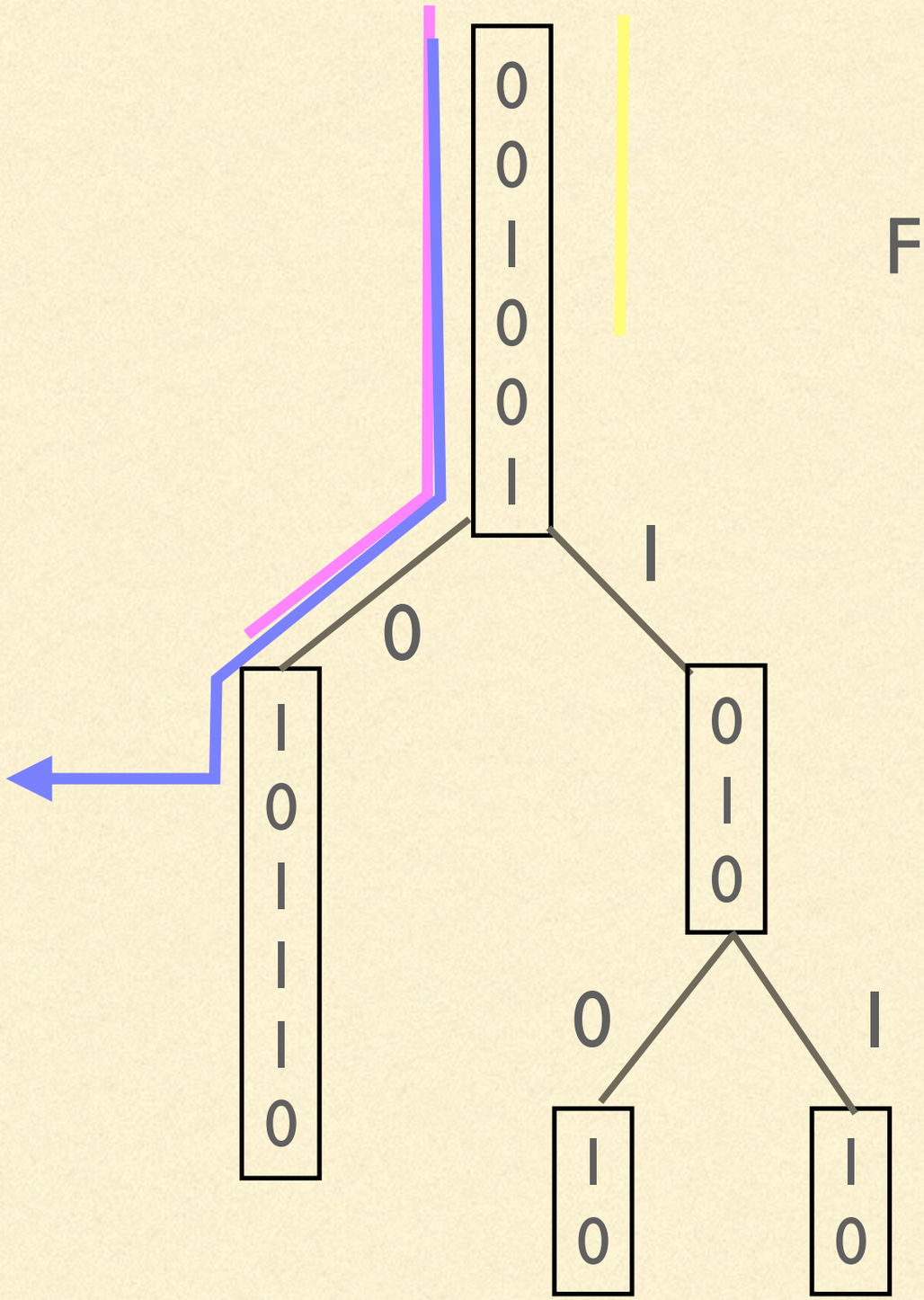
$(l..r) = (0..13) \Rightarrow f = 8$

Function returns $0010010101110 \nleqslant x$

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

Function returns $001001 \leqslant x$

$x = 0010010100110$



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

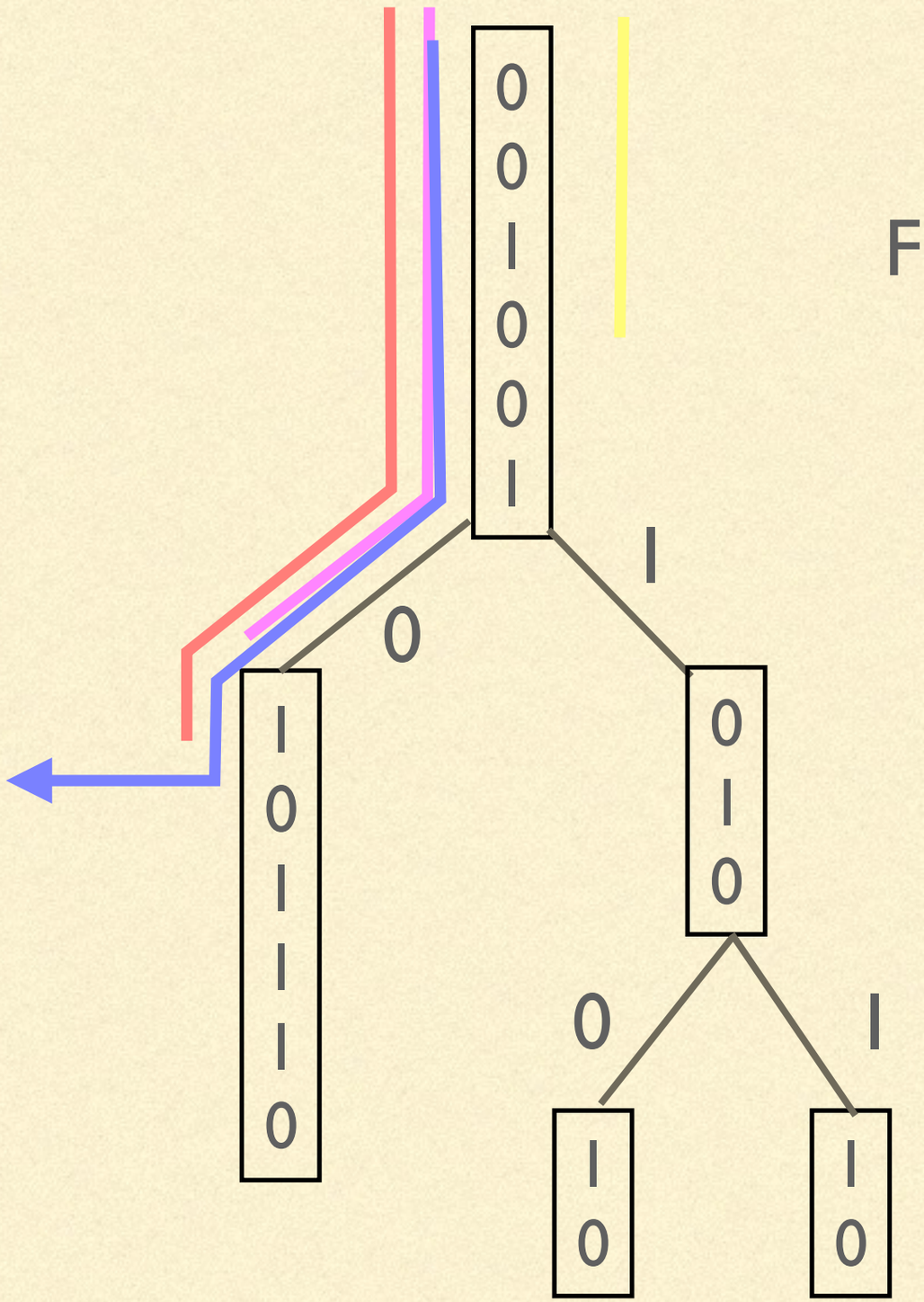
Function returns $0010010101110 \nleqslant x$

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

Function returns $001001 \leqslant x$

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

$x = 0010010100110$



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

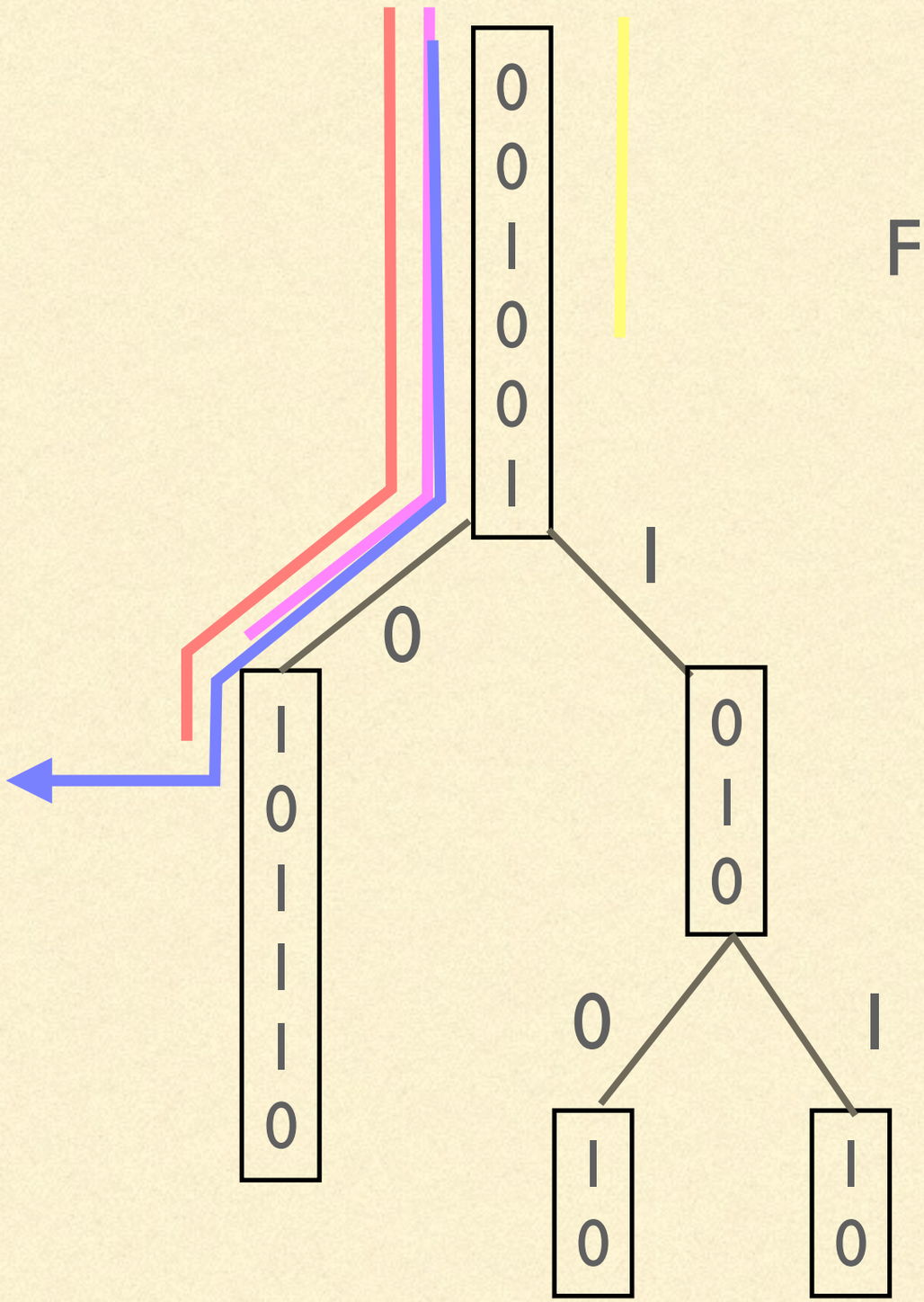
Function returns $0010010101110 \nleqslant x$

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

Function returns $001001 \leqslant x$

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

$x = 0010010100110$



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

Function returns $0010010101110 \nleqslant x$

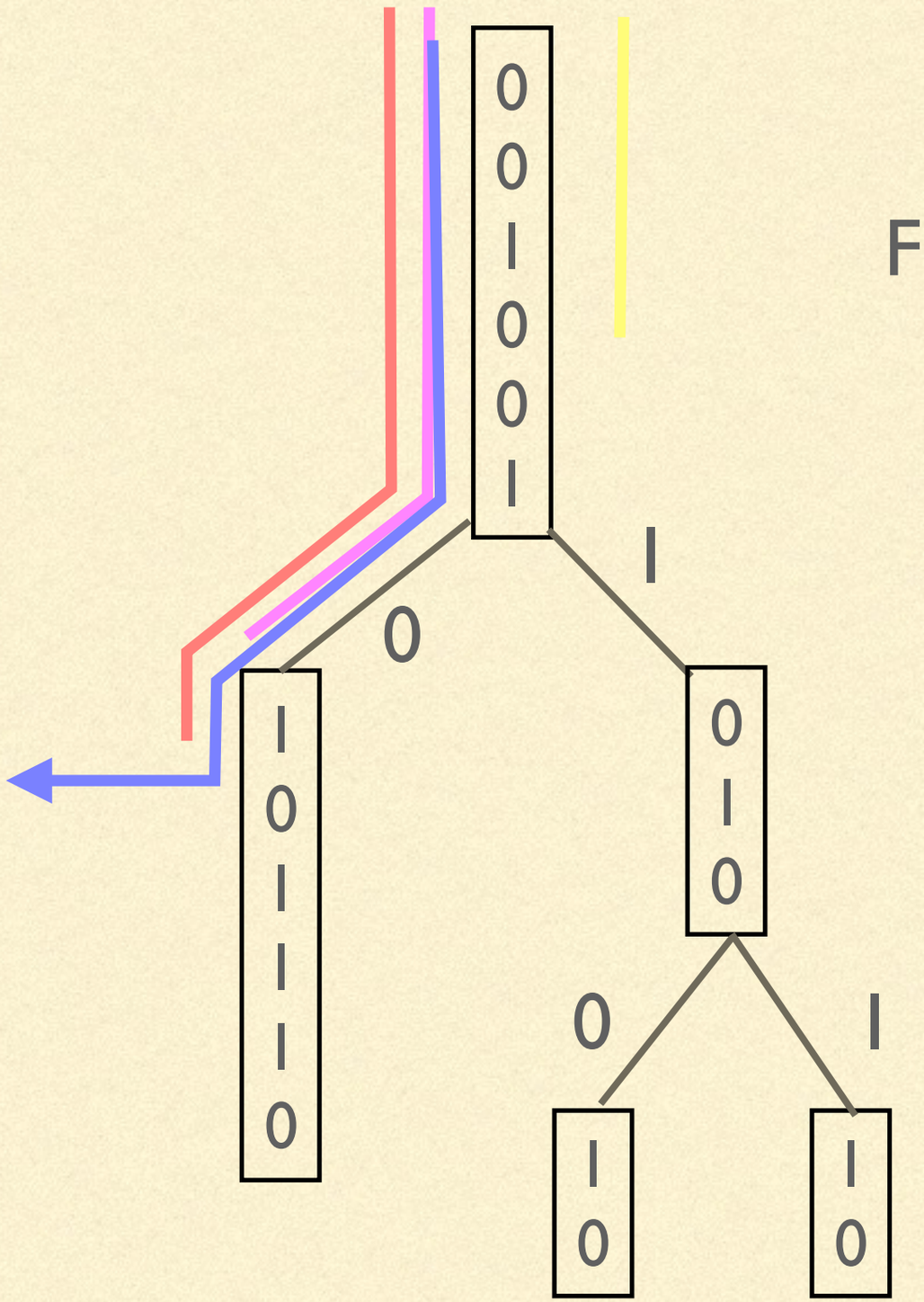
$\Rightarrow (l..r) = (0..8) \Rightarrow f = 4$

Function returns $001001 \leqslant x$

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

Function returns $\perp$

$x = 0010010100110$



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

Function returns $0010010101110 \preccurlyeq x$

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

Function returns $001001 \preccurlyeq x$

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

Function returns $\perp$

$\Rightarrow$ x exits at node x[1,….,8]

x = 0010010100110

WE DON’T KEEP THE PATHS!

WE DON'T KEEP THE PATHS!

- The main point: we don't map handles to extents, but handles to (length, log w bits signatures) of extents!

WE DON'T KEEP THE PATHS!

- The main point: we don't map handles to extents, but handles to (length, log w bits signatures) of extents!
- Space: O(log w) bits per node!

WE DON'T KEEP THE PATHS!

- The main point: we don't map handles to extents, but handles to (length, log w bits signatures) of extents!
- Space: O(log w) bits per node!
- (1) We might make mistakes

WE DON'T KEEP THE PATHS!

- The main point: we don't map handles to extents, but handles to (length, log w bits signatures) of extents!
- Space: O(log w) bits per node!
- (1) We might make mistakes
- (2) We don't know in which direction we exit

WE DON'T KEEP THE PATHS!

- The main point: we don't map handles to extents, but handles to (length, log w bits signatures) of extents!
- Space: O(log w) bits per node!
- (1) We might make mistakes
- (2) We don't know in which direction we exit
- (3) The only information we get is the name of the exit node—not a bucket!

WE DON'T KEEP THE PATHS!

- The main point: we don't map handles to extents, but handles to (length, log w bits signatures) of extents!
- Space: O(log w) bits per node!
- (1) We might make mistakes
- (2) We don't know in which direction we exit
- (3) The only information we get is the name of the exit node—not a bucket!
- The latter is interesting *per se*

WE DON'T KEEP THE PATHS!

- The main point: we don't map handles to extents, but handles to (length, log w bits signatures) of extents!
- Space: O(log w) bits per node!
- (1) We might make mistakes
- (2) We don't know in which direction we exit
- (3) The only information we get is the name of the exit node—not a bucket!
- The latter is interesting *per se*

(1) ERRORS

(1) ERRORS

- 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)

(2) EXIT DIRECTION

(2) EXIT DIRECTION

- We need to know whether we exit left/right…

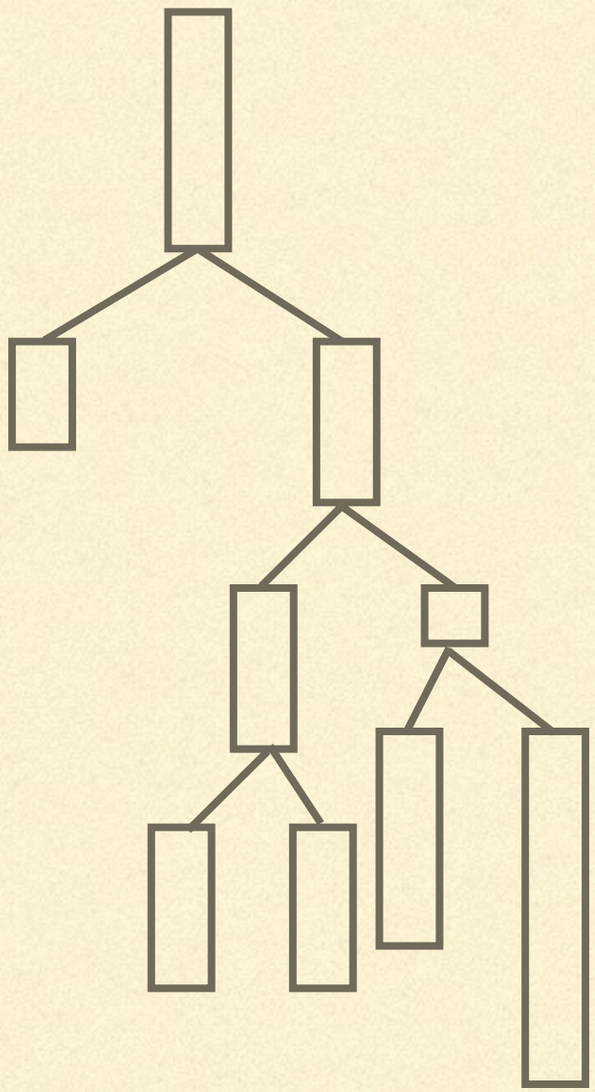
(2) EXIT DIRECTION

- We need to know whether we exit left/right…
- We can just store a function $S \to 2$ to record this fact

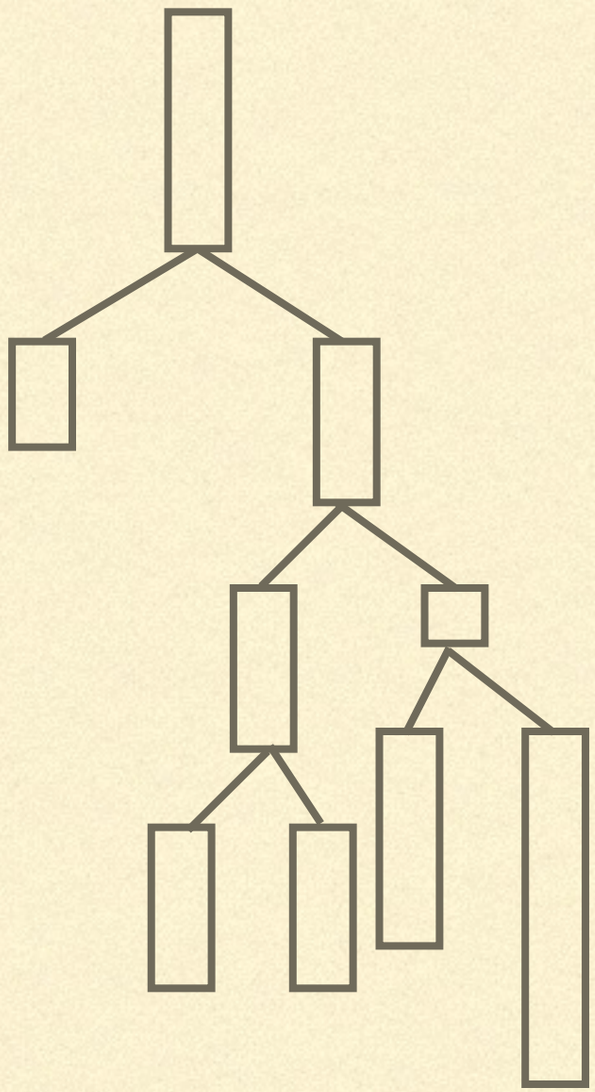
(2) EXIT DIRECTION

- We need to know whether we exit left/right…
- We can just store a function $S \rightarrow 2$ to record this fact
- Space needed: 2n bits

(3) FROM NAMES TO RANKS

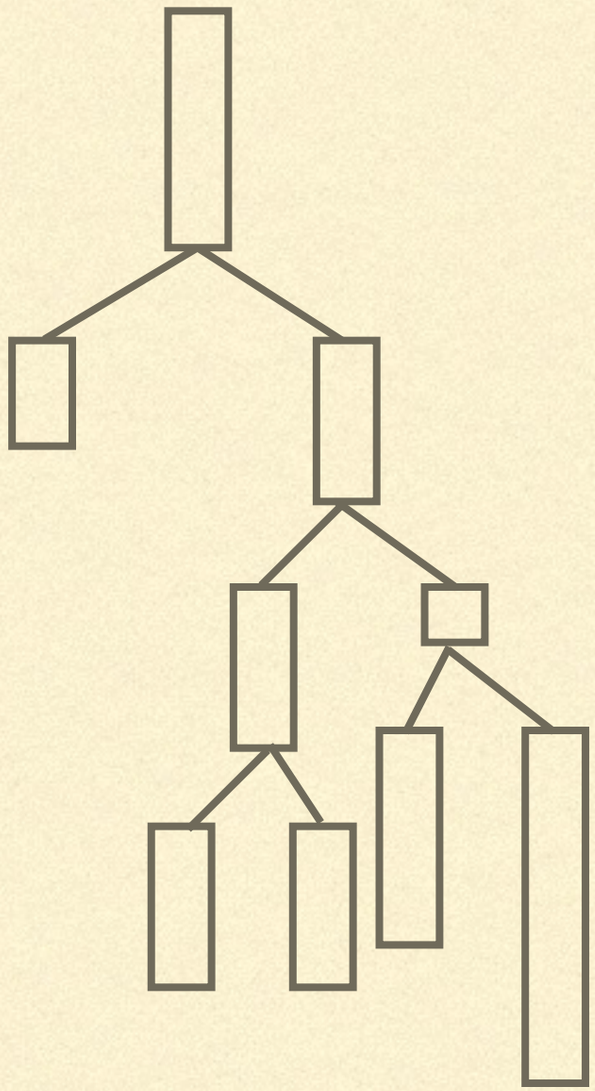


(3) FROM NAMES TO RANKS



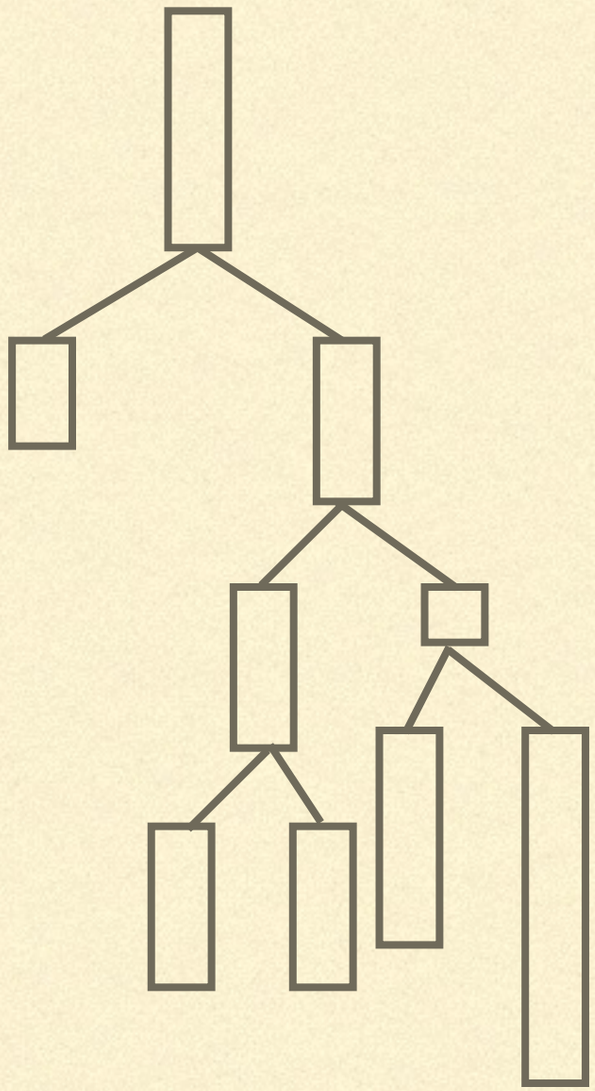
- Given the name of the exit node & the exit direction

(3) FROM NAMES TO RANKS



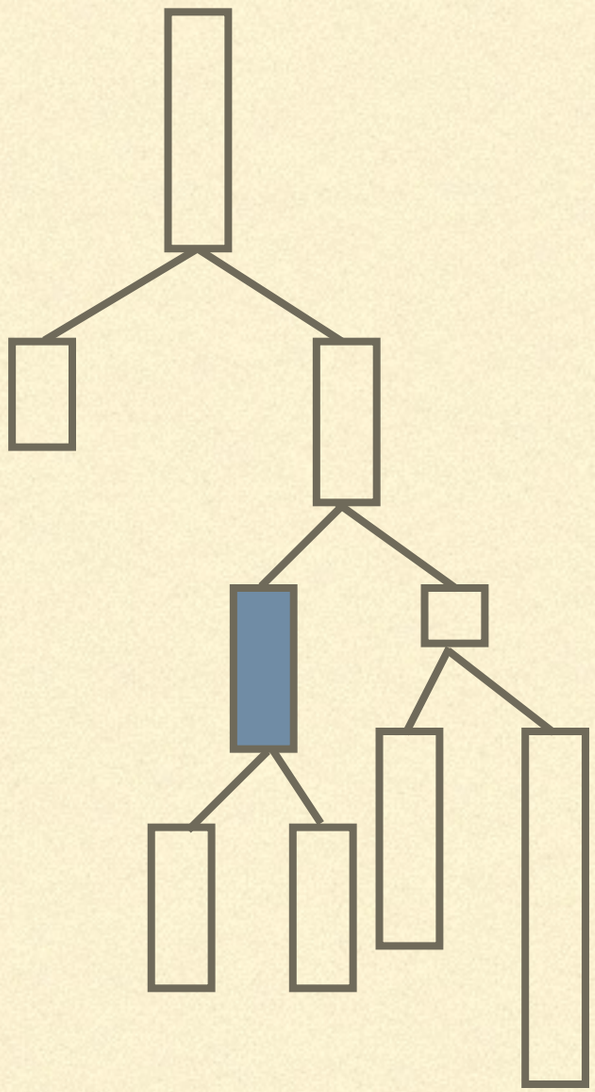
- Given the name of the exit node & the exit direction
- We need to know how many leaves we leave *on our left*

(3) FROM NAMES TO RANKS

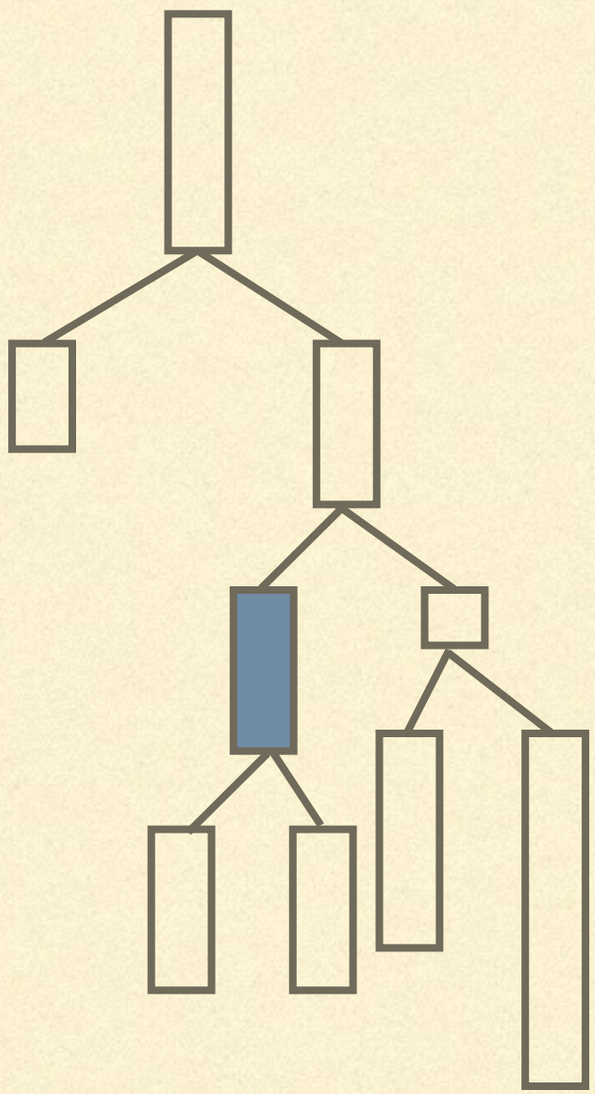


- Given the name of the exit node & the exit direction
- We need to know how many leaves we leave *on our left*
- Four cases

(3) FROM NAMES TO RANKS

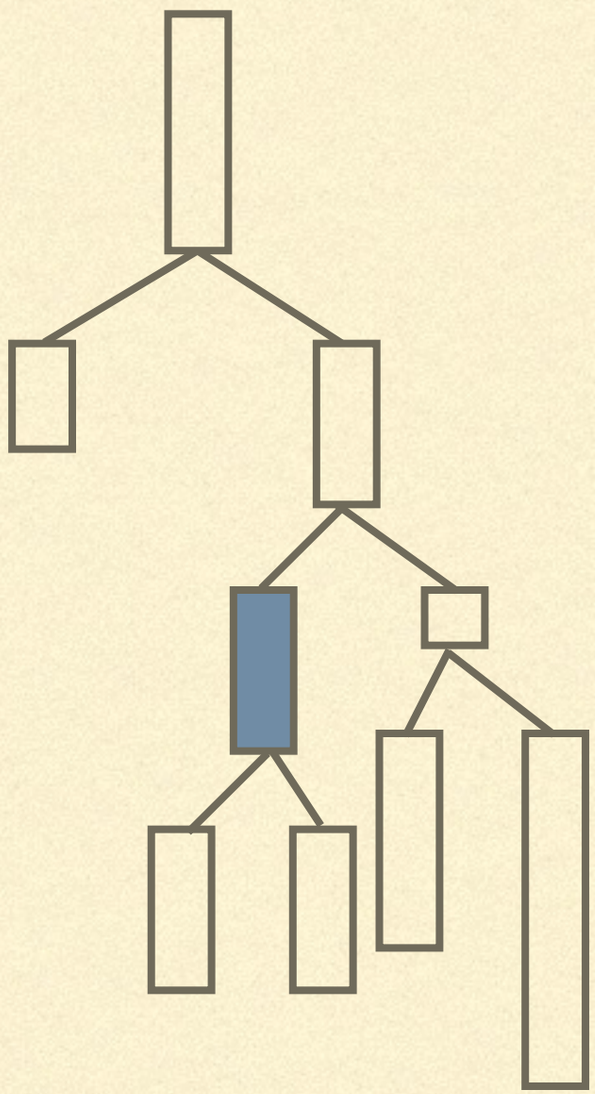


(3) FROM NAMES TO RANKS



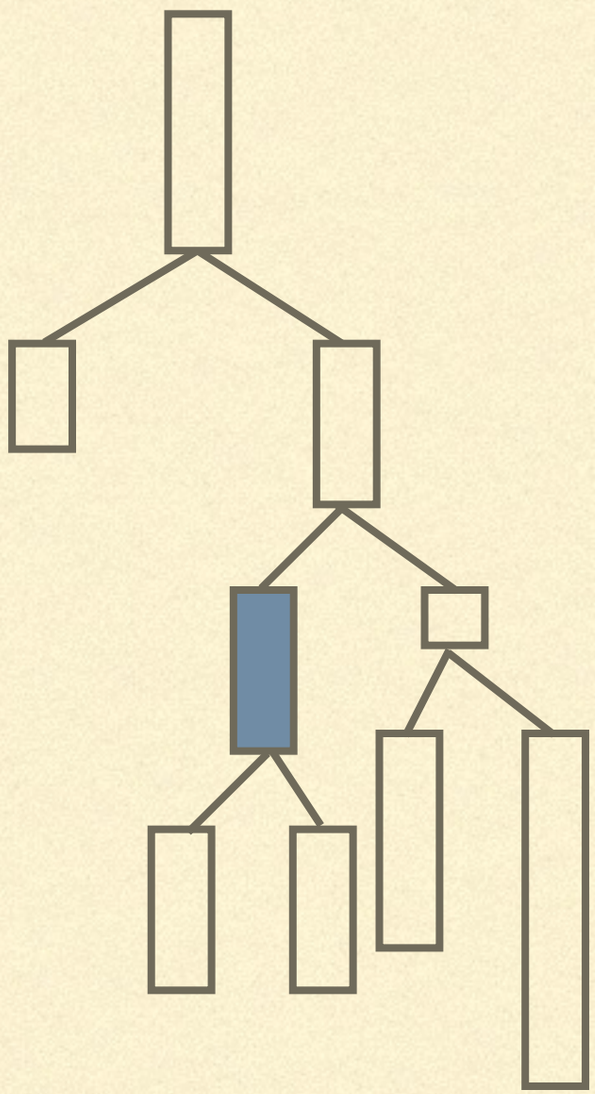
- Exit left on a 0 (i.e. name=e0)

(3) FROM NAMES TO RANKS



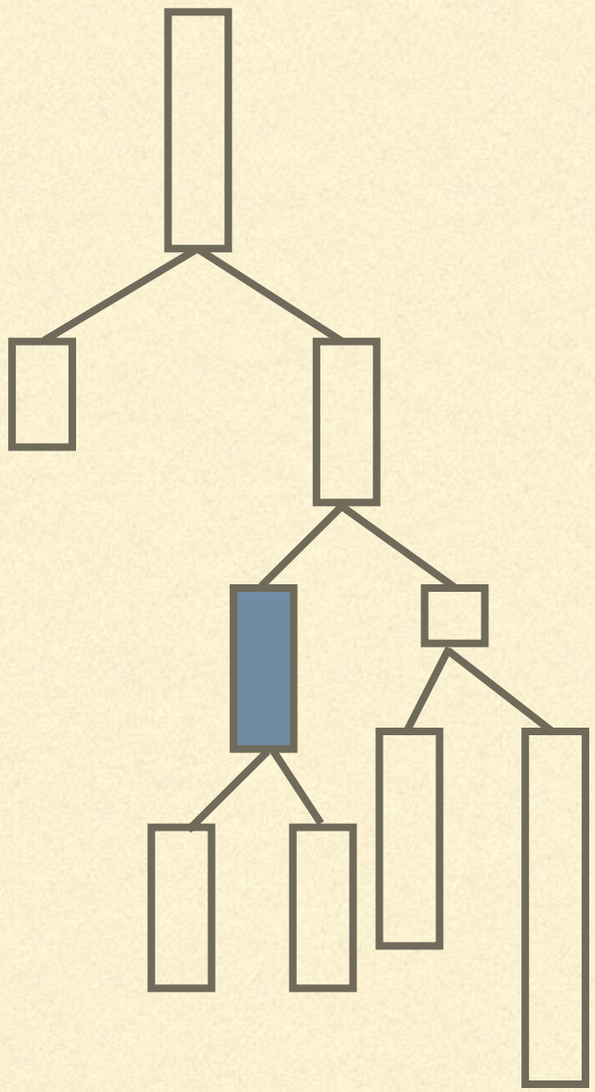
- Exit left on a 0 (i.e. name=e0)
  - count the leaves on the *left*

(3) FROM NAMES TO RANKS



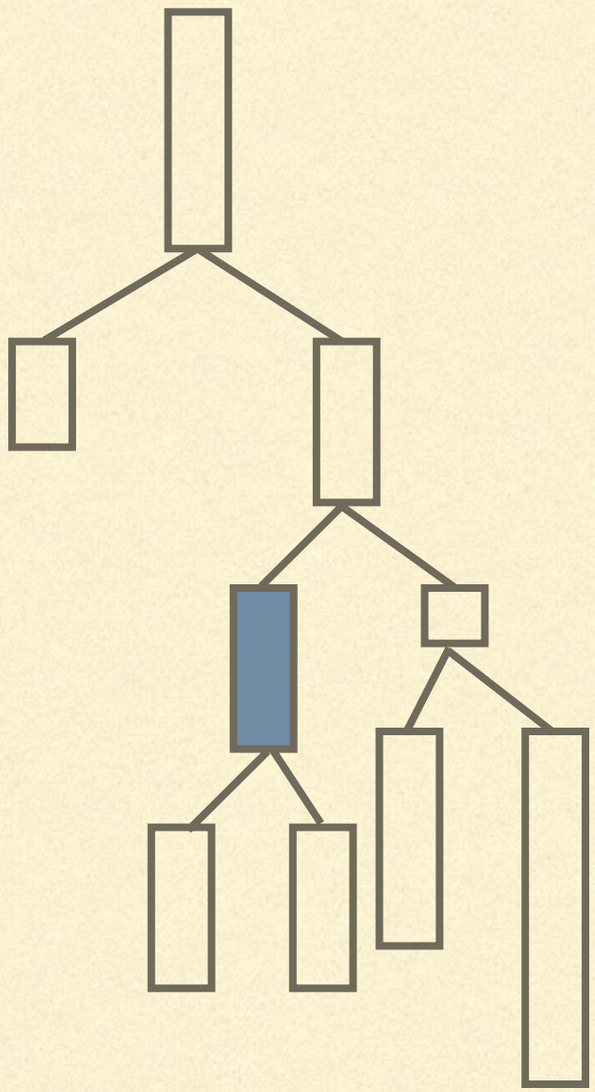
- Exit left on a 0 (i.e. name=e0)
  - count the leaves on the *left*

(3) FROM NAMES TO RANKS

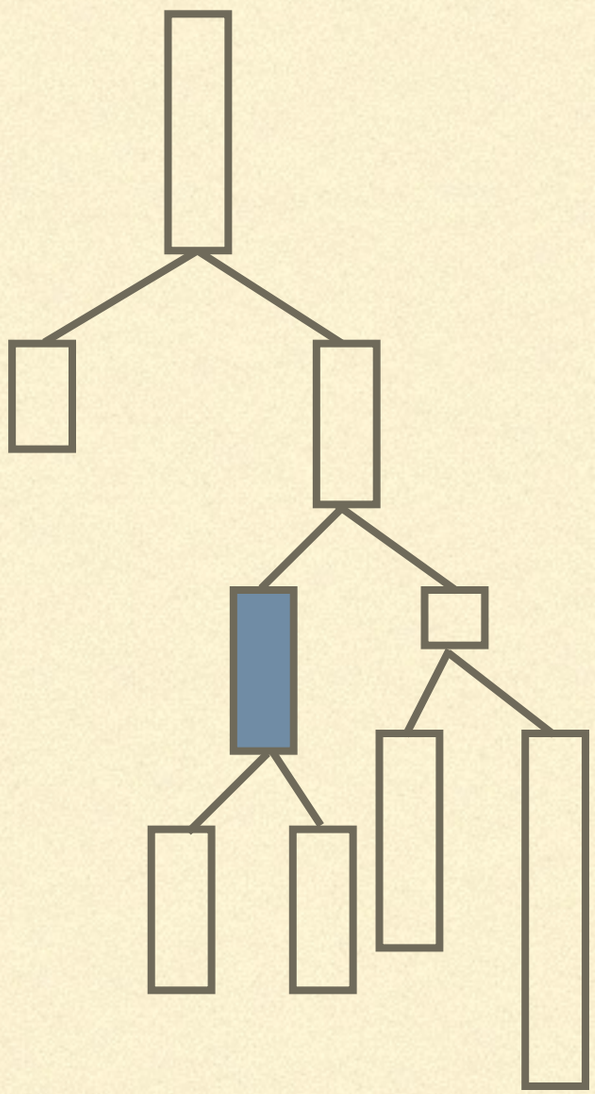


- Exit left on a 0 (i.e. name=e0)
  - count the leaves on the *left*
  - =those that are lexicographically smaller than the e0

(3) FROM NAMES TO RANKS

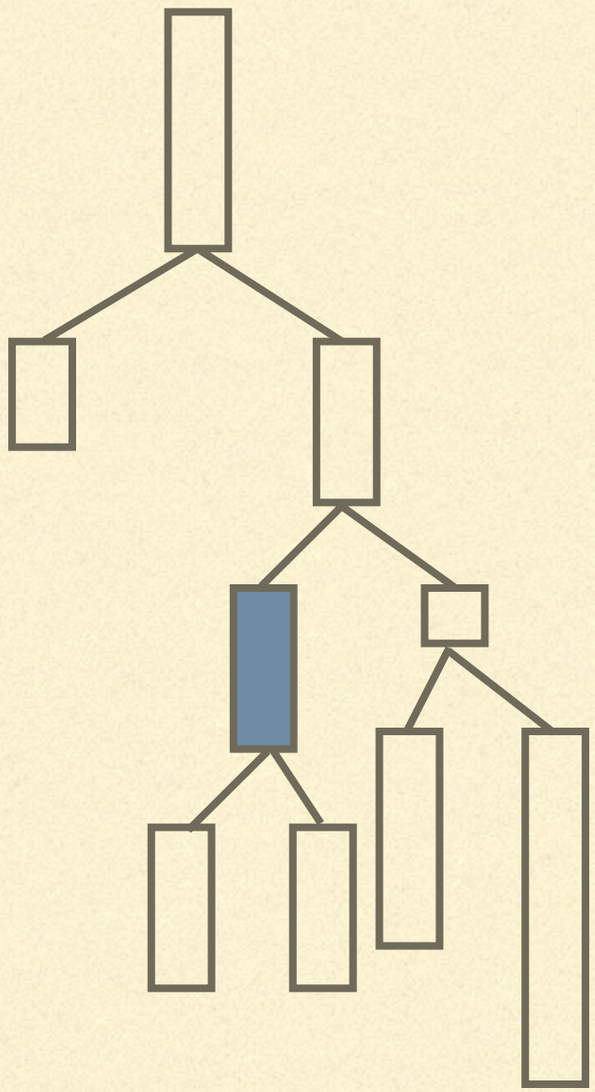


(3) FROM NAMES TO RANKS



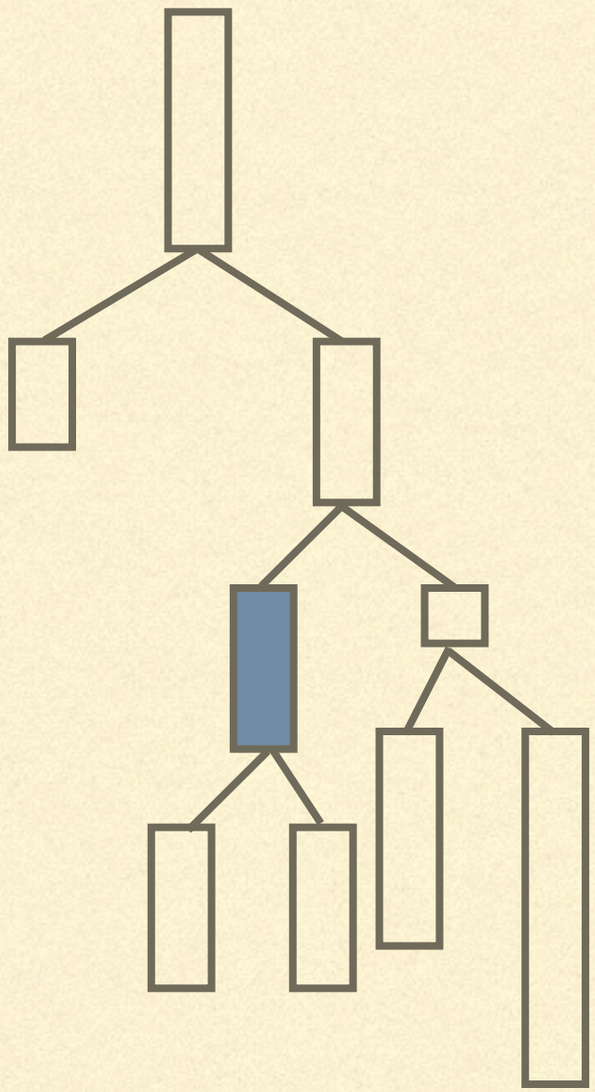
- Exit right on a 0 (i.e. name=e0)

(3) FROM NAMES TO RANKS



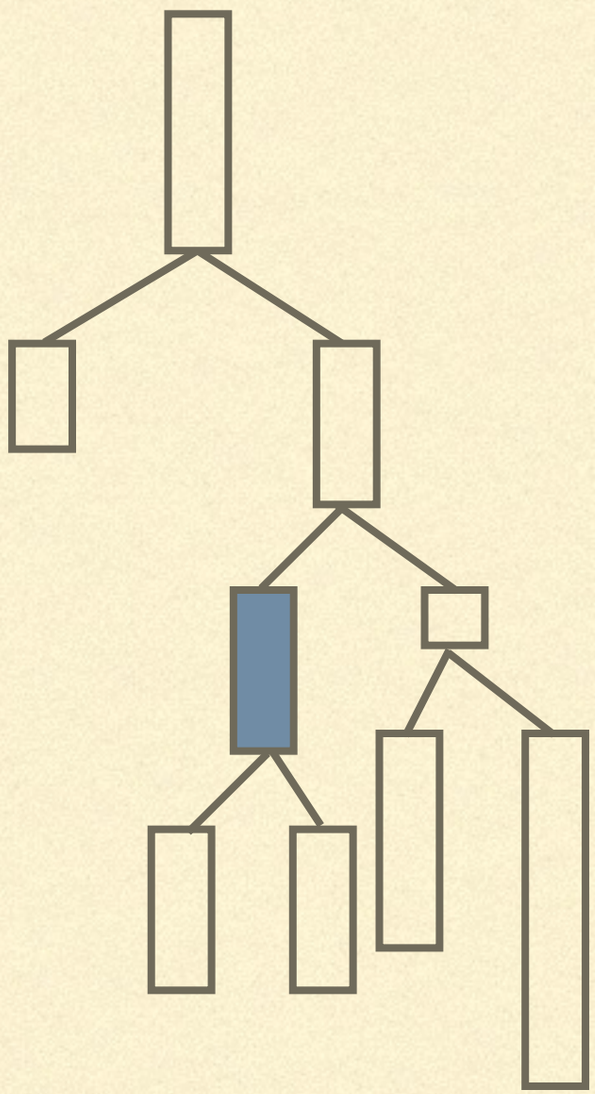
- Exit right on a 0 (i.e. name=e0)
  - count the leaves on the *left and below*

(3) FROM NAMES TO RANKS



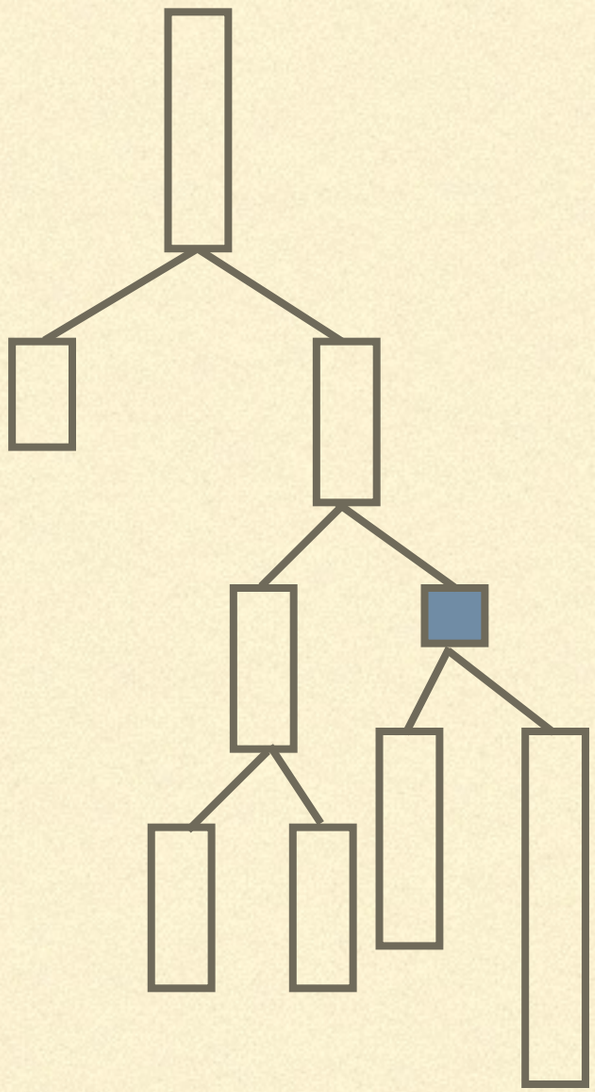
- Exit right on a 0 (i.e. name=e0)
  - count the leaves on the *left and below*

(3) FROM NAMES TO RANKS

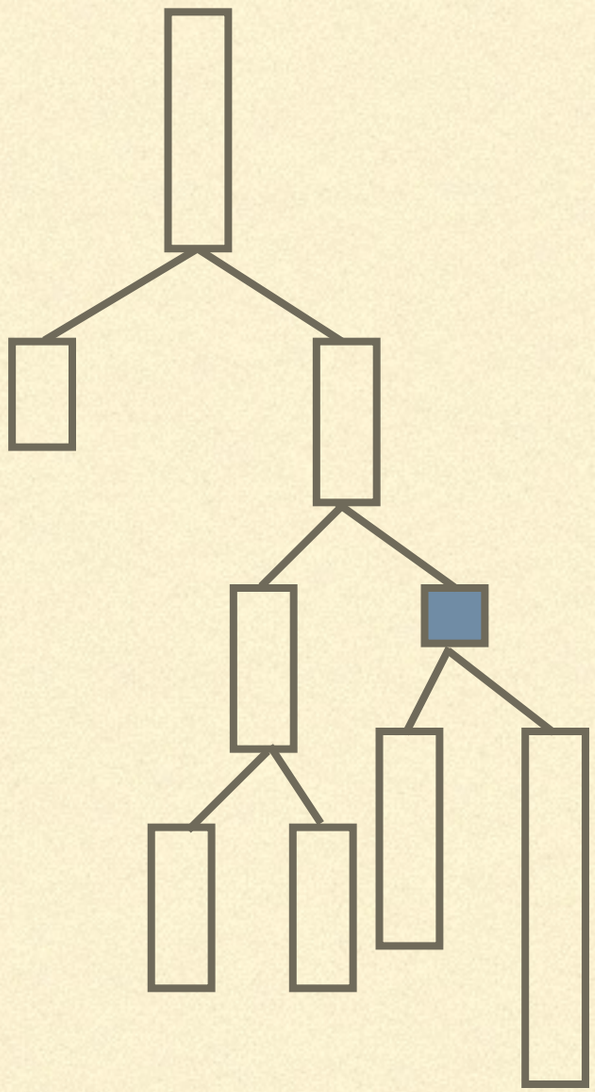


- Exit right on a 0 (i.e. name=e0)
  - count the leaves on the *left and below*
  - =those that are lexicographically smaller than e1

(3) FROM NAMES TO RANKS

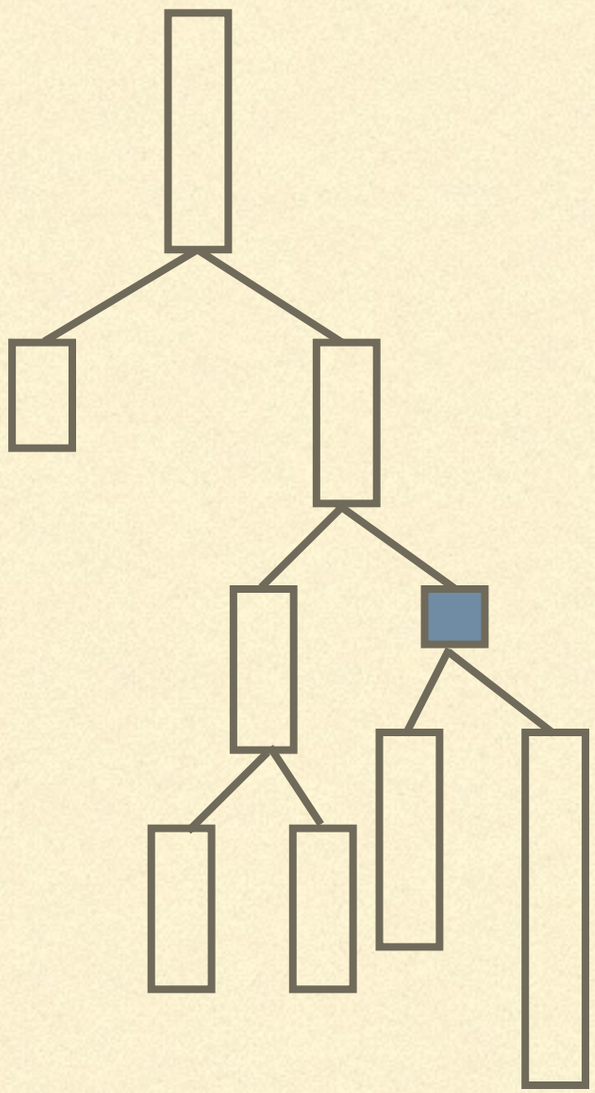


(3) FROM NAMES TO RANKS



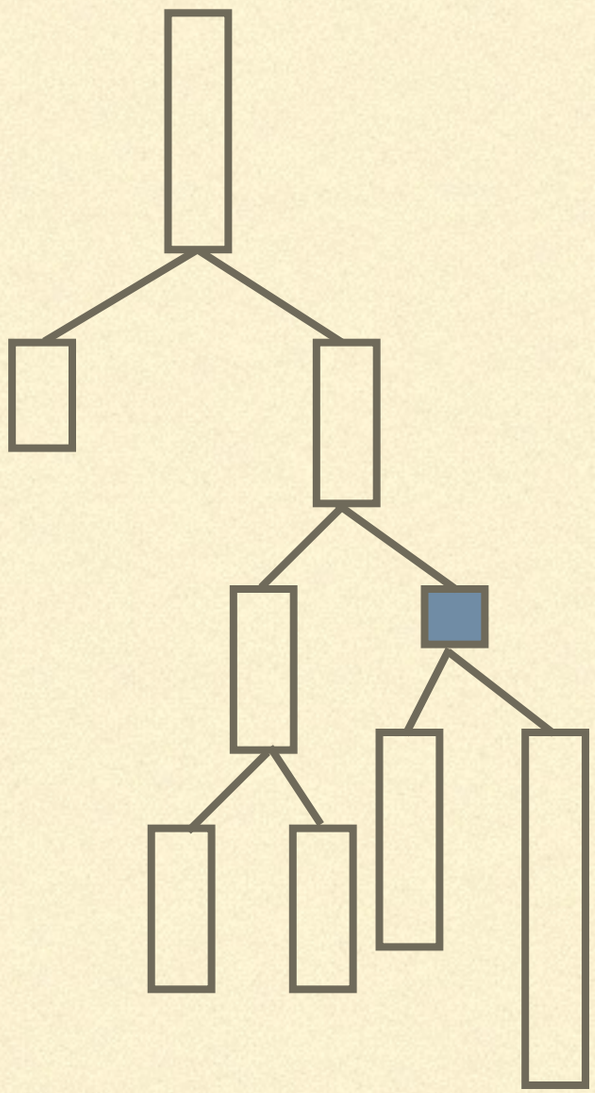
- Exit left on a 1 (i.e. name=e1)

(3) FROM NAMES TO RANKS



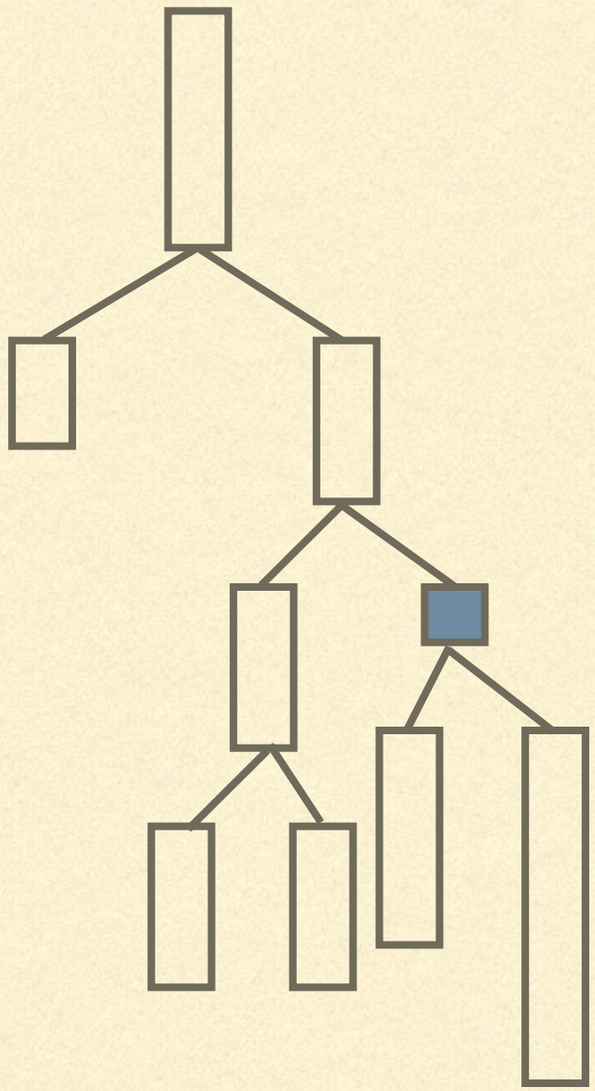
- Exit left on a 1 (i.e. name=e1)
  - count the leaves on the *left*

(3) FROM NAMES TO RANKS



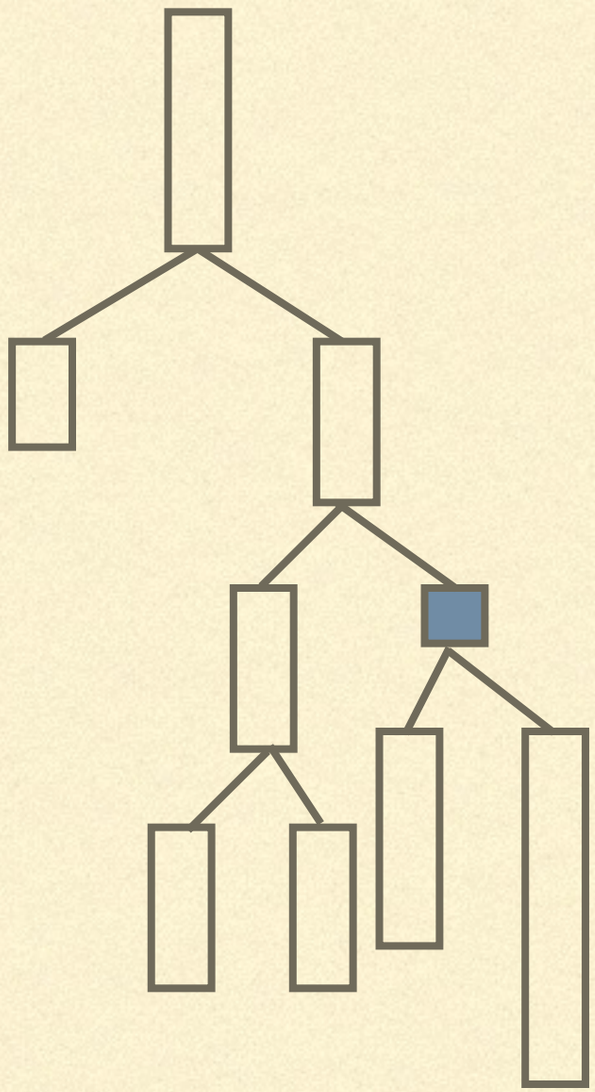
- Exit left on a 1 (i.e. name=e1)
  - count the leaves on the *left*

(3) FROM NAMES TO RANKS

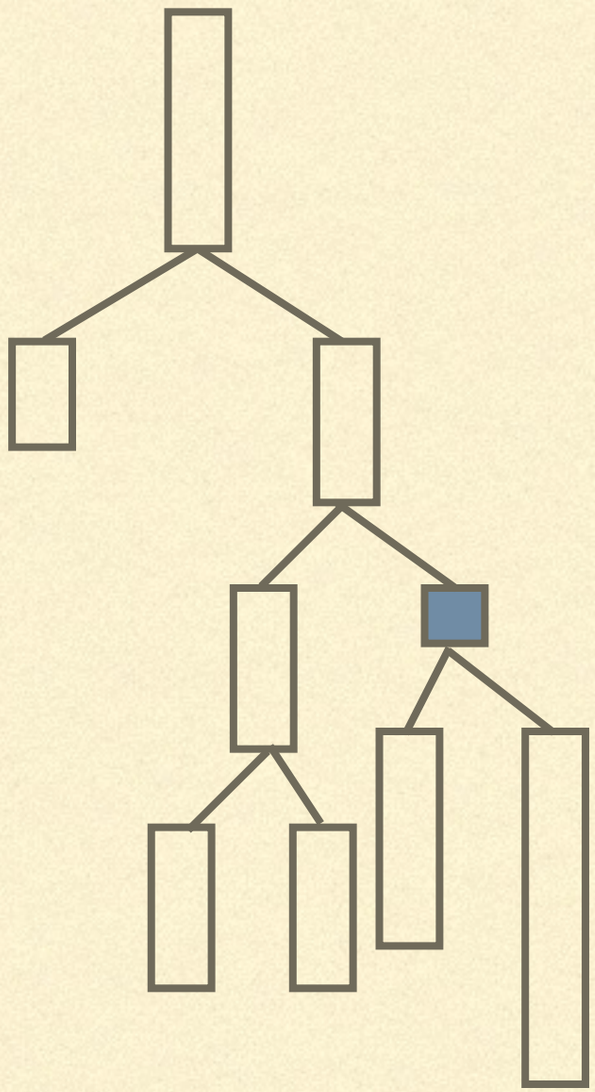


- Exit left on a 1 (i.e. name=e1)
  - count the leaves on the *left*
  - =those that are lexicographically smaller than e1

(3) FROM NAMES TO RANKS

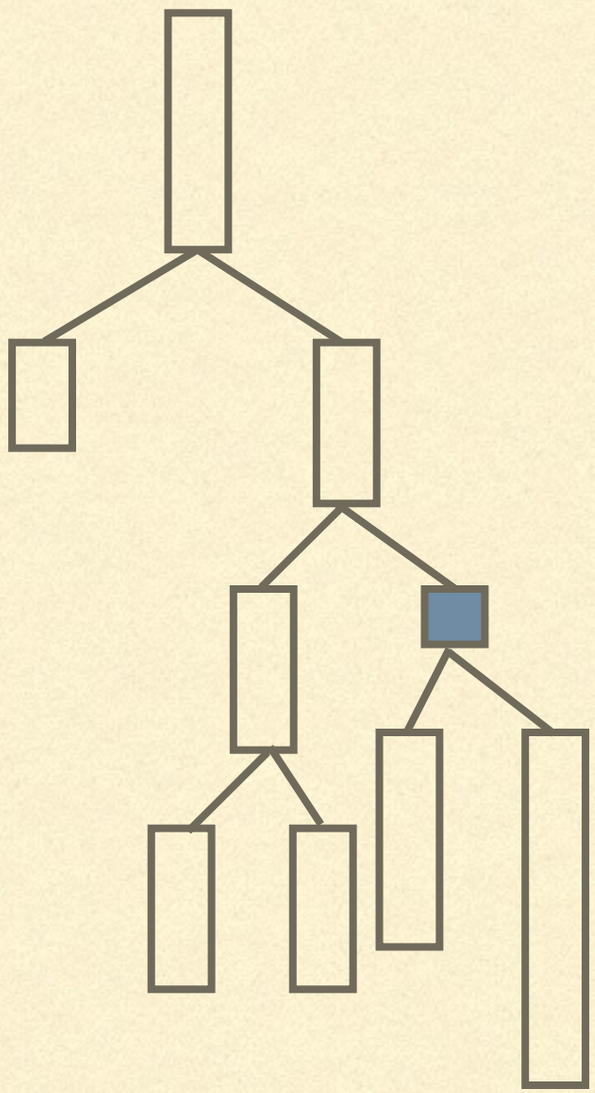


(3) FROM NAMES TO RANKS



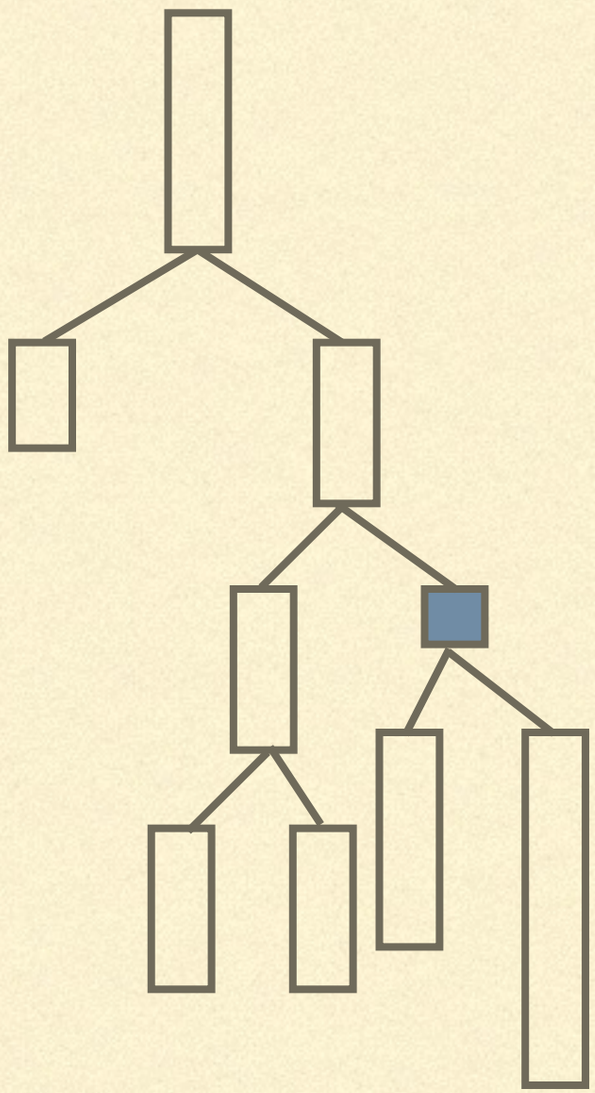
- Exit right on a 1 (i.e. name=e1)

(3) FROM NAMES TO RANKS



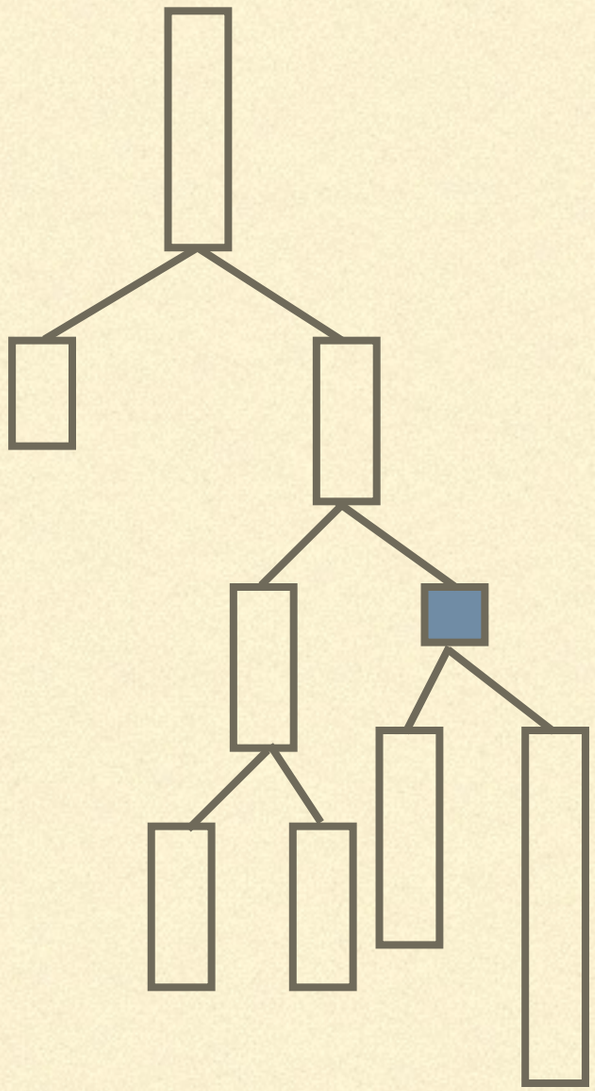
- Exit right on a 1 (i.e. name=e1)
  - count the leaves on the *left and below*

(3) FROM NAMES TO RANKS



- Exit right on a 1 (i.e. name=e1)
  - count the leaves on the *left and below*

(3) FROM NAMES TO RANKS



- Exit right on a 1 (i.e. name=e1)
  - count the leaves on the *left and below*
  - =those that are lexicographically smaller than e1+ (the successor of e1)

(3) FROM NAMES TO RANKS

(3) FROM NAMES TO RANKS

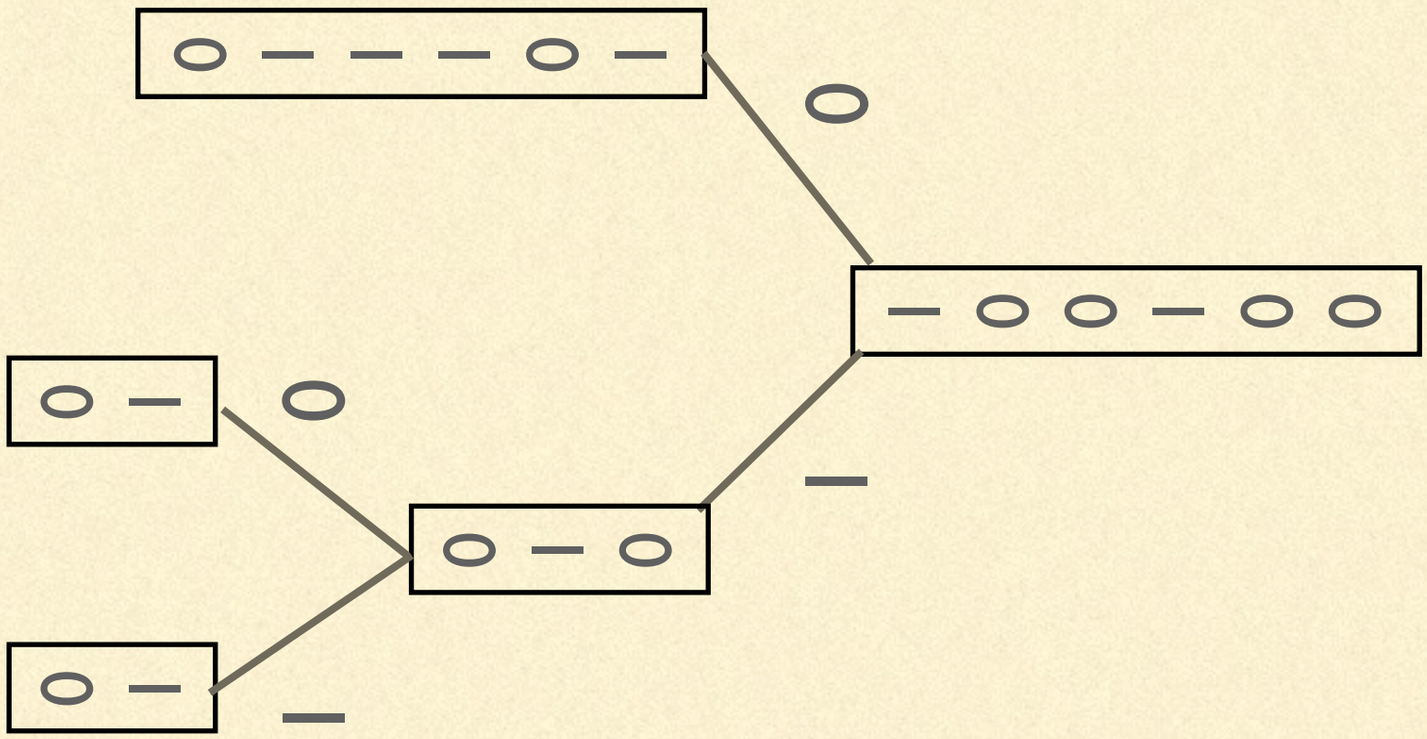
- We build a set P containing, for each extent e of an internal node, the strings e0, e1, e1+ (the successor of e1)

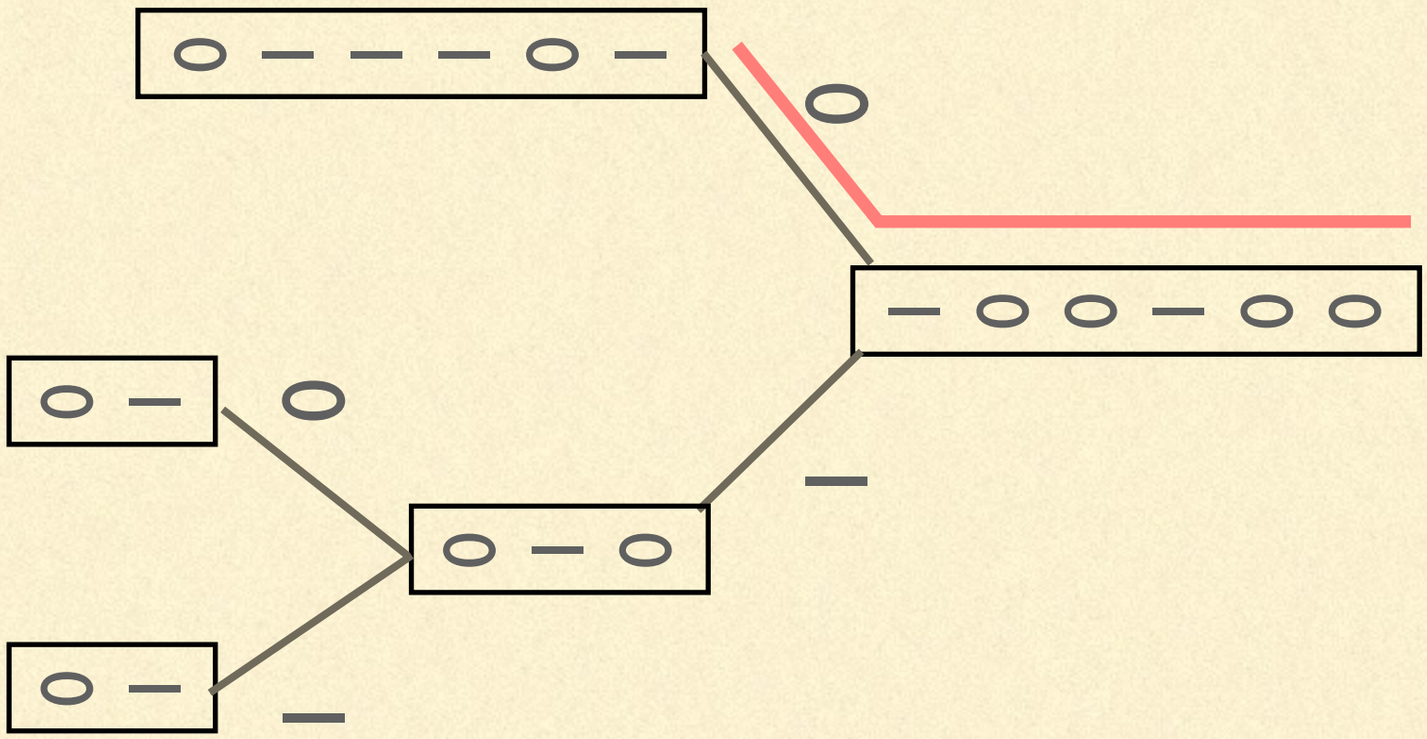
(3) FROM NAMES TO RANKS

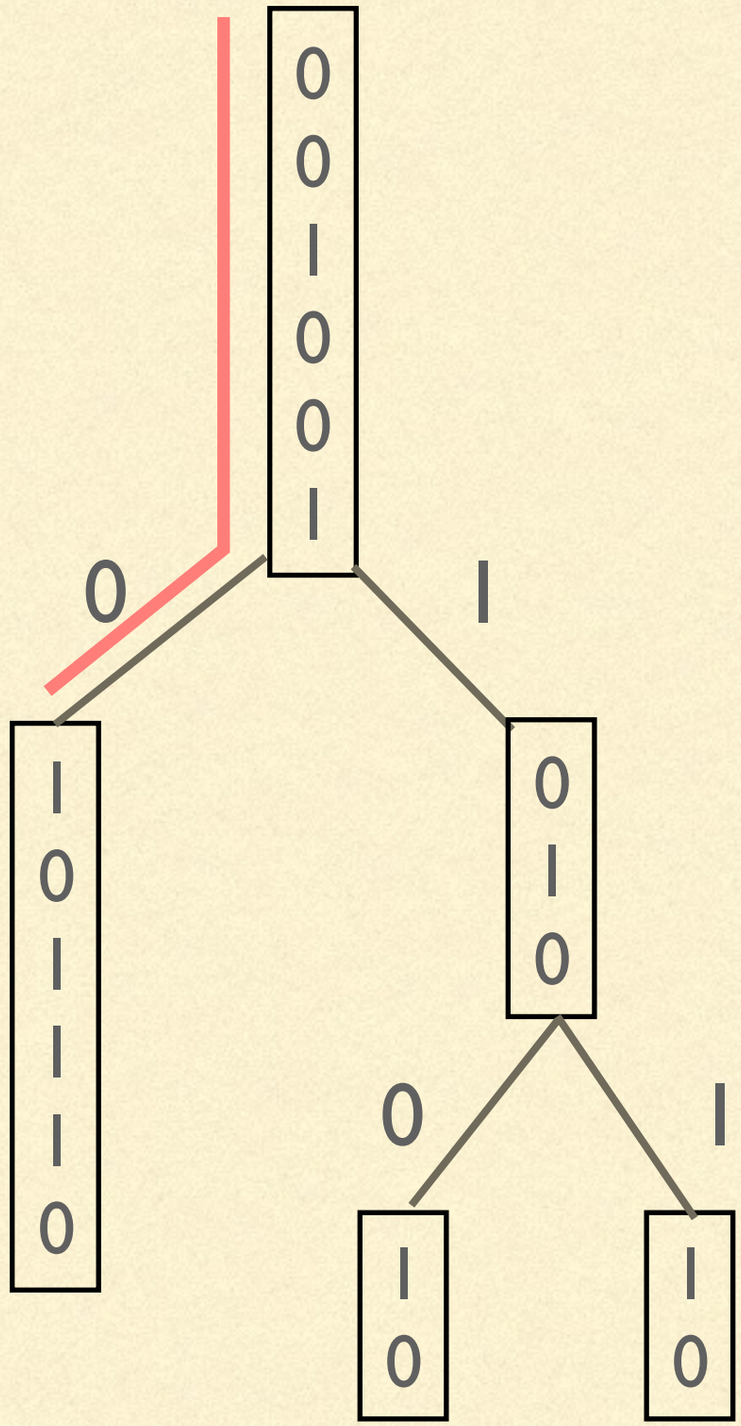
- We build a set P containing, for each extent e of an internal node, the strings e0, e1, e1+ (the successor of e1)
- An LCP MMPH f to map P ↦ bit vector containing 1 in the positions corresponding to names of leaves

(3) FROM NAMES TO RANKS

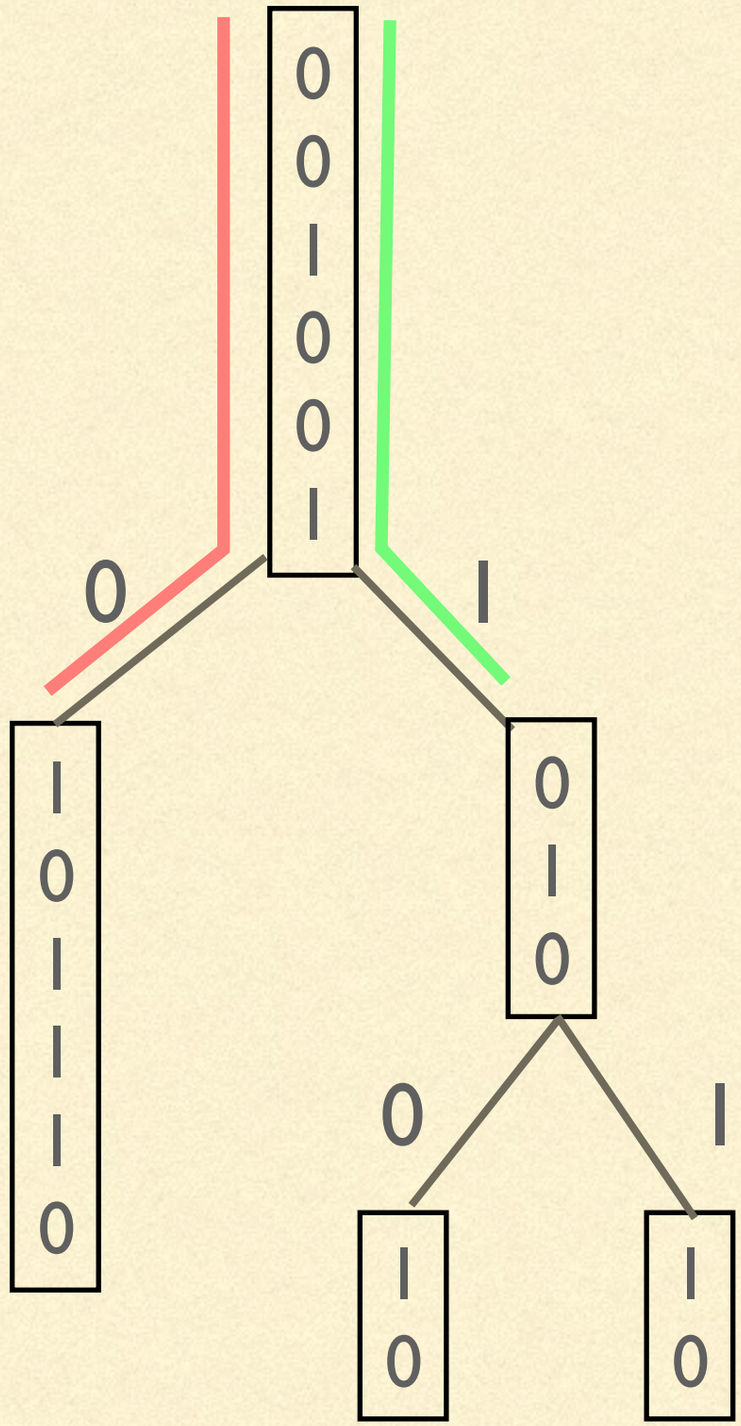
- We build a set P containing, for each extent e of an internal node, the strings e0, e1, e1+ (the successor of e1)
- An LCP MMPH f to map P ↦ bit vector containing 1 in the positions corresponding to names of leaves
- Leave count by ranking (e.g., if we exit right from e0, rank(f(e1)))



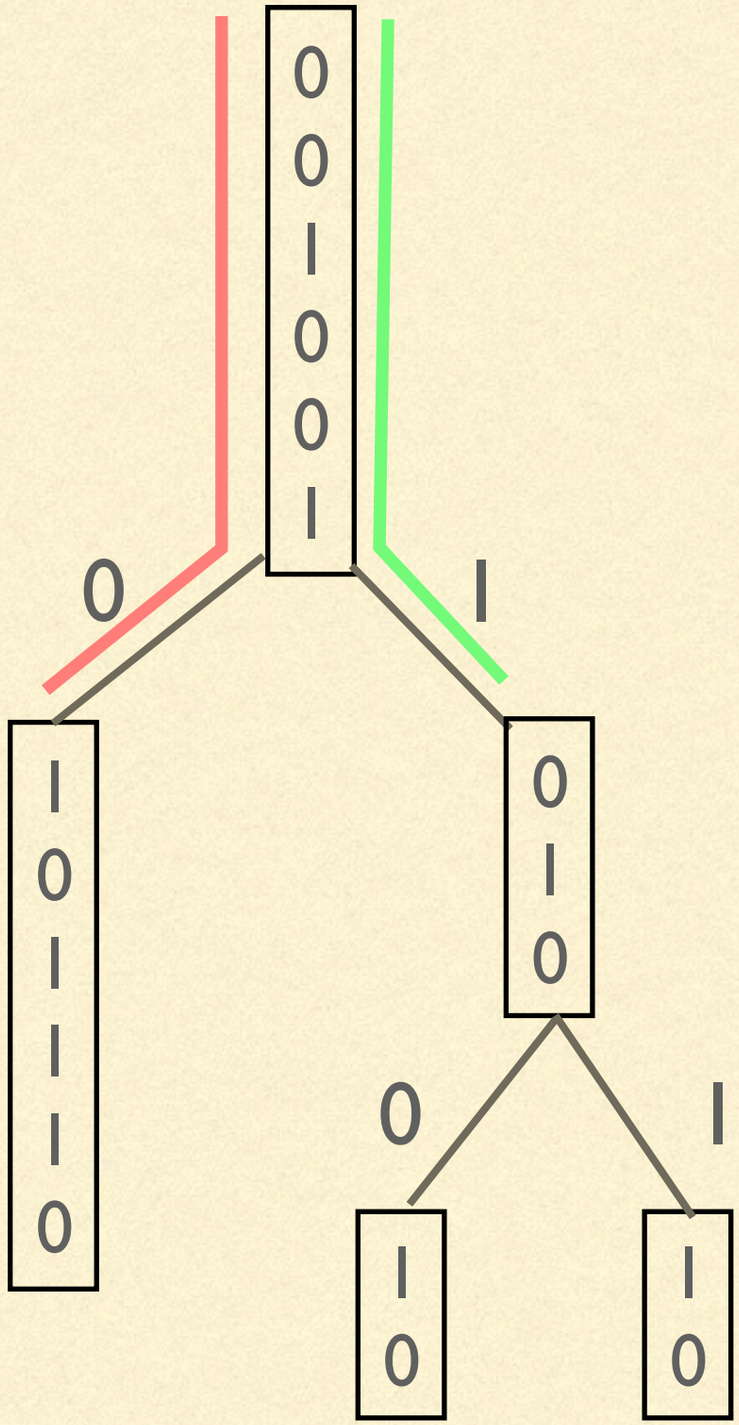




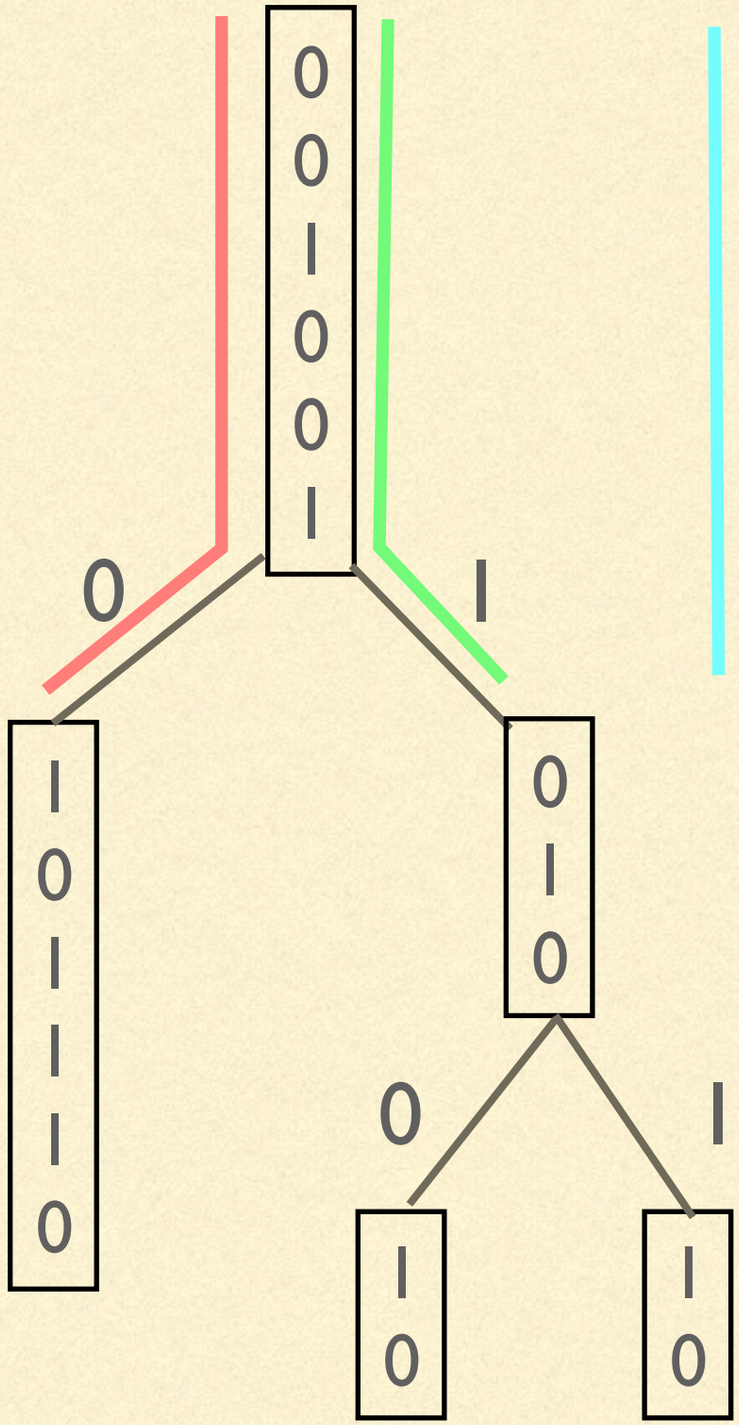
0
0
1
0
0
1
0



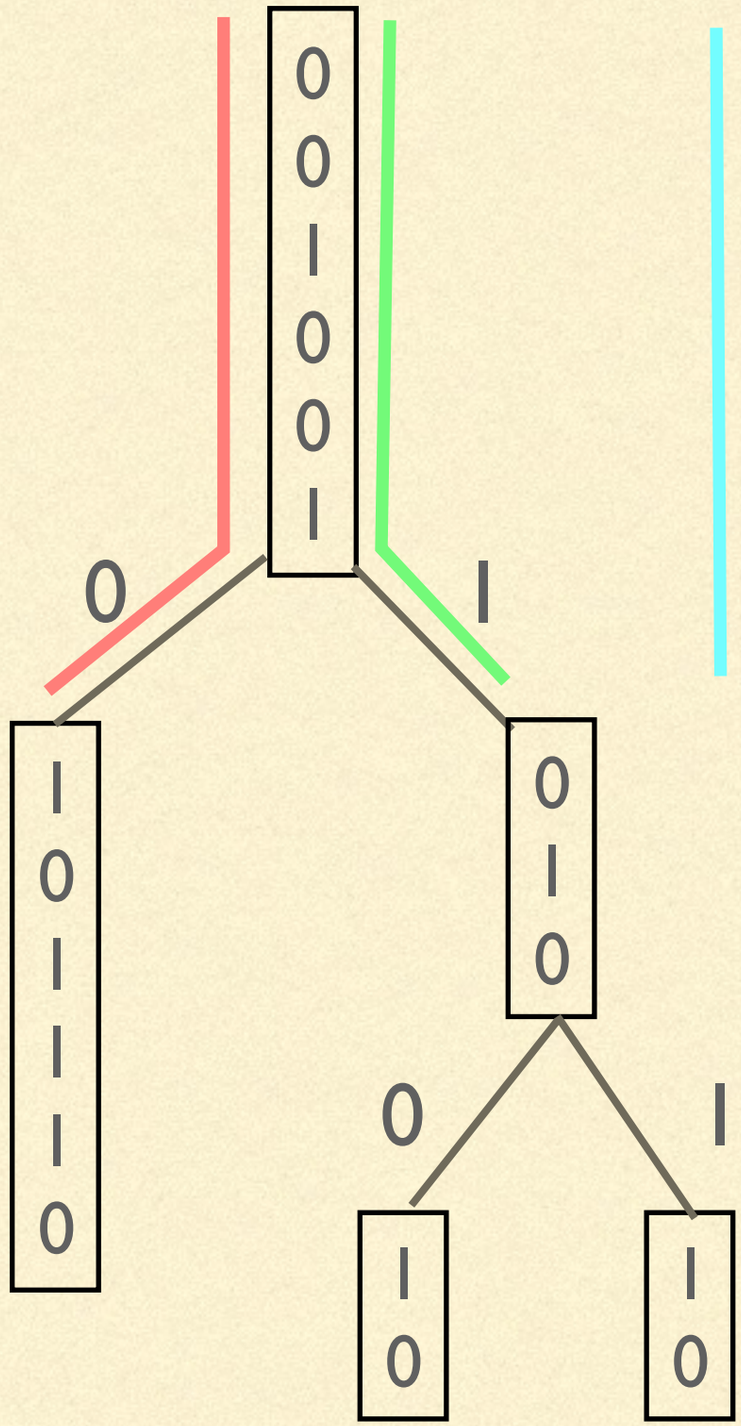
0
0
1
0
0
1
0



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

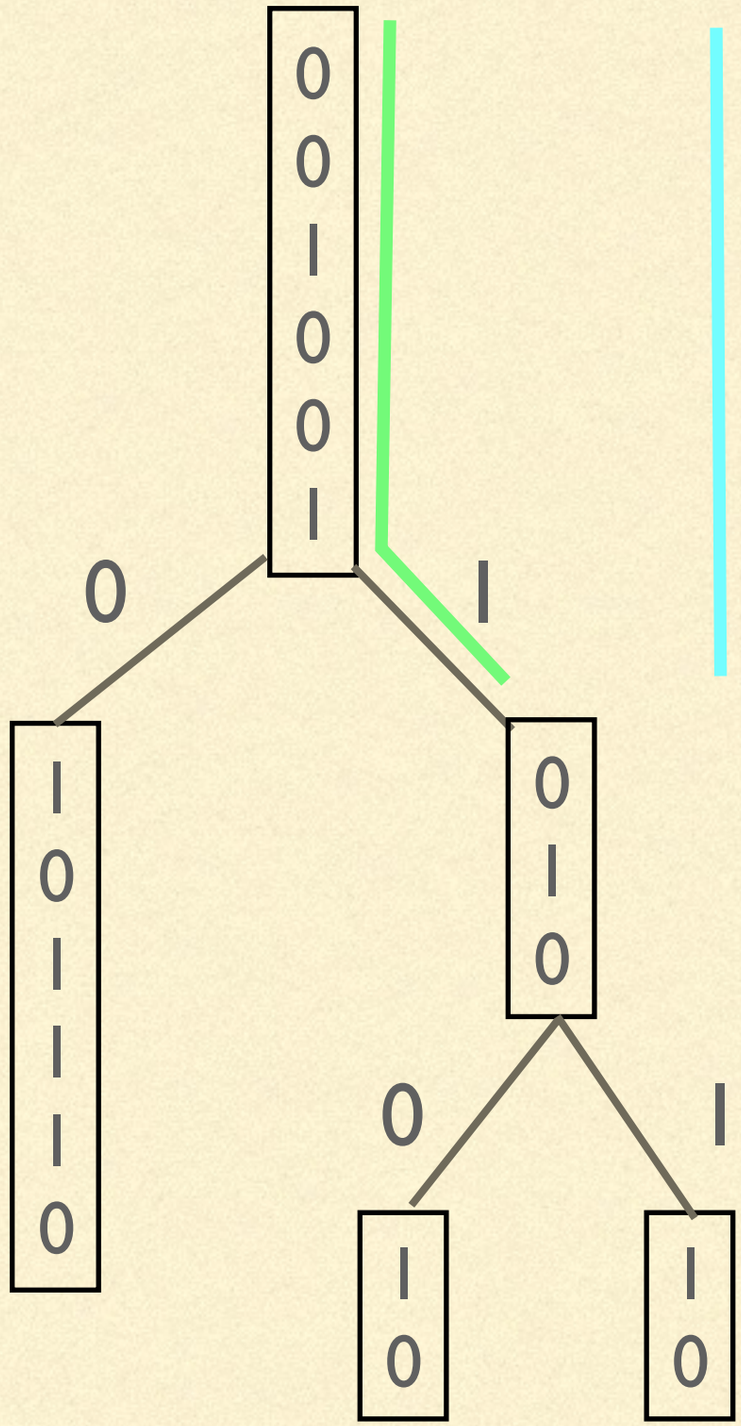


0	0
0	0
1	1
0	0
0	0
1	1
0	1



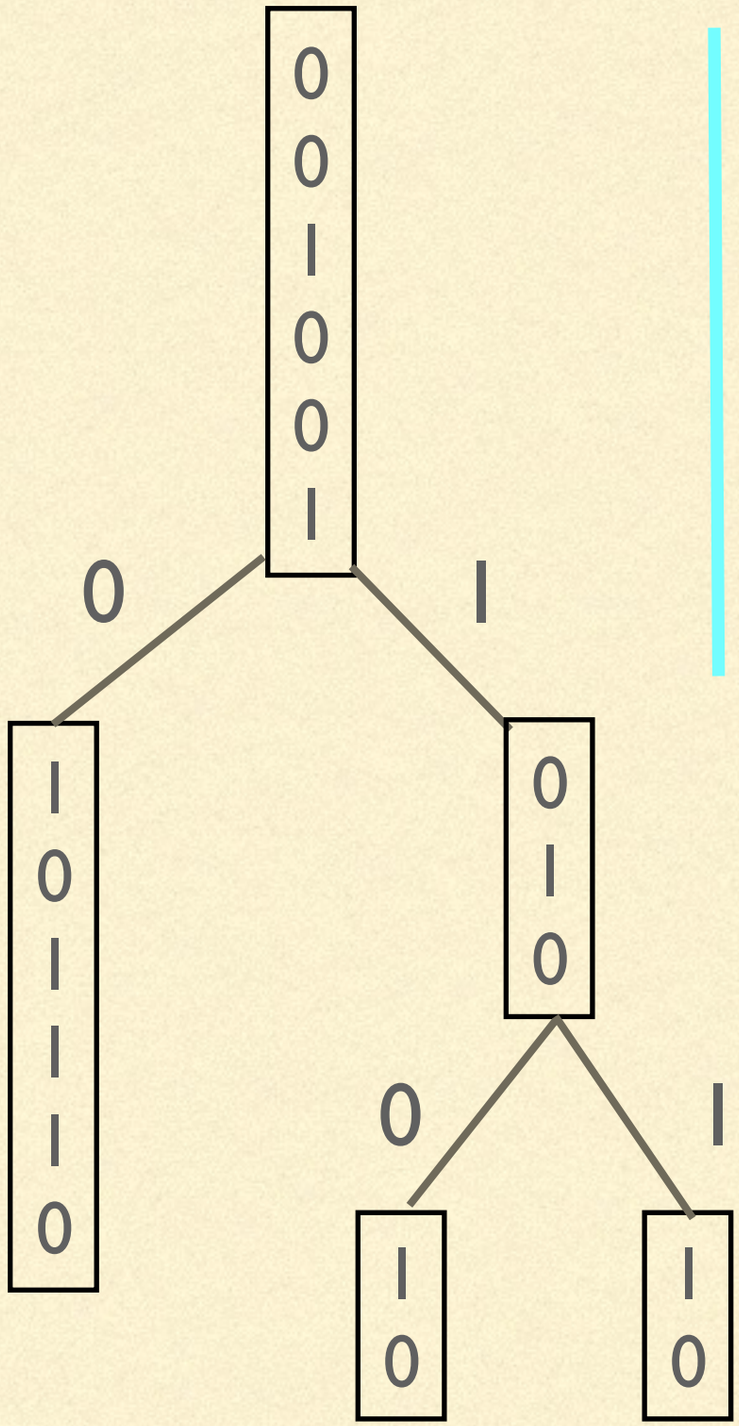
0	0
0	0
1	1
0	0
0	0
1	1
0	1

0
0
1
0
1
0
0

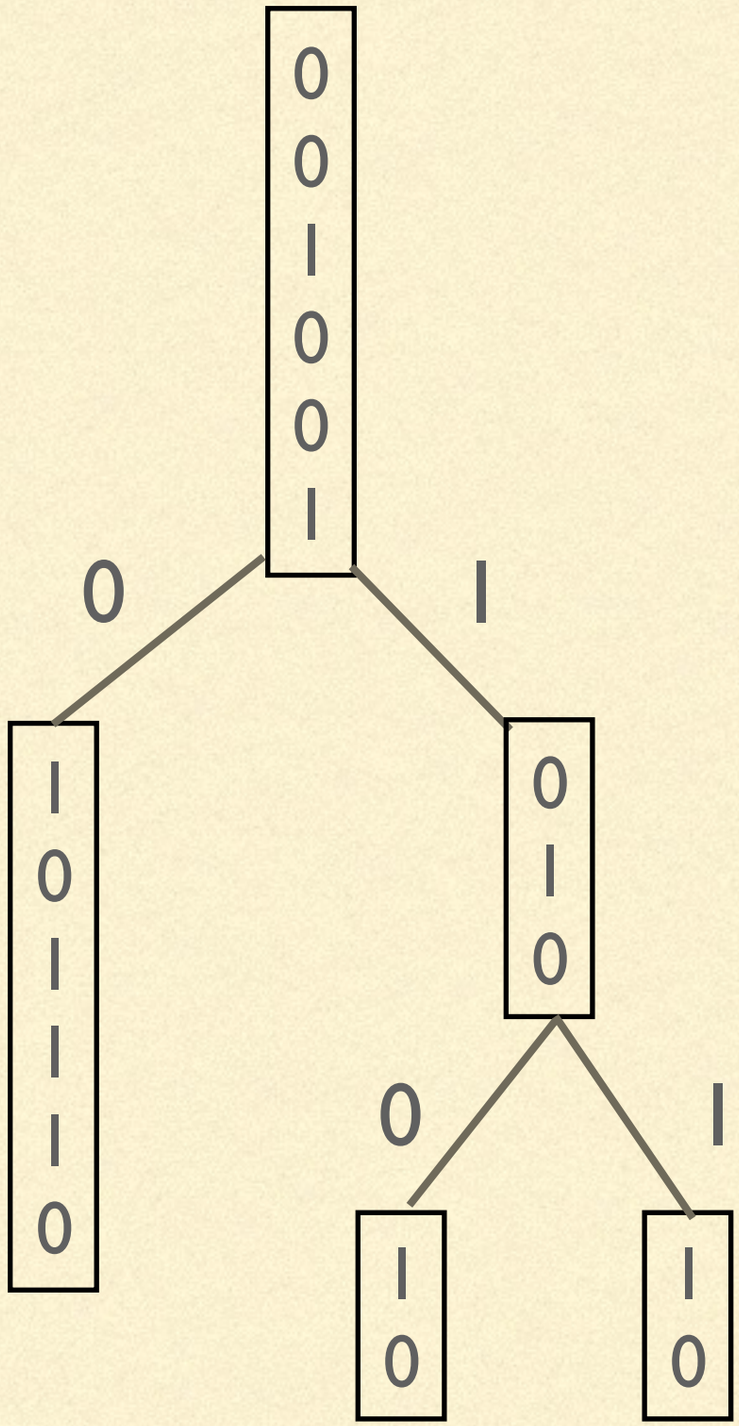


0	0
0	0
1	1
0	0
0	0
1	1
0	1

0
0
1
0
1
0
0

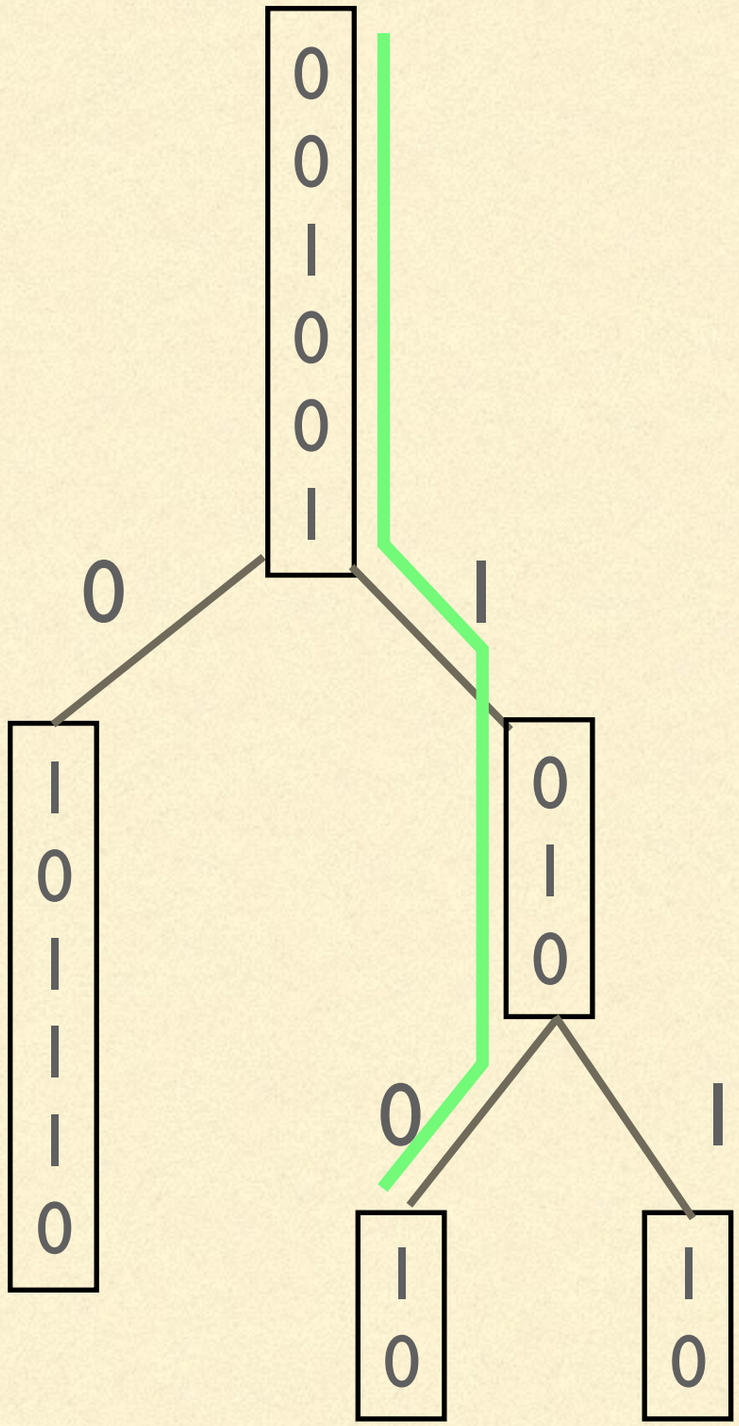


0	0	0
0	0	0
1	1	1
0	0	0
0	0	1
1	1	0
0	1	0



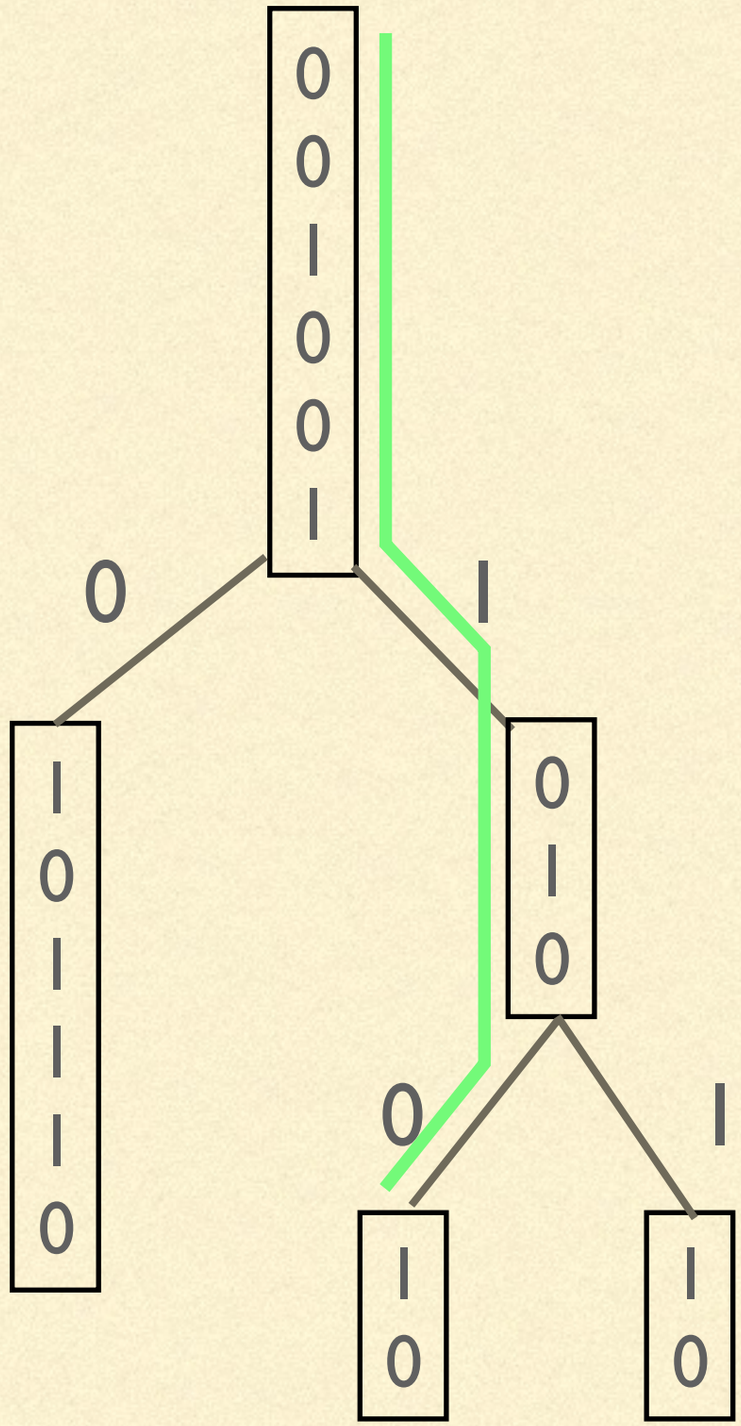
0	0
0	0
1	1
0	0
0	0
1	1
0	1

0
0
1
0
1
0
0

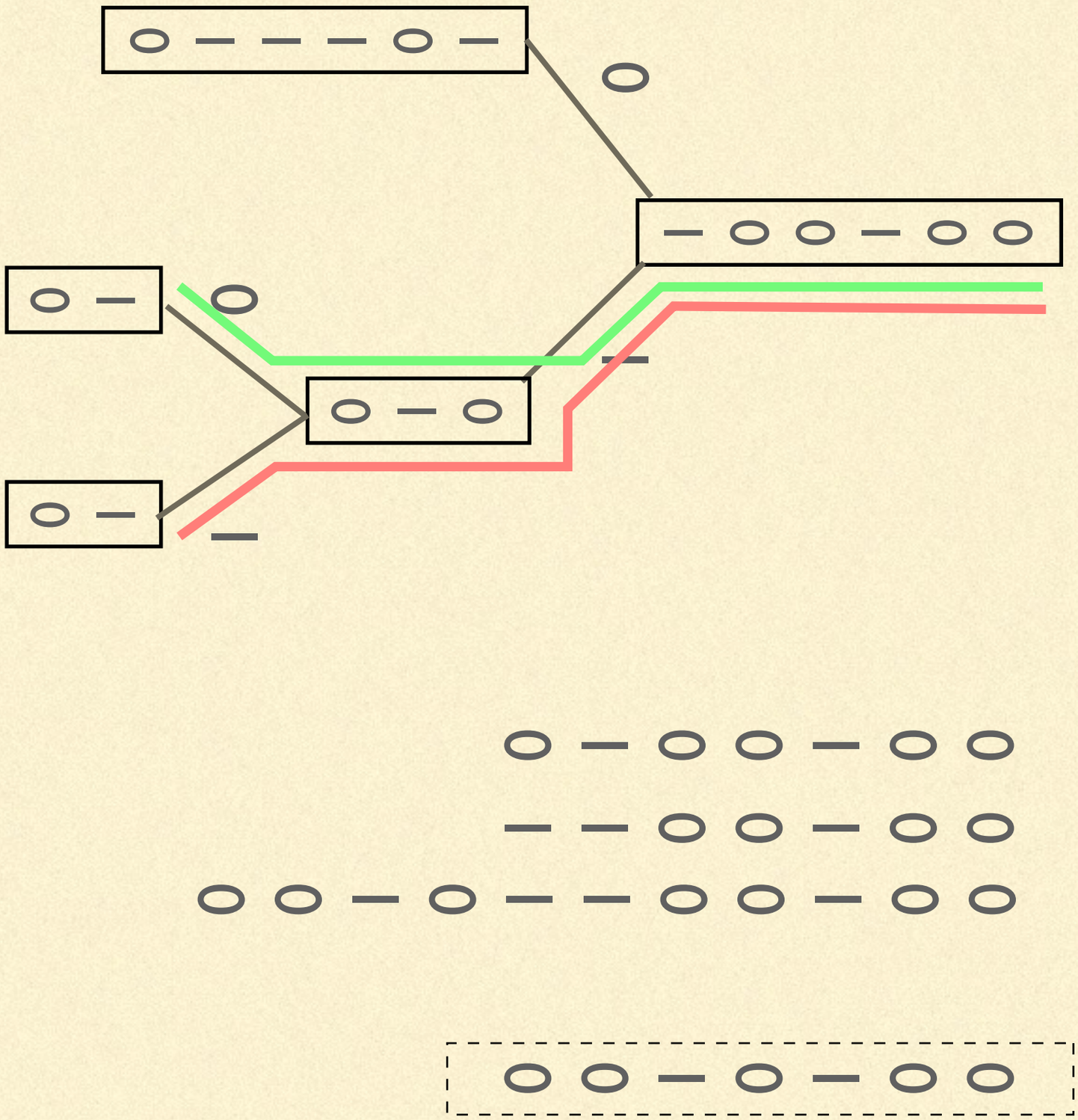


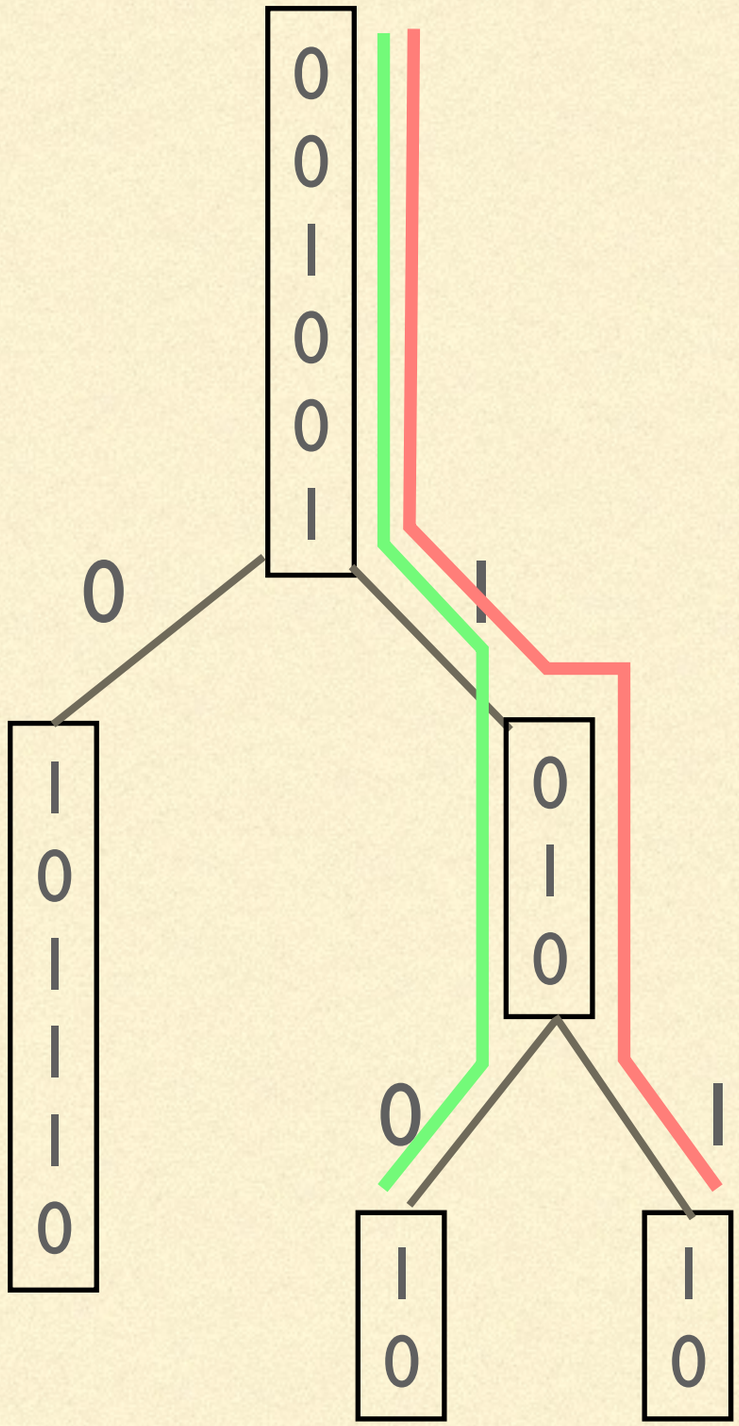
0	0
0	0
1	1
0	0
0	0
1	1
0	1

0
0
1
0
1
0
0

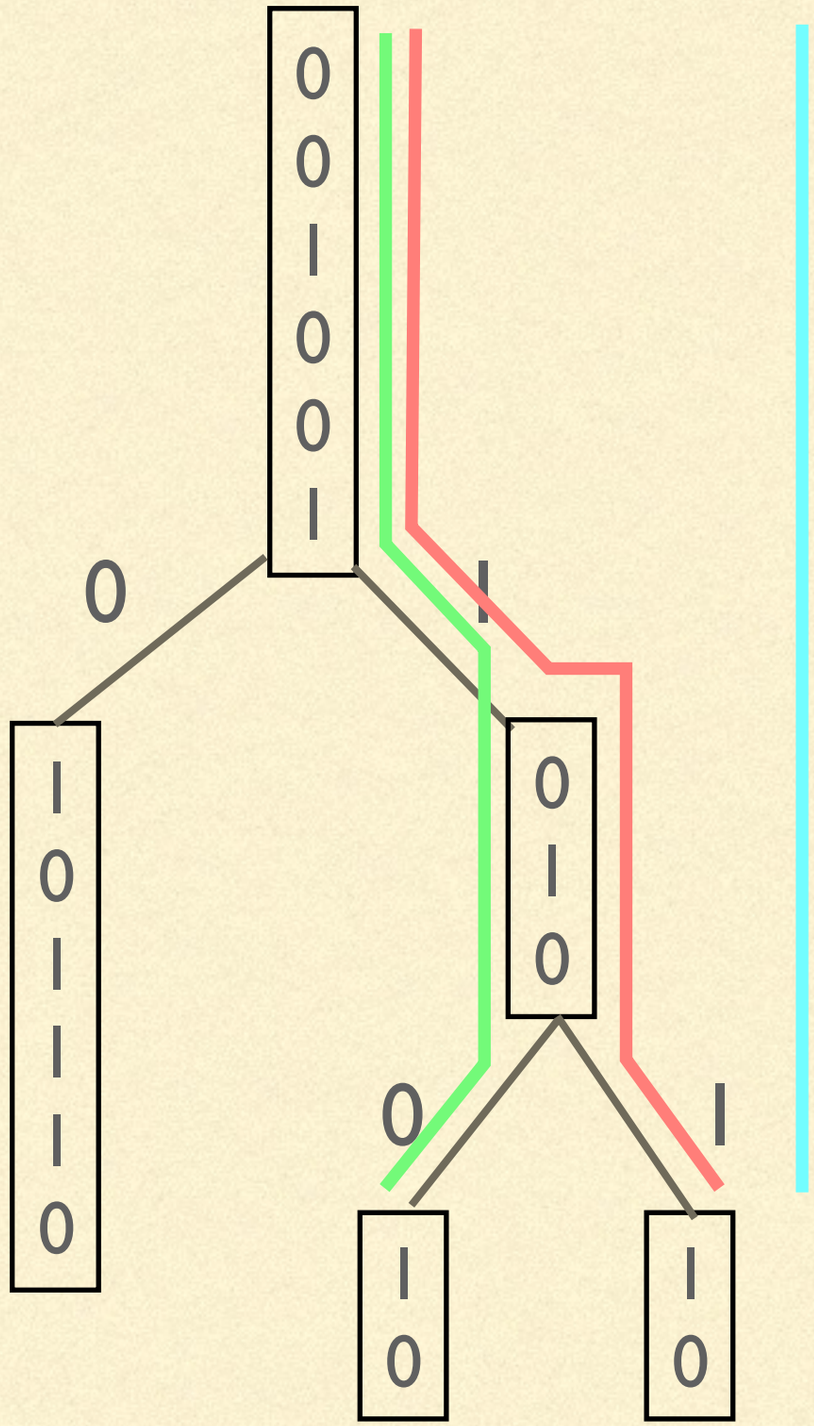

0
0
1
0
0
1
0
1
1
1
1
0
0
1
0
0
1
1
0
1
0

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
1
0
0

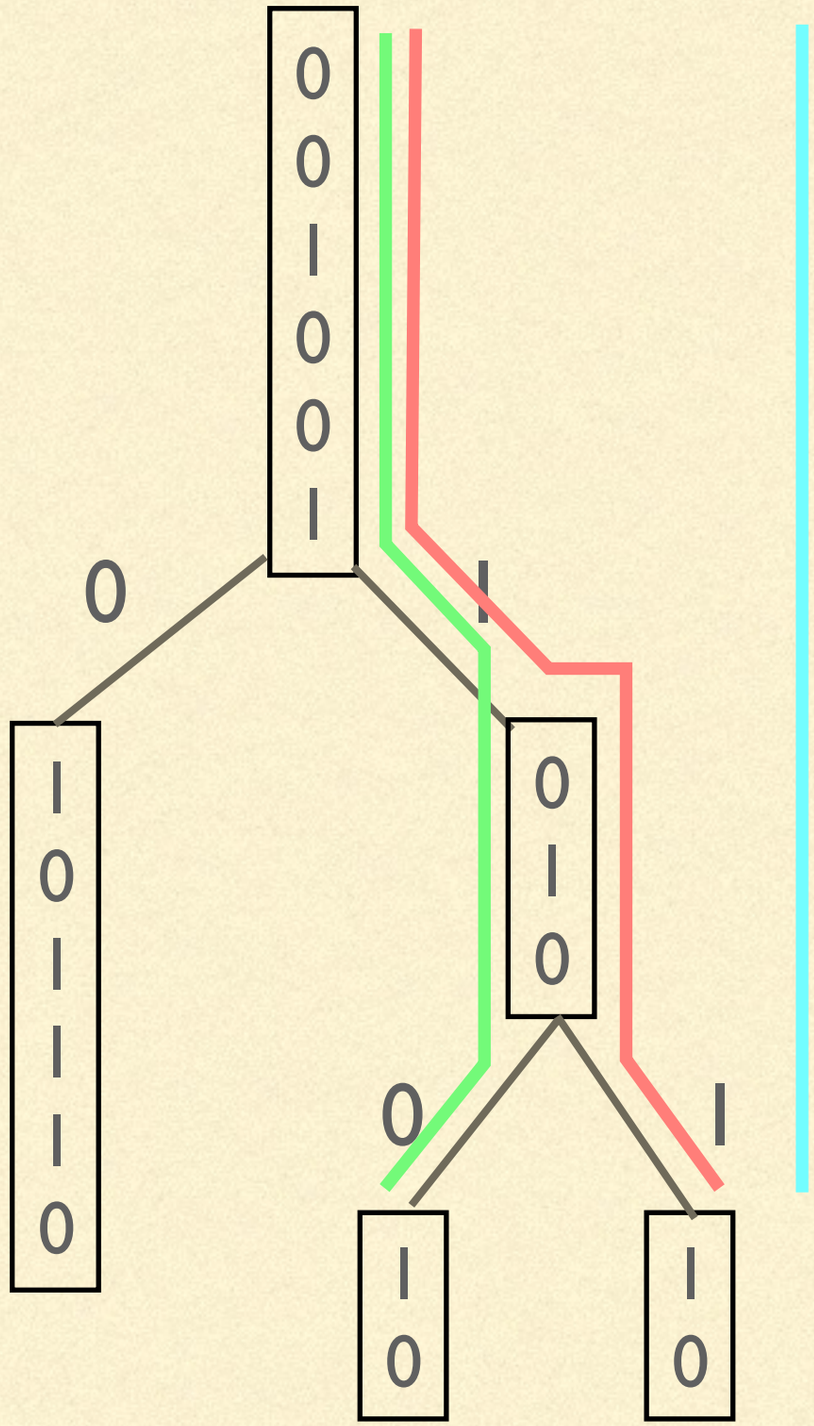




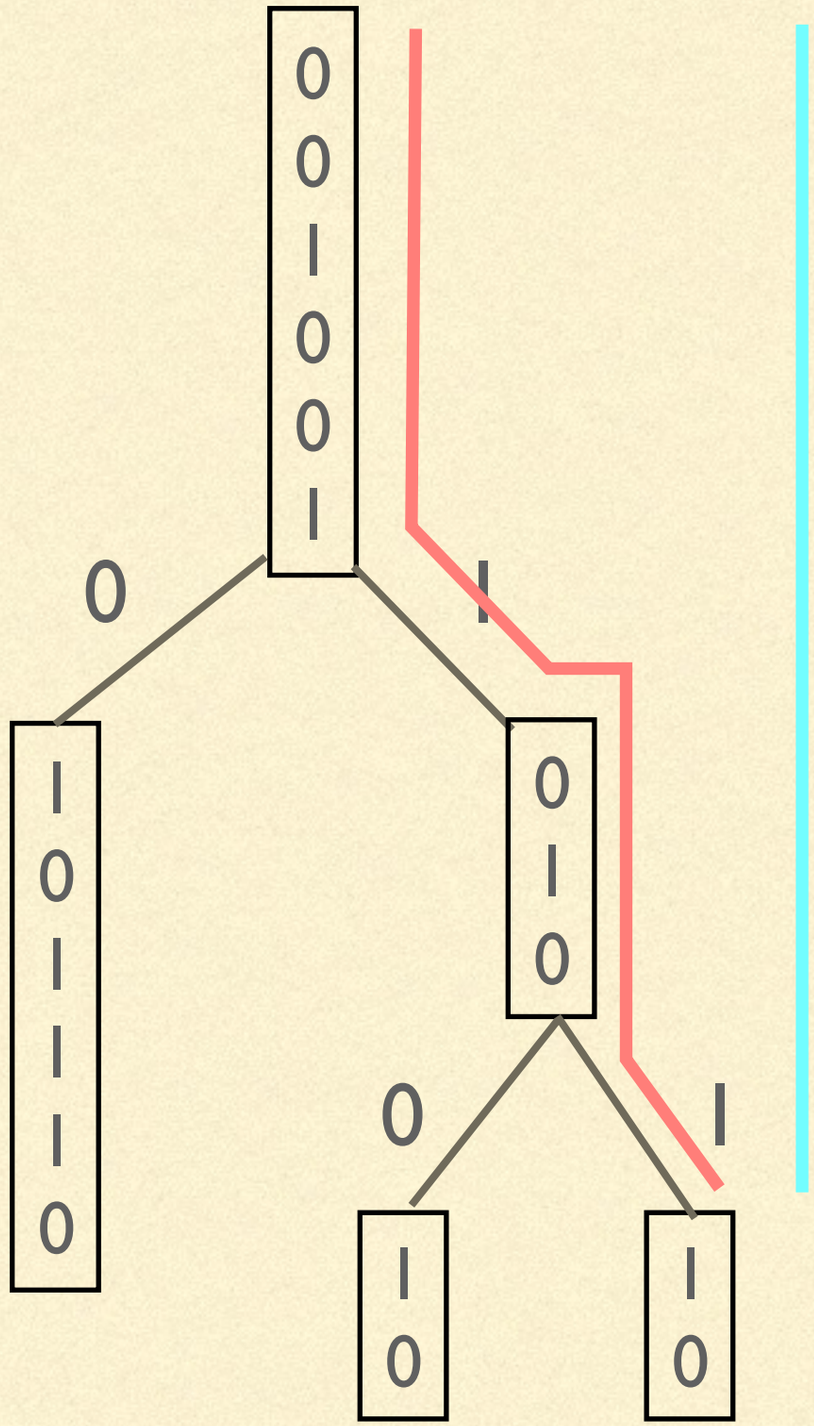
0	0	0	0	0
0	0	0	0	0
1	1	1	1	1
0	0	0	0	0
0	0	0	0	1
1	1	1	1	0
0	1	1	1	0
		0	0	
		1	1	
		0	0	
		0	1	



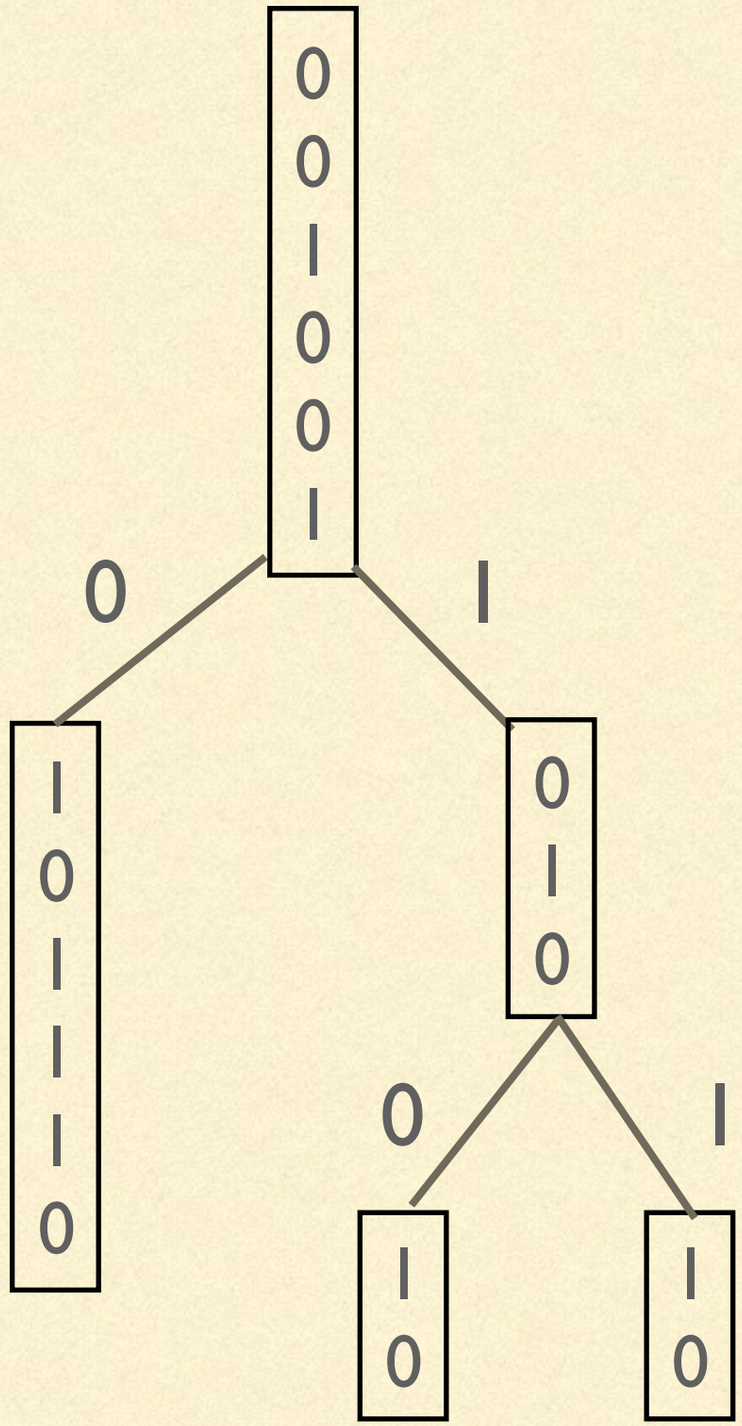
0	0	0	0	0
0	0	0	0	0
1	1	1	1	1
0	0	0	0	0
0	0	0	0	1
1	1	1	1	0
0	1	1	1	0
		0	0	
		1	1	
		0	0	
		0	1	



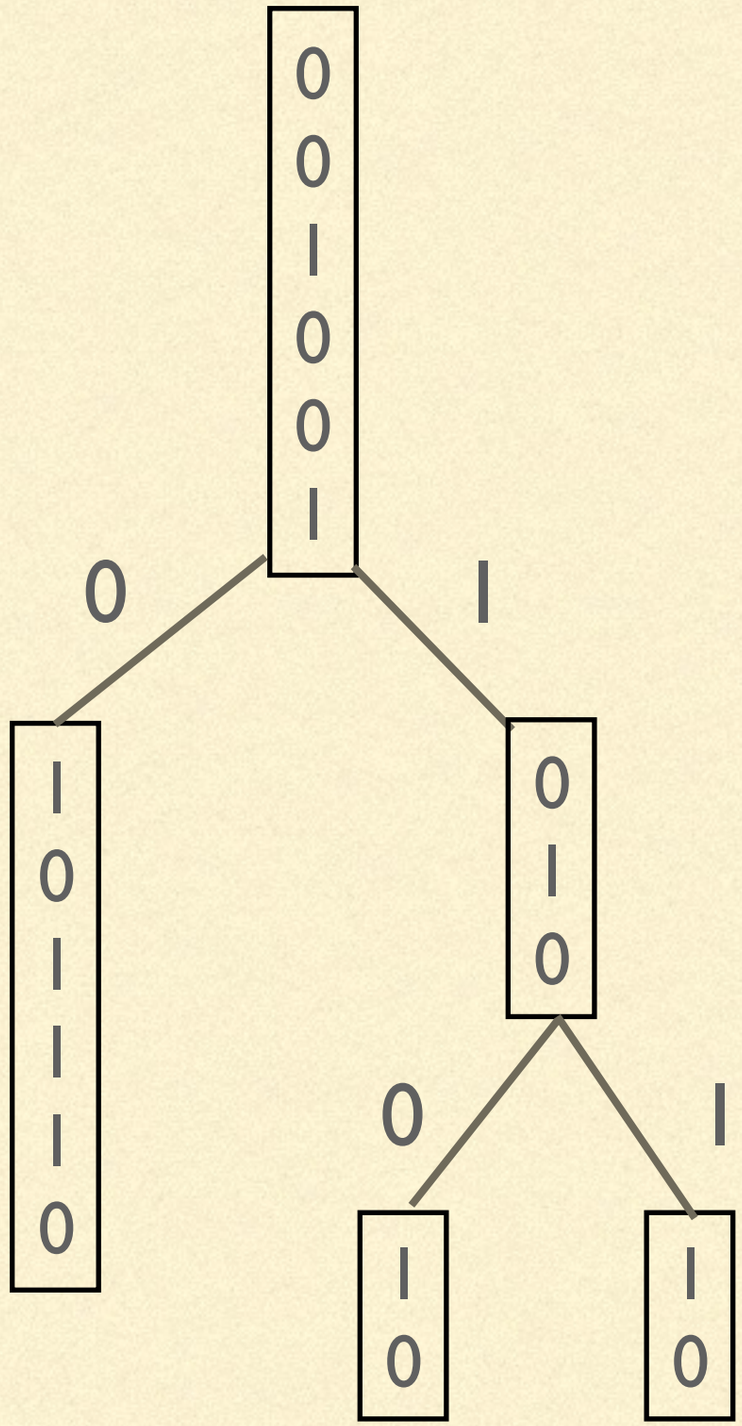
0	0	0	0	0	0
0	0	0	0	0	0
1	1	1	1	1	1
0	0	0	0	0	0
0	0	0	0	1	0
1	1	1	1	0	1
0	1	1	1	0	1
		0	0		0
		1	1		1
		0	0		1
		0	1		0



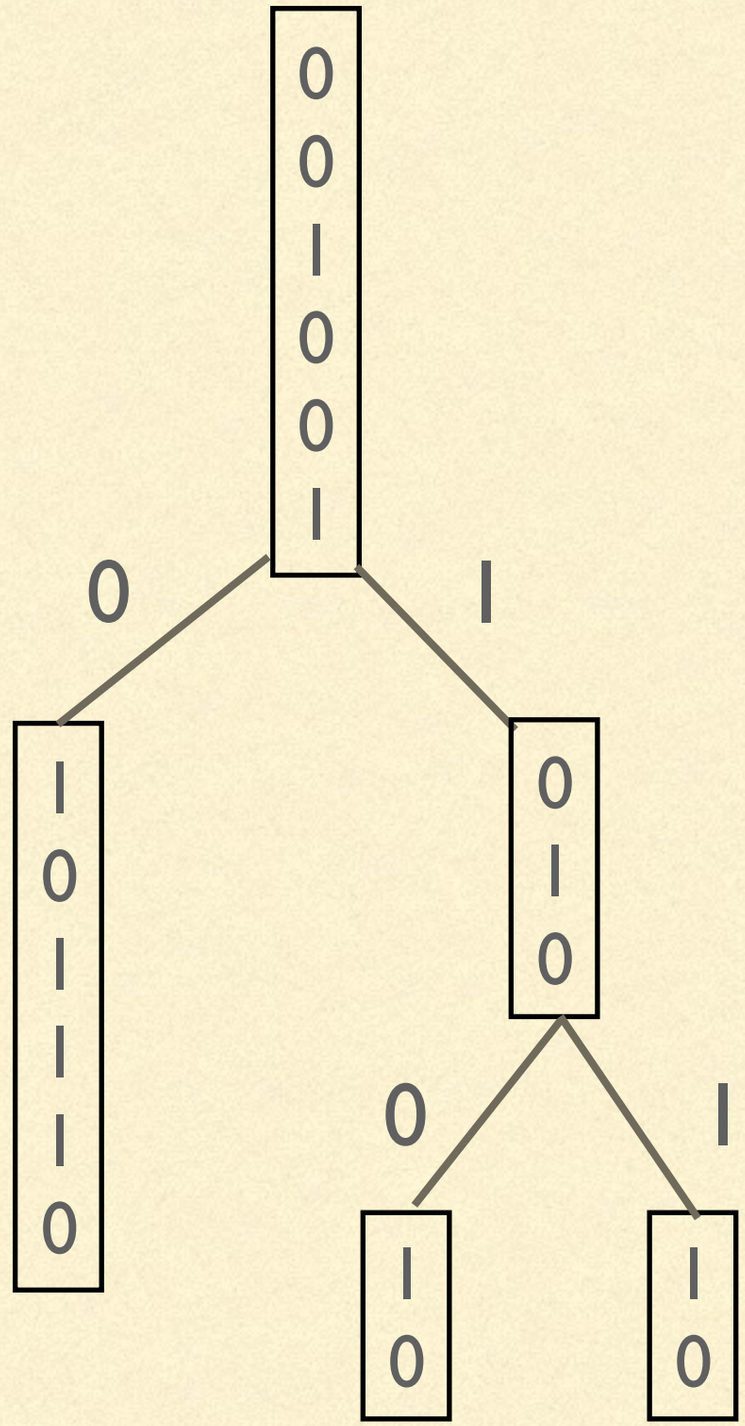
0	0	0	0	0	0
0	0	0	0	0	0
1	1	1	1	1	1
0	0	0	0	0	0
0	0	0	0	1	0
1	1	1	1	0	1
0	1	1	1	0	1
		0	0		0
		1	1		1
		0	0		1
		0	1		0

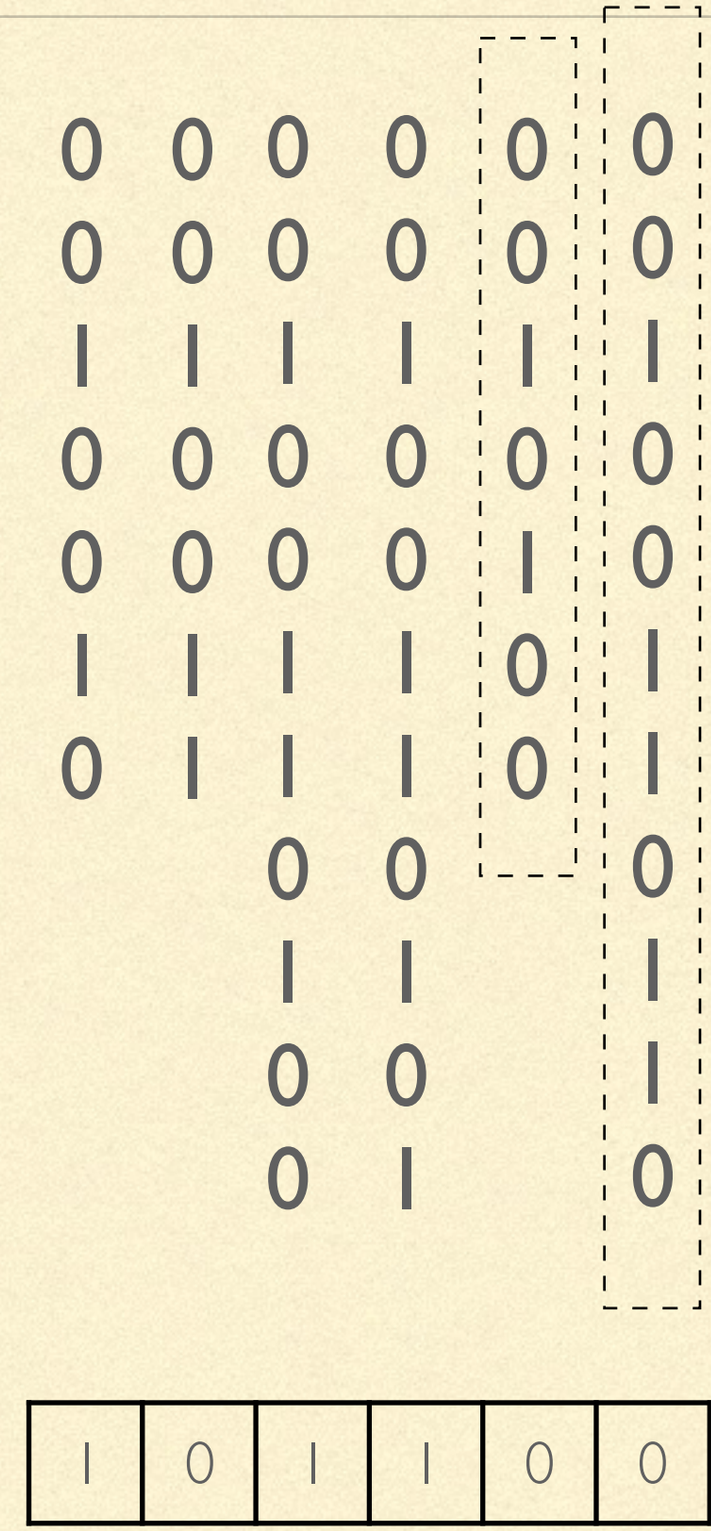

0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
0
1
0
1
0

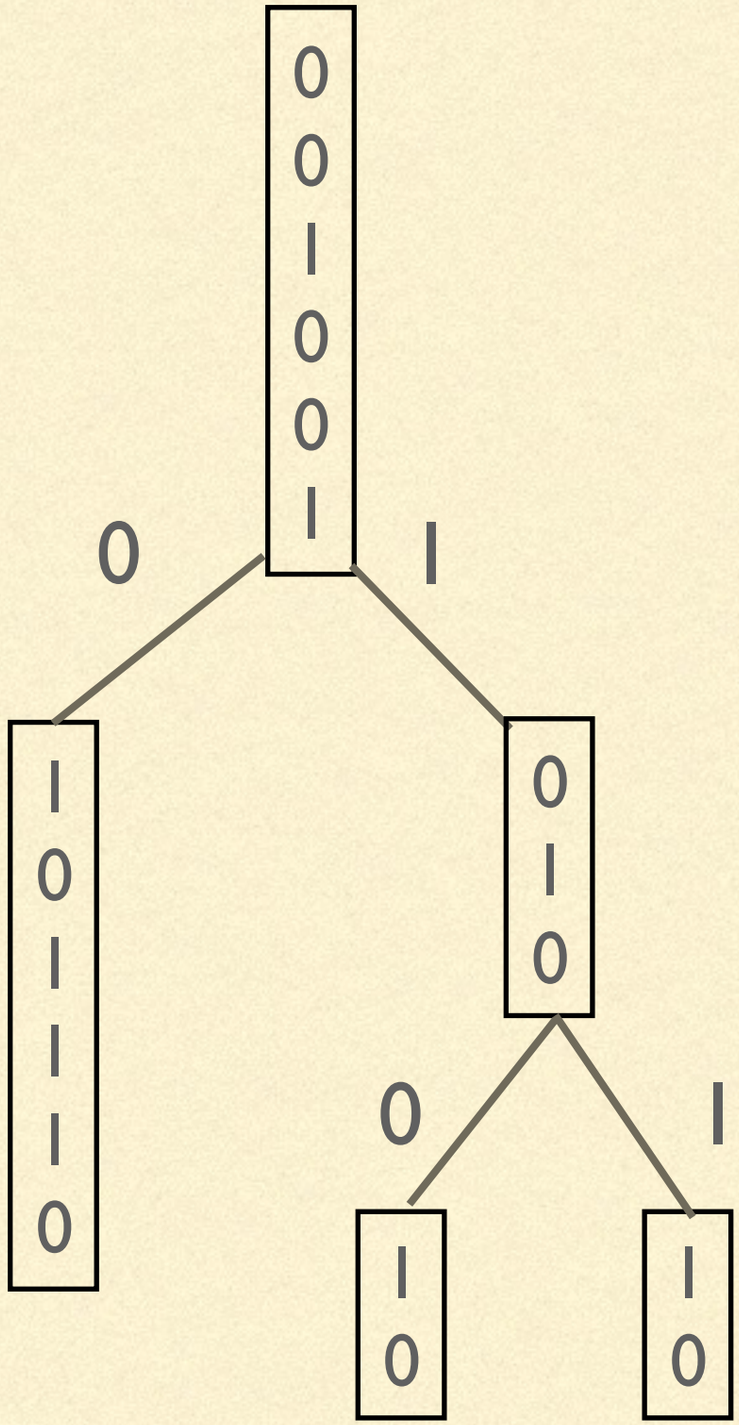
0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
0 0 0 0 0 0
0 0 0 0 1 0
1 1 1 1 0 1
0 1 1 1 0 1
0 0 1 0
1 1 0 1
0 0 1 1
0 1 0

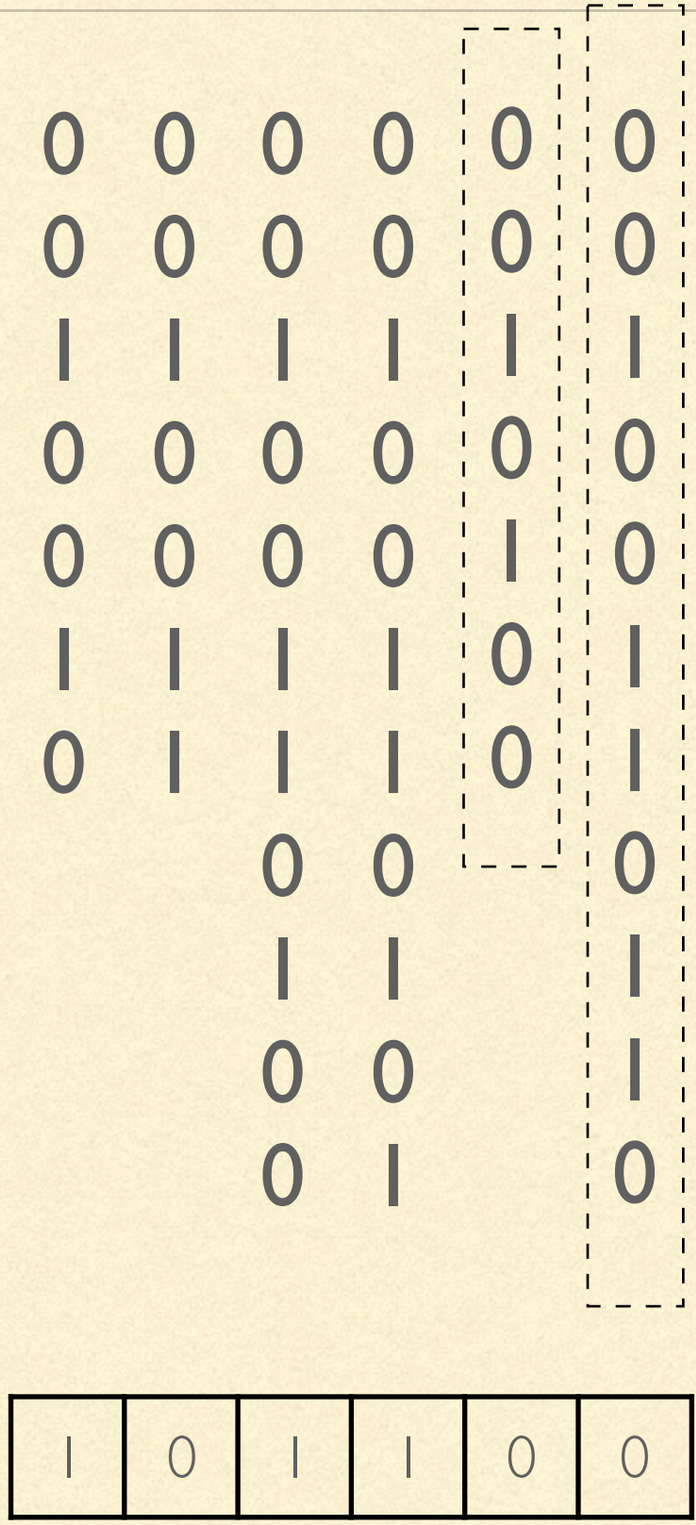

0
0
1
0
0
1
0
1
1
0
1
1
1
0
0
1
0
1
1
0

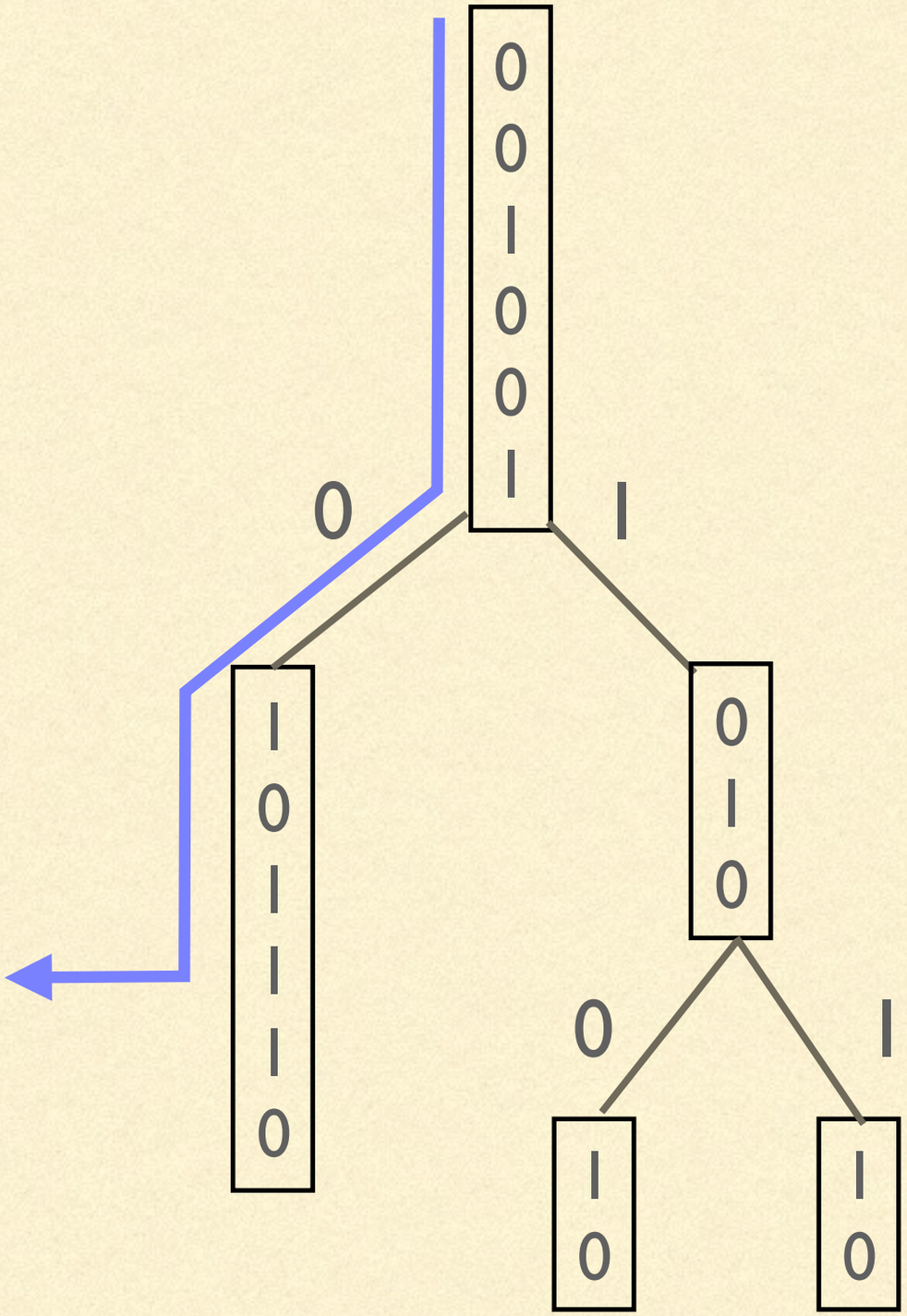
0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
0 0 0 0 0 0
0 0 0 0 1 0
1 1 1 1 0 1
0 1 1 1 0 1
0 0 1 1 0
1 1 0 1
0 0 1
0 1 0


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


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

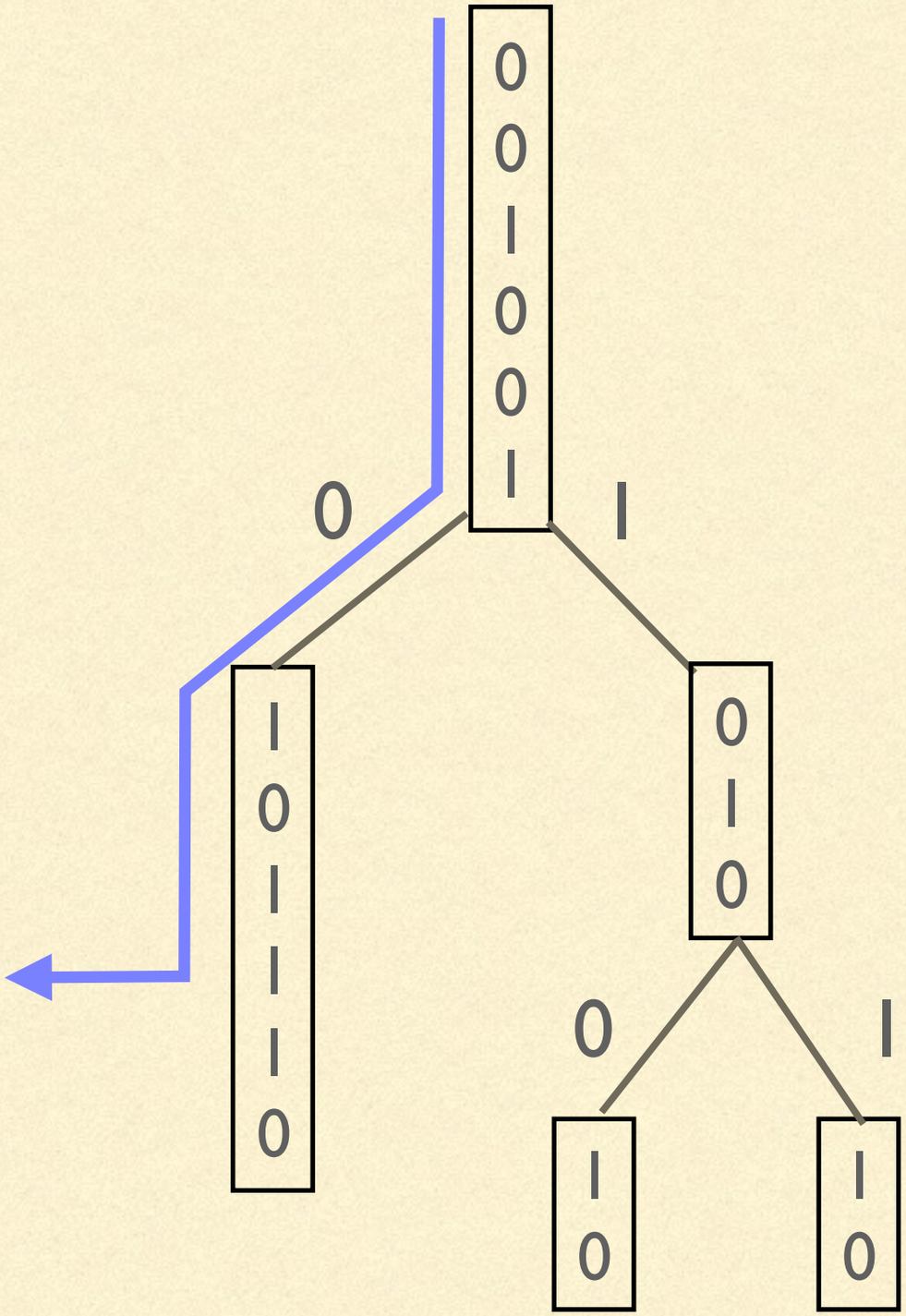

0
0
1
0
0
1
0
1
1
1
1
0
0
1
0
0
1
0
1
0


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



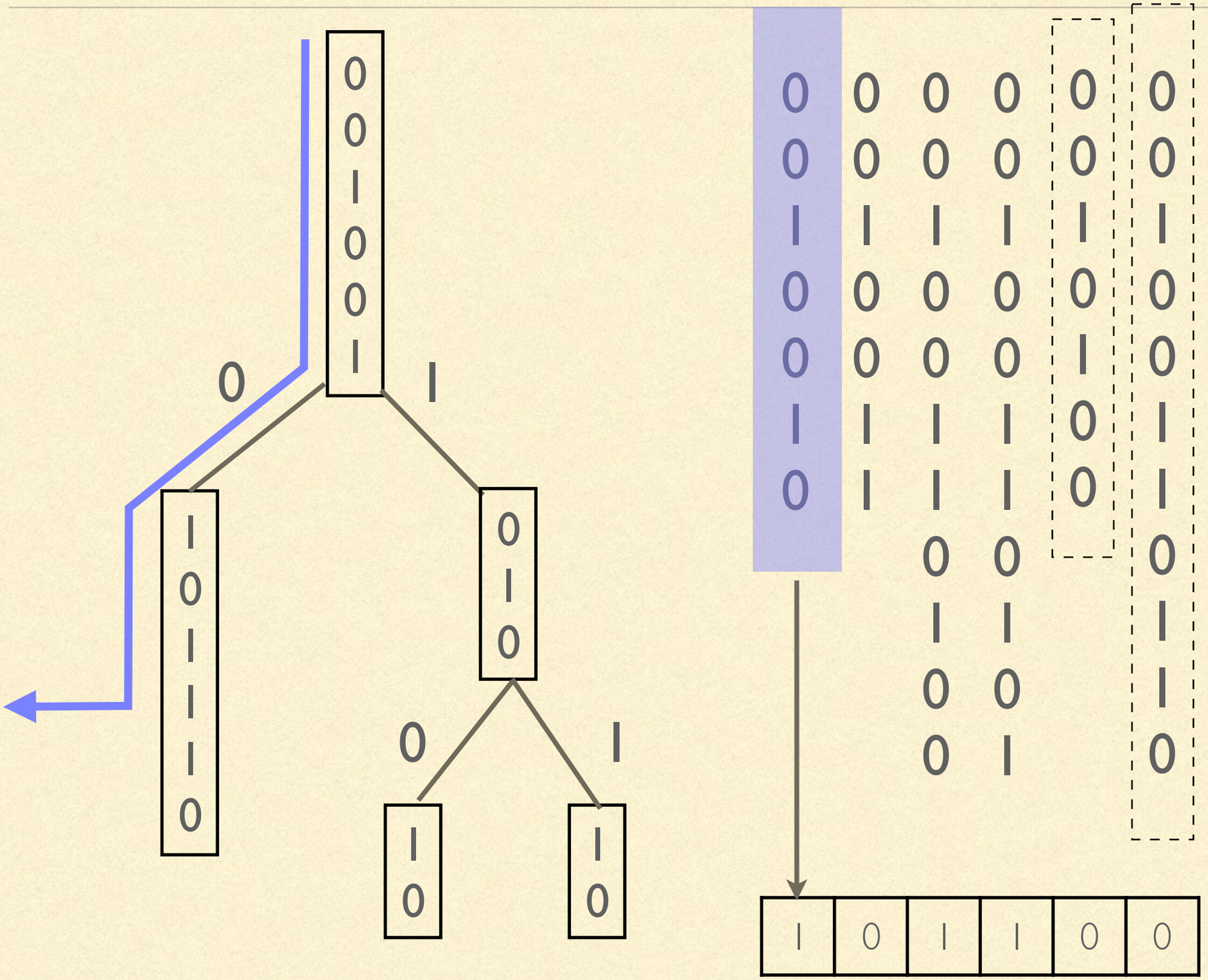
0	0	0	0	0	0
0	0	0	0	0	0
1	1	1	1	1	1
0	0	0	0	0	0
0	0	0	0	1	0
1	1	1	1	0	1
0	1	1	1	0	1
		0	0		0
		1	1		1
		0	0		1
		0	1		0

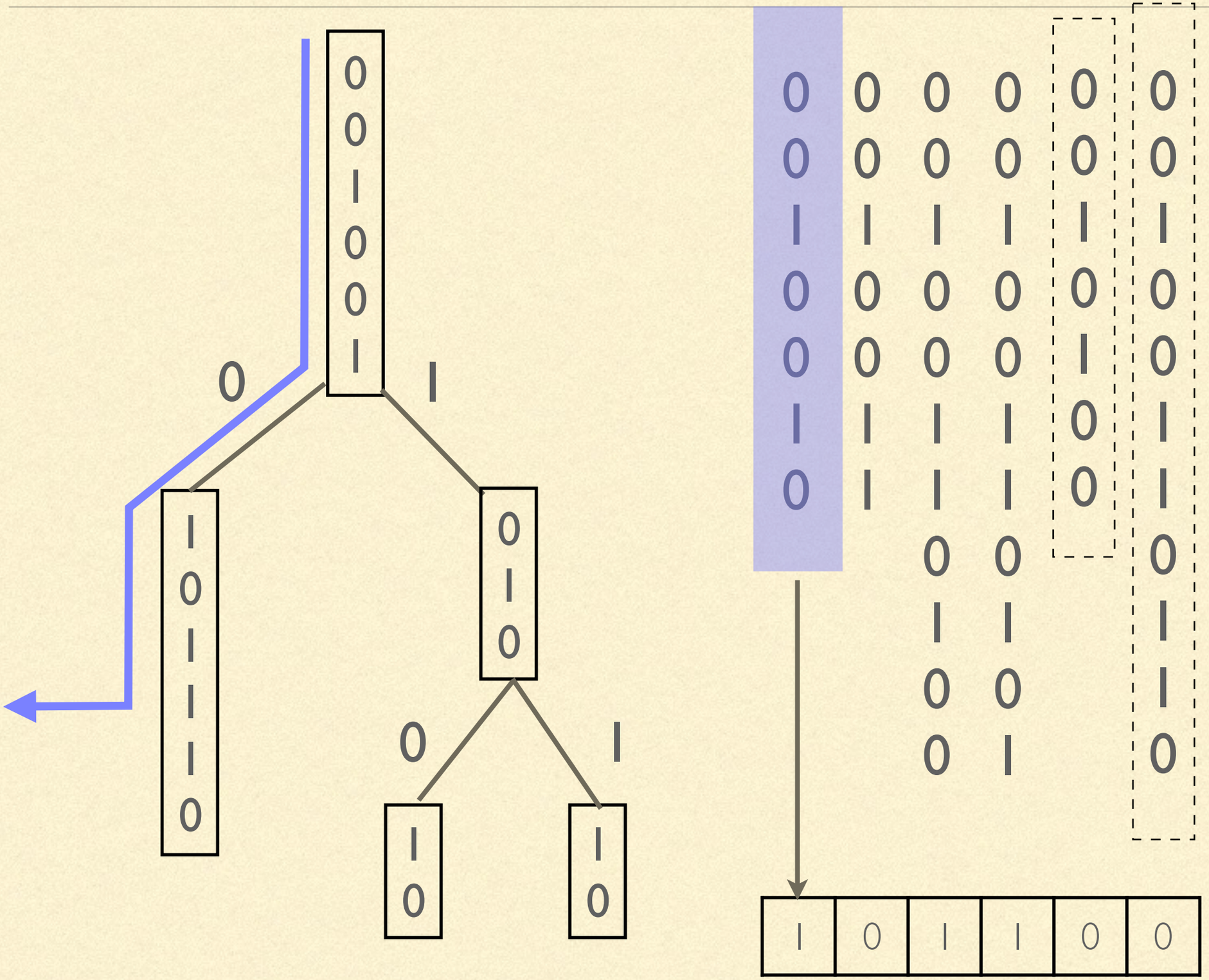
1	0	1	1	0	0



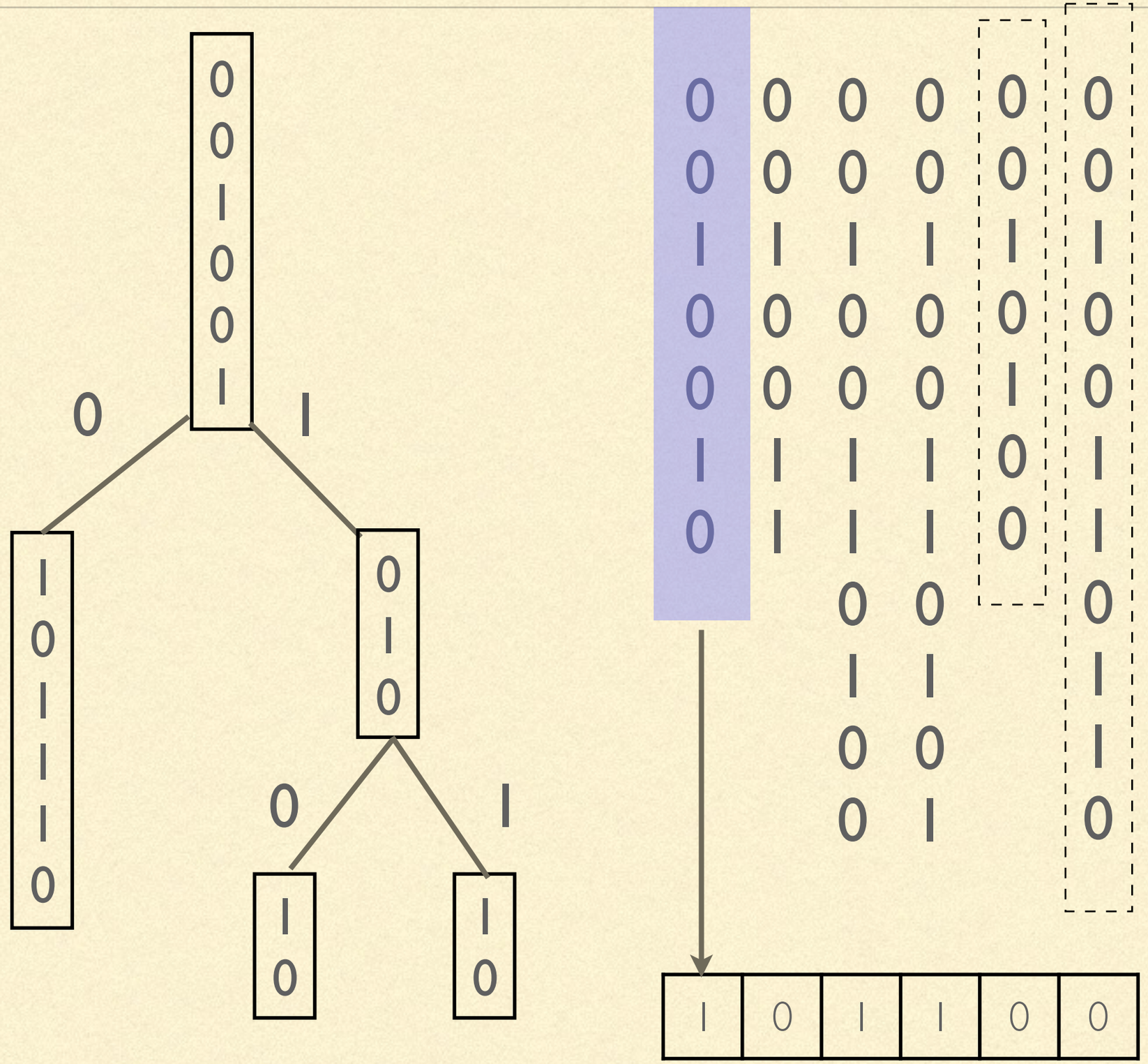
0	0	0	0	0	0
0	0	0	0	0	0
1	1	1	1	1	1
0	0	0	0	0	0
0	0	0	0	1	0
1	1	1	1	0	1
0	1	1	1	0	1
		0	0		0
		1	1		1
		0	0		1
		0	1		0

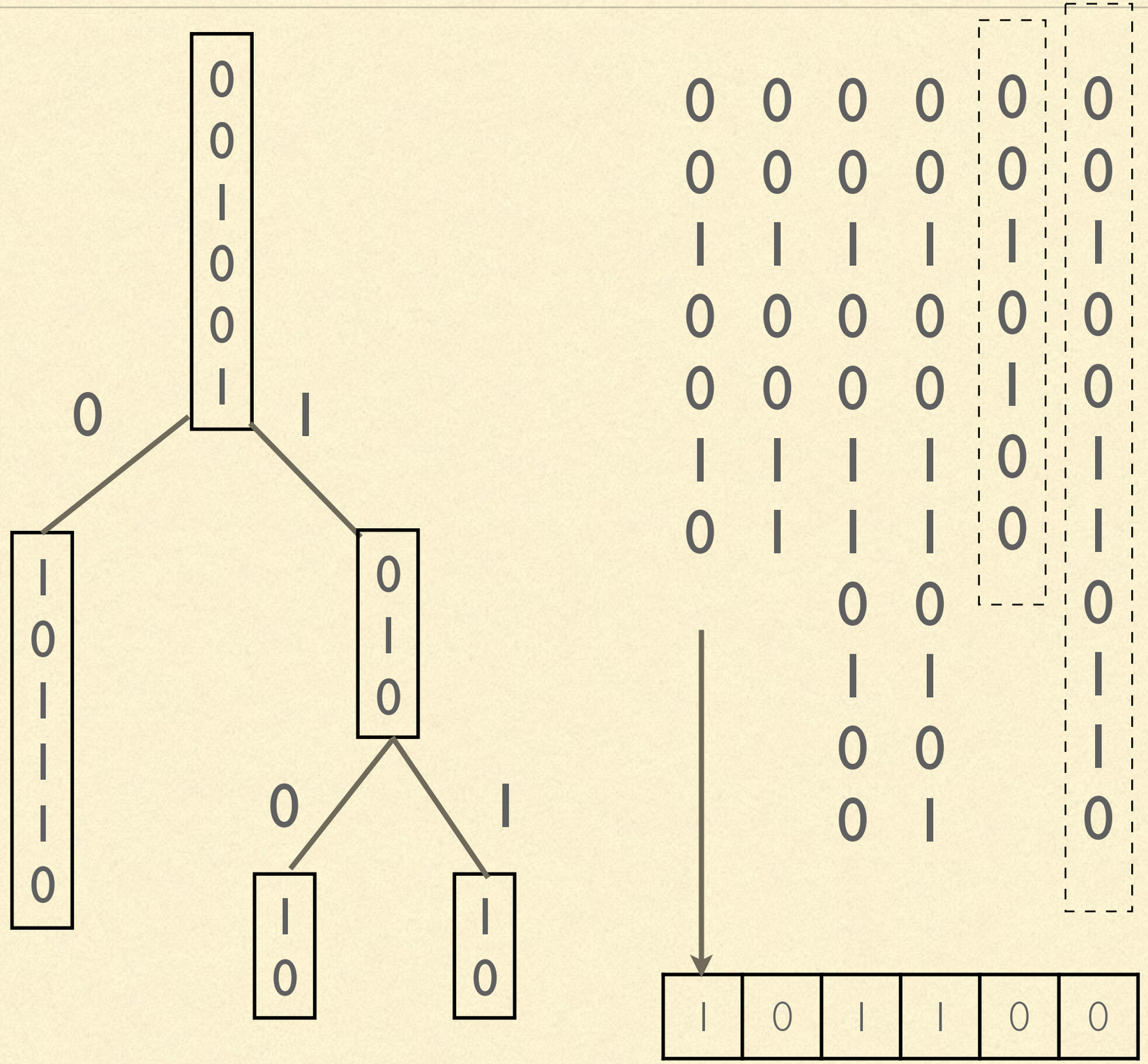
1	0	1	1	0	0


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

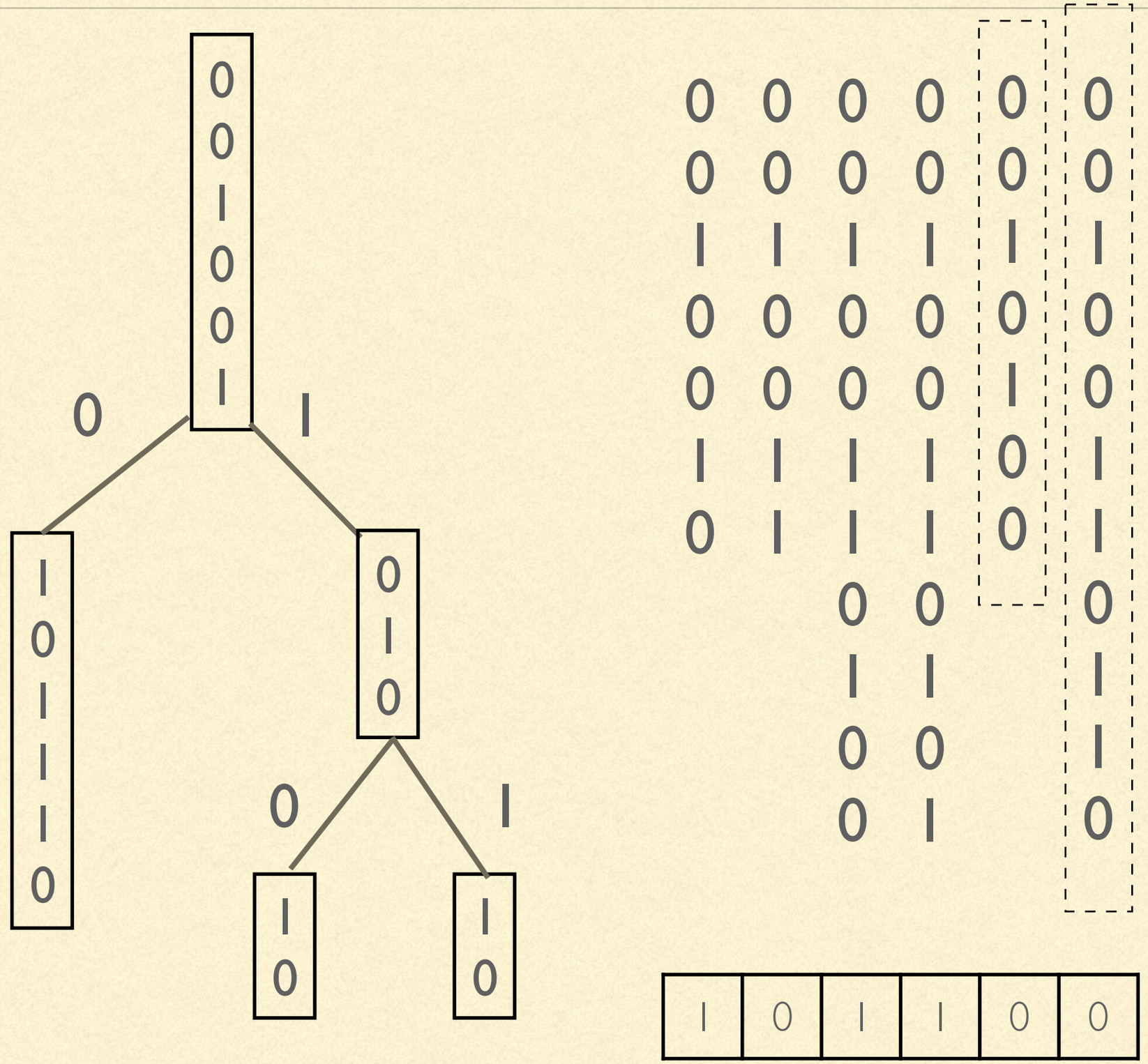


Rank 0 ⇒ bucket 0!

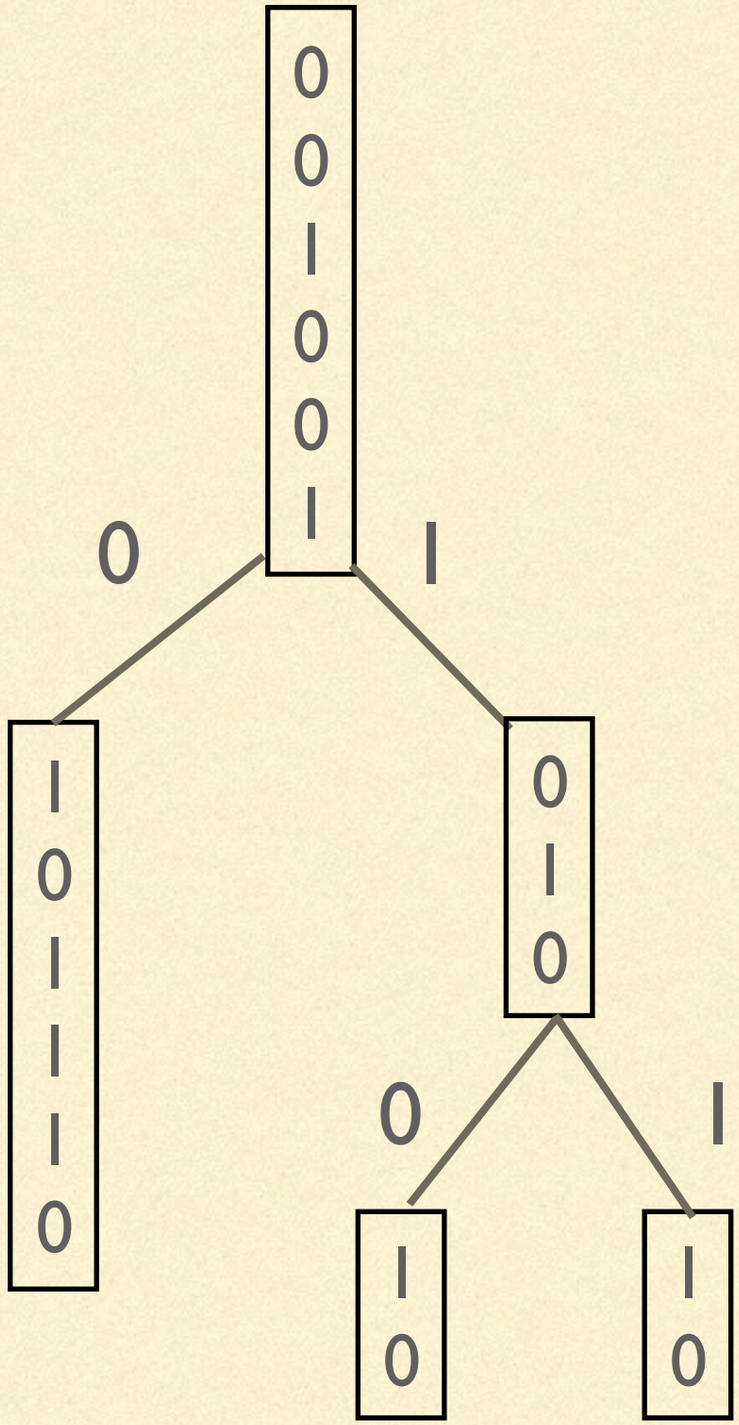

0
1
0
1
0
0
1
0
1
0
1

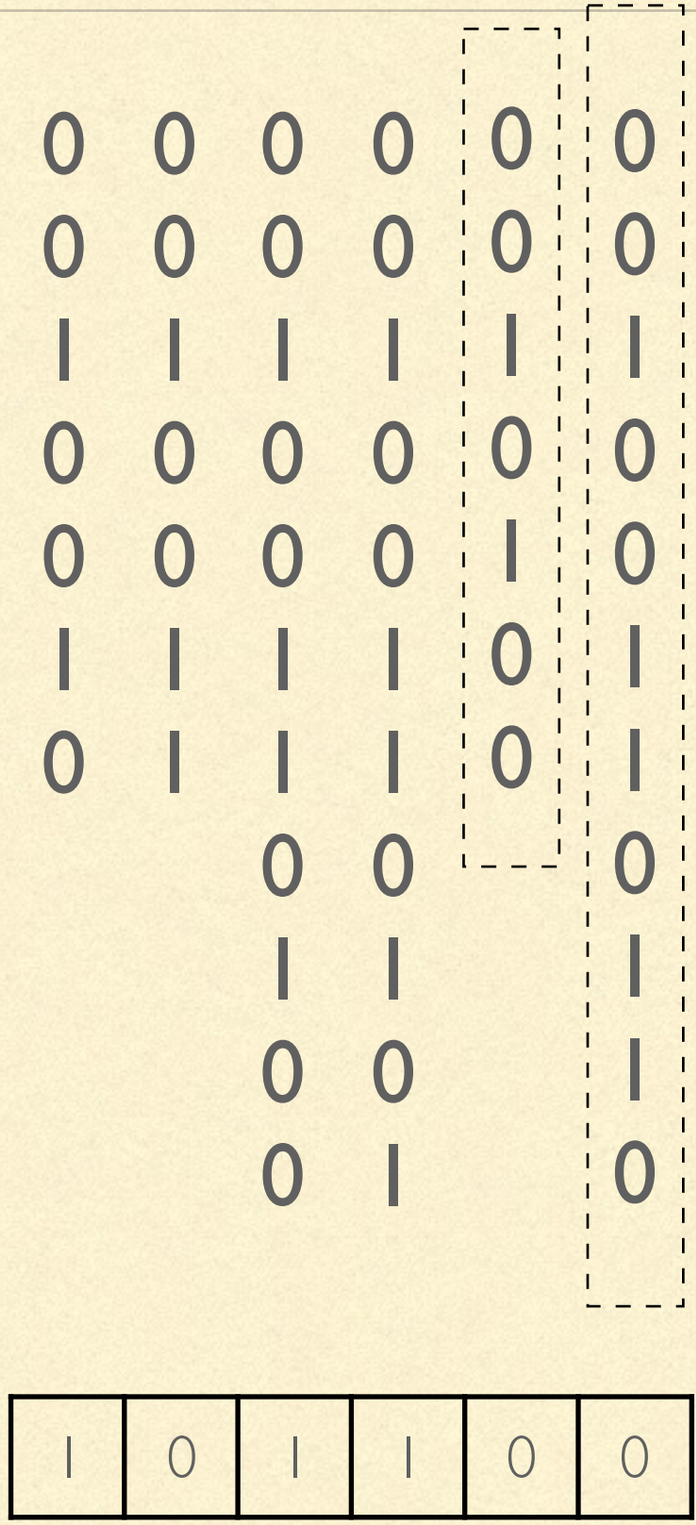


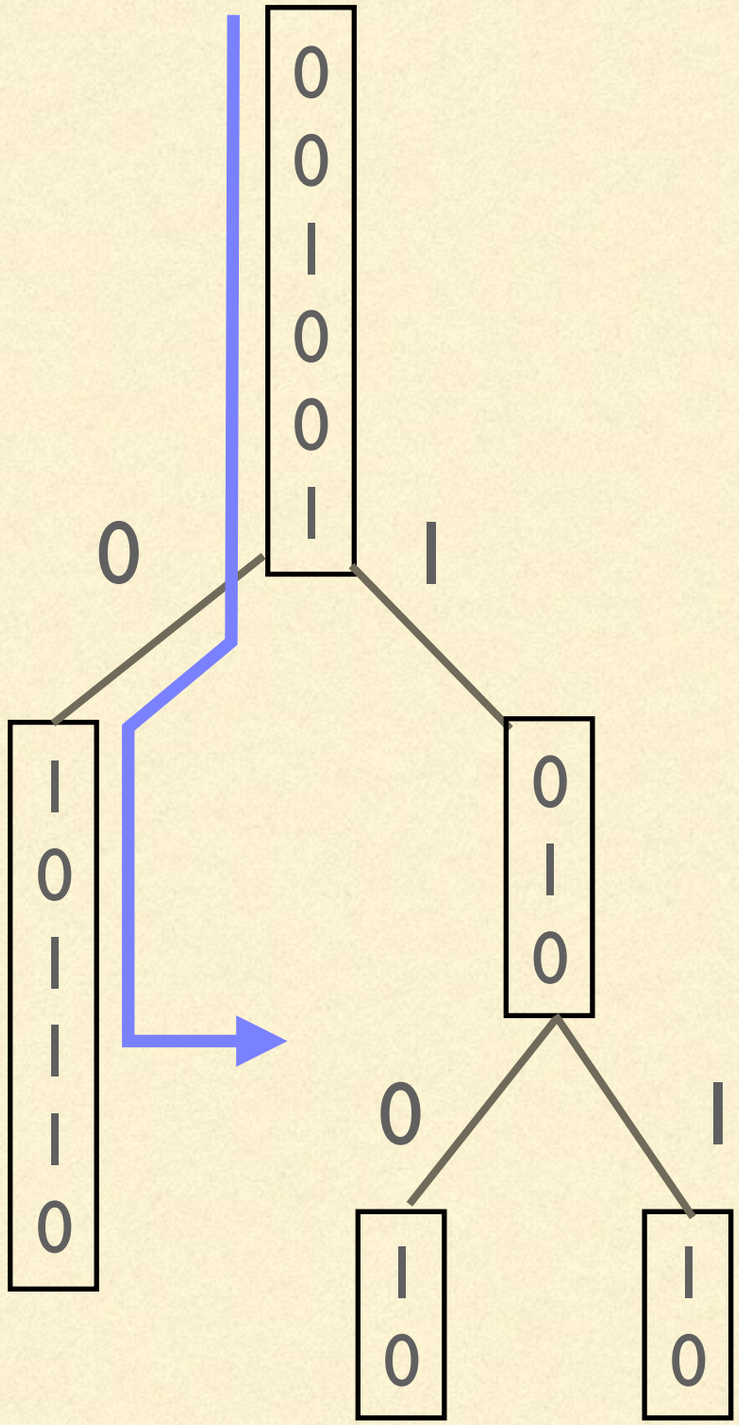
Rank 0 ⇒ bucket 0!



Rank 0 ⇒ bucket 0!

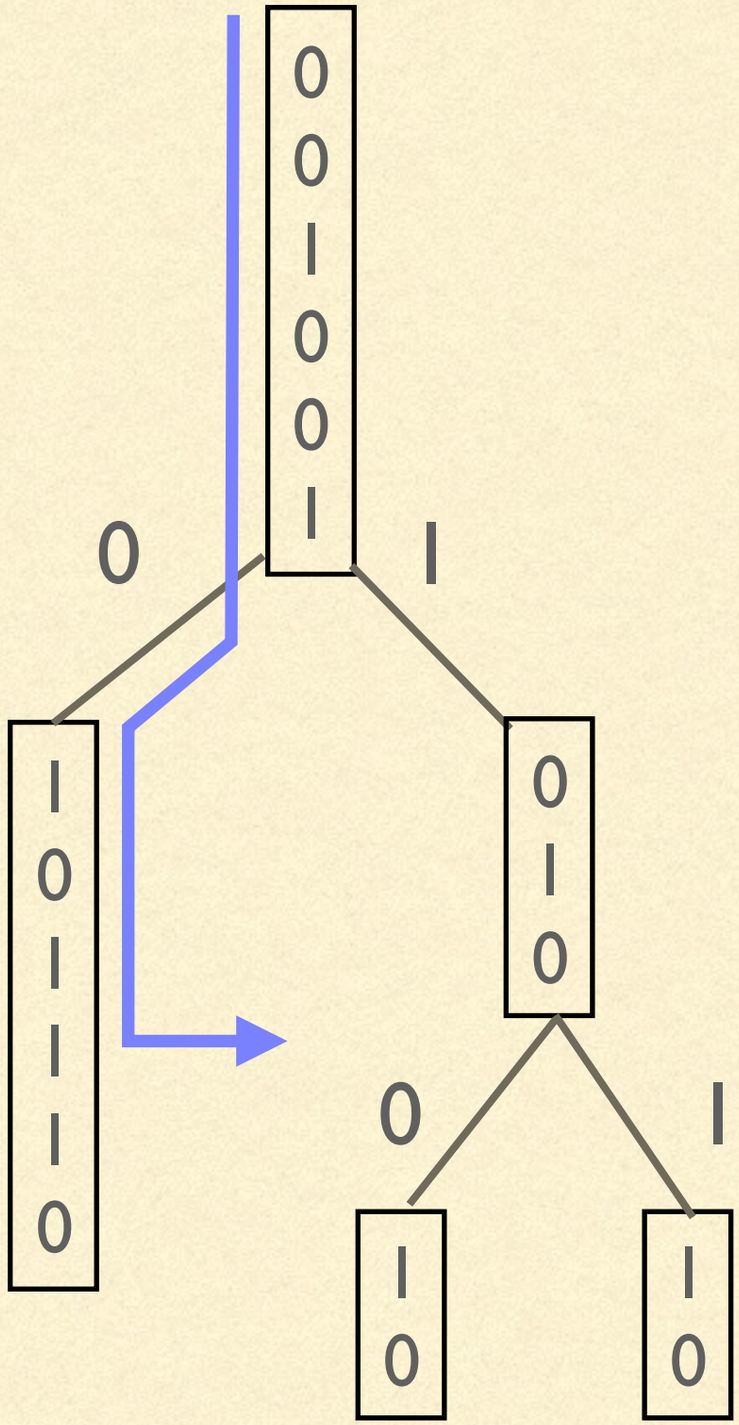

0
0
1
0
0
1
0
1
1
1
1
0
0
1
0
0
1
0
1
0


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



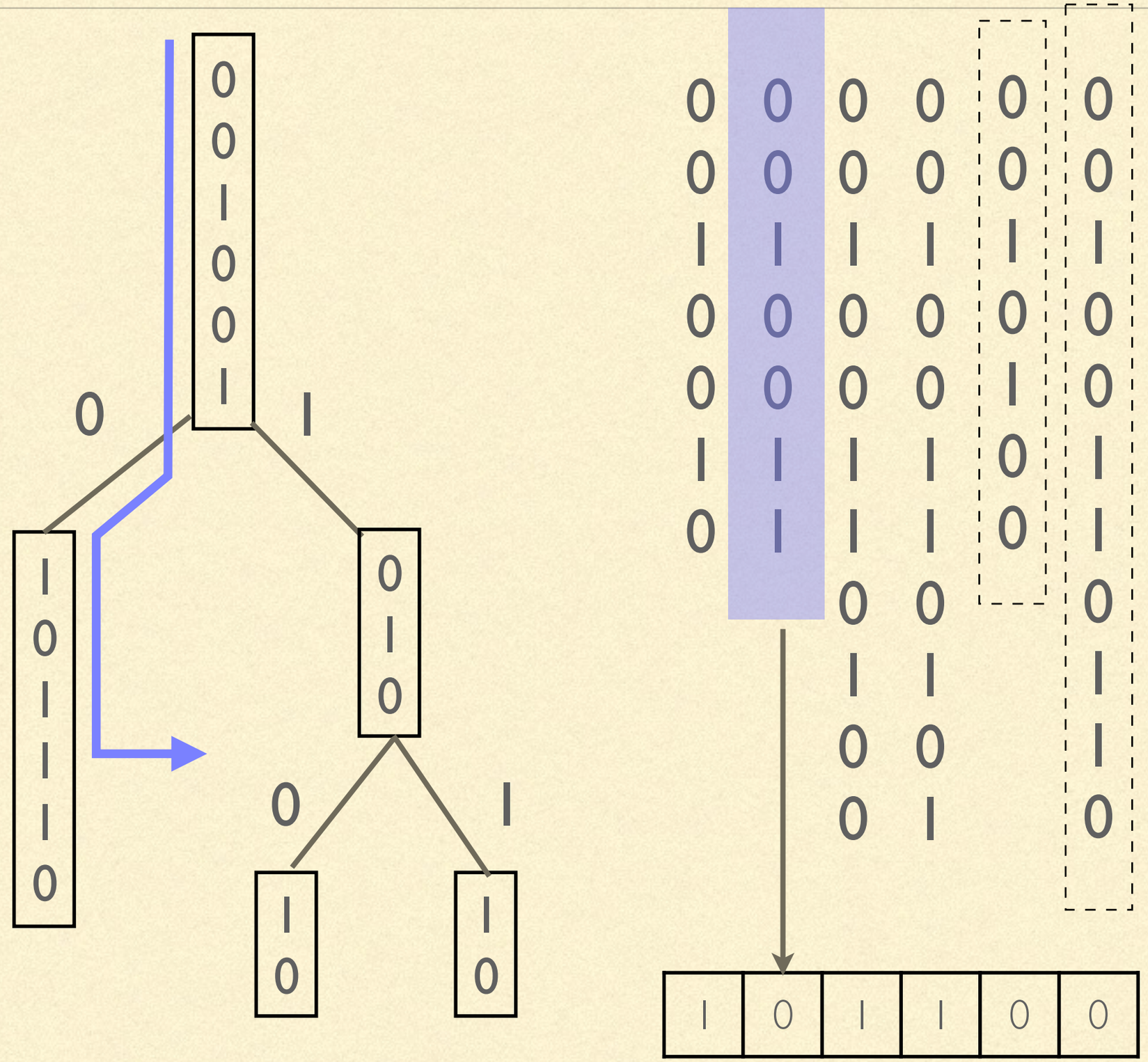
0	0	0	0	0	0
0	0	0	0	0	0
1	1	1	1	1	1
0	0	0	0	0	0
0	0	0	0	1	0
1	1	1	1	0	1
0	1	1	1	0	1
		0	0		0
		1	1		1
		0	0		1
		0	1		0

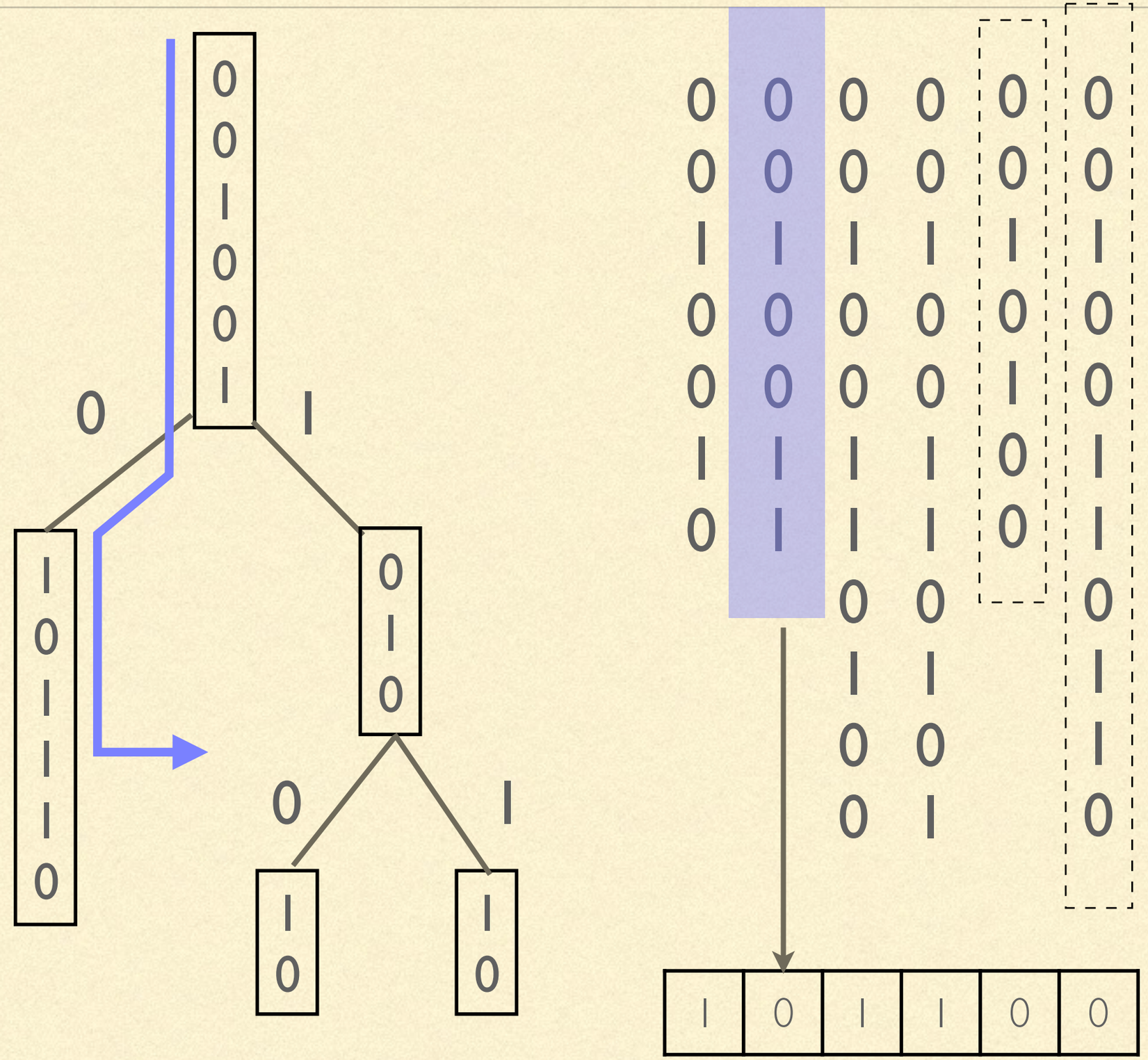
1	0	1	1	0	0



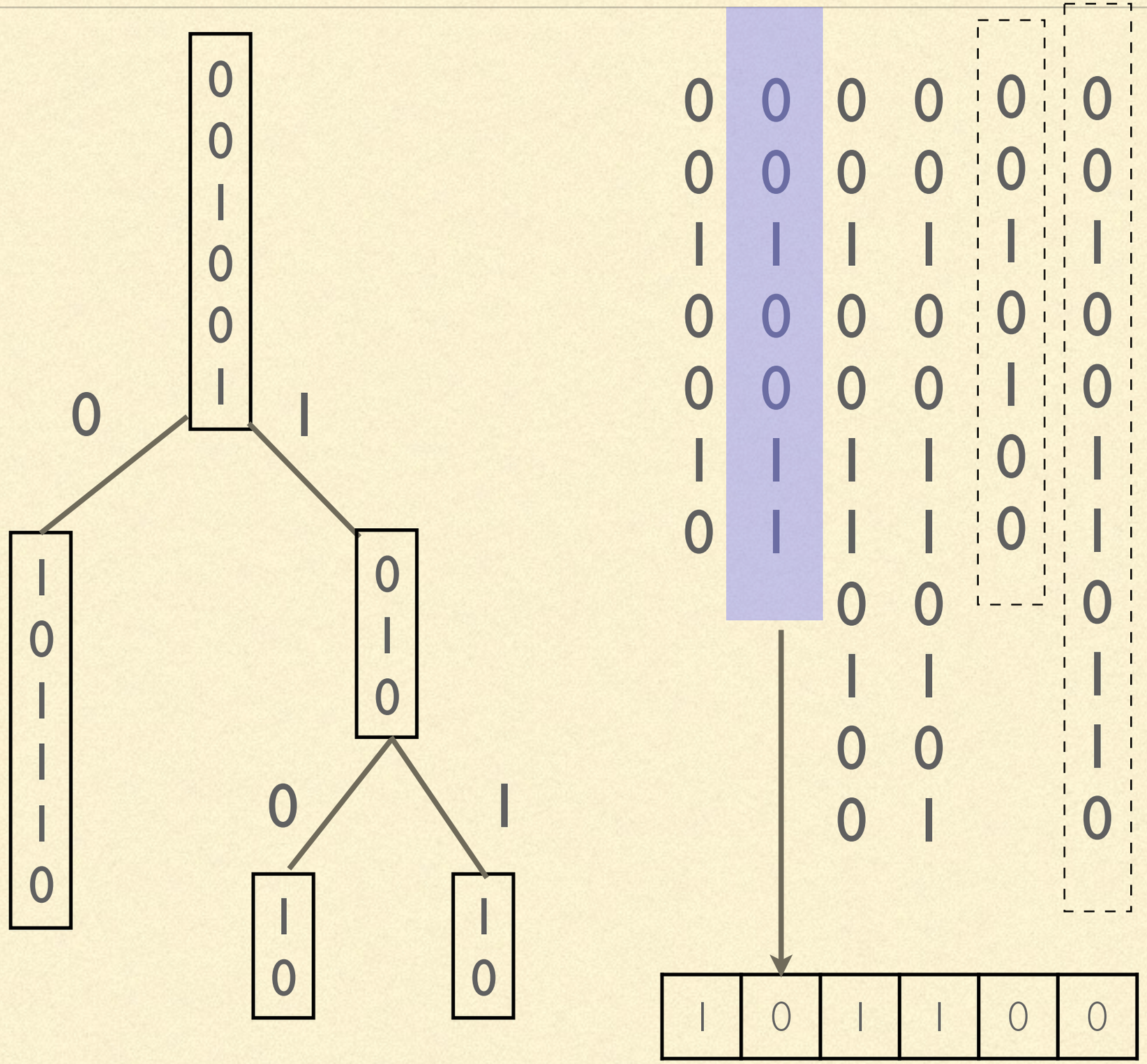
0	0	0	0	0	0
0	0	0	0	0	0
1	1	1	1	1	1
0	0	0	0	0	0
0	0	0	0	1	0
1	1	1	1	0	1
0	1	1	1	0	1
		0	0		0
		1	1		1
		0	0		1
		0	1		0

1	0	1	1	0	0

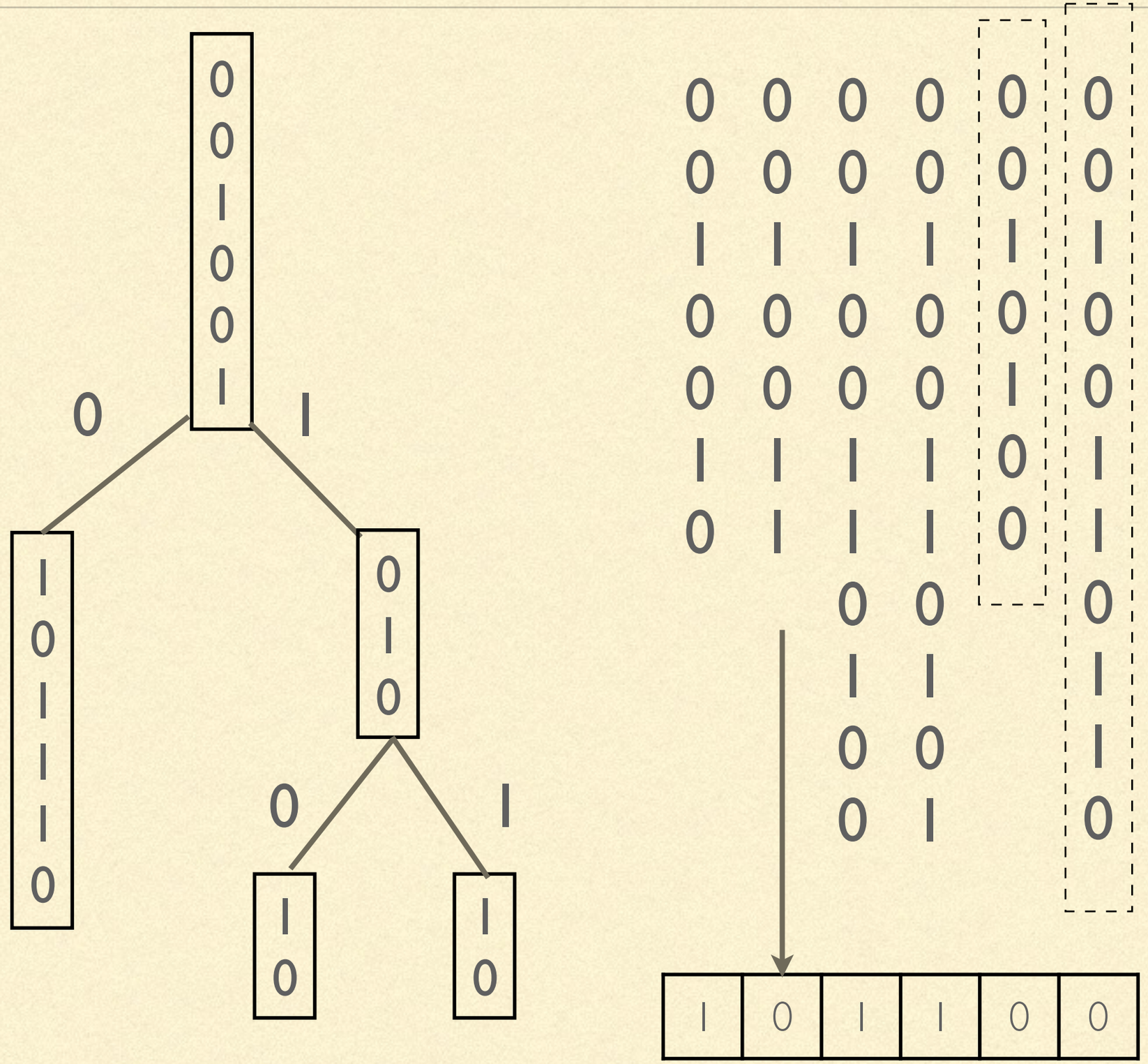

0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 1 0
0
1
1 0
1 0
0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
0 0 0 0 0 0
0 0 0 0 1 0
1 1 1 1 0 1
0 1 1 1 0 1
0 0 1 1 0
1 1 1 1
0 0 1 1
0 1 0
1 0 1 1 0 0



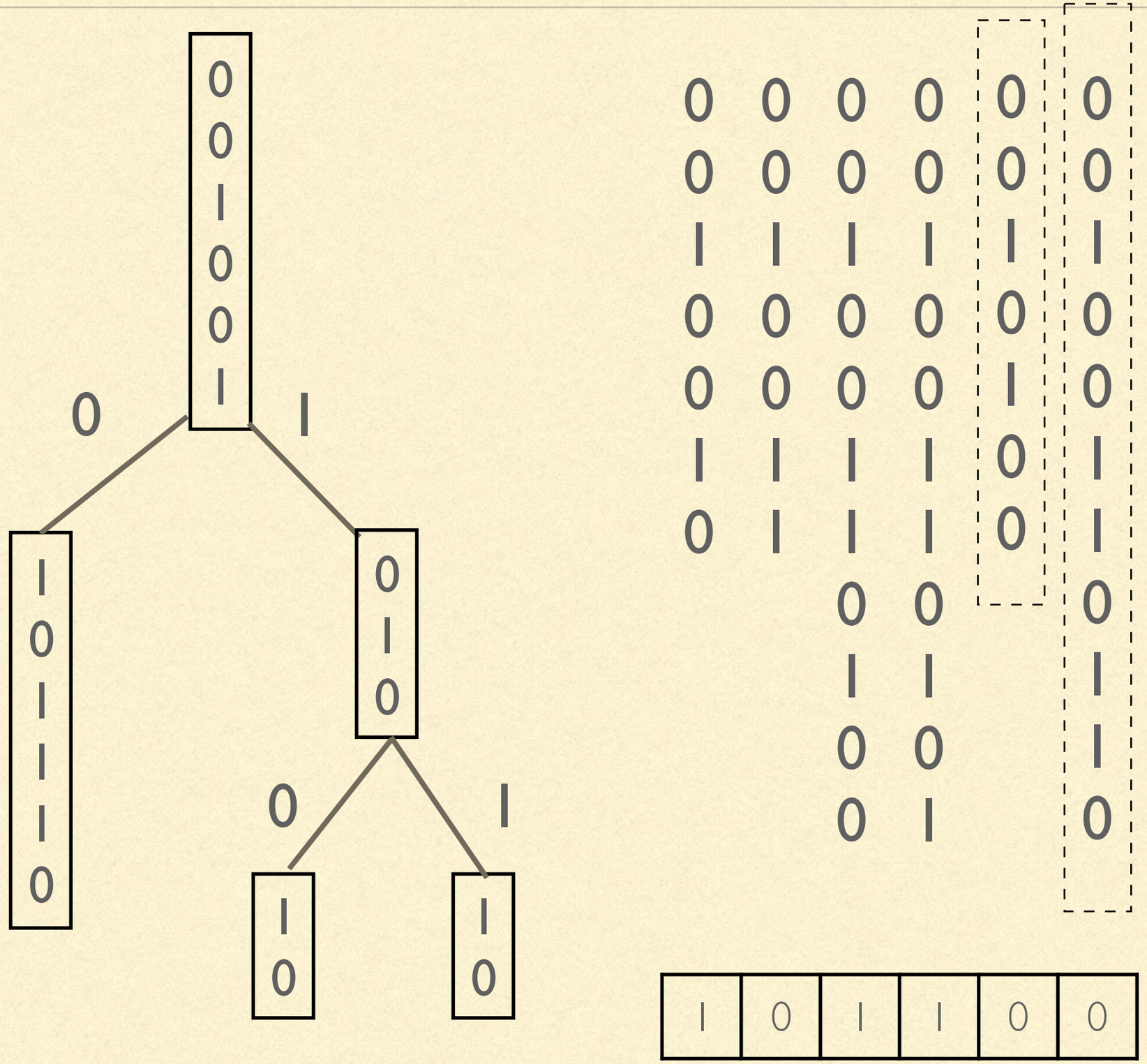
Rank 1 ⇒ bucket 1!



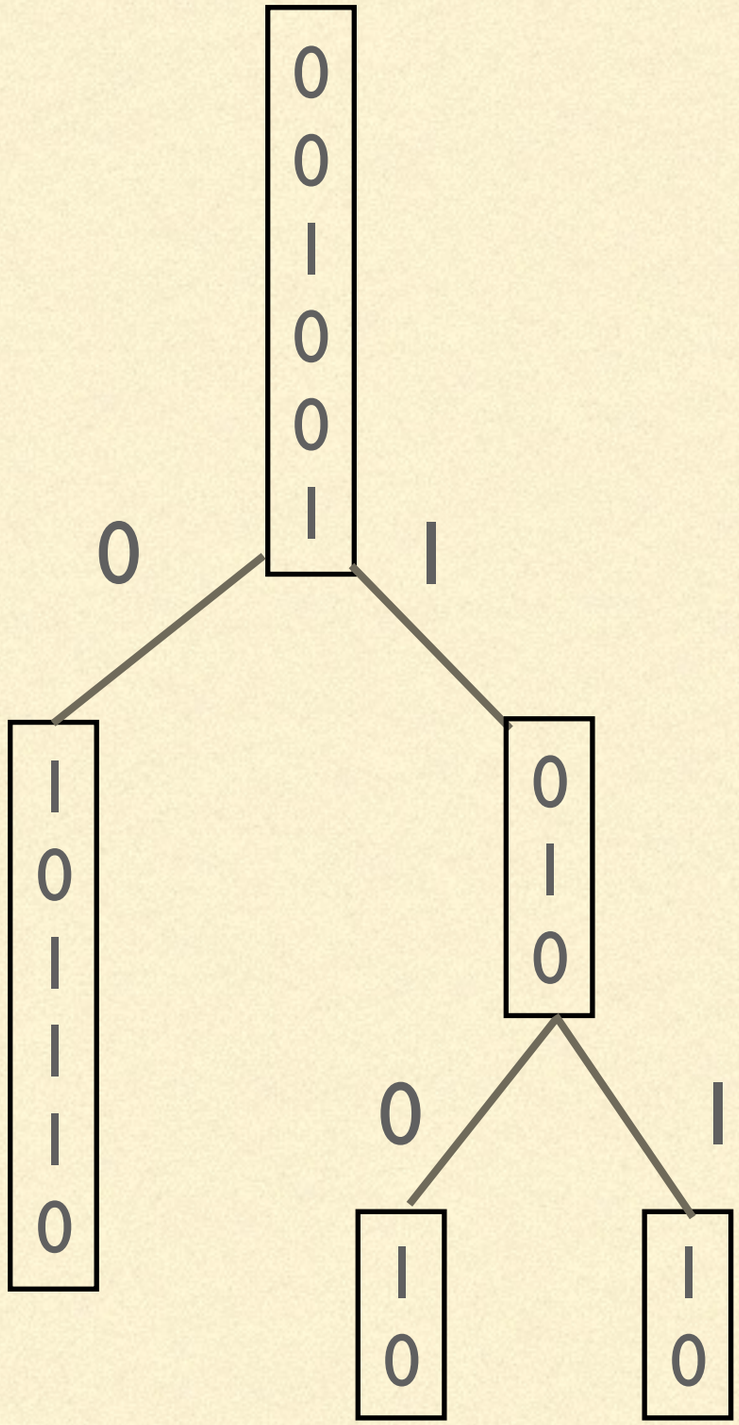
Rank 1 ⇒ bucket 1!

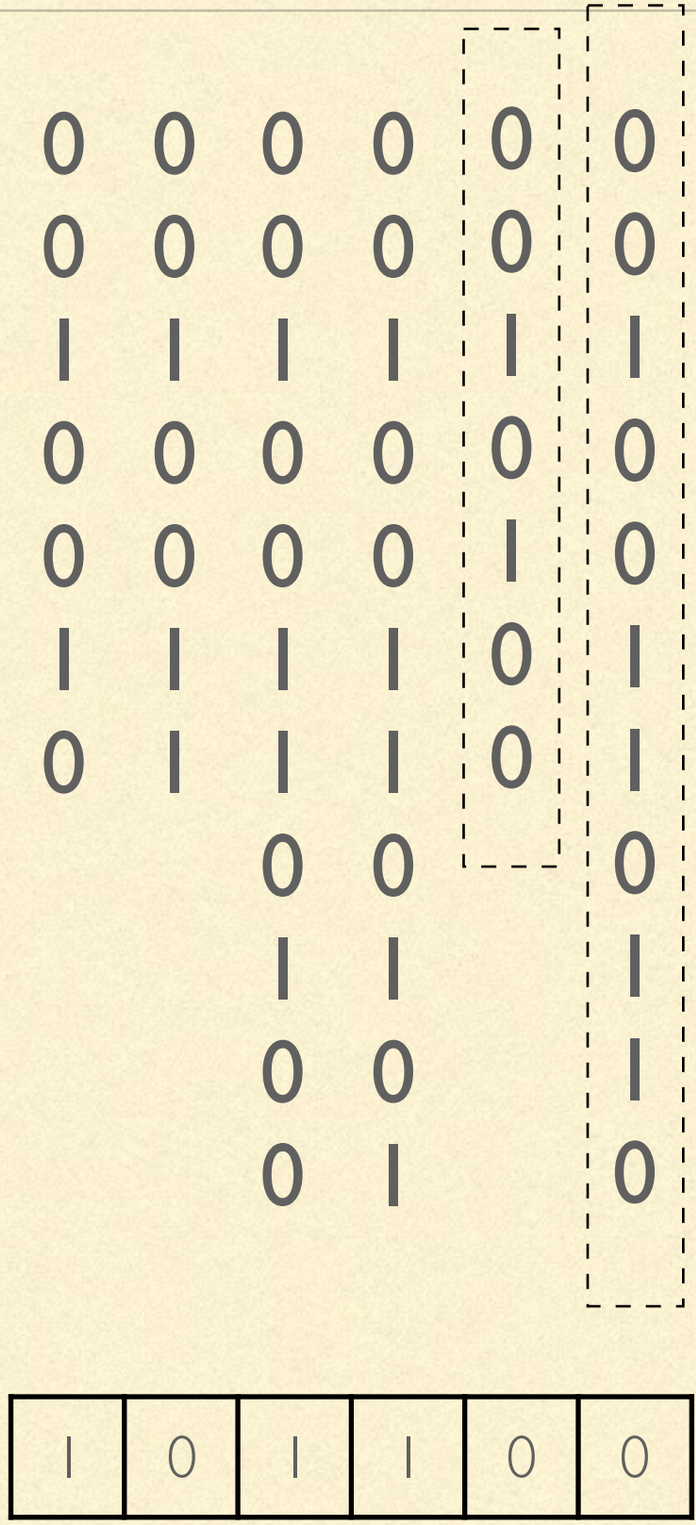


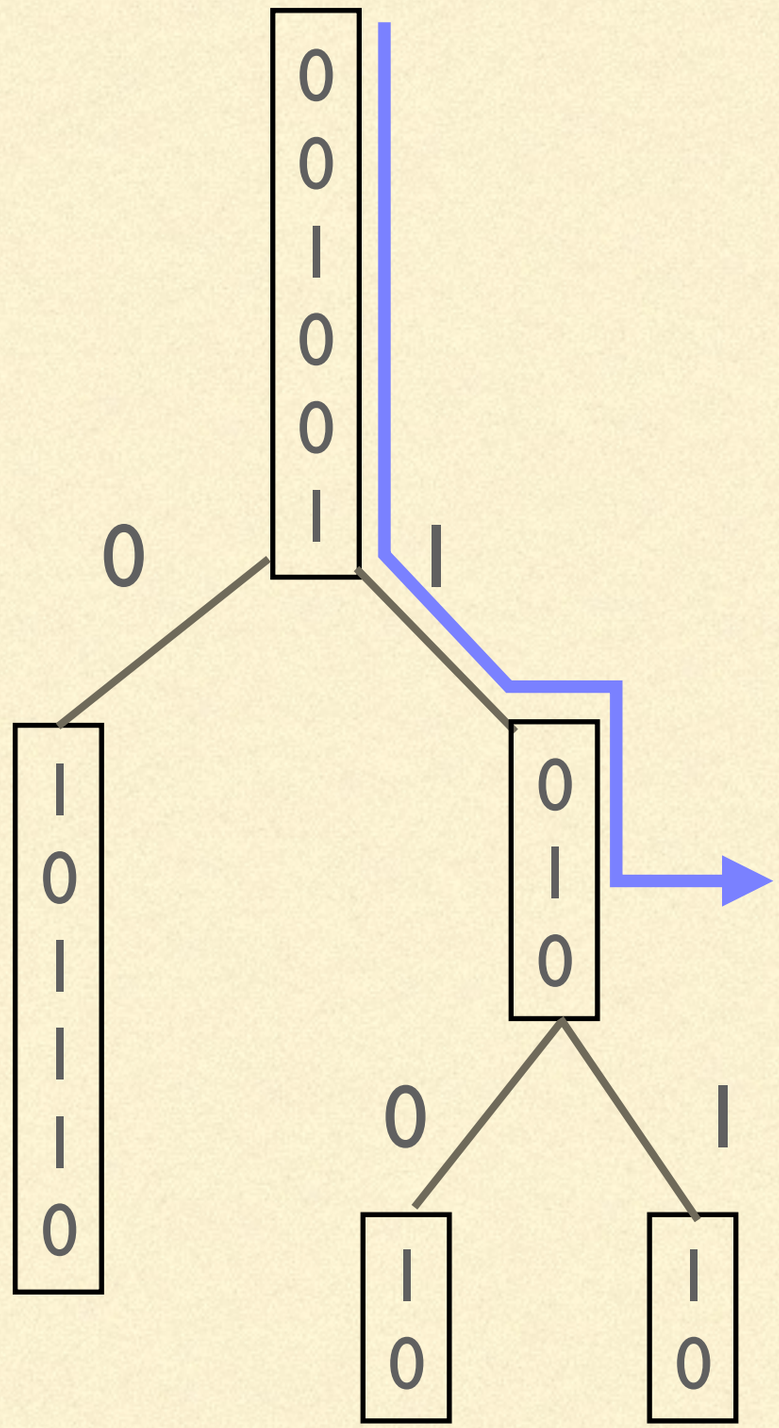
Rank 1 ⇒ bucket 1!

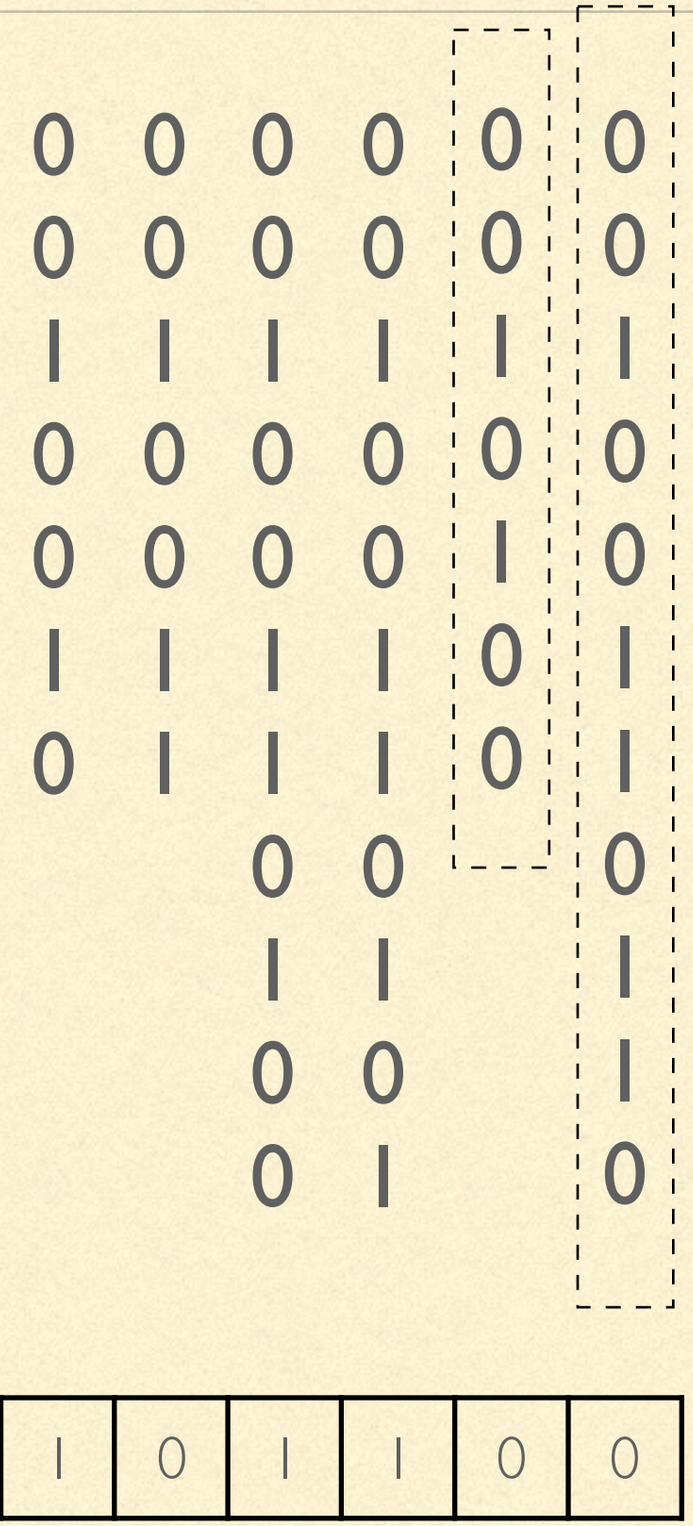


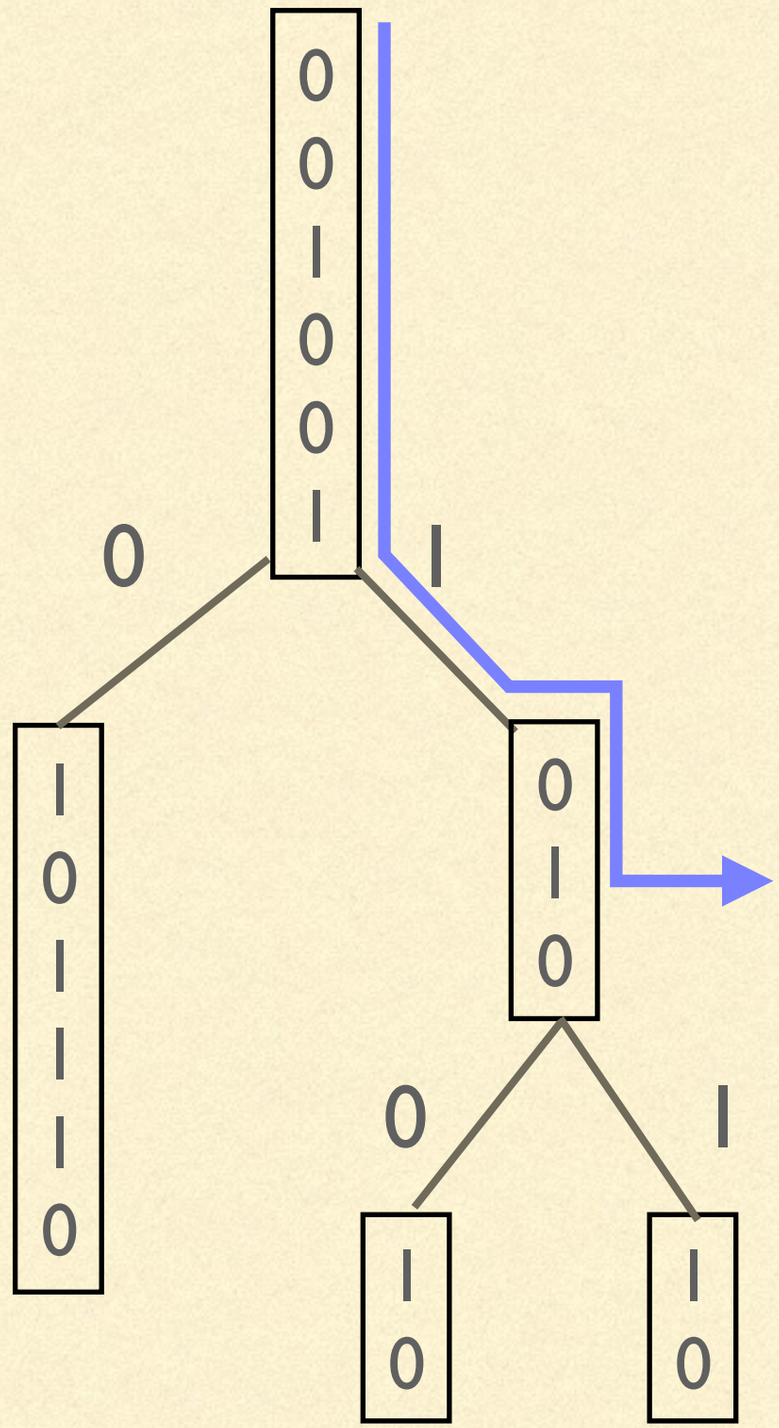
Rank 1 ⇒ bucket 1!

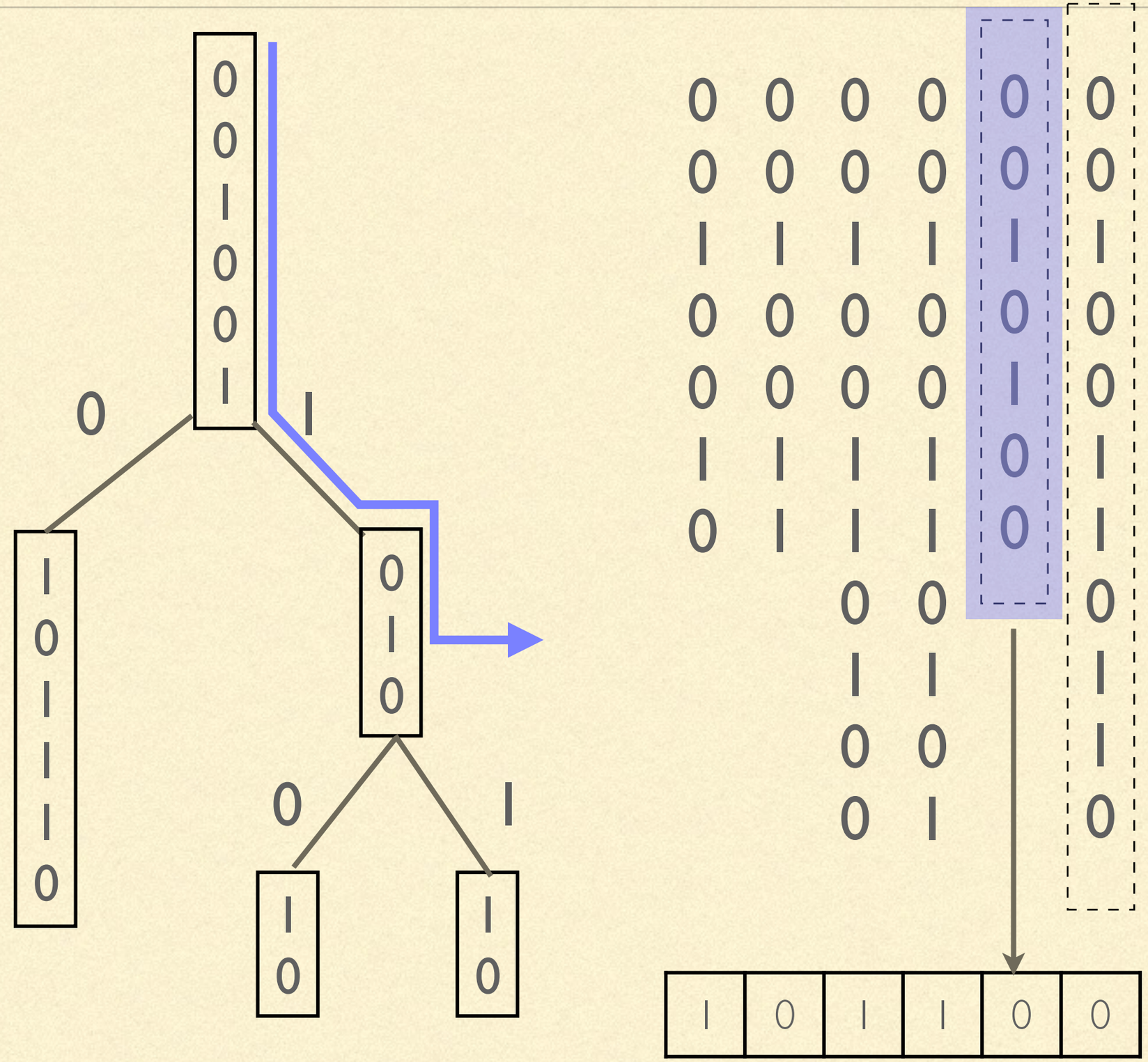

0
0
1
0
0
1
0
1
1
1
1
0
0
1
0
0
1
0
1
0

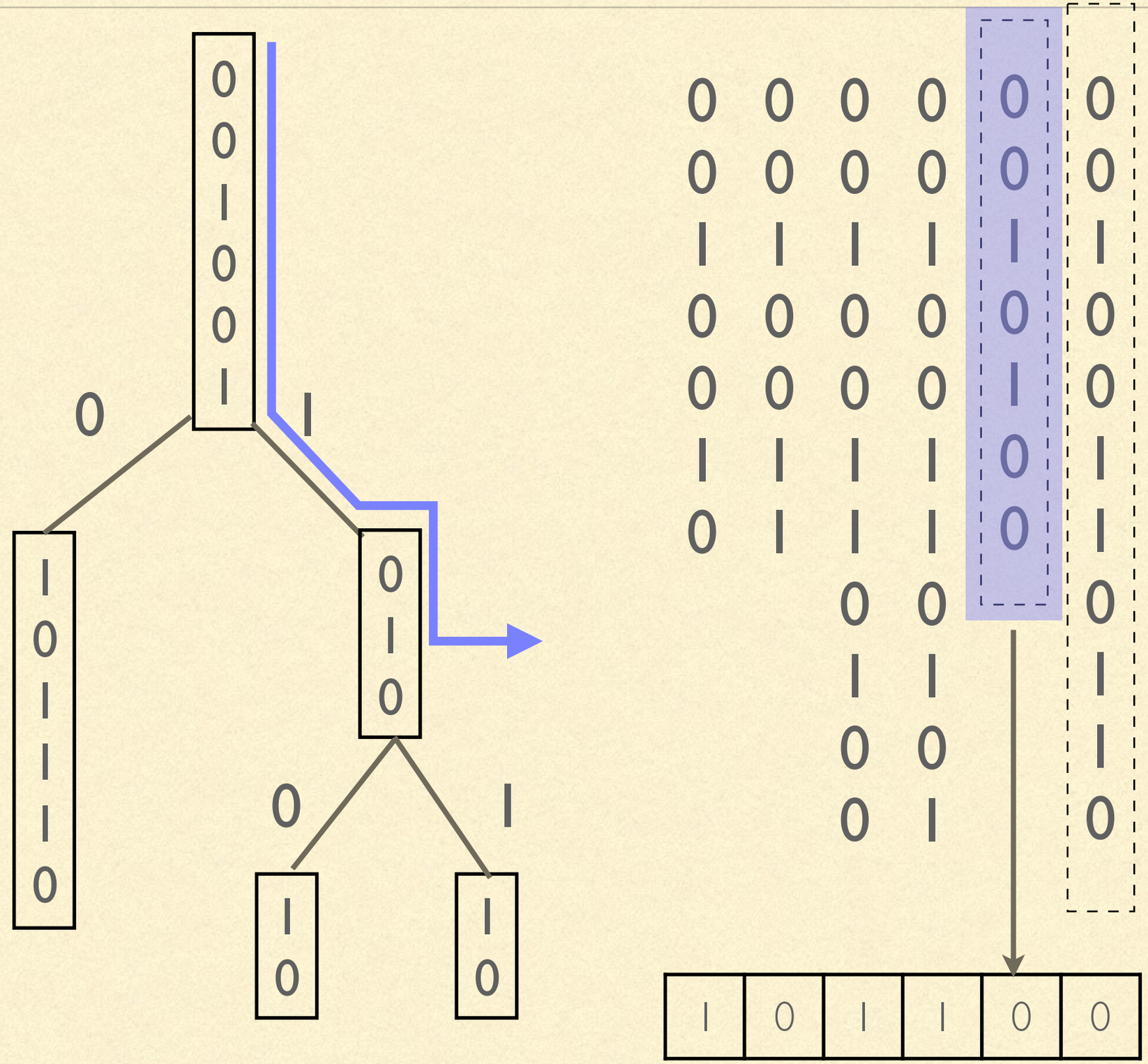

0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
0 0 0 0 0 0
0 0 0 0 1 0
1 1 1 1 0 1
0 1 1 1 0 1
0 0 1 1
0 1 0 1
0 1 1 0


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

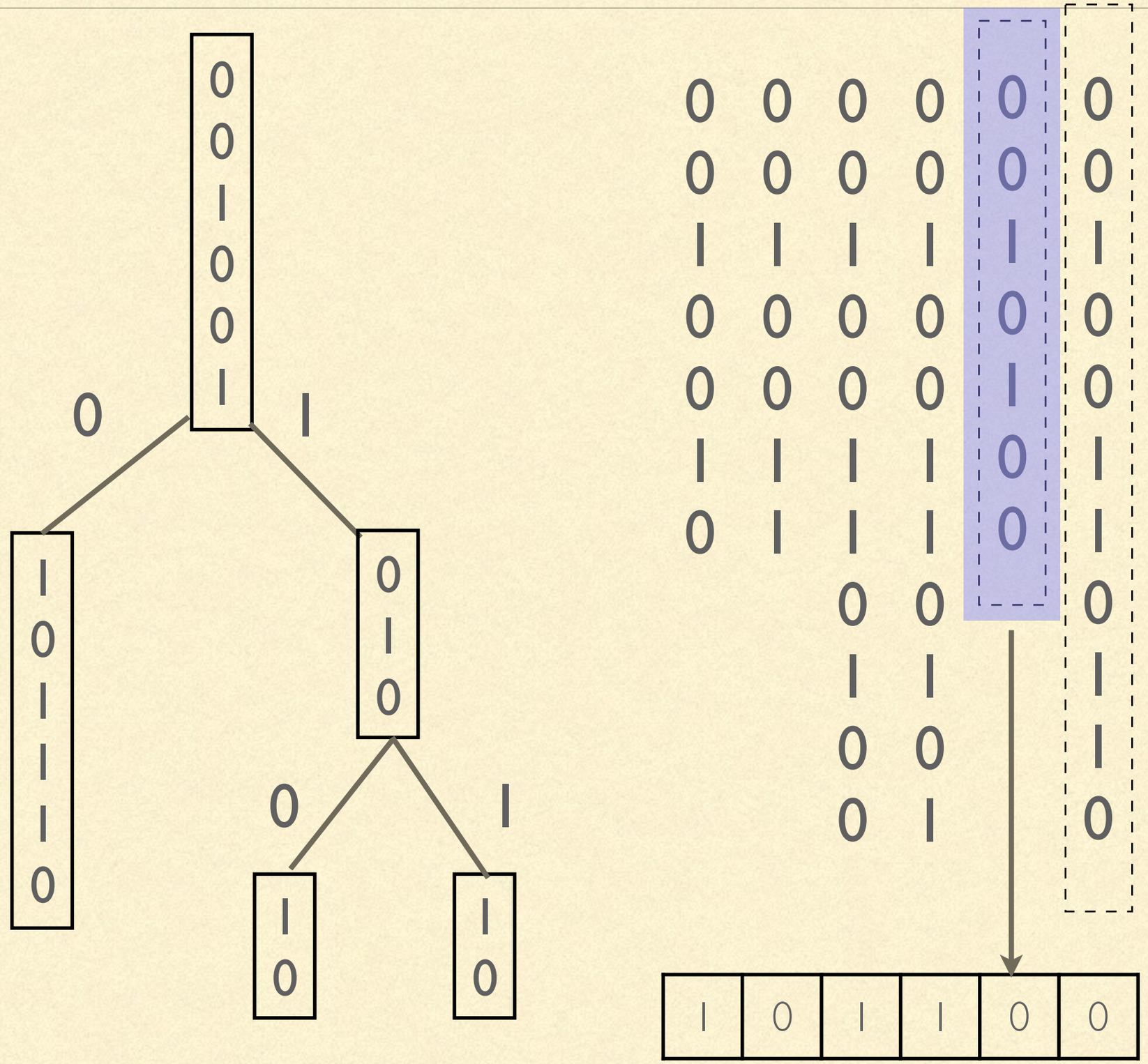

0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
0 0 0 0 0 0
0 0 0 0 1 0
1 1 1 1 1 1
0 1 1 1 0 0
0 0 1 1 1 0
0 1 0 1 1 0
0 1 0 1 0


0
1
0
1
0
1

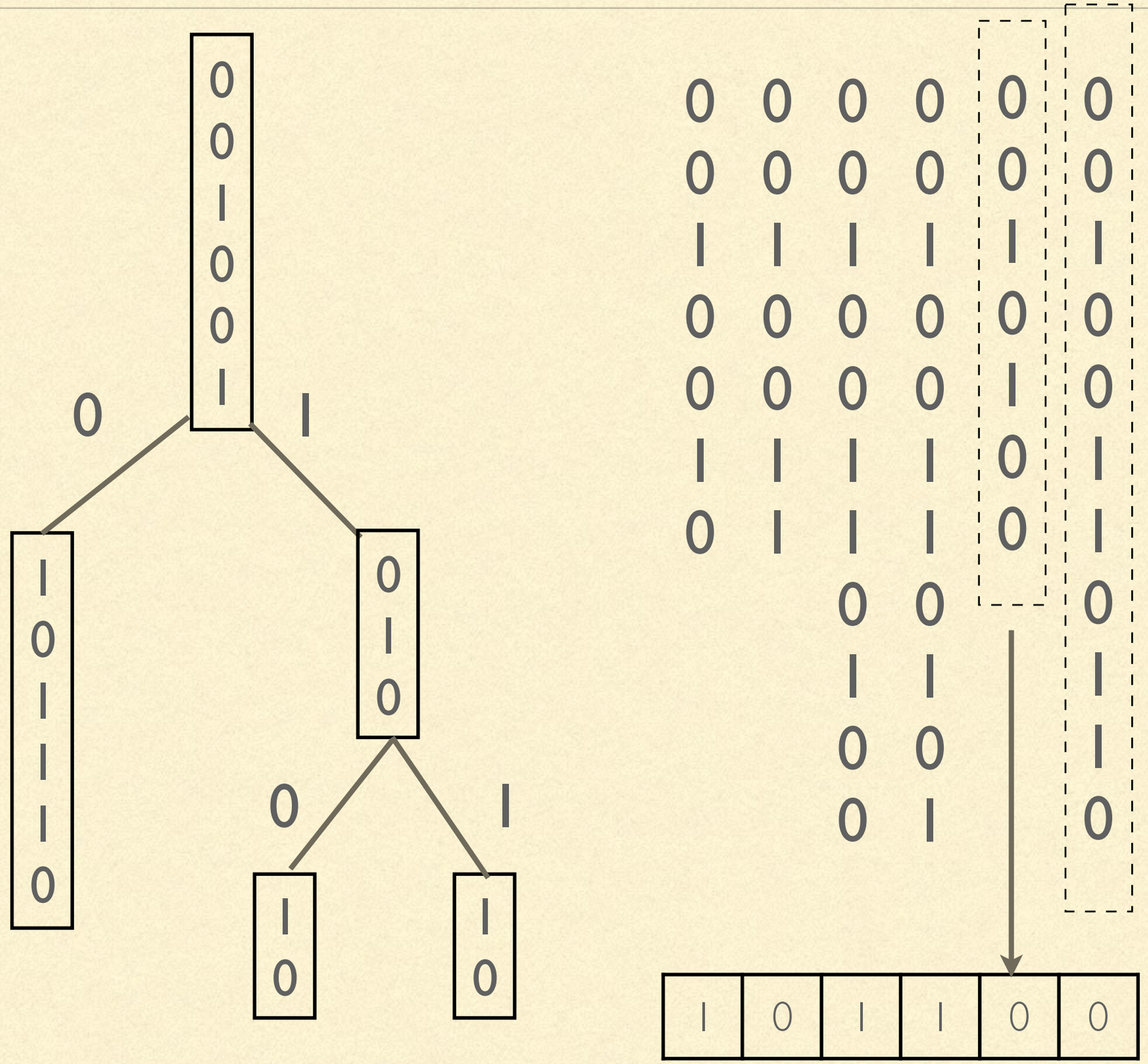

0 0 1 0 0 1
0
1
1 0 1 1 1 0
0 1 0
0
1
1 0
1 0
0 0 0 0 0 0
0 0 0 0 0 0
1 1 1 1 1 1
0 0 0 0 0 0
0 0 0 0 1 0
1 1 1 1 0 1
0 1 1 1 0 1
0 0 1 1 0 1
0 0 1 1 1 0
1 0 1 1 0 0

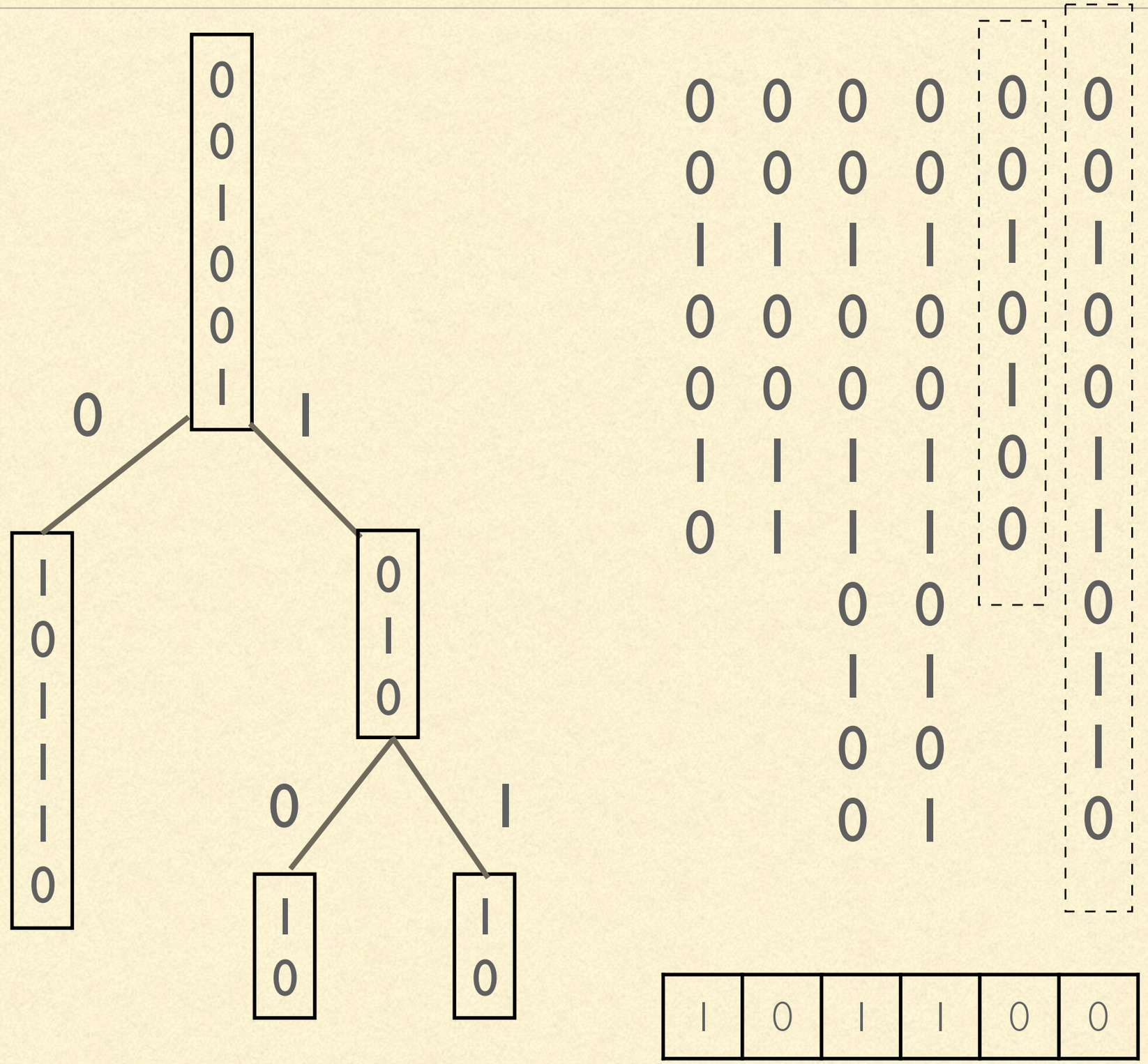


Rank 3 ⇒ bucket 3!

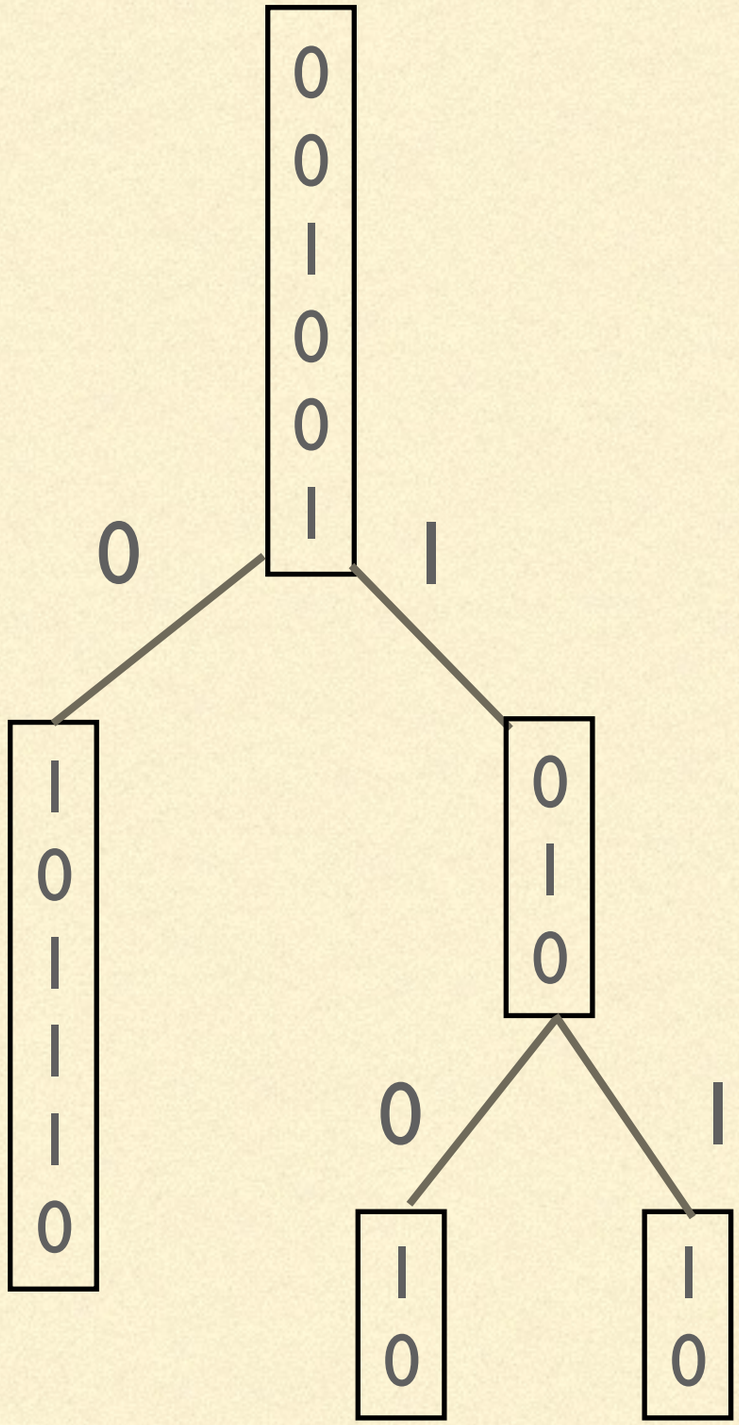


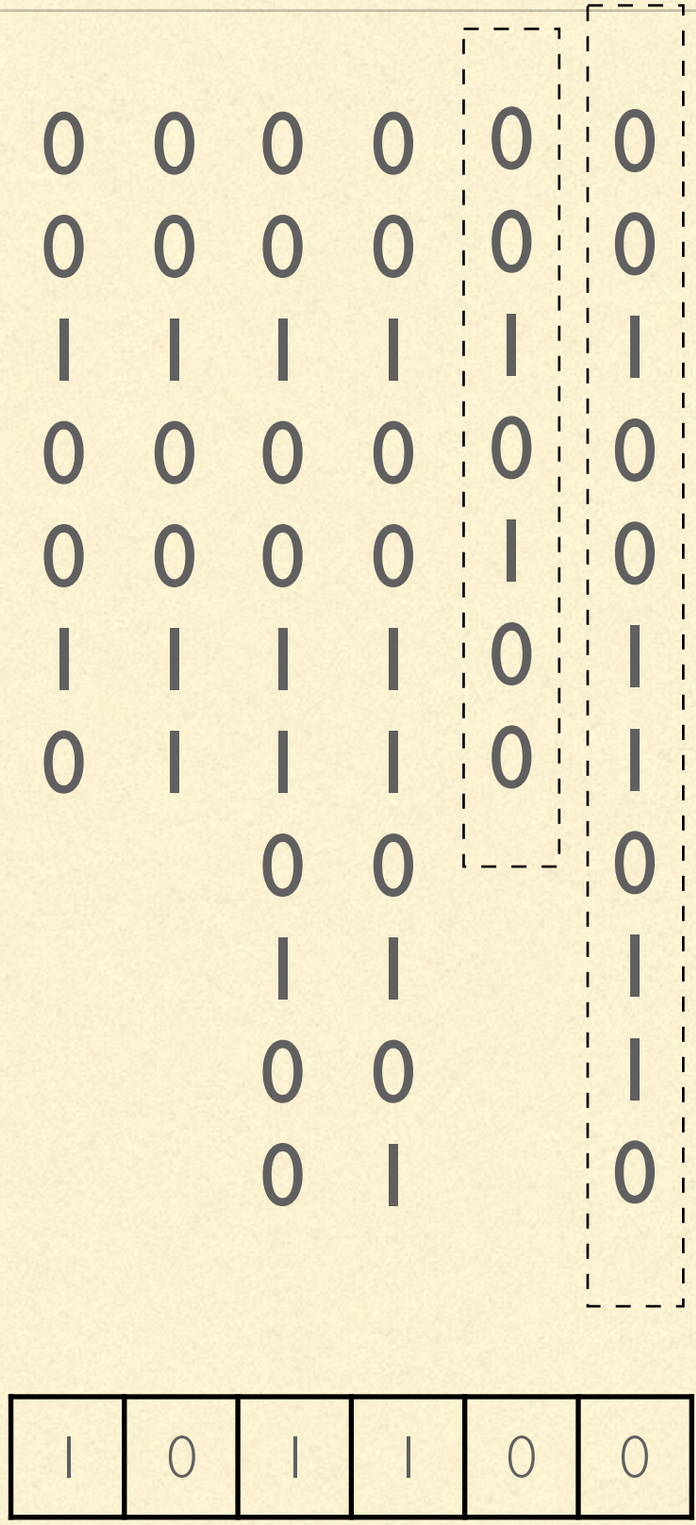
Rank 3 ⇒ bucket 3!

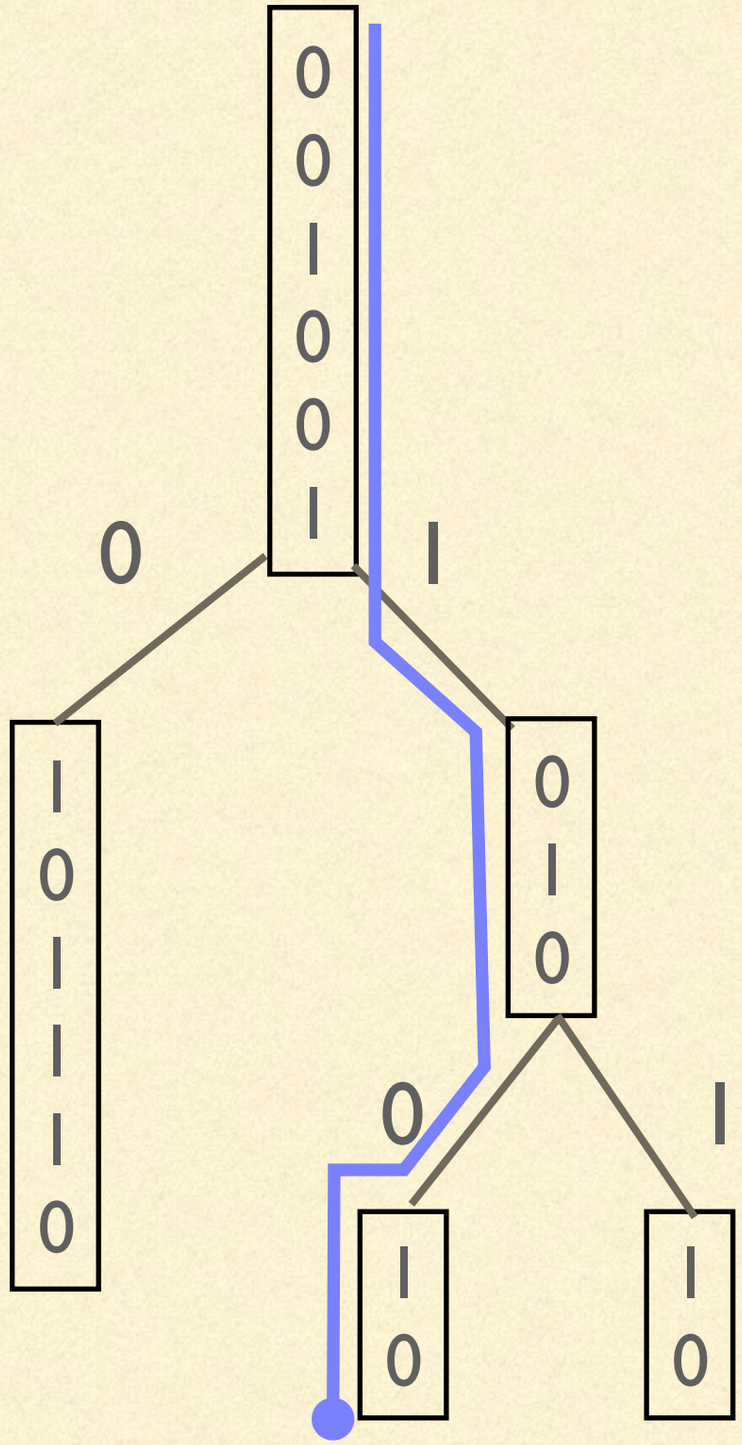

0
1
0
1
0
1



Rank 3 ⇒ bucket 3!

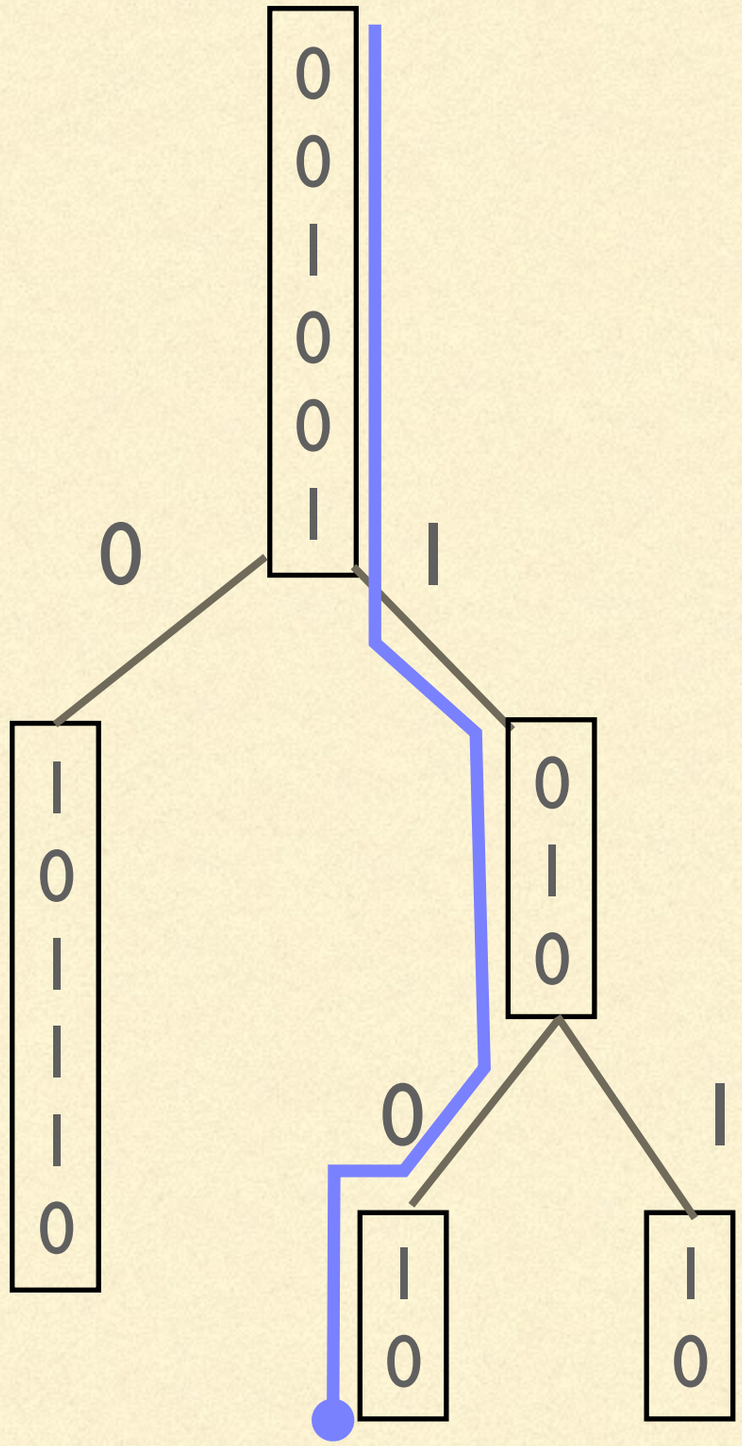

0
0
1
0
0
1
0
1
1
1
1
0
0
1
0
0
1
0
1
0


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



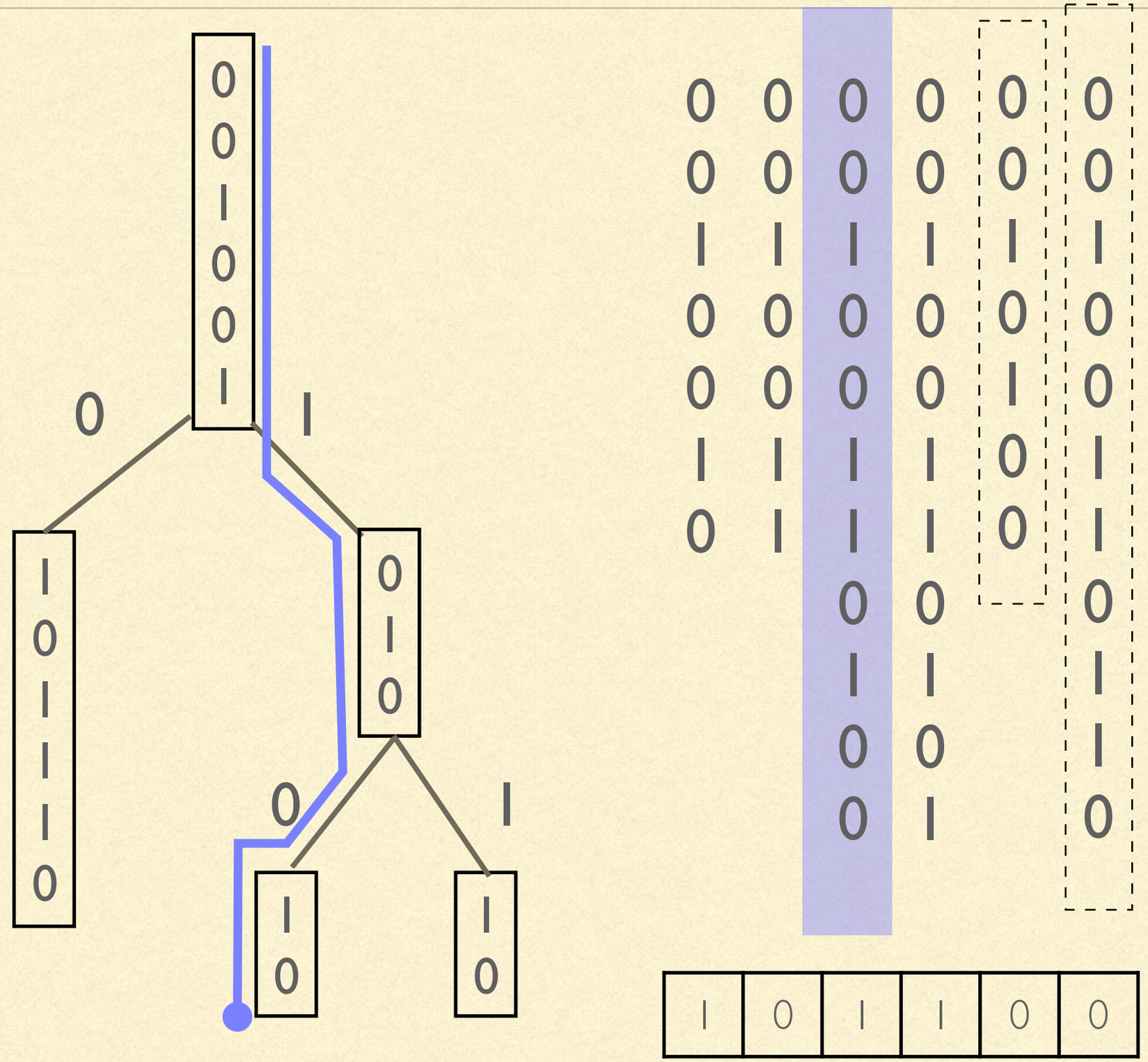
0	0	0	0	0	0
0	0	0	0	0	0
1	1	1	1	1	1
0	0	0	0	0	0
0	0	0	0	1	0
1	1	1	1	0	1
0	1	1	1	0	1
		0	0		0
		1	1		1
		0	0		1
		0	1		0

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

1	0	1	1	0	0



Rank 1 ⇒ bucket 1!

ALL IN ALL

ALL IN ALL

- All in all, using buckets of size log w…

ALL IN ALL

- All in all, using buckets of size log w…
- …we can do MMPH

ALL IN ALL

- All in all, using buckets of size log w…
- …we can do MMPH
  - in time O(log w)=O(log log u)

ALL IN ALL

- All in all, using buckets of size log w…
- …we can do MMPH
  - in time O(log w)=O(log log u)
  - using O(n log log w)=O(n log log log u) bits (i.e., constant cost per key to all practical purposes)