Functional Vectors, Maps, and Sets
in Julia

Zach Allaun
Strange Loop 2013

Julia lang

- MATLAB-like syntax, poised as a language for technical computing
- dynamic
- fast, JIT compiled

Julia and Types

- run-time type tags, not compile-time types
- performance and expressiveness, not safety or correctness
- primary unit of abstraction: the generic function

Generic Functions

```
abstract BinaryTree

immutable Node <: BinaryTree
    data
    left::BinaryTree
    right::BinaryTree
end

immutable Leaf <: BinaryTree
    data
end

Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))
```

Generic Functions

```
abstract BinaryTree

immutable Node <: BinaryTree
    data
    left::BinaryTree
    right::BinaryTree
end

immutable Leaf <: BinaryTree
    data
end

Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))
```

Generic Functions

```
abstract BinaryTree

immutable Node <: BinaryTree
    data
    left::BinaryTree
    right::BinaryTree
end

immutable Leaf <: BinaryTree
    data
end

Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))
```

Generic Functions

```
abstract BinaryTree

immutable Node <: BinaryTree
    data
    left::BinaryTree
    right::BinaryTree
end

immutable Leaf <: BinaryTree
    data
end

Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))
```

Generic Functions

```
3 in Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))

# parses to

Base.in(
    3,
    Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))
)
```

Generic Functions

```
3 in Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))

# parses to

Base.in(
    3,
    Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))
)
```

Generic Functions

```
Base.in(data, l::Leaf) = data == l.data

function Base.in(data, n::Node)
    (data == n.data
     || data in n.left
     || data in n.right)
end

3 in Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))
#=> true
```

Generic Functions

```
Base.in(data, l::Leaf) = data == l.data

function Base.in(data, n::Node)
    (data == n.data
     || data in n.left
     || data in n.right)
end

3 in Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))
#=> true
```

Generic Functions

```
Base.in(data, l::Leaf) = data == l.data

function Base.in(data, n::Node)
    (data == n.data
     || data in n.left
     || data in n.right)
end

3 in Node(1, Leaf(2), Node(3, Leaf(4), Leaf(5)))
#=> true
```

Functional Data Structures

- immutable values, not mutable objects
- “change” returns a new value, leaving the old one unmodified
- they’re persistent
- they’re fast

Vectors

```
a = [1, 2, 3, 4, 5]

a[1]
#=> 1

push!(a, 6)

length(a)
#=> 6
```

Vectors

```
a = [1, 2, 3, 4, 5]

a[1]
#=> 1

push!(a, 6)

length(a)
#=> 6
```

Maps

```
knights = ["Sir Galahad" => "the pure",
           "Sir Lancelot" => "the brave",
           "Sir Bedevere" => "the wise"]

knights["Sir Galahad"]
#=> "the pure"

knights["Sir Robin"] =
    "the not-quite-so-brave-as-Sir-Lancelot"

knights["Sir Robin"]
#=> "the not-quite-so-brave-as-Sir-Lancelot"
```

Maps

```
knights = ["Sir Galahad" => "the pure",
           "Sir Lancelot" => "the brave",
           "Sir Bedevere" => "the wise"]

knights["Sir Galahad"]
#=> "the pure"

knights["Sir Robin"] =
    "the not-quite-so-brave-as-Sir-Lancelot"

knights["Sir Robin"]
#=> "the not-quite-so-brave-as-Sir-Lancelot"
```

Sets

```
alpha = Set("abc"...)

push!(alpha, 'd')

'd' in alpha
#=> true
```

Sets

```
alpha = Set("abc"...)

push!(alpha, 'd')

'd' in alpha
#=> true
```

Persistent Vectors

```
a = @Persistent [1, 2, 3, 4, 5]

a[1]
#=> 1

push(a, 6)
#=> Persistent{Int64}[1, 2, 3, 4, 5, 6]

length(a)
#=> 5
```

Persistent Vectors

```
a = @Persistent [1, 2, 3, 4, 5]

a[1]
#=> 1

push(a, 6)
#=> Persistent{Int64}[1, 2, 3, 4, 5, 6]

length(a)
#=> 5
```

Persistent Maps

```
knights = @Persistent ["Sir Galahad" => "the pure",
                       "Sir Lancelot" => "the brave",
                       "Sir Bedevere" => "the wise"]

knights["Sir Galahad"]
#=> "the pure"

assoc(knights, "Sir Robin",
      "the not-quite-so-brave-as-Sir-Lancelot")

knights["Sir Robin"]
#=> ERROR: key not found
```

Persistent Maps

```
knights = @Persistent ["Sir Galahad" => "the pure",
                       "Sir Lancelot" => "the brave",
                       "Sir Bedevere" => "the wise"]

knights["Sir Galahad"]
#=> "the pure"

assoc(knights, "Sir Robin",
      "the not-quite-so-brave-as-Sir-Lancelot")

knights["Sir Robin"]
#=> ERROR: key not found
```

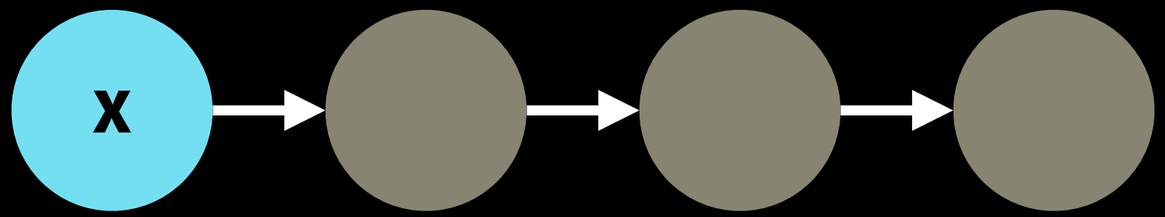
Persistent Sets

```
alpha = @Persistent Set("abc"...)

push(alpha, 'd')
#=> PersistentSet{Char}('a', 'c', 'b', 'd')


'd' in alpha
#=> false
```

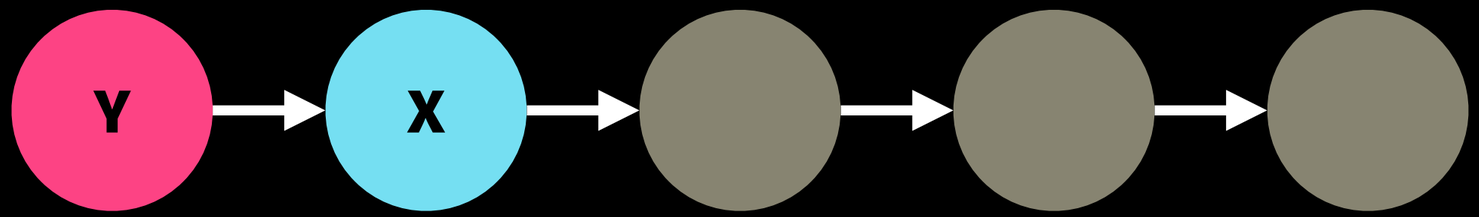
Persistent Sets

```
alpha = @Persistent Set("abc"...)

push(alpha, 'd')
#=> PersistentSet{Char}('a', 'c', 'b', 'd')

'd' in alpha
#=> false
```

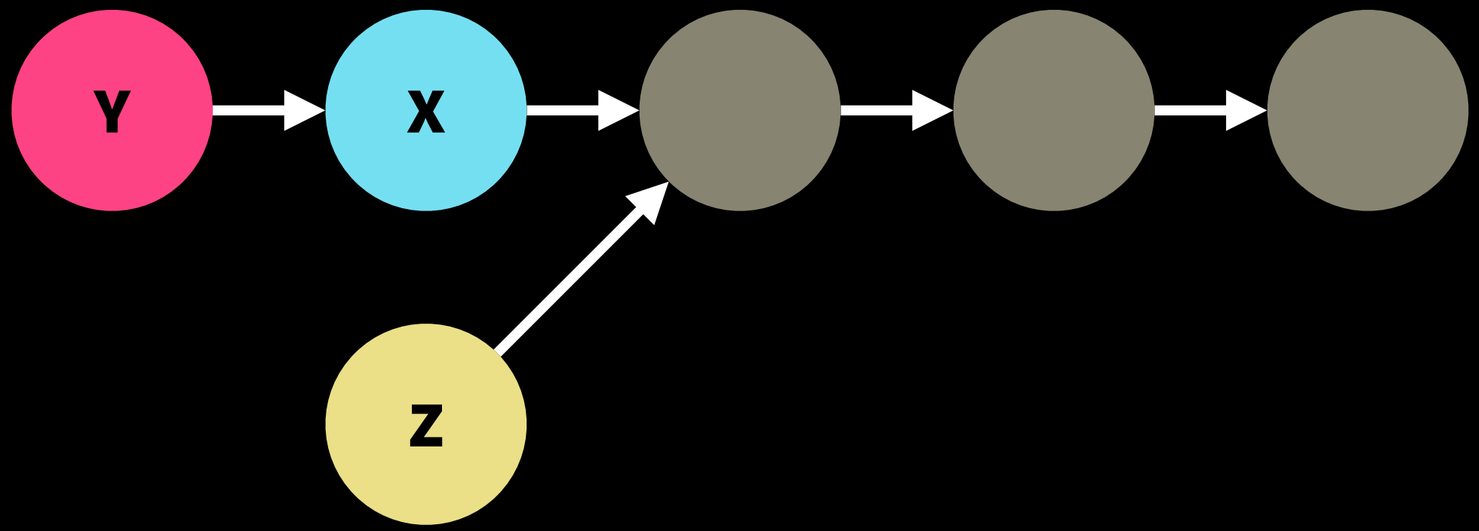
Simple example: Linked List



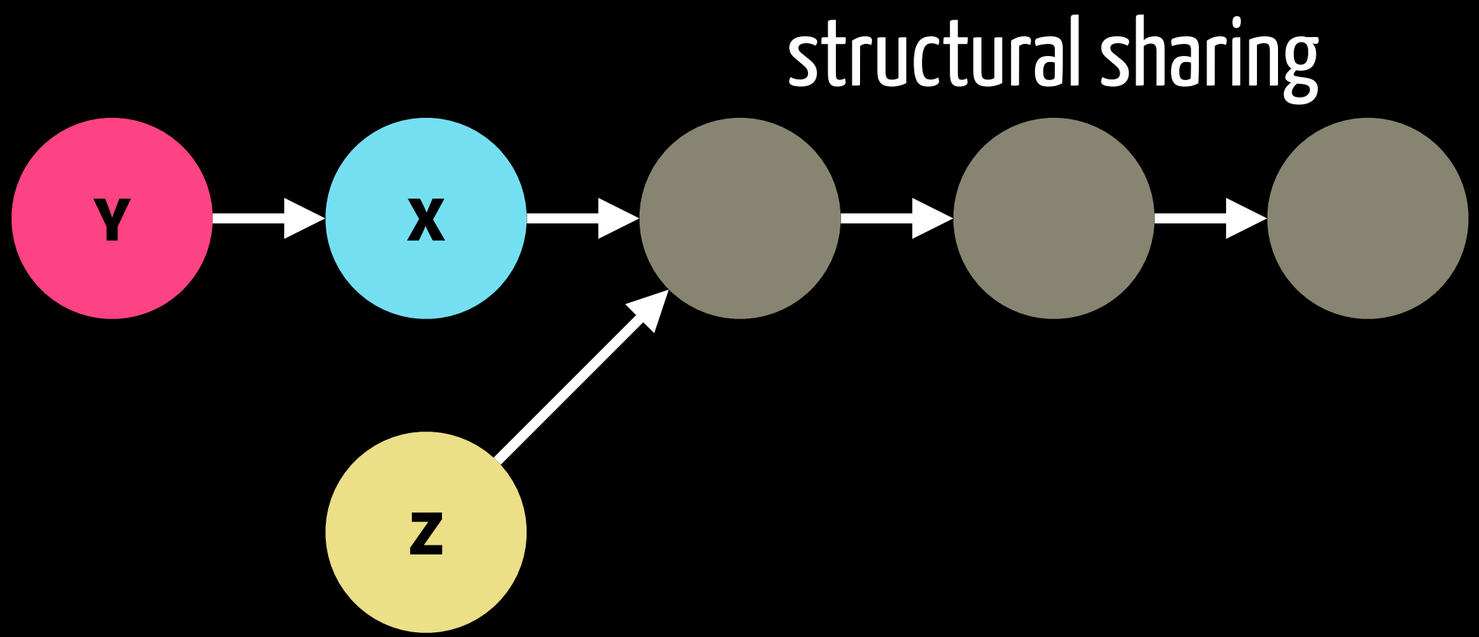
Simple example: Linked List



Simple example: Linked List



Simple example: Linked List



Sharing structure

- space efficiency
- computational efficiency – avoids copying

Phil Bagwell

- Array Mapped Trie
- Hash Array Mapped Trie

Phil Bagwell
+
Rich Hickey

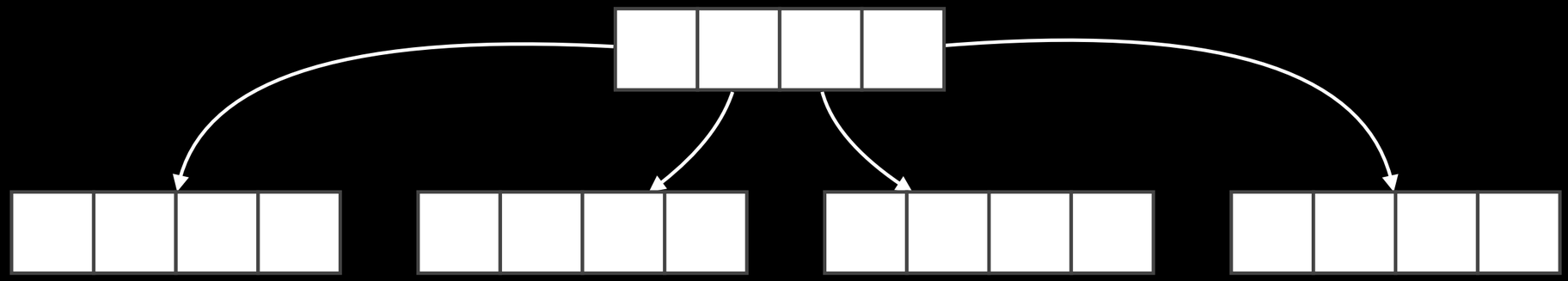
Bitmapped Vector Trie

- data lives in the leaves
- e.g. prefix tree used for string lookup
- bitwise trie

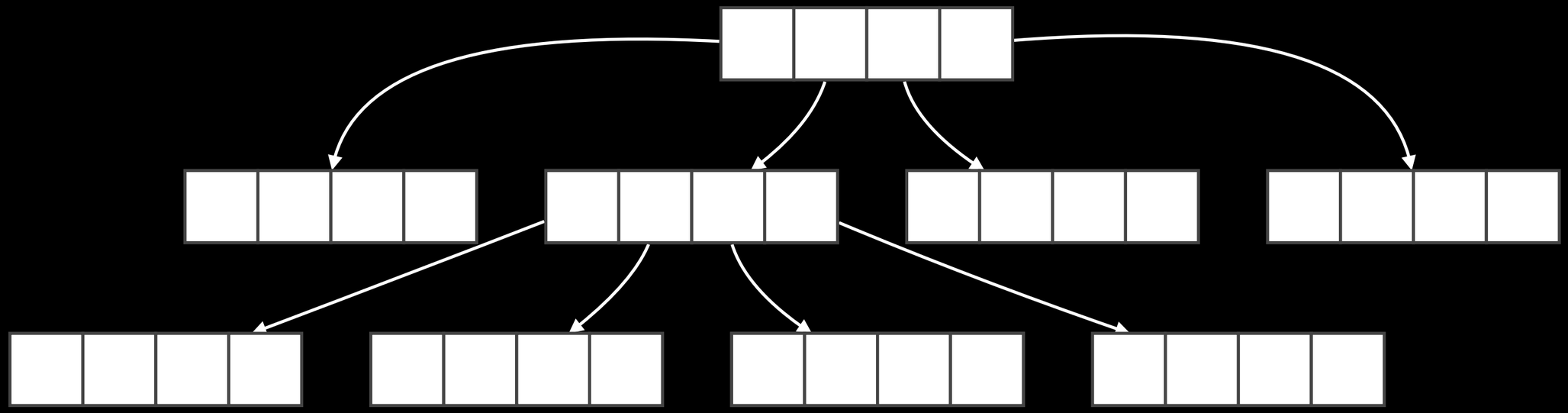
Persistent Vector

Persistent Vector

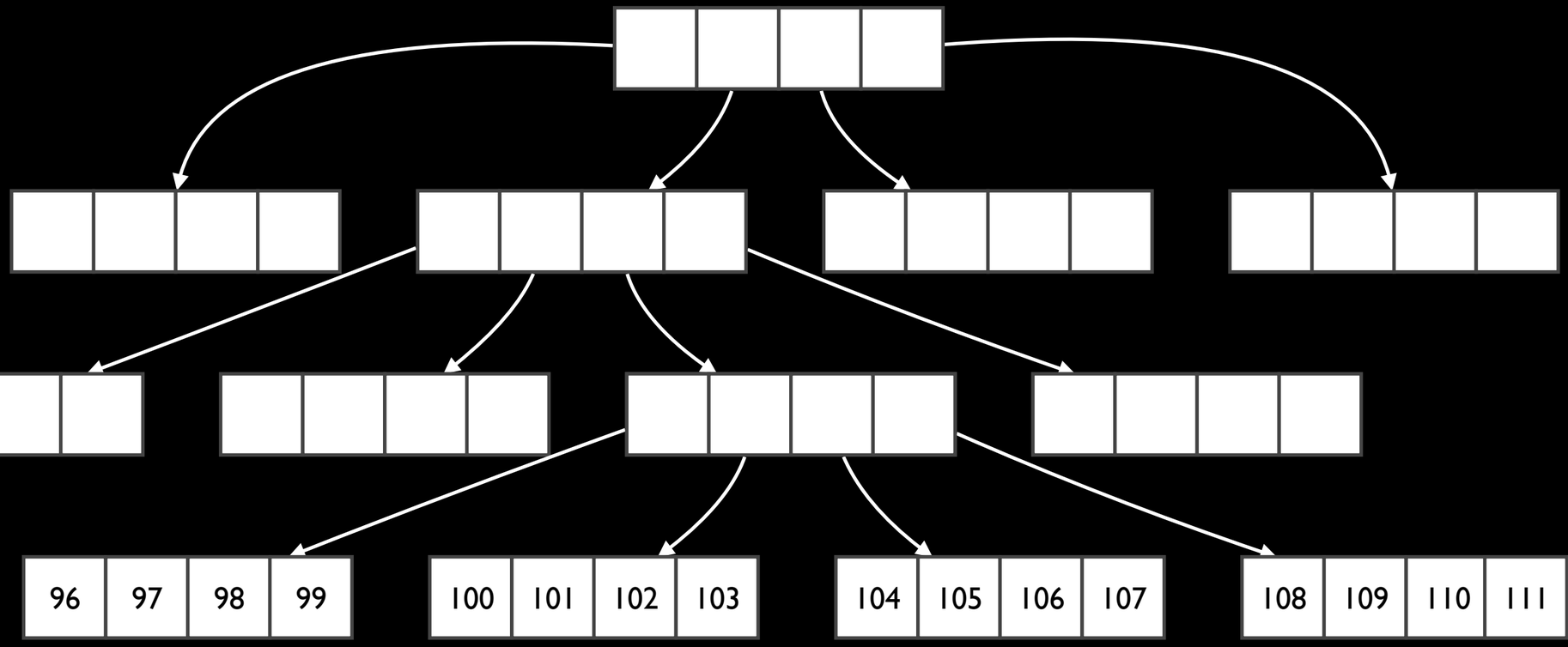
Persistent Vector



Persistent Vector



Persistent Vector



Persistent Vector

```
abstract BitmappedTrie

immutable ArrayNode <: BitmappedTrie
    arr::Vector{BitmappedTrie}
    shift::Int
    length::Int
    maxlength::Int
end

immutable ArrayLeaf <: BitmappedTrie
    arr::Vector
end
```

Persistent Vector

```
abstract BitmappedTrie

immutable ArrayNode <: BitmappedTrie
    arr::Vector{BitmappedTrie}
    shift::Int
    length::Int
    maxlength::Int
end

immutable ArrayLeaf <: BitmappedTrie
    arr::Vector
end
```

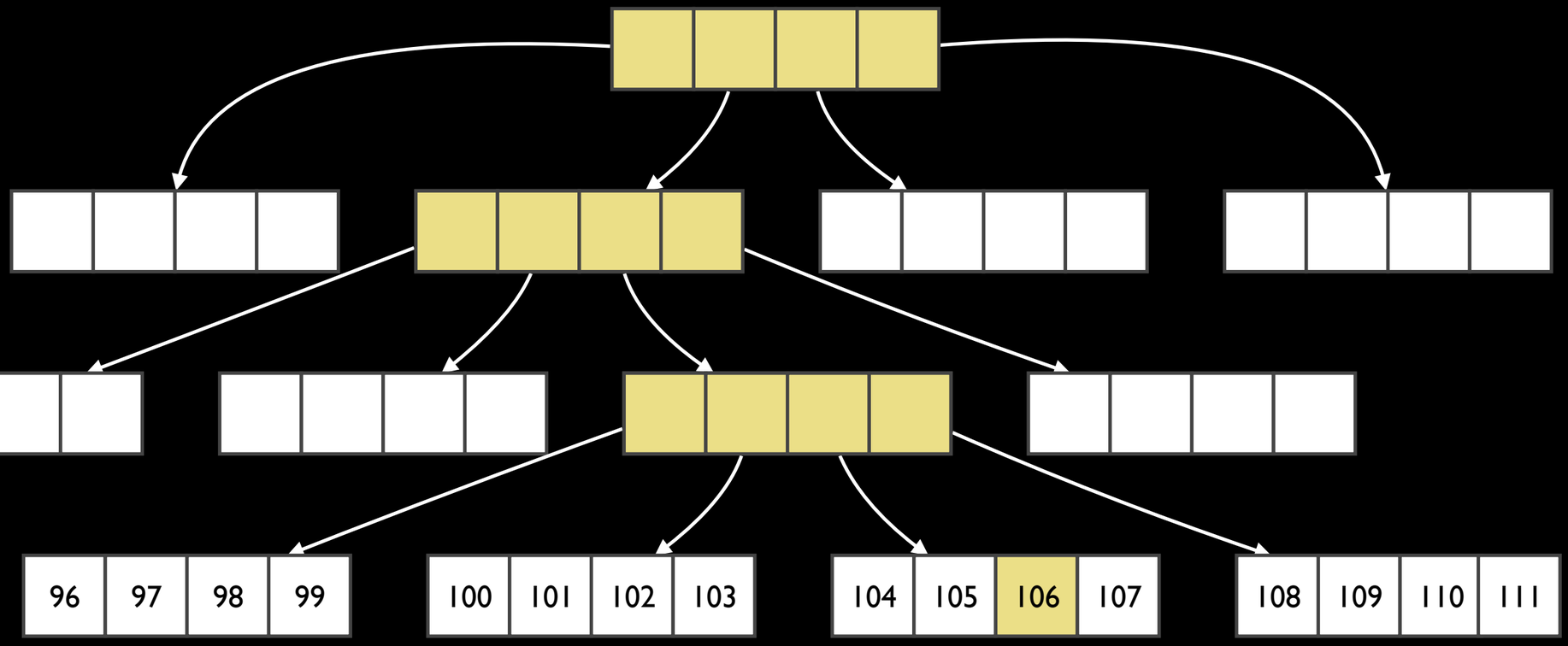
Persistent Vector

```
abstract BitmappedTrie

immutable ArrayNode <: BitmappedTrie
    arr::Vector{BitmappedTrie}
    shift::Int
    length::Int
    maxlength::Int
end

immutable ArrayLeaf <: BitmappedTrie
    arr::Vector
end
```

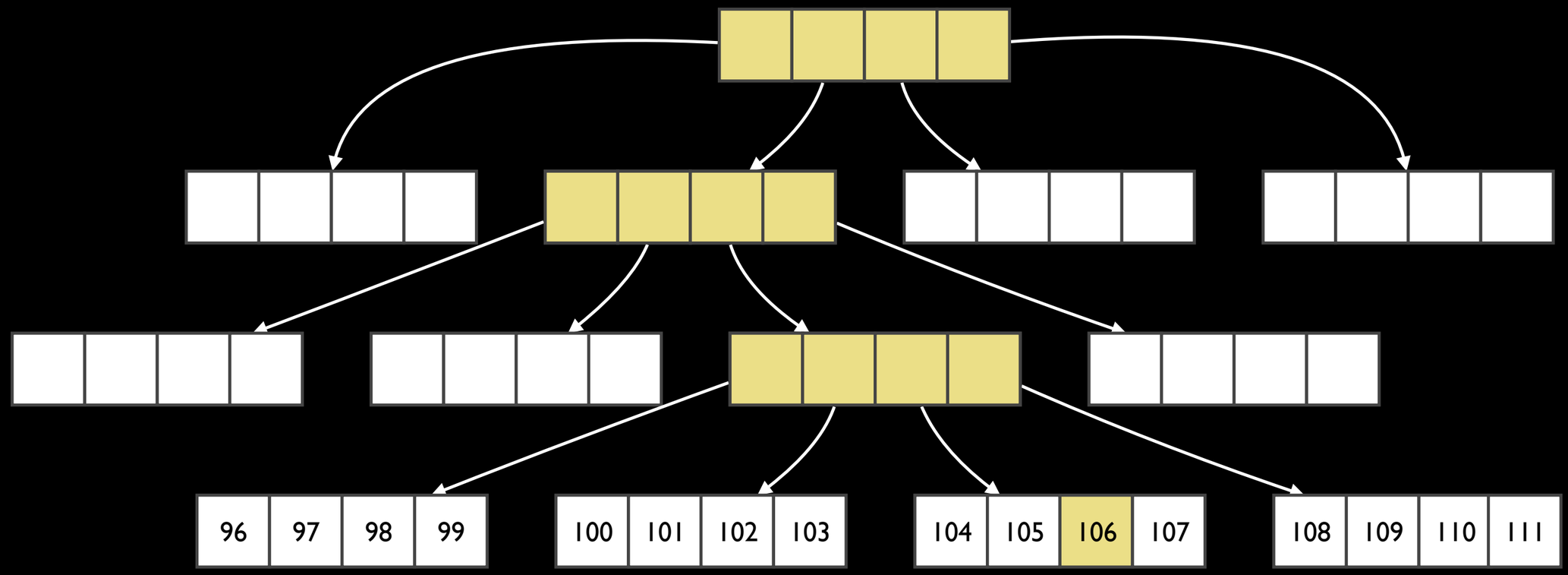
Persistent Vector

```
getindex
```

Persistent Vector

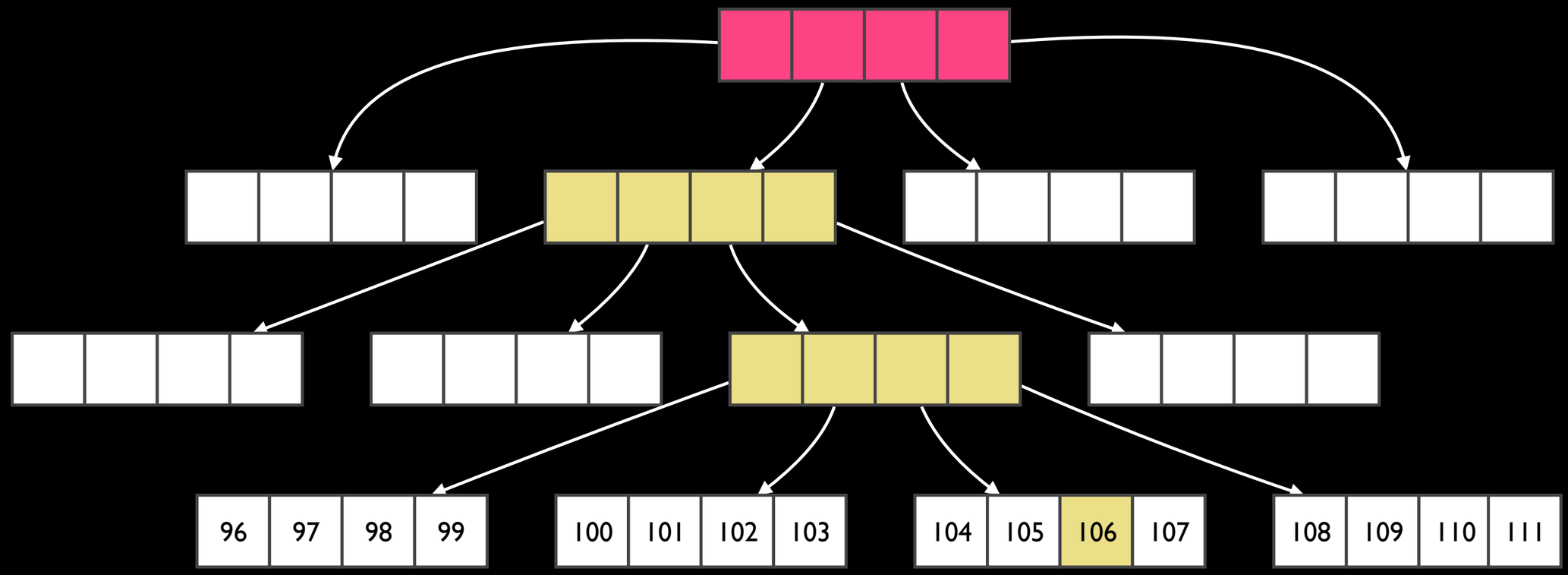


Persistent Vector



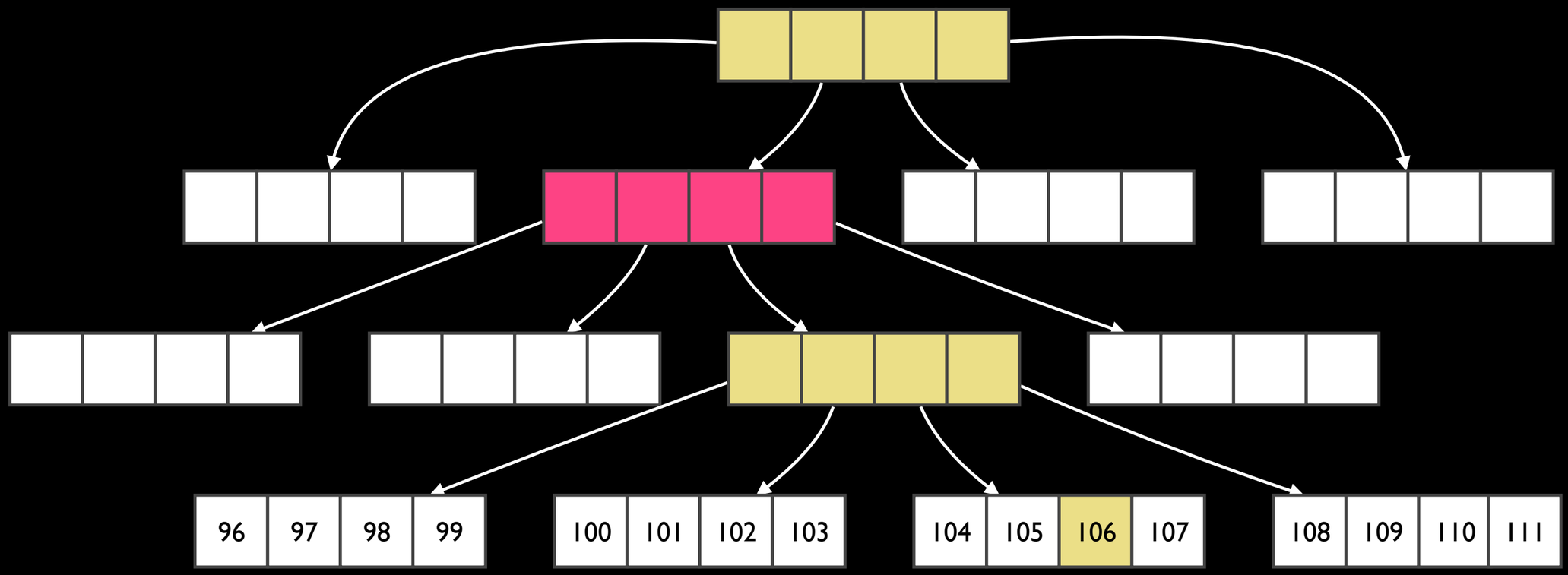
0b01101010

Persistent Vector



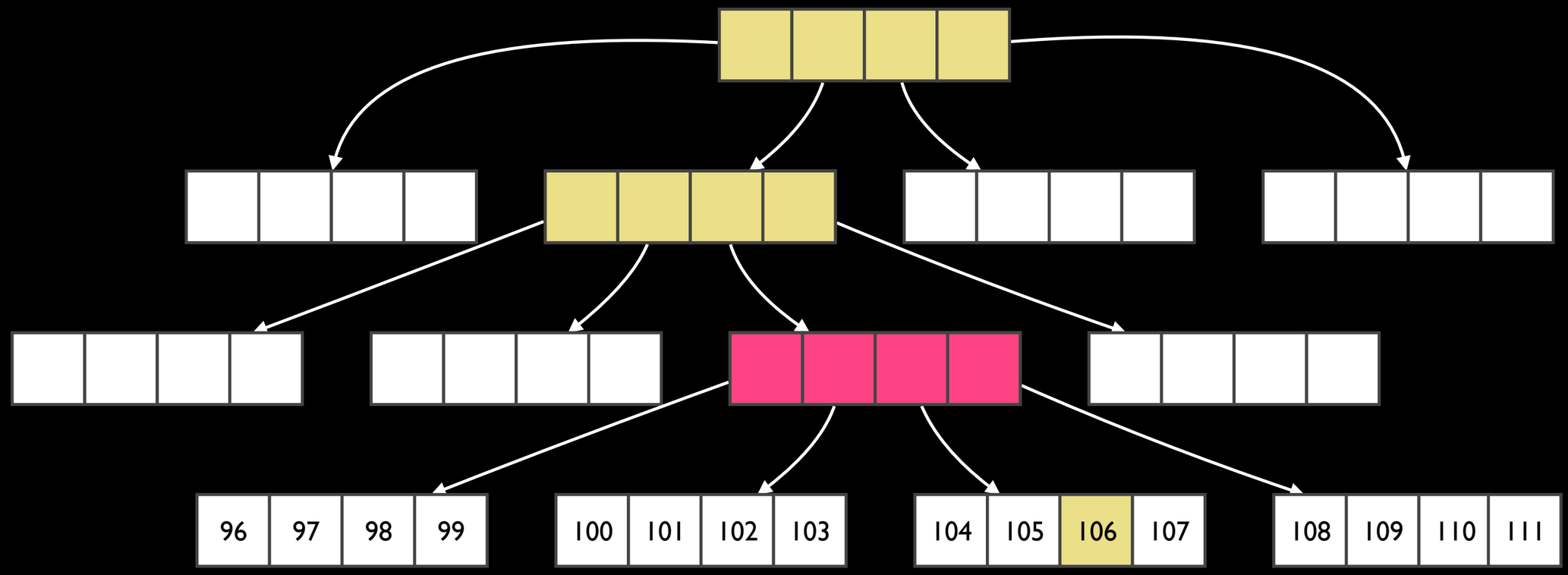
0b01101010

Persistent Vector



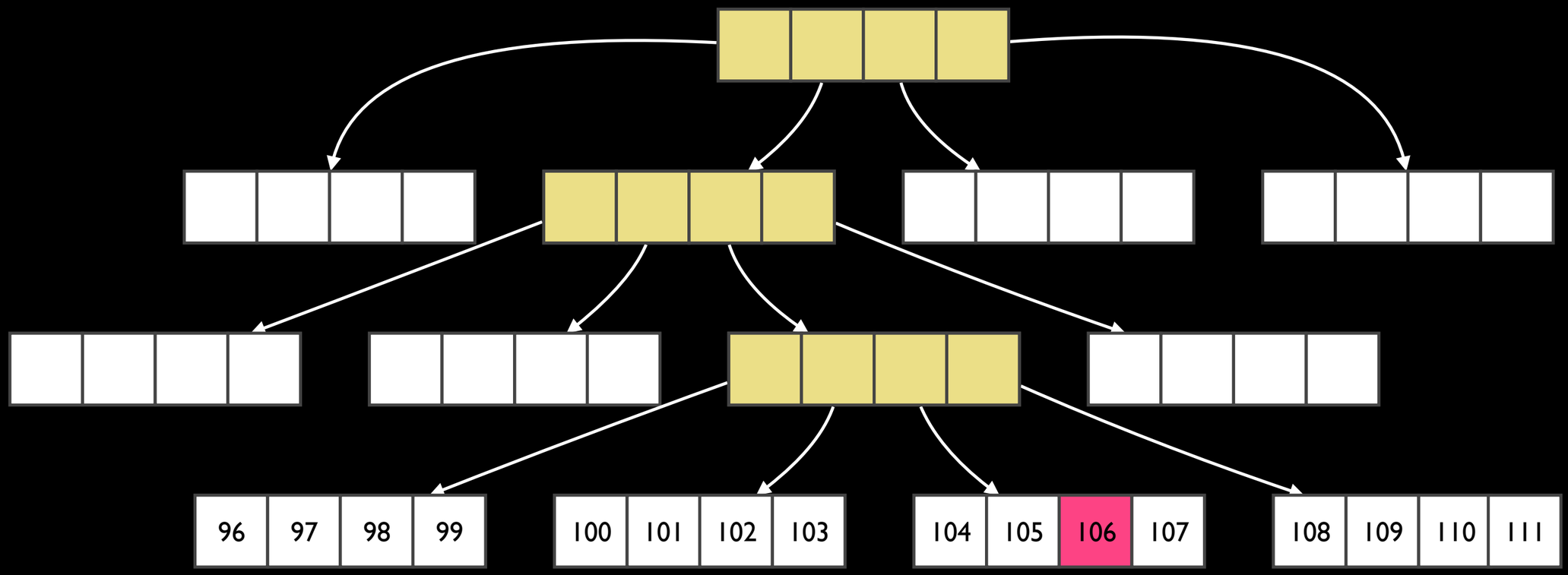
0b01101010

Persistent Vector



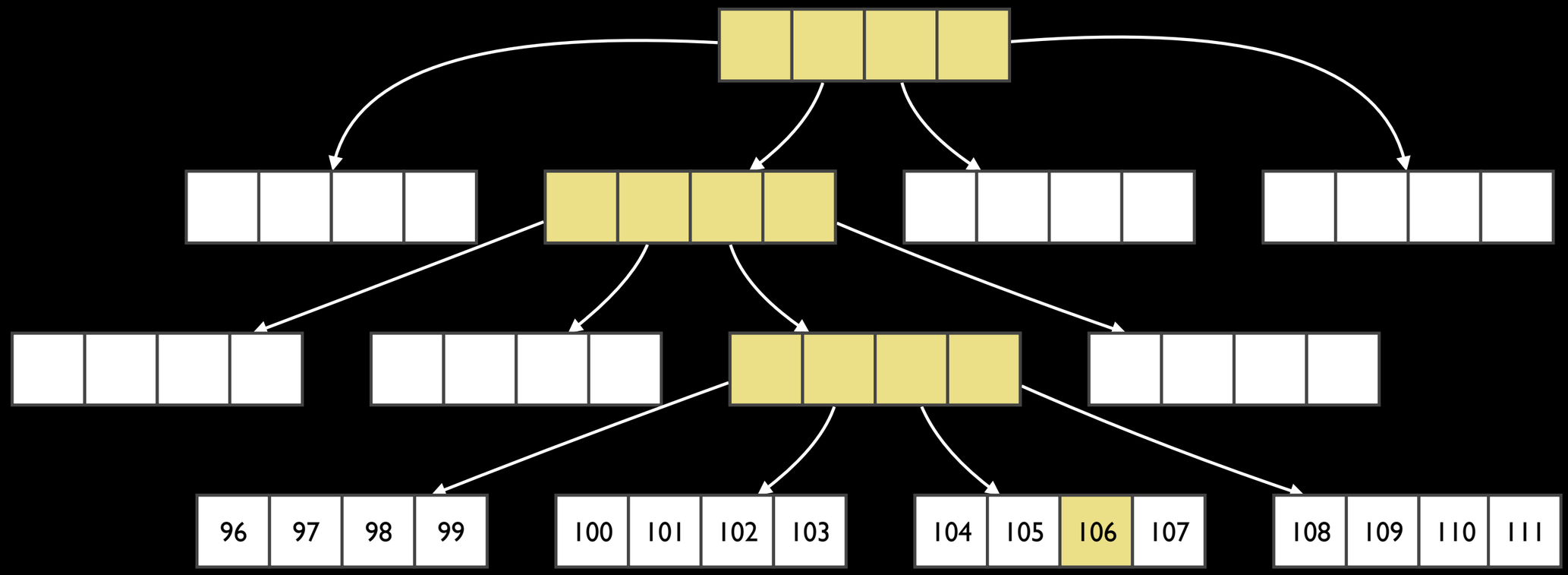
0b01101010

Persistent Vector



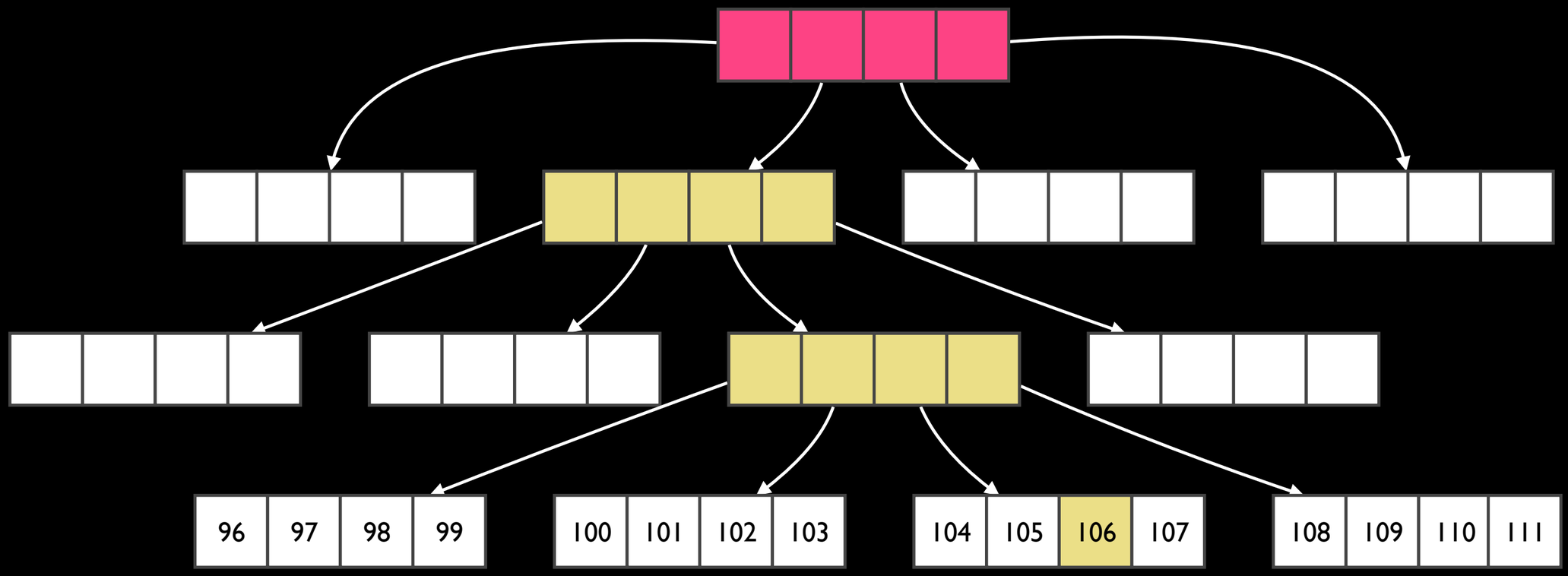
0b01101010

Persistent Vector



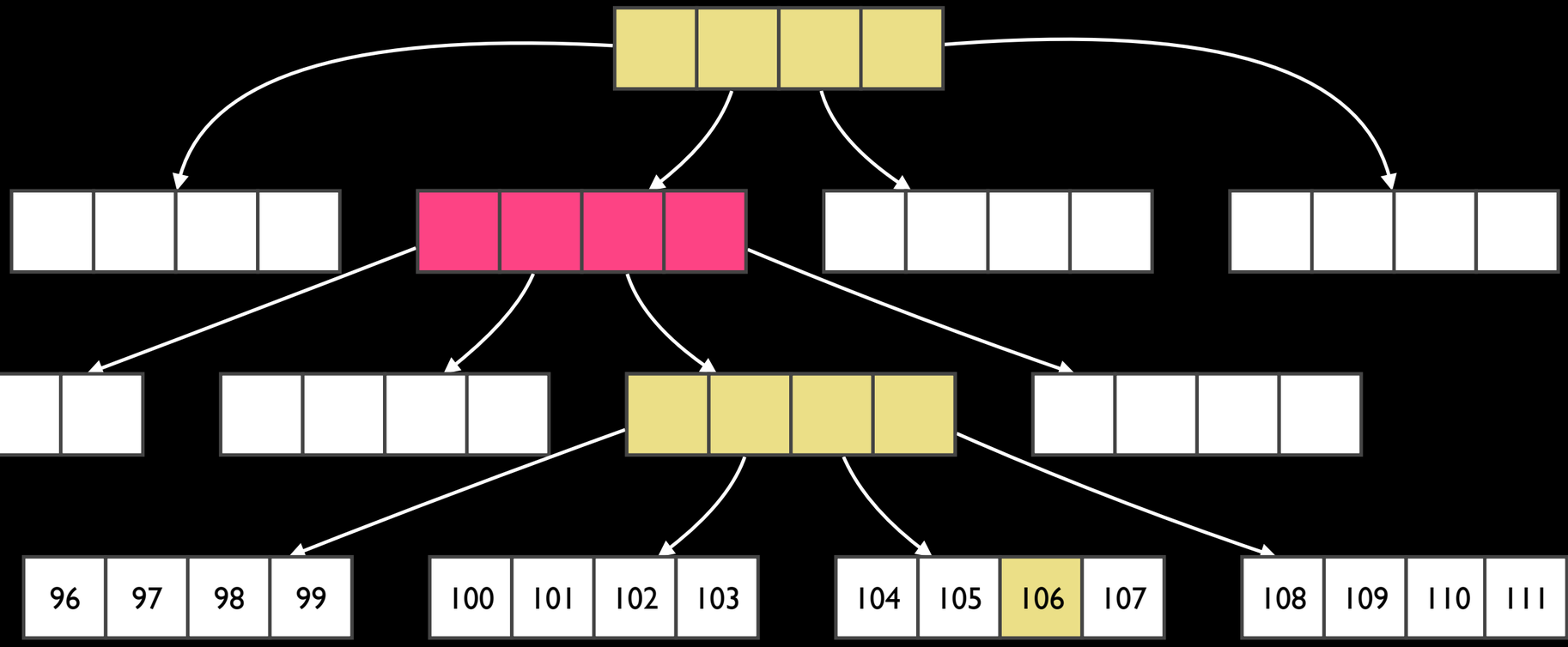

```
n = 0b01101010
```

Persistent Vector



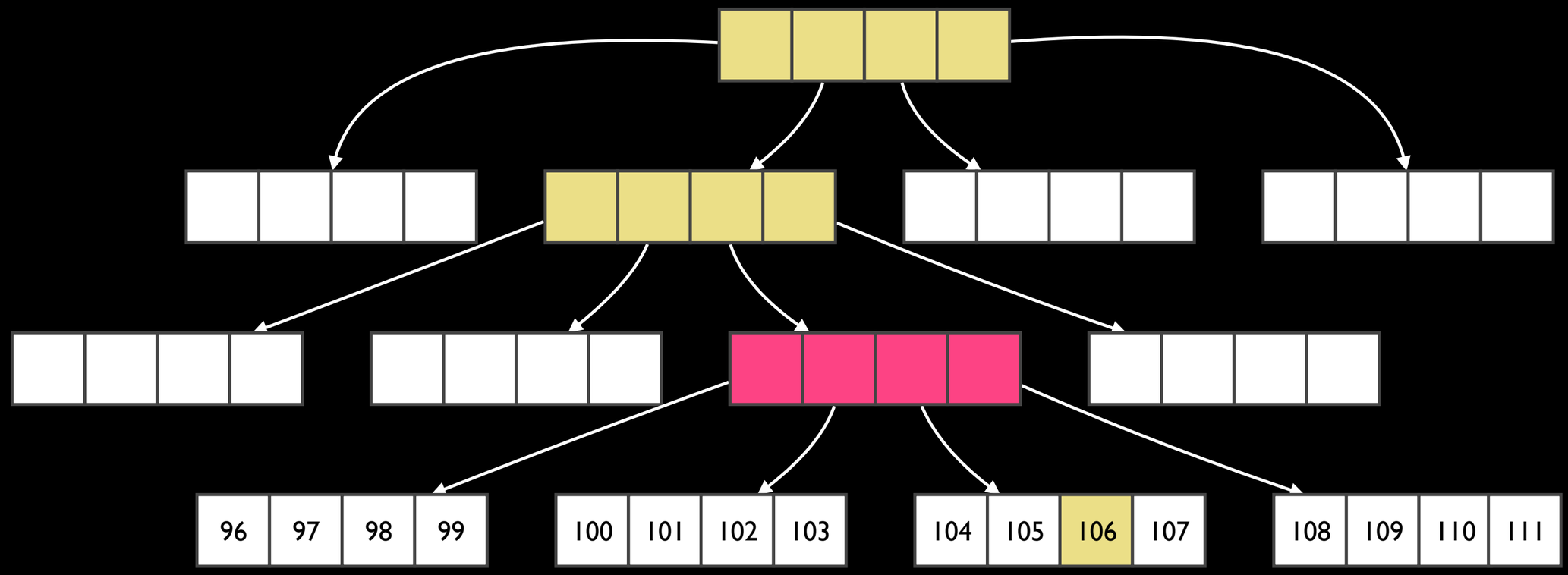

```
n = 0b01101010
(n >> 6) & 0b11
```

Persistent Vector



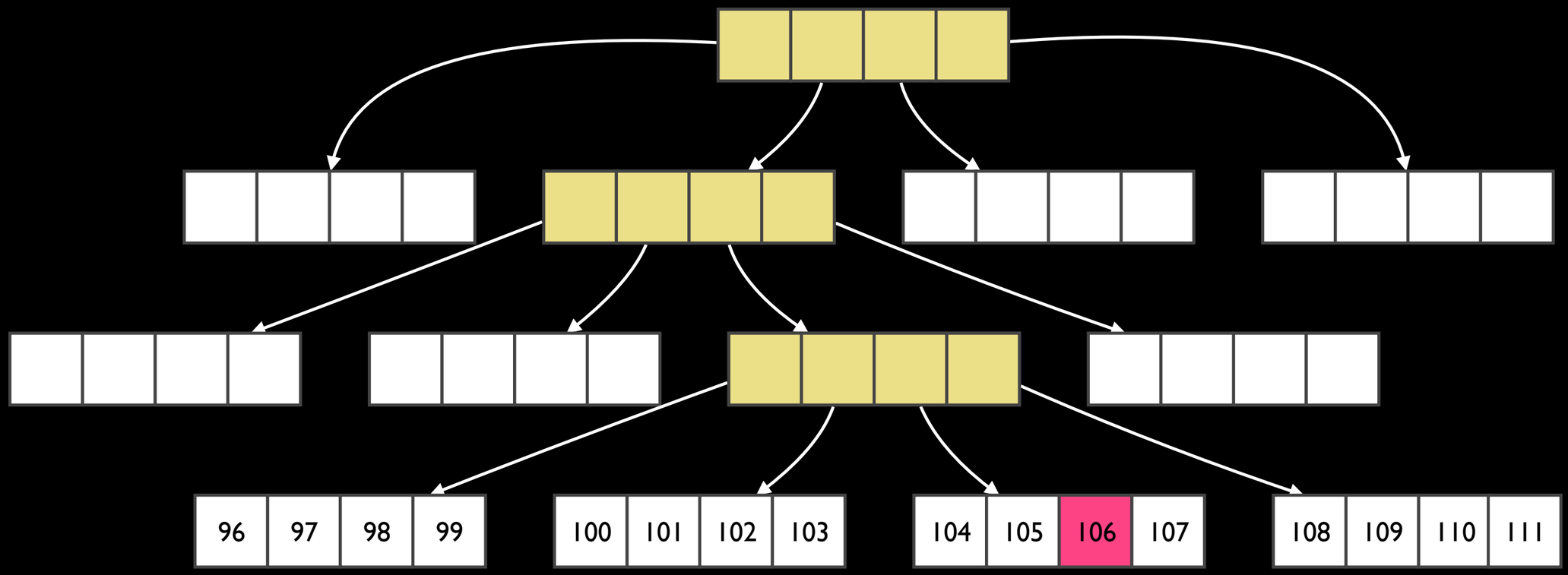

```
n = 0b01101010
(n >> 4) & 0b11
```

Persistent Vector




```
n = 0b01101010
(n >> 2) & 0b11
```

Persistent Vector




```
n = 0b01101010
(n >> 0) & 0b11
```

Persistent Vector

```
shiftval(n::ArrayNode) = n.shift
shiftval(l::ArrayLeaf) = 0

function mask(t::BitmappedTrie, i)
    i -= 1 # remember: Julia is indexed from 1
    view = (i >> shiftval(t)) & 3
    view + 1
end
```

Persistent Vector

```
shiftval(n::ArrayNode) = n.shift
shiftval(l::ArrayLeaf) = 0

function mask(t::BitmappedTrie, i)
    i -= 1 # remember: Julia is indexed from 1
    view = (i >> shiftval(t)) & 3
    view + 1
end
```

Persistent Vector

```
function Base.getindex(leaf::ArrayLeaf, i)
    leaf.arr[mask(leaf, i)]
end

function Base.getindex(node::ArrayNode, i)
    node.arr[mask(node, i)][i]
end
```

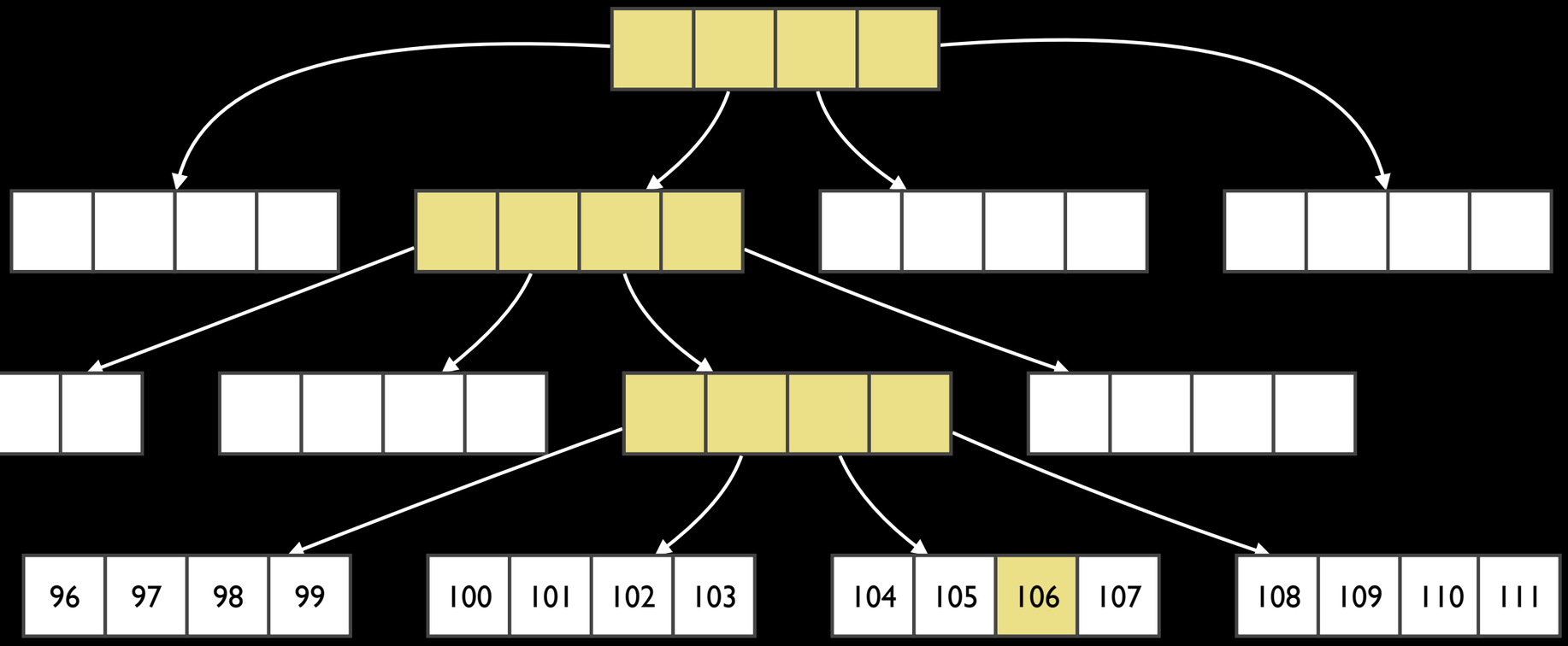
Persistent Vector

```
function Base.getindex(leaf::ArrayLeaf, i)
    leaf.arr[mask(leaf, i)]
end

function Base.getindex(node::ArrayNode, i)
    node.arr[mask(node, i)][i]
end
```

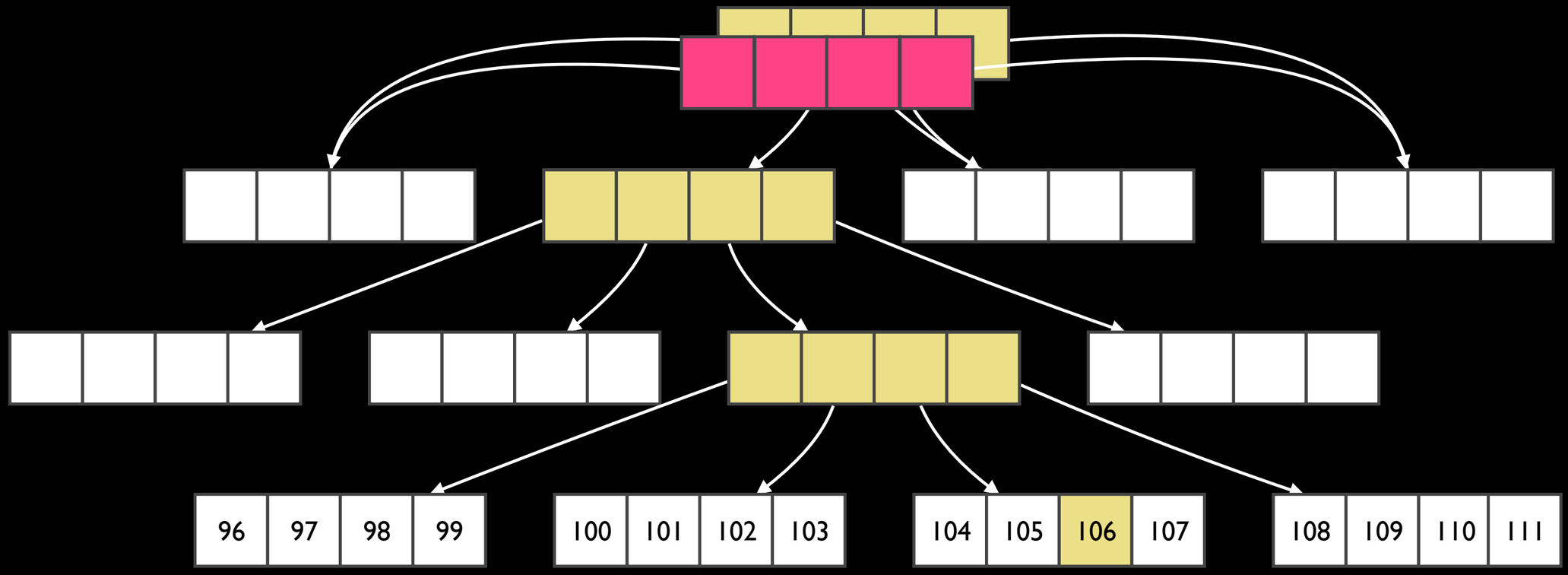
Persistent Vector

assoc

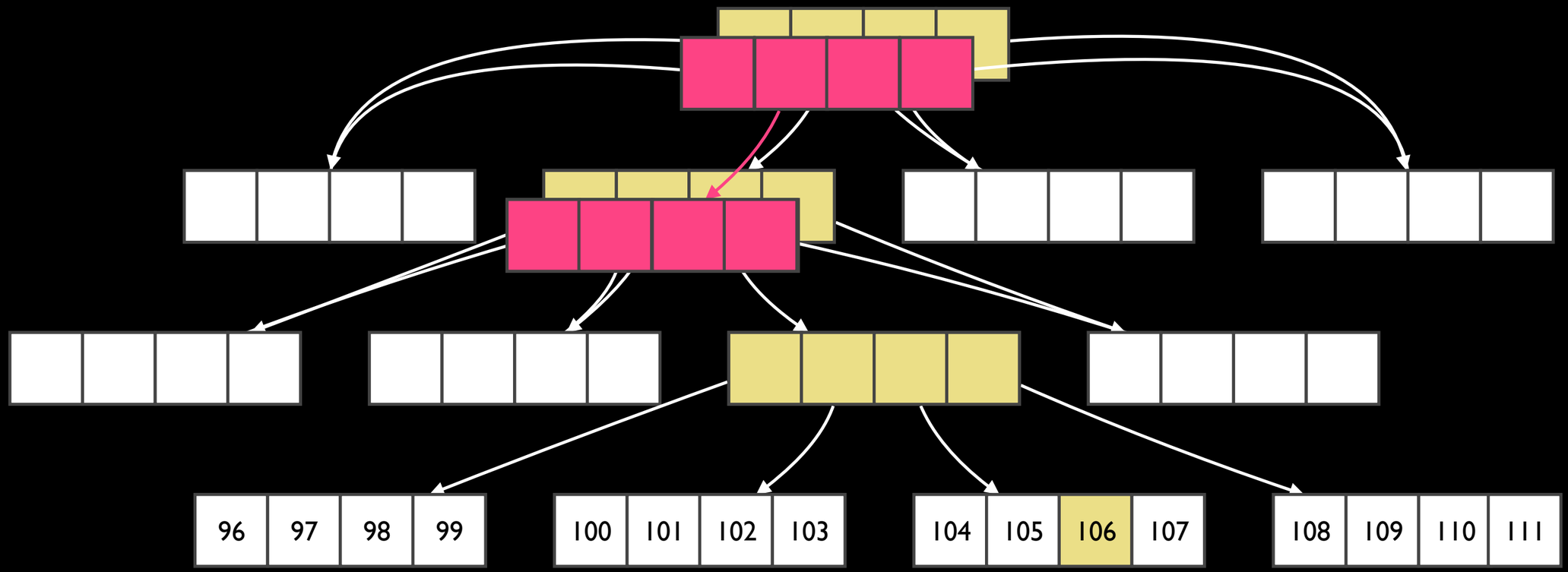
Persistent Vector



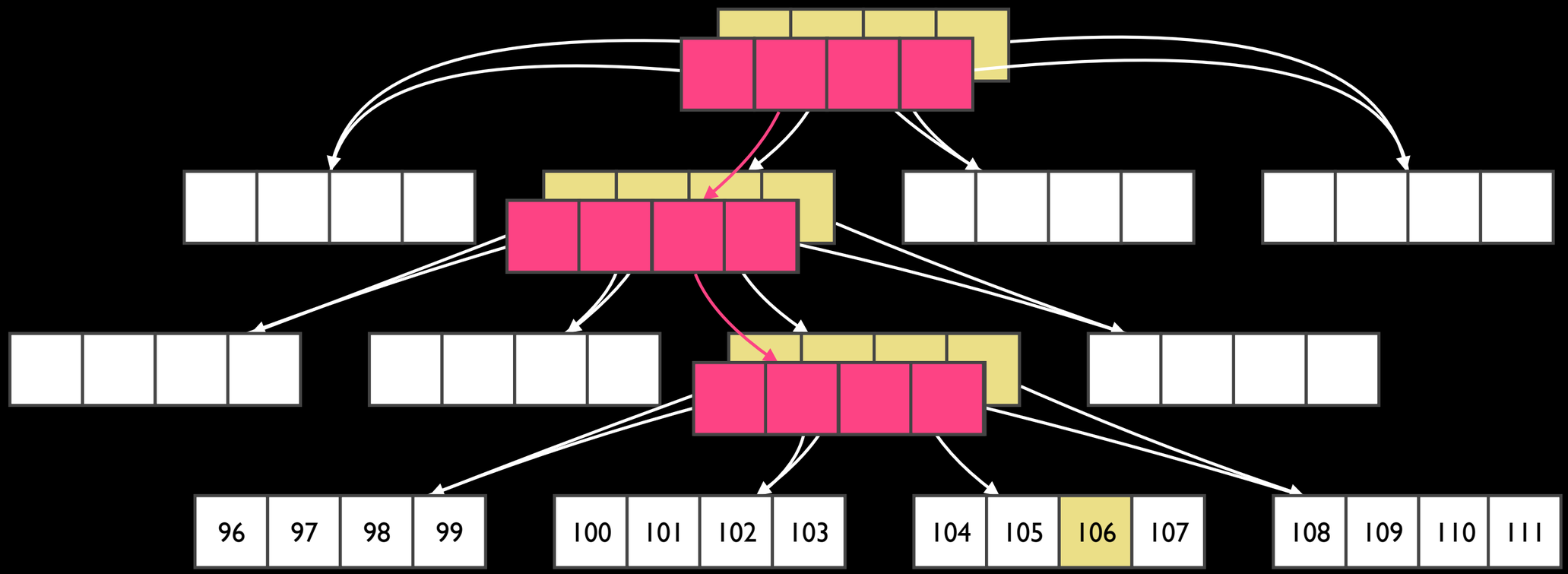
Persistent Vector



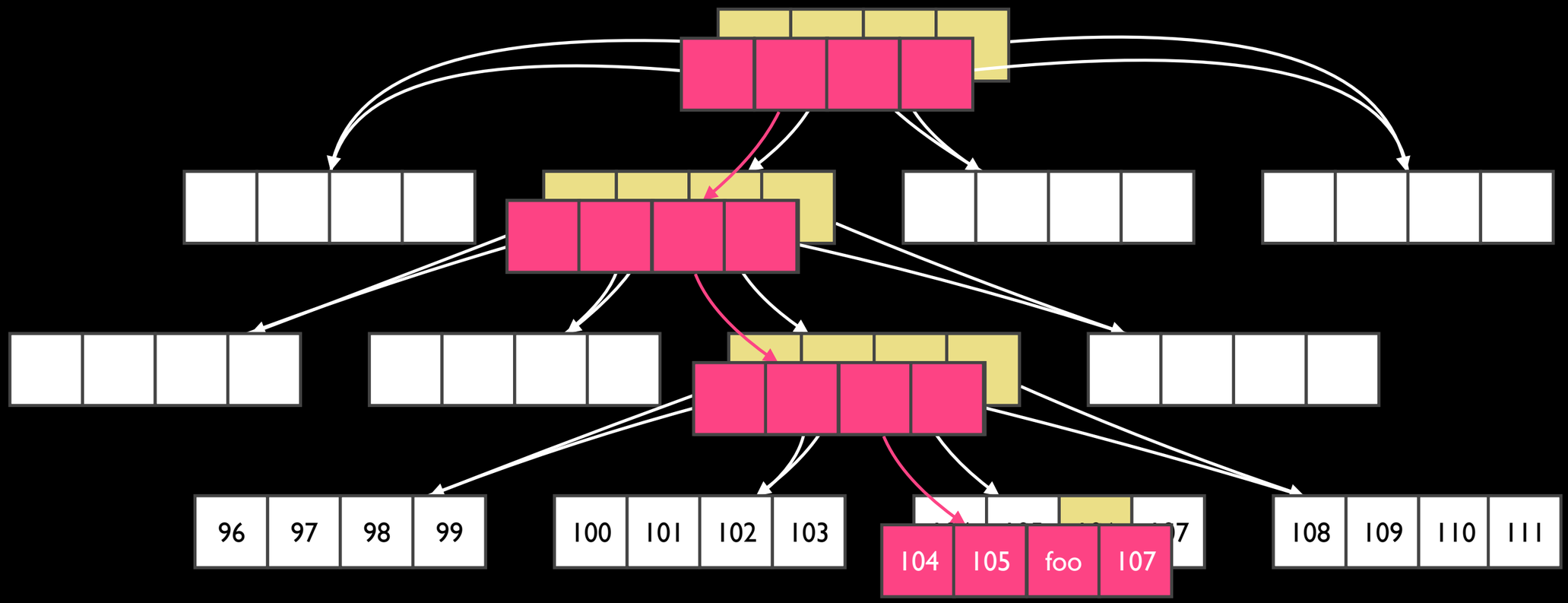
Persistent Vector



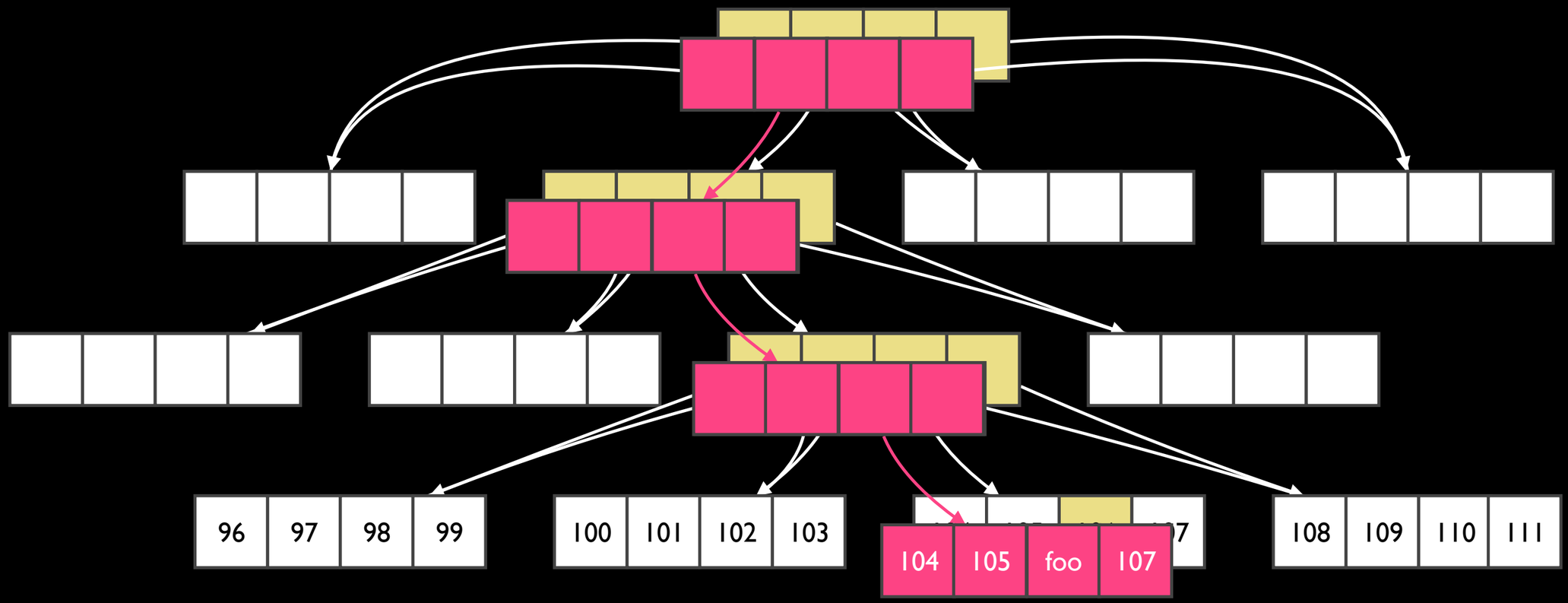
Persistent Vector



Persistent Vector



Persistent Vector



Functional data structures are made with arrows

Persistent Vector

```
function assoc(leaf::ArrayLeaf, i, el)
    newarr = leaf.arr[:]
    newarr[mask(leaf, i)] = el
    ArrayLeaf(newarr)
end
```

Persistent Vector

```
function assoc(node::ArrayNode, i, el)
    newarr = node.arr[:]
    idx = mask(node, i)
    newarr[idx] = assoc(newarr[idx], i, el)
    ArrayNode(newarr,
              node.shift,
              node.length,
              node.maxlength)
end
```

Persistent Vector

push

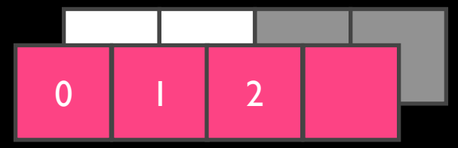
Persistent Vector

push

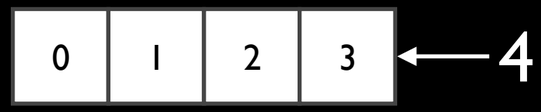
- ArrayLeaf
  - full or not
- ArrayNode
  - last child has room, last child is full, ArrayNode is full

Persistent Vector

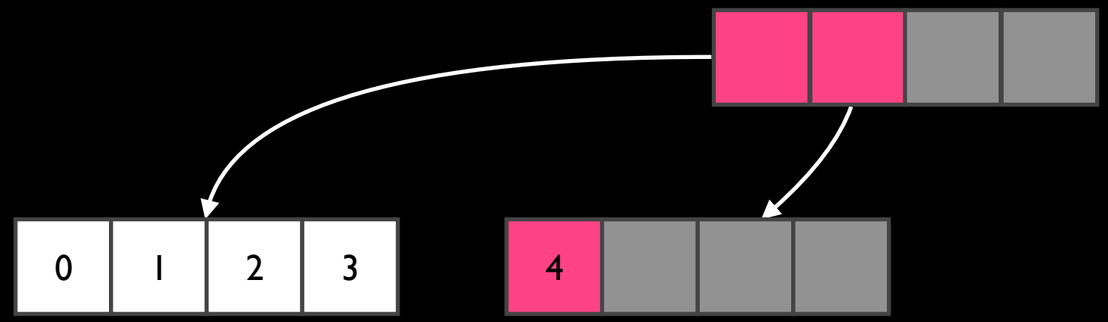
Persistent Vector



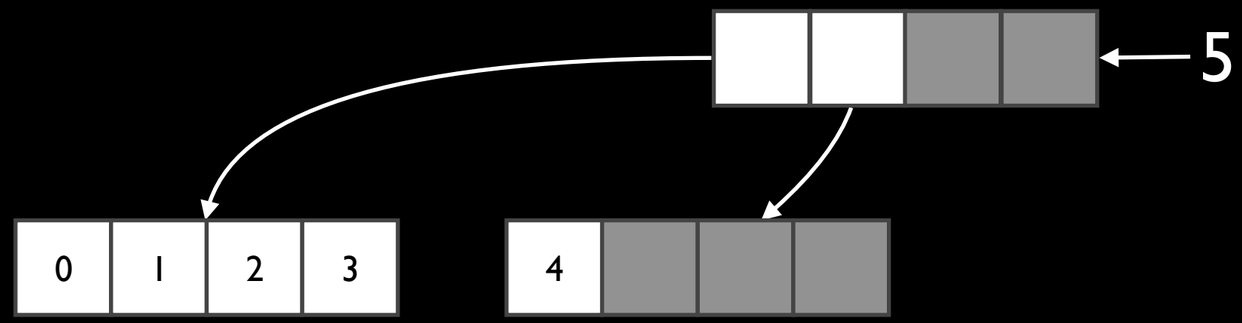
Persistent Vector



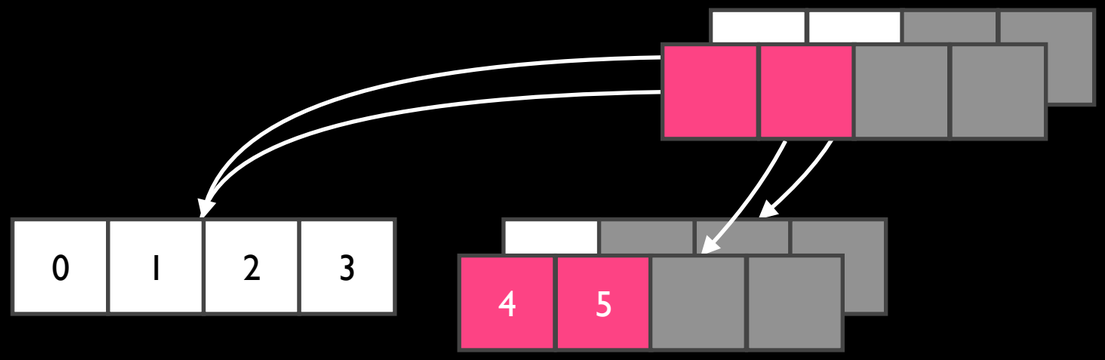
Persistent Vector



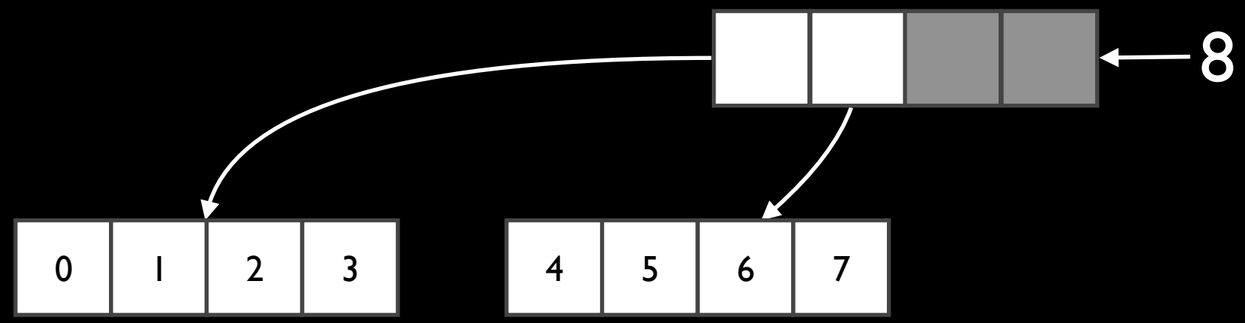
Persistent Vector



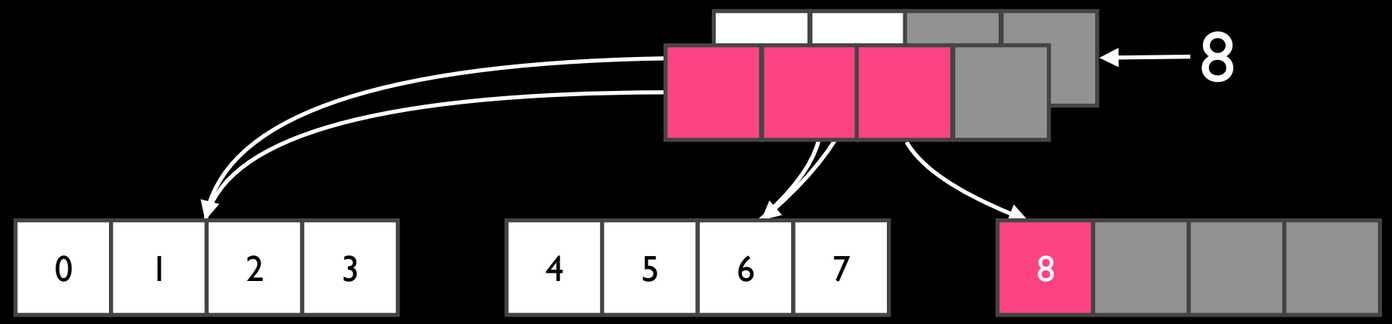
Persistent Vector



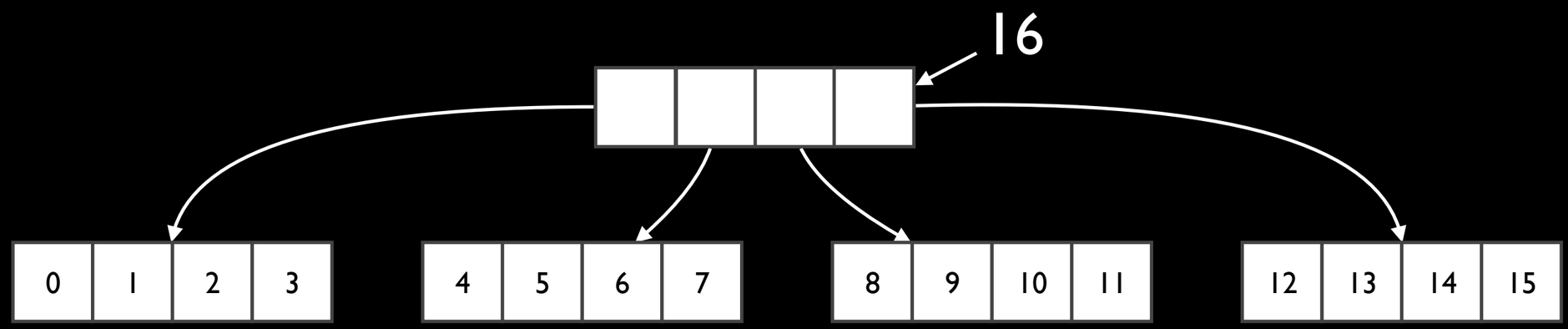
Persistent Vector



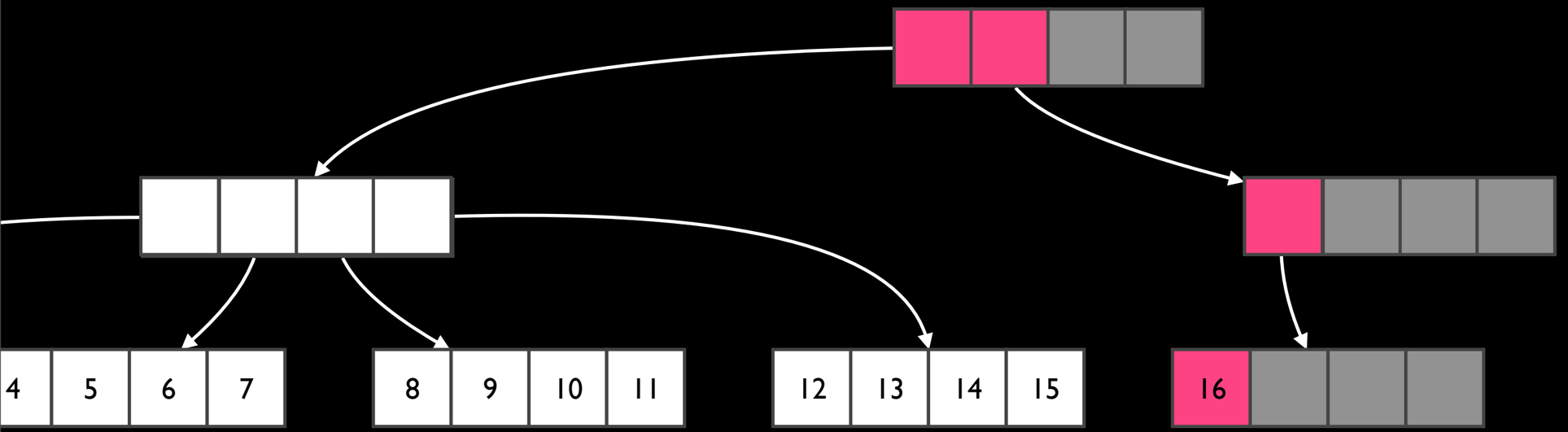
Persistent Vector



Persistent Vector



Persistent Vector



Persistent Vector

</code>

Persistent Vector

Where is the `PersistentVector` type?

Persistent Vector

```
immutable PersistentVector
    trie::BitmappedTrie
    tail::Vector
    length::Int
end
```

Persistent Vector

- In examples, we appended our data to the trie
- What if we append full vectors?
- Very fast access to the tail
- Only update the trie when the tail is full

Persistent Vector

Length 4 internal vectors?

Persistent Vector

32

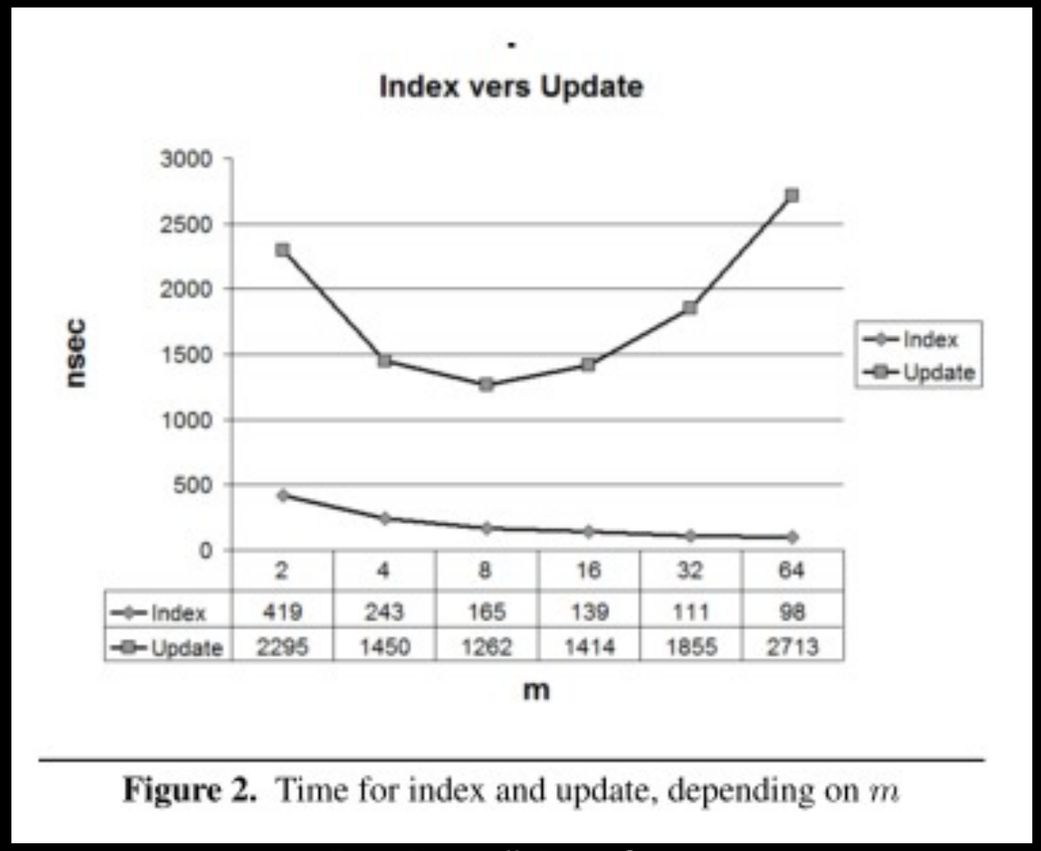


Figure 2. Time for index and update, depending on m

From Bagwell, Rompf 2011

Persistent Vector Further Reading

RRB-Trees: Efficient Immutable Vectors

Bagwell and Rompf, 2011

Mutable Hash Map

- Built on Array
- Start at a fixed size, insert elements based on some hash
- Resize tables when 2/3 full to avoid collisions

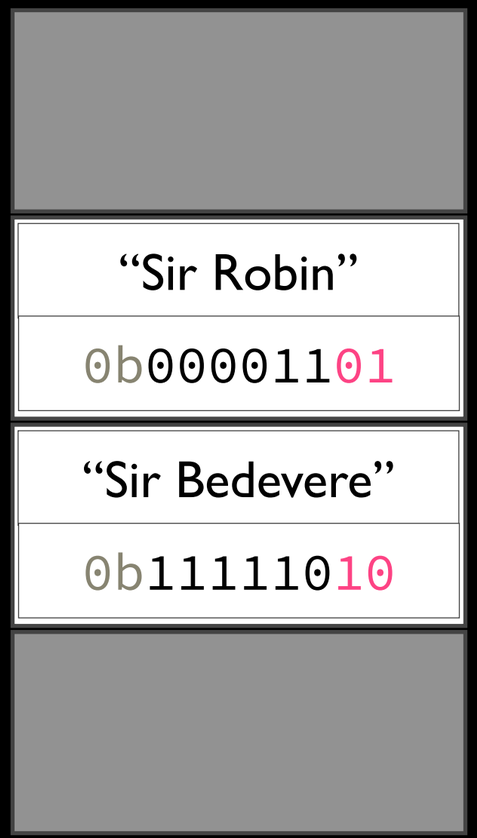
Persistent Hash Map

- Handle collisions
- Avoid resizing

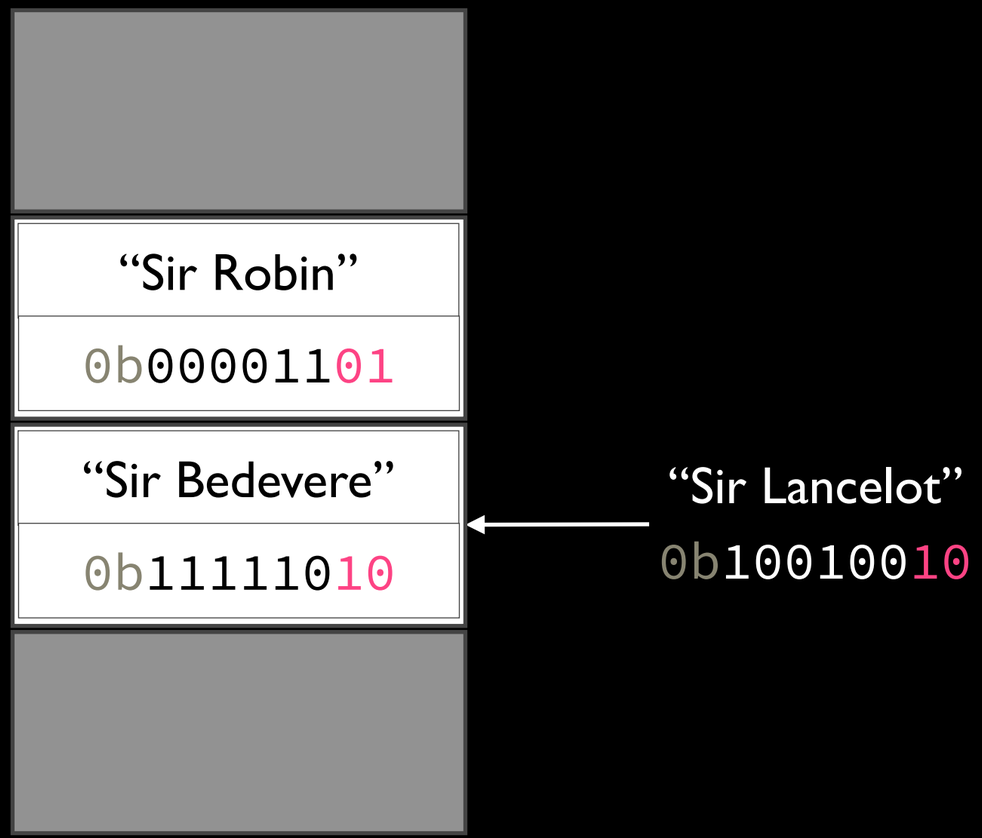
Persistent Hash Map

assoc

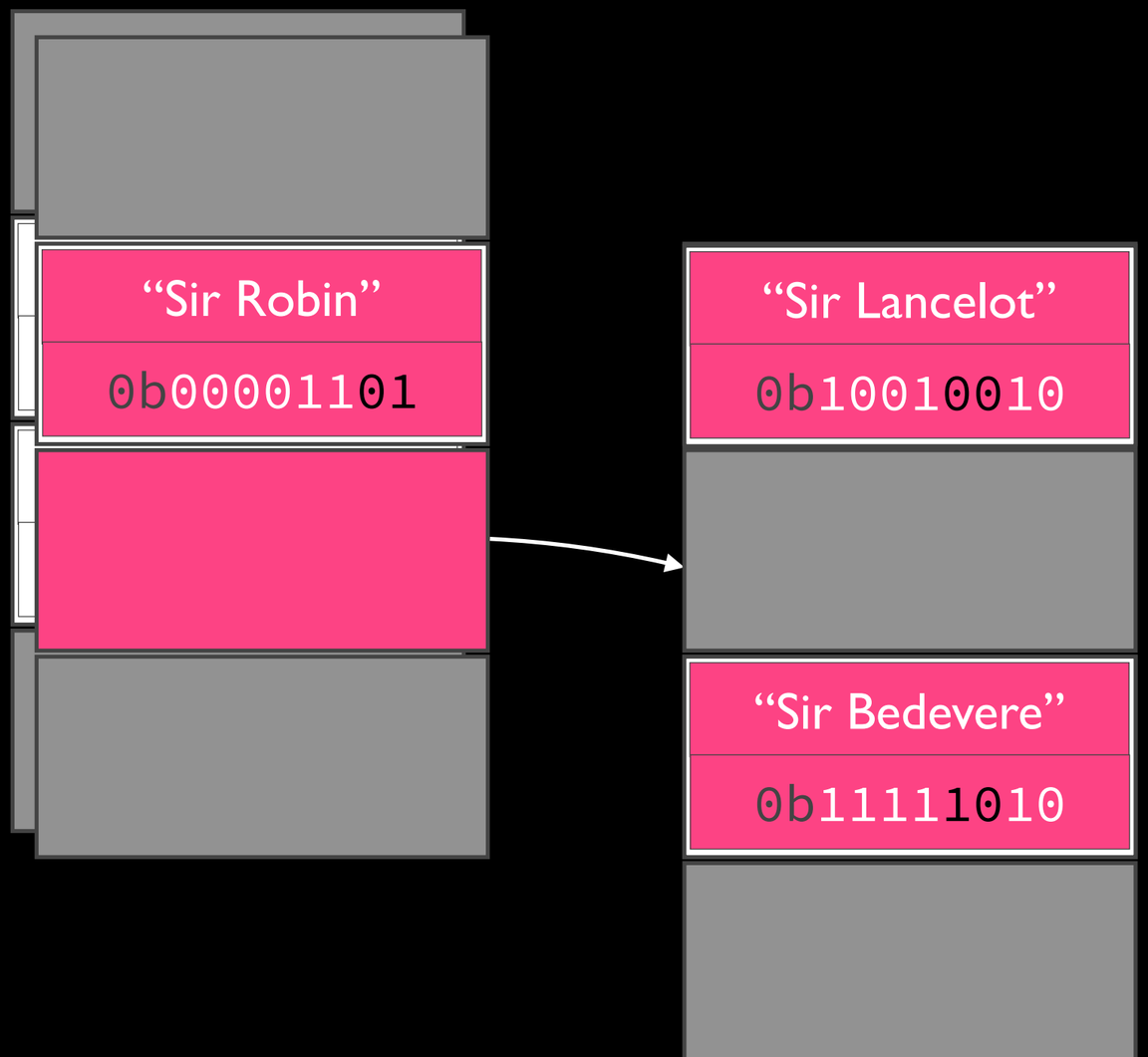
Persistent Hash Map



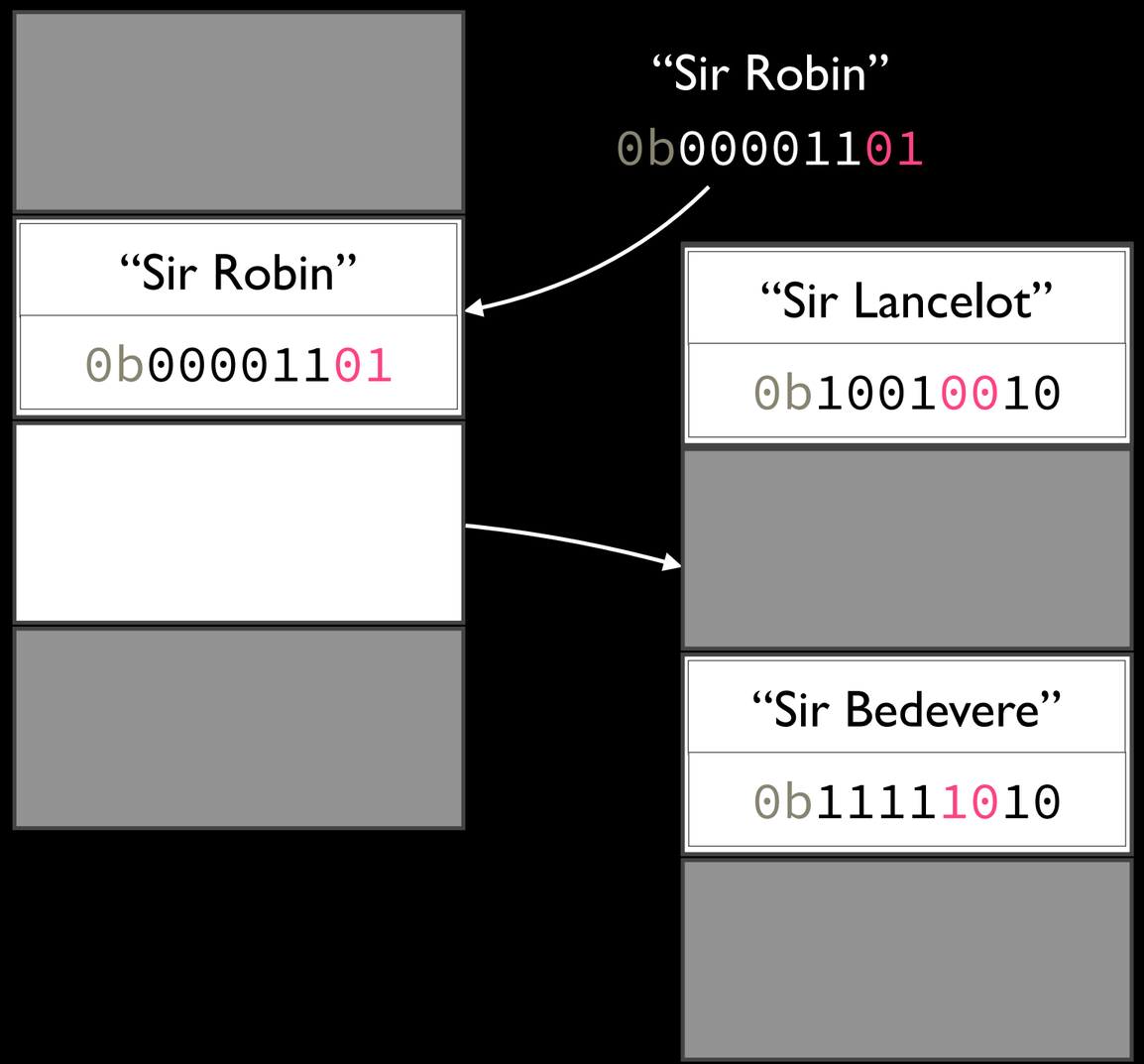
Persistent Hash Map



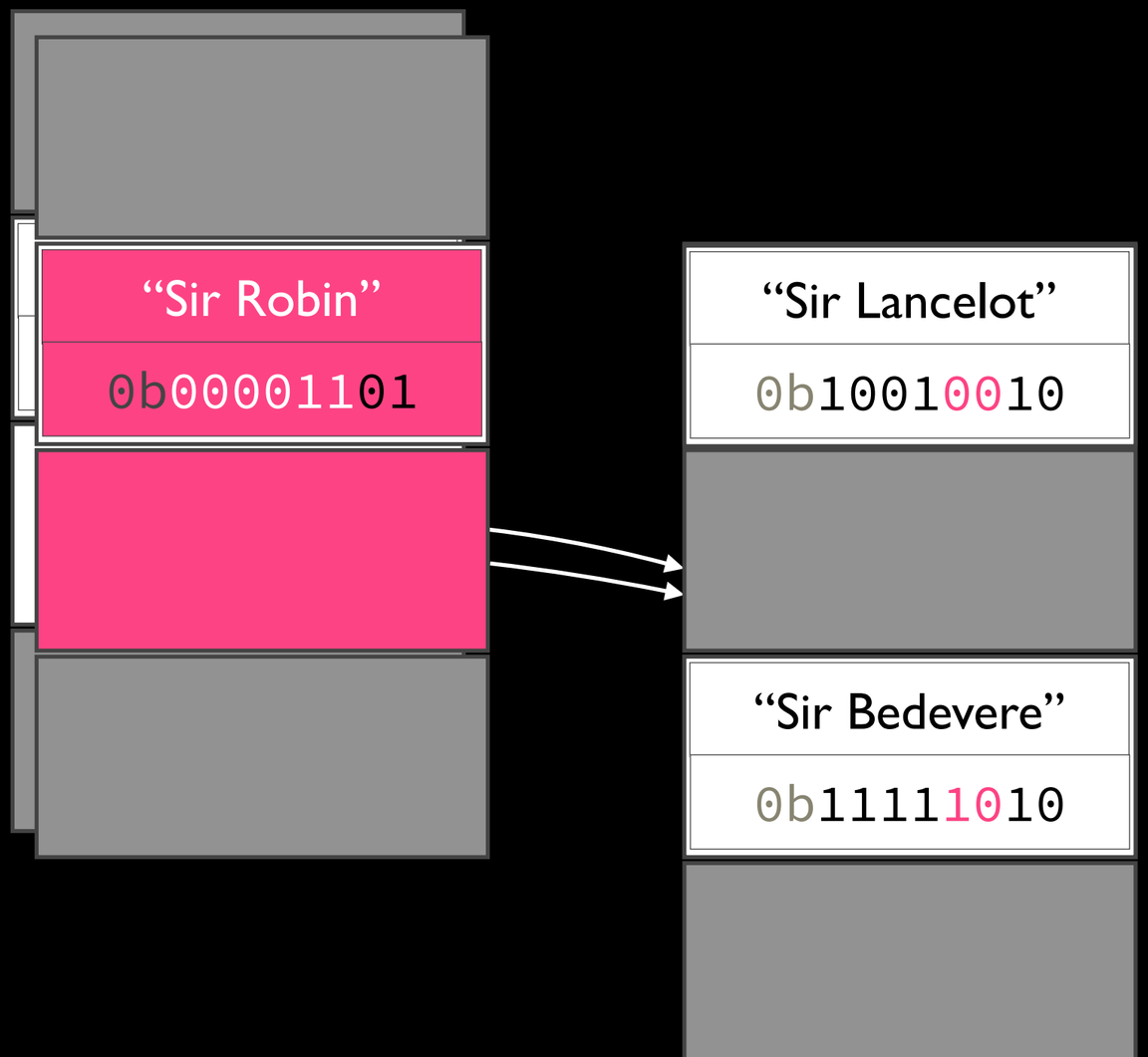
Persistent Hash Map



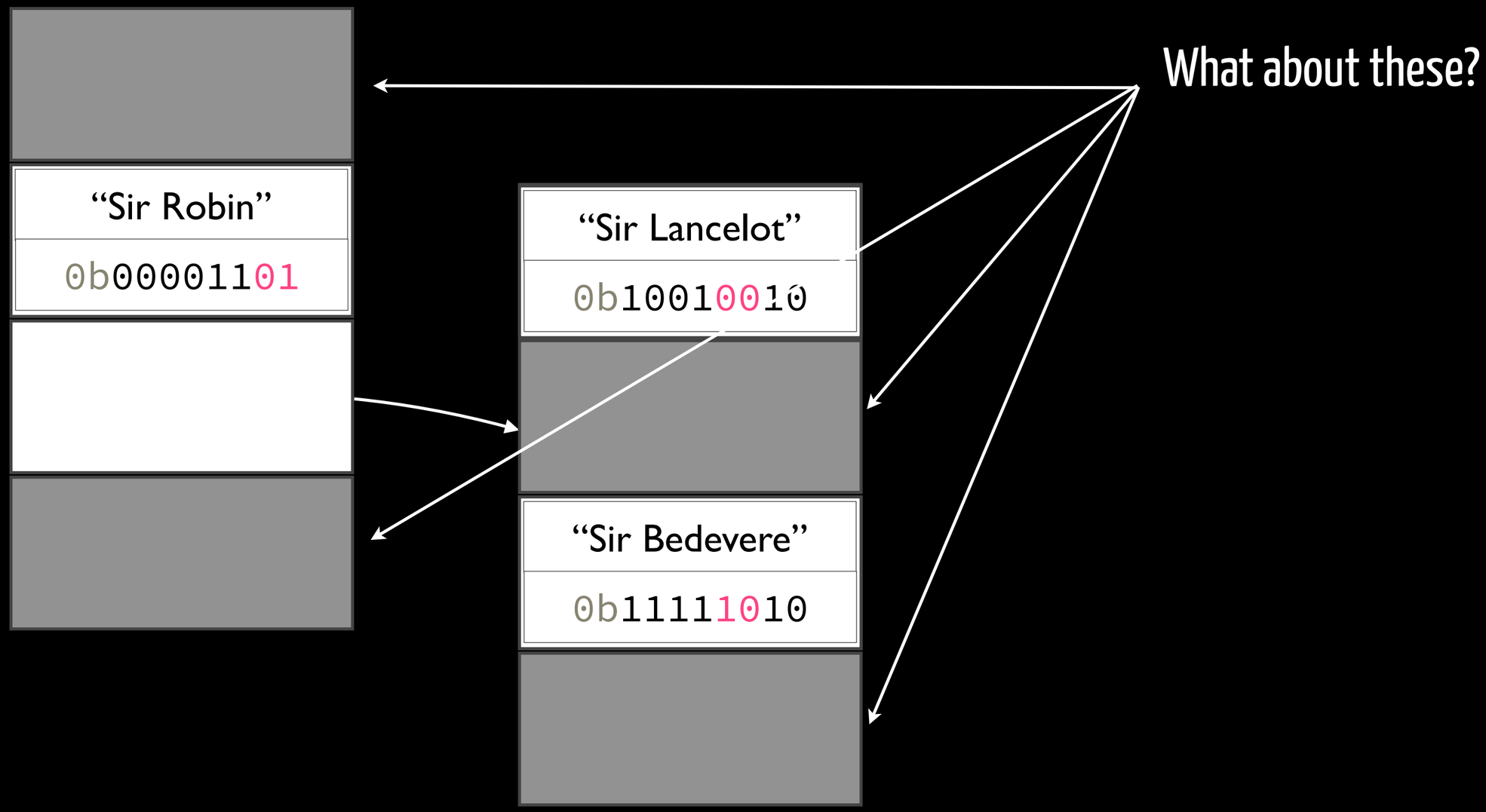
Persistent Hash Map



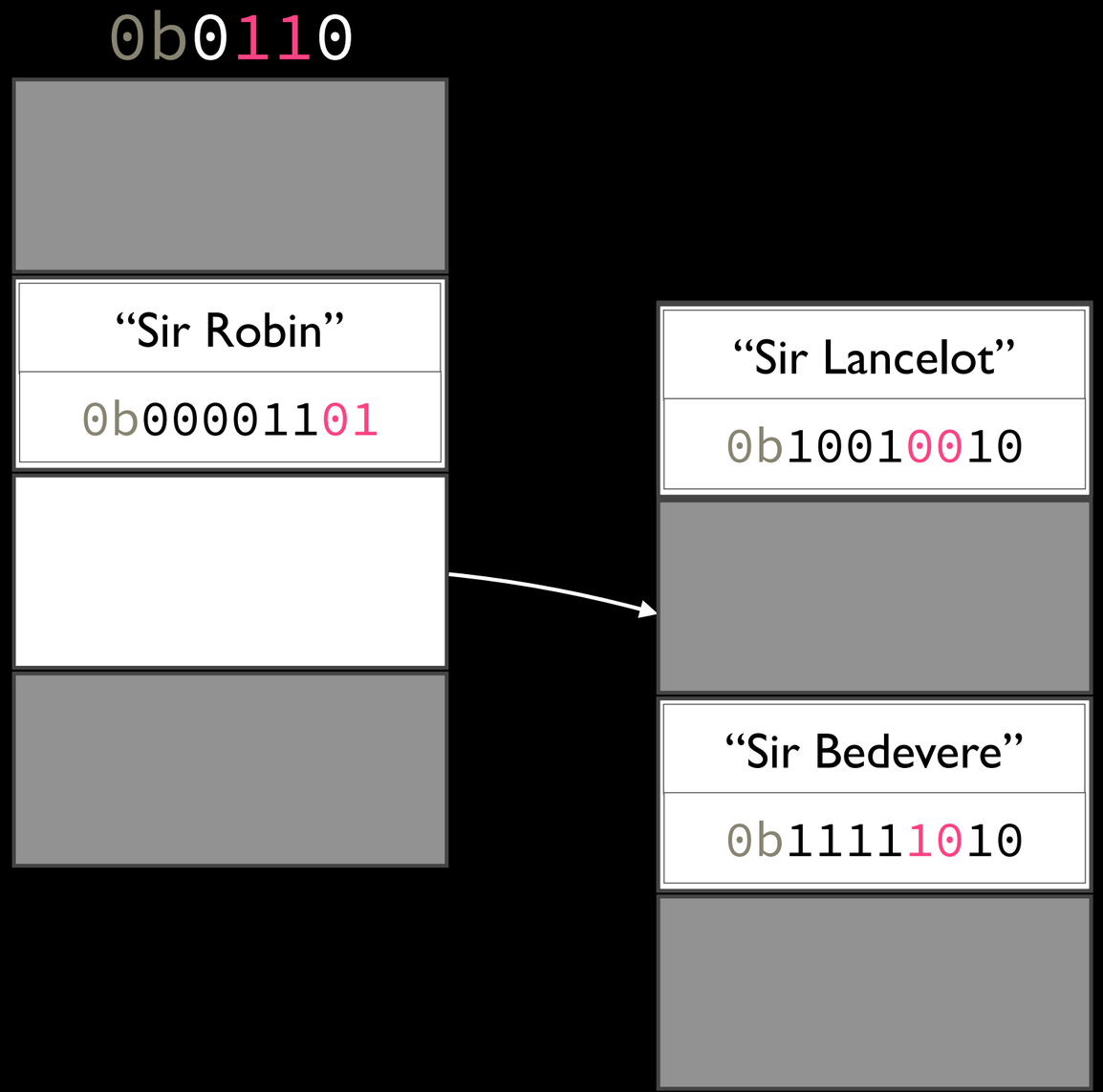
Persistent Hash Map



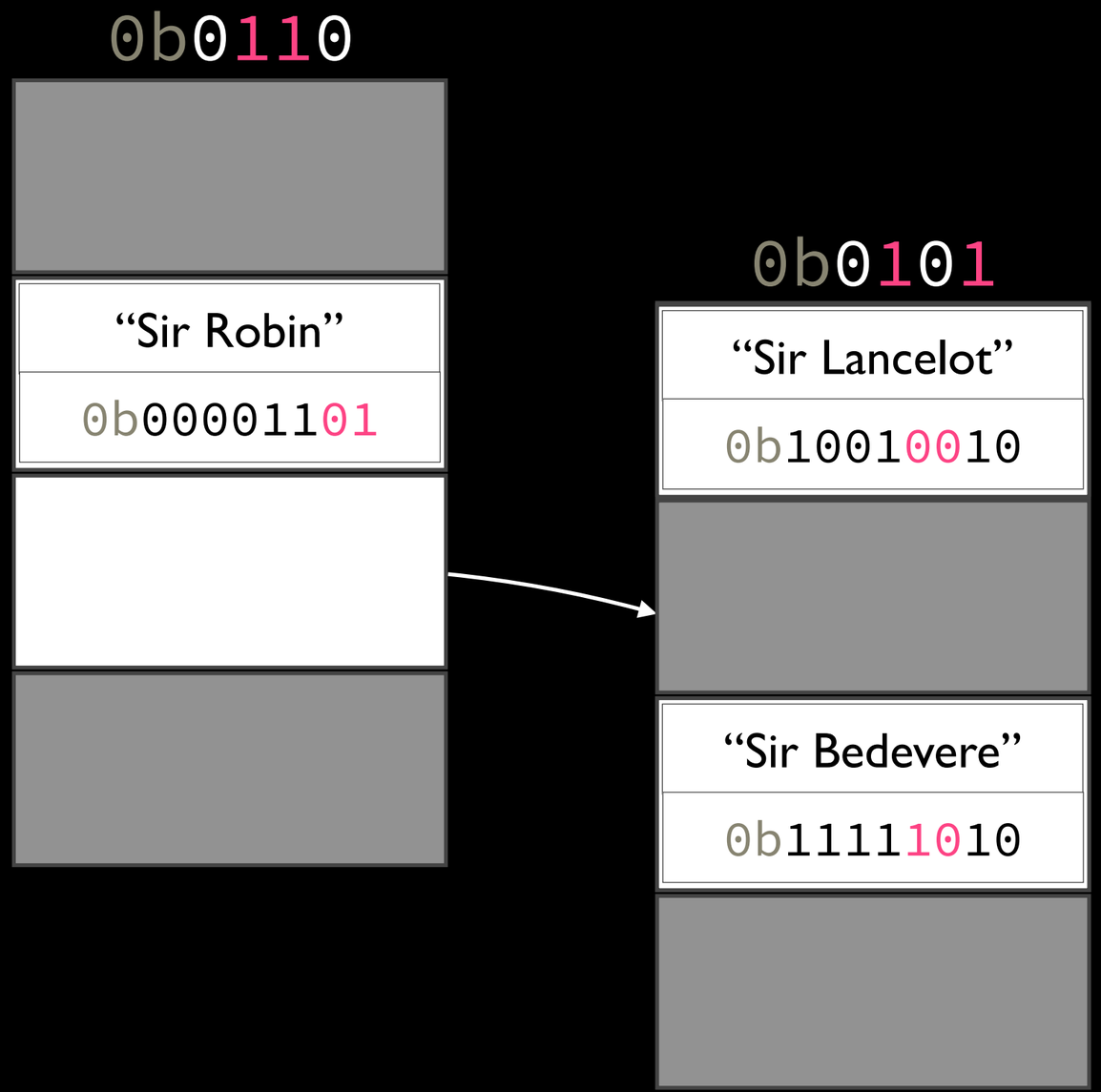
Persistent Hash Map



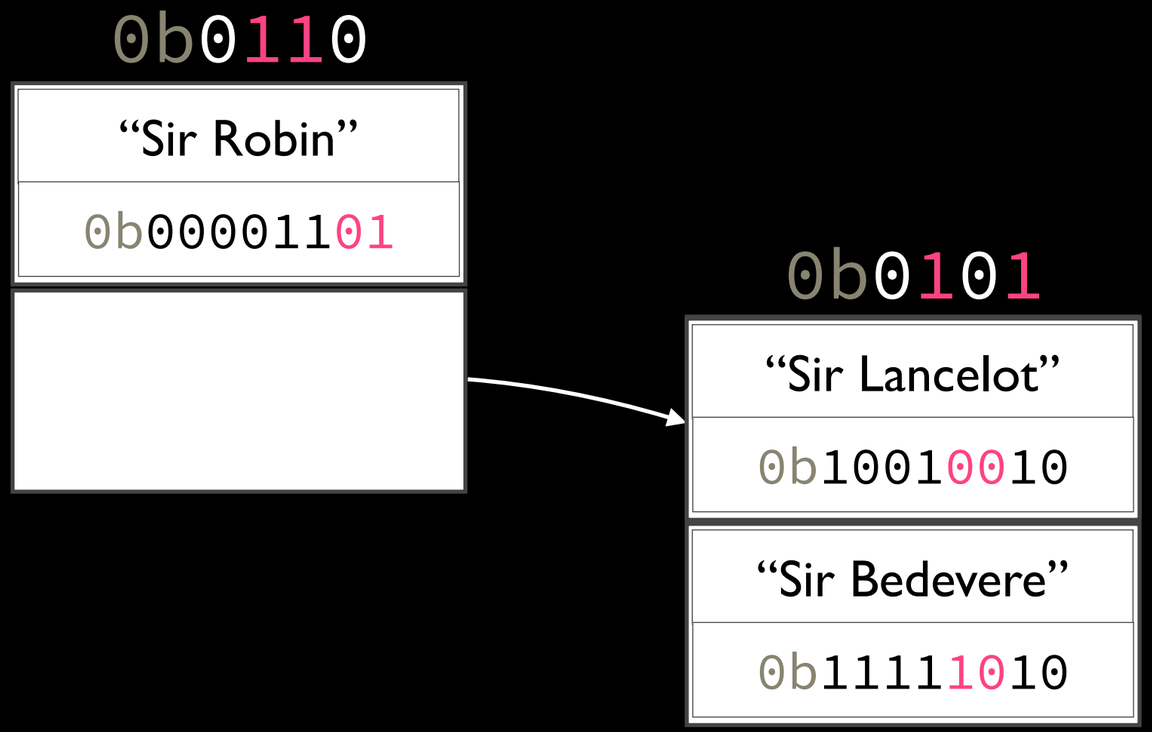
Persistent Hash Map



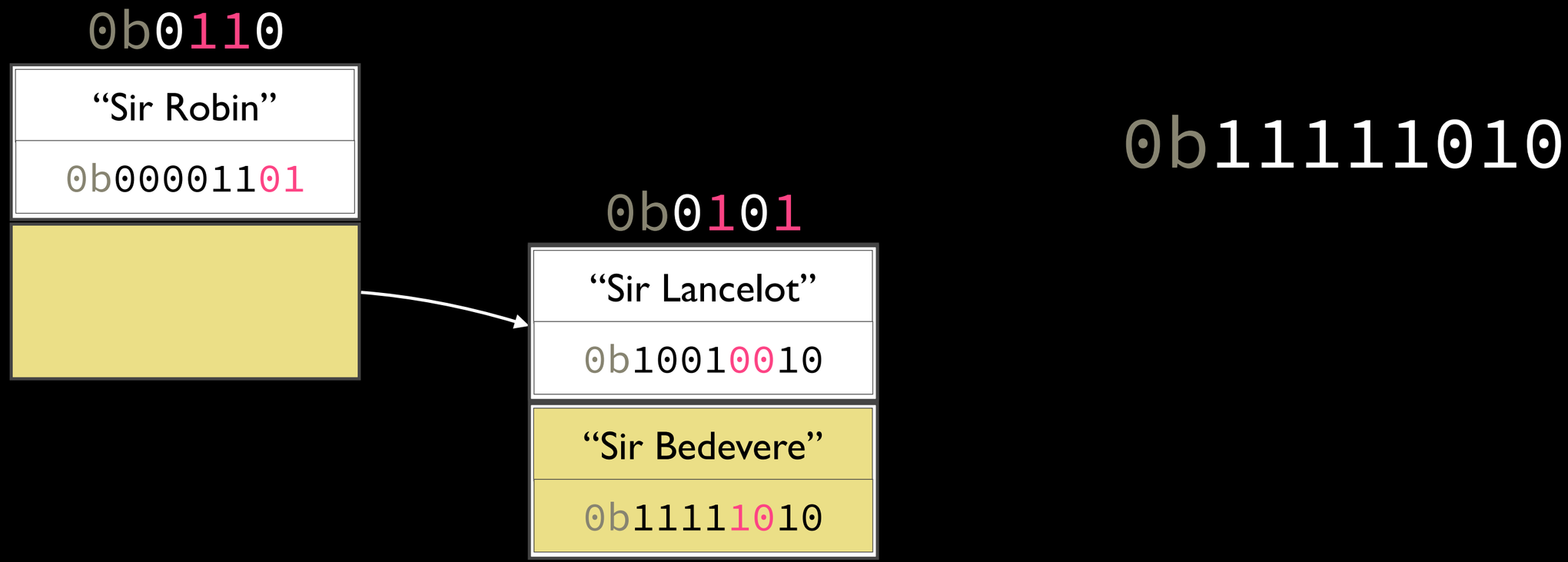
Persistent Hash Map



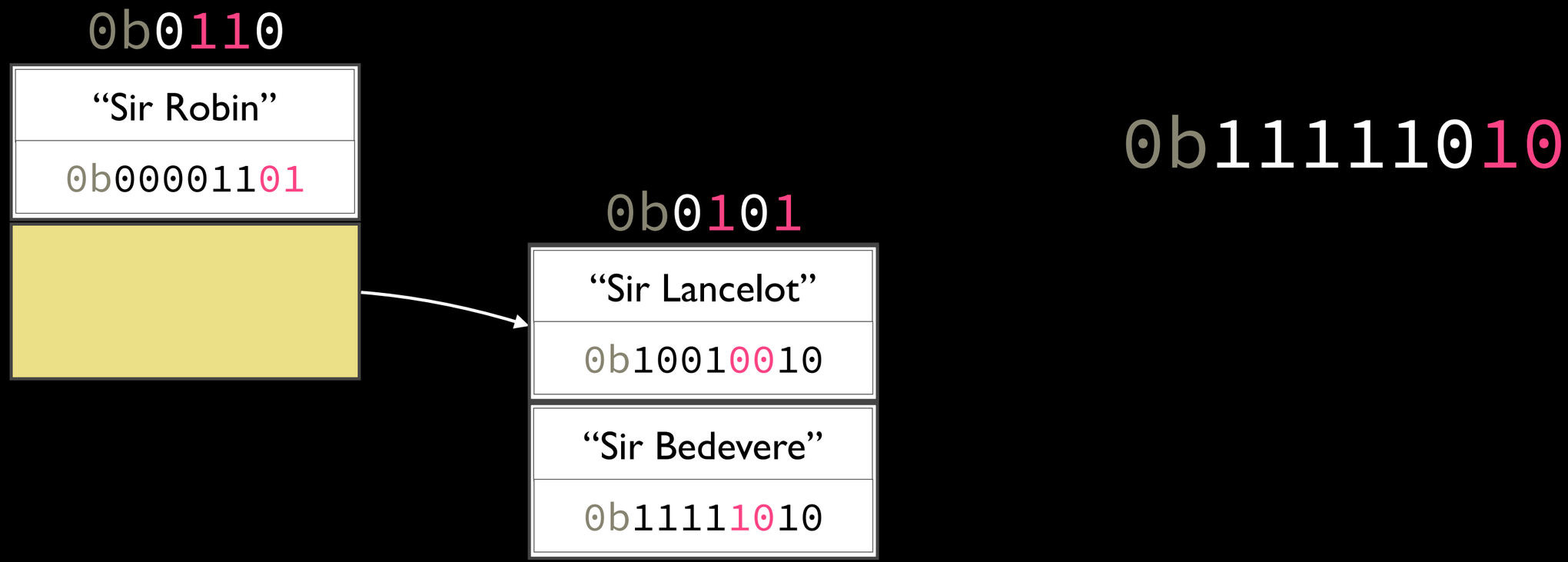
Persistent Hash Map



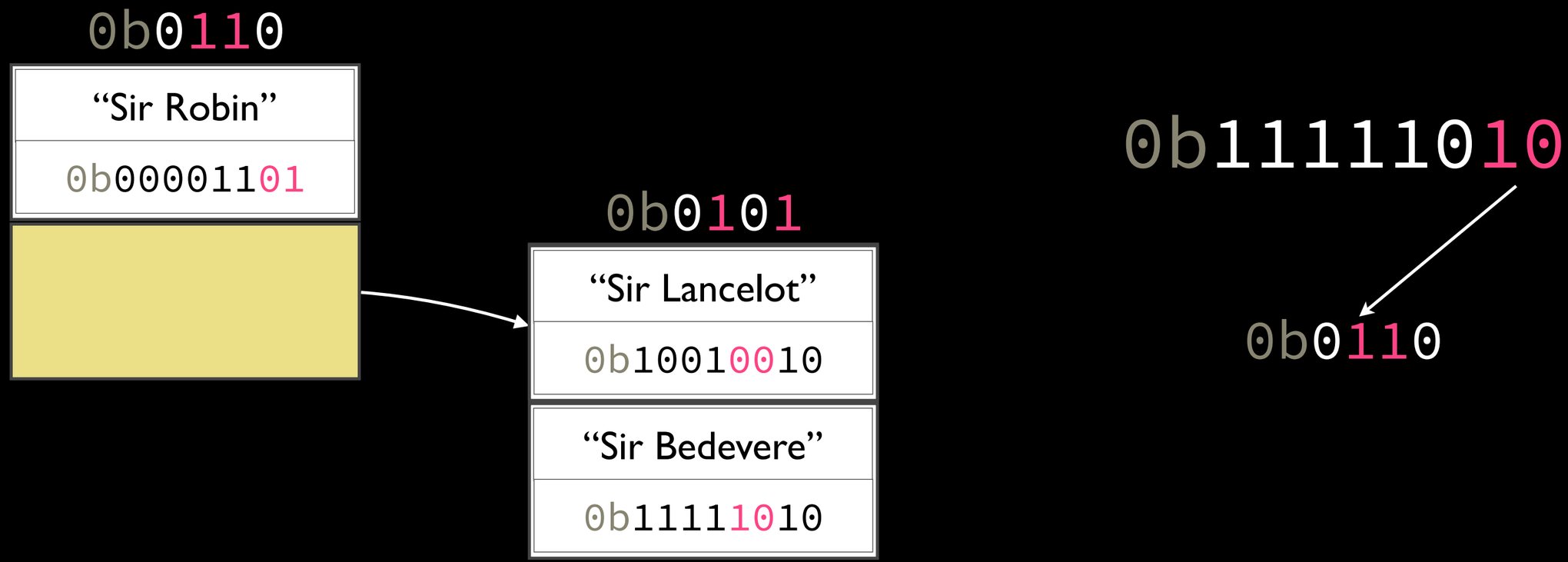
Persistent Hash Map



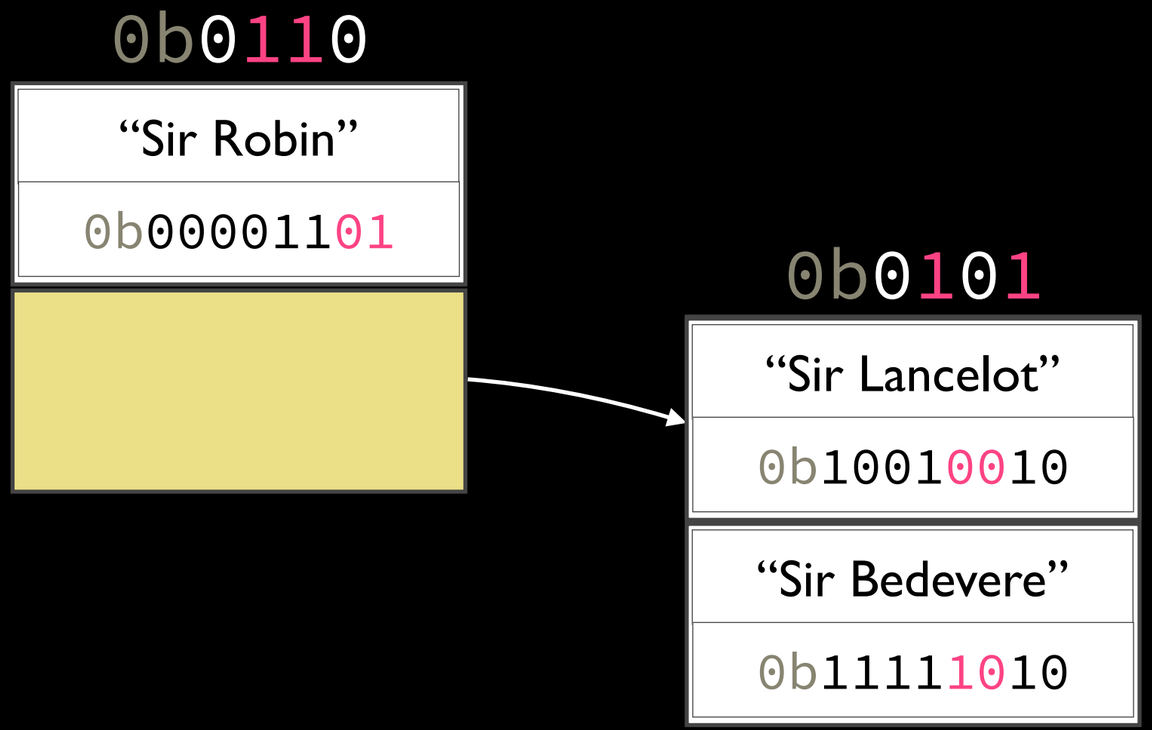
Persistent Hash Map



Persistent Hash Map



Persistent Hash Map

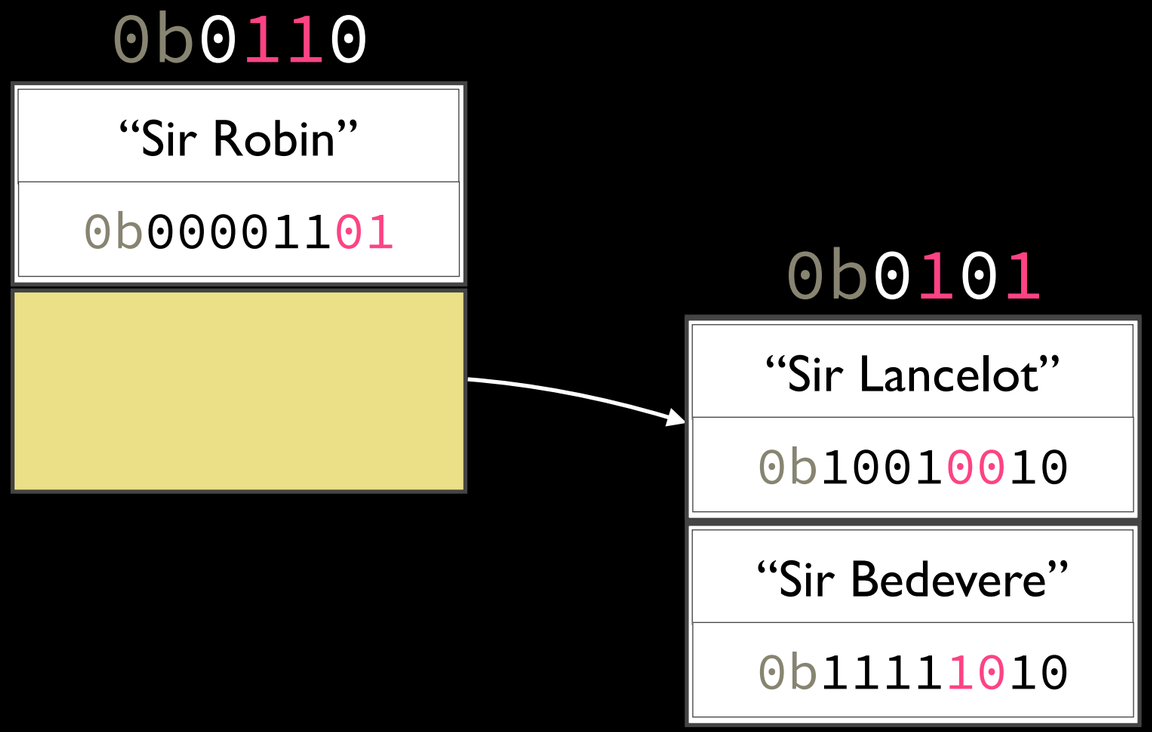


0b11111010

mask off top bits

0b0010

Persistent Hash Map

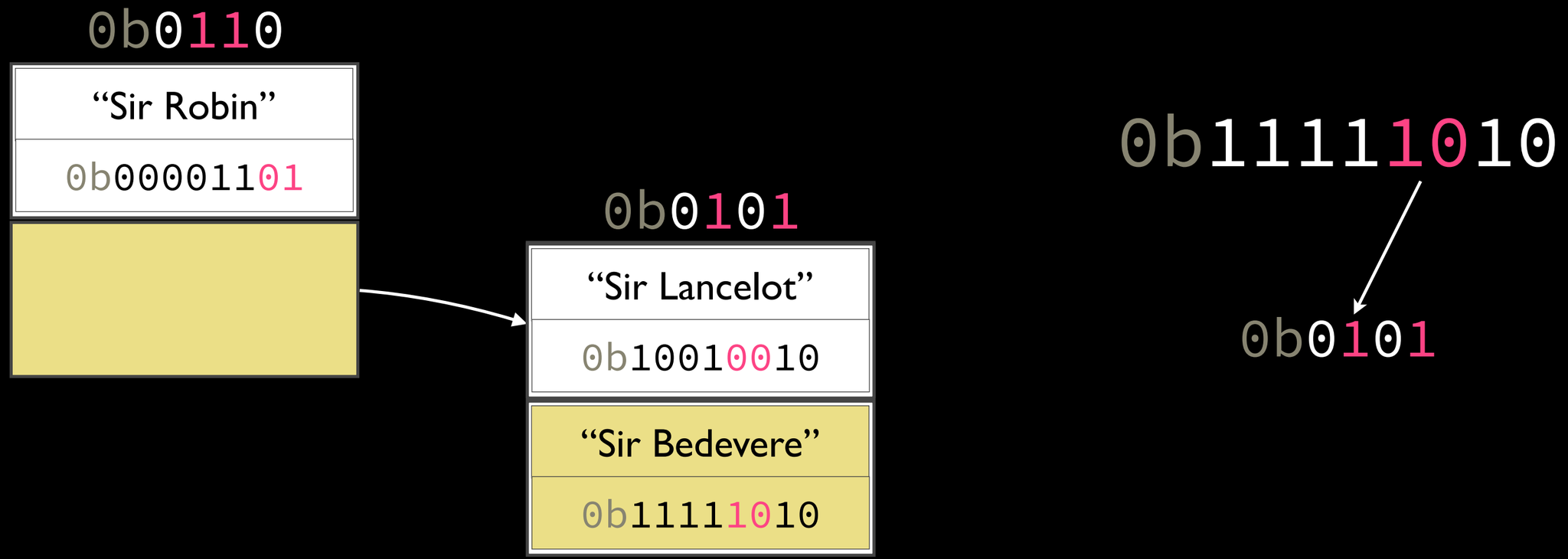


0b11111010

bitcount to get true index

1

Persistent Hash Map



Persistent Hash Map

0b0110

“Sir Robin”
0b00001101

0b0101

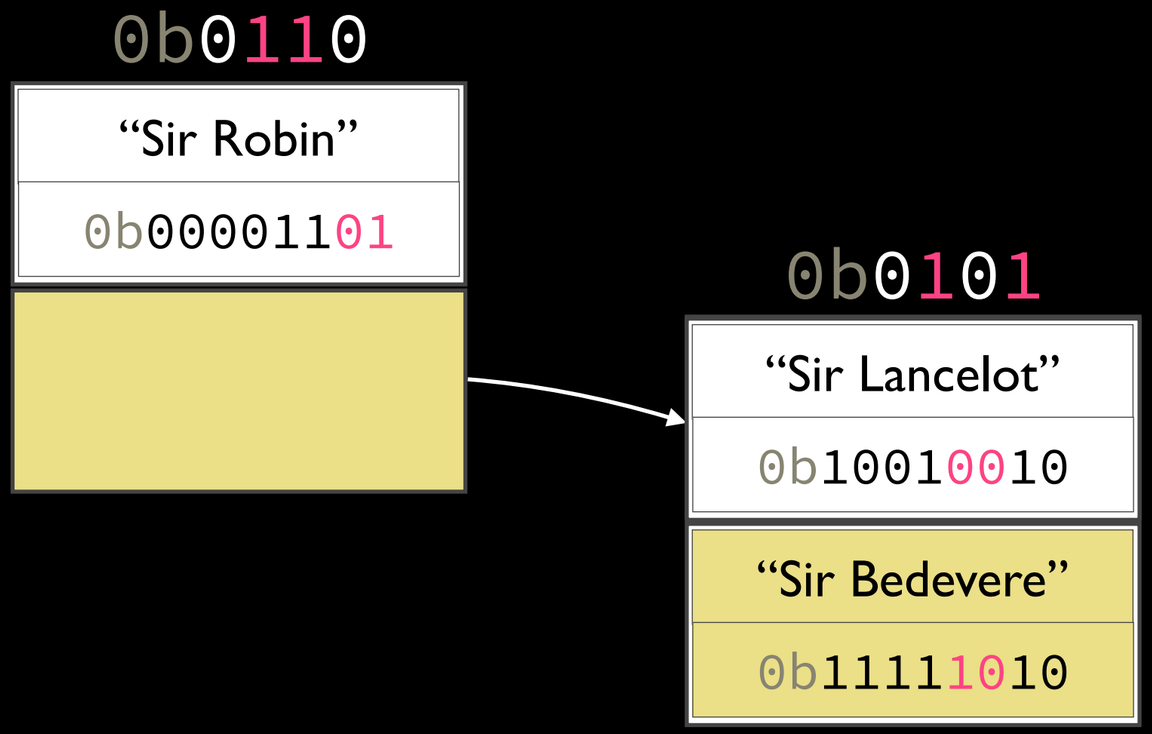
“Sir Lancelot”
0b10010010
“Sir Bedevere”
0b11111010

0b11111010

mask off top bits

0b0001

Persistent Hash Map



0b11111010

bitcount to get true index
1

Persistent Hash Map

```
immutable HashMapEntry
    hash::Int
    key
    value
end
```

Persistent Hash Map

```
abstract SparseBitmappedTrie <: BitmappedTrie

immutable SparseArrayNode <: SparseBitmappedTrie
    arr::Vector{Union(SparseBitmappedTrie,
                      HashMapEntry)}
    shift::Int
    length::Int
    maxlength::Int
    bitmap::Int
end

shiftval(n::SparseArrayNode) = s.shift
```

Persistent Hash Map

```
abstract SparseBitmappedTrie <: BitmappedTrie

immutable SparseArrayNode <: SparseBitmappedTrie
    arr::Vector{Union(SparseBitmappedTrie,
                      HashMapEntry)}
    shift::Int
    length::Int
    maxlength::Int
    bitmap::Int
end

shiftval(n::SparseArrayNode) = s.shift
```

Persistent Hash Map

```
abstract SparseBitmappedTrie <: BitmappedTrie

immutable SparseArrayNode <: SparseBitmappedTrie
    arr::Vector{Union(SparseBitmappedTrie,
                      HashMapEntry)}
    shift::Int
    length::Int
    maxlength::Int
    bitmap::Int
end

shiftval(n::SparseArrayNode) = s.shift
```

Persistent Hash Map

```
function bitpos(t::SparseBitmappedTrie, i)
    1 << (mask(t, i) - 1)
end

function hasindex(t::SparseBitmappedTrie, i)
    t.bitmap & bitpos(t, i) != 0
end

function slot(t::SparseBitmappedTrie, i)
    1 + count_ones(t.bitmap & (bitpos(t, i) - 1))
end
```

Persistent Hash Map

```
function bitpos(t::SparseBitmappedTrie, i)
    1 << (mask(t, i) - 1)
end

function hasindex(t::SparseBitmappedTrie, i)
    t.bitmap & bitpos(t, i) != 0
end

function slot(t::SparseBitmappedTrie, i)
    1 + count_ones(t.bitmap & (bitpos(t, i) - 1))
end
```

Persistent Hash Map

```
function bitpos(t::SparseBitmappedTrie, i)
    1 << (mask(t, i) - 1)
end

function hasindex(t::SparseBitmappedTrie, i)
    t.bitmap & bitpos(t, i) != 0
end

function slot(t::SparseBitmappedTrie, i)
    1 + count_ones(t.bitmap & (bitpos(t, i) - 1))
end
```

Persistent Hash Map TODO

- deletions
- true hash collisions

Persistent Set

wrap the map!

Bitmapped Vector Trie Performance

lookups: $O(\log_{32} N)$

Bitmapped Vector Trie Performance

lookups: O(1)

Bitmapped Vector Trie Performance

updates: O(logN)

Some conclusions

- Julia's a great language for this
- you get a lot from one fundamental data structure
- functional abstractions in a mutable world

github:

zachallaun/FunctionalCollections.jl

Hacker School

I love feedback: zach@hackerschool.com

github: zachallaun

twitter: riotonthebay