Protocol-oriented Persistent Data Structures

in Swift

Vincent Esche | @regexident
blog.definiteloops.com

Protocol-oriented Persistent Data Structures

in Swift

Vincent Esche | @regexident
blog.definiteloops.com

Protocol-oriented Persistent Data Structures

in Swift

Vincent Esche | @regexident
blog.definiteloops.com

Protocol-oriented Programming
Persistent Data Structures

in Swift

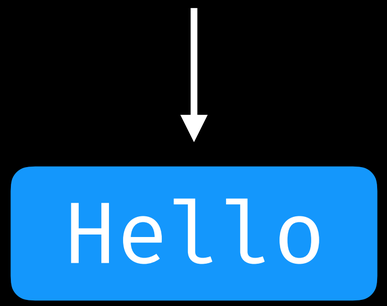
Vincent Esche | @regexident
blog.definiteloops.com

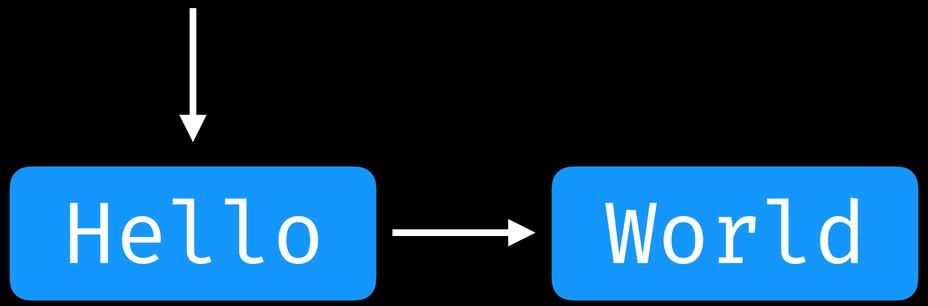
Persistent Data Structures

“In computing, a persistent data structure is a data structure that **always preserves the previous version of itself** when it is **modified**. Such data structures are **effectively immutable**, as their operations do not (visibly) update the structure in-place, but instead **always yield a new updated structure**.” – Wikipedia

```
print(“Hello World”)
```

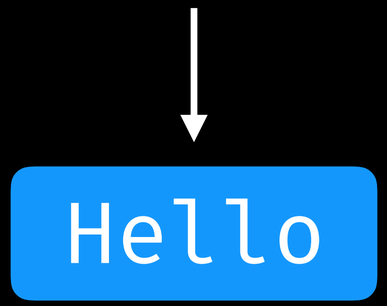
Hello

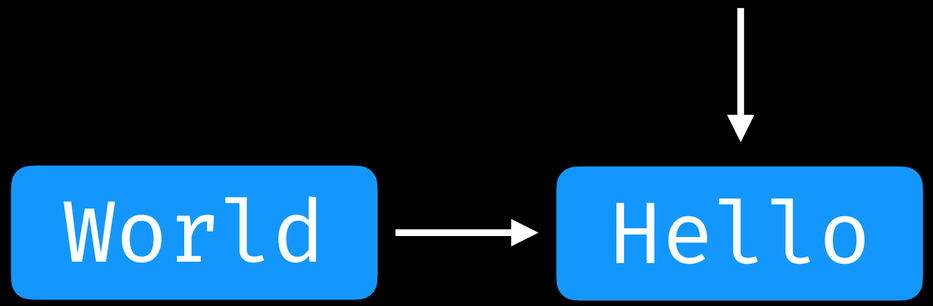

Hello

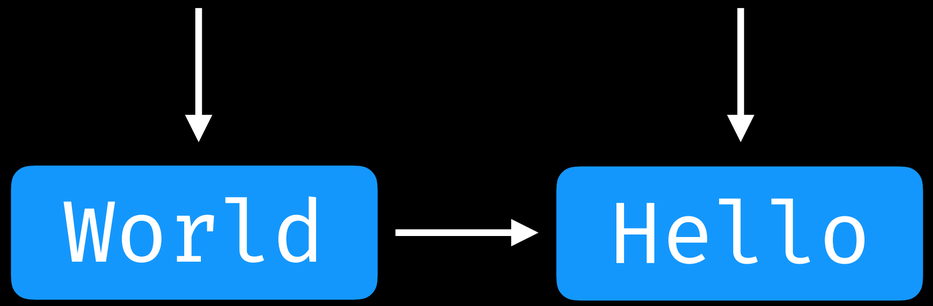

Hello
World

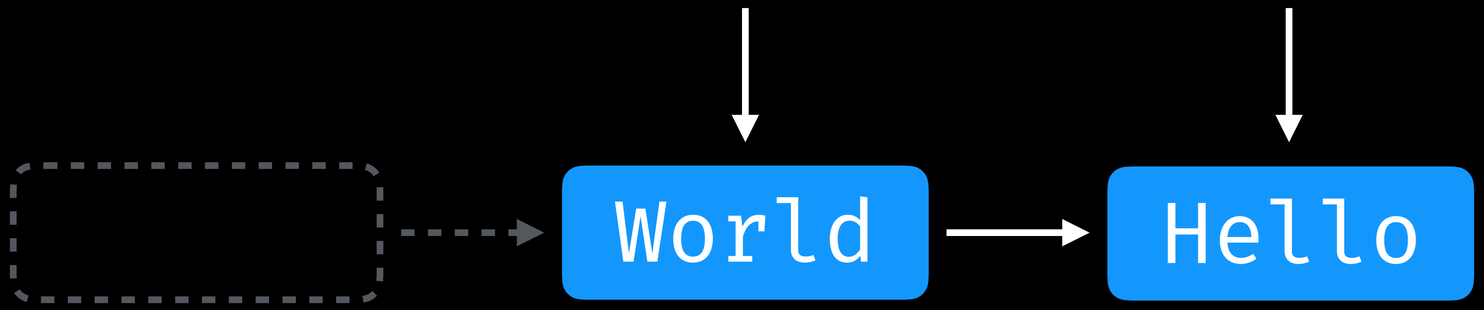

Hello
World

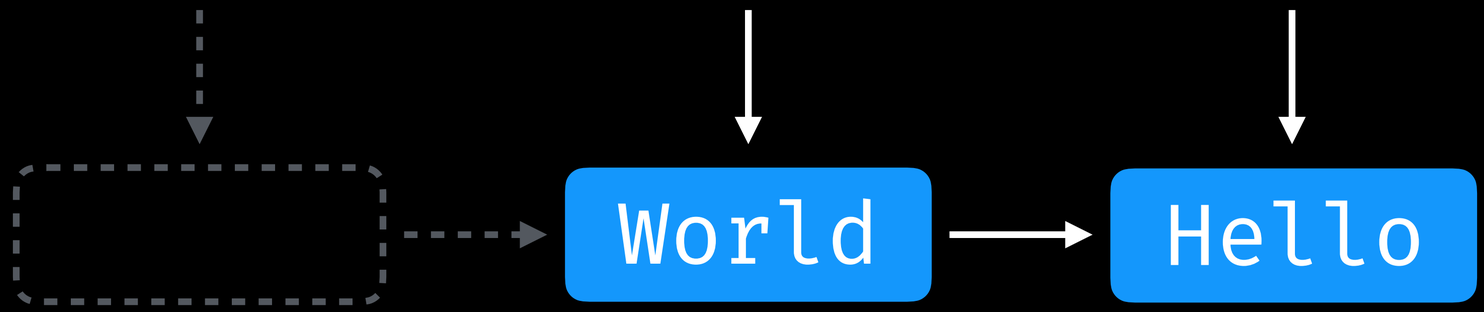
Hello


Hello


World
Hello


World
Hello


World
Hello


World
Hello

```
enum LinkedList<T> {
  indirect case cons(T, LinkedList<T>)
  case none

  // …
}
```

```
let empty: LinkedList<Int> = .none
let list1 = .cons(1, empty)
let list2 = .cons(2, list1)
let list3 = .cons(3, list2)
```

```
enum LinkedList<T> {
  // …
  init() { self = .none }

  init(
    head: T,
    tail: LinkedList<T> = .none
  ) {
    self = .cons(head, tail)
  }
  // …
}
```

```
enum LinkedList<T> {
  // …
  init() { self = .none }

  init(
    head: T,
    tail: LinkedList<T> = .none
  ) {
    self = .cons(head, tail)
  }
  // …
}
```

```
enum LinkedList<T> {
  // …
  init() { self = .none }

  init(
    head: T,
    tail: LinkedList<T> = .none
  ) {
    self = .cons(head, tail)
  }
  // …
}
```

```
let empty = LinkedList<Int>()
let list1 = LinkedList(head: 1, tail: empty)
let list2 = LinkedList(head: 2, tail: list1)
let list3 = LinkedList(head: 3, tail: list2)
```

```
enum LinkedList<T> {
  // …
  func walk(closure: (T) → ()) {
    if case let .cons(head, tail) = self {
      closure(head)
      tail.walk(closure: closure)
    }
  }
  // …
}
```

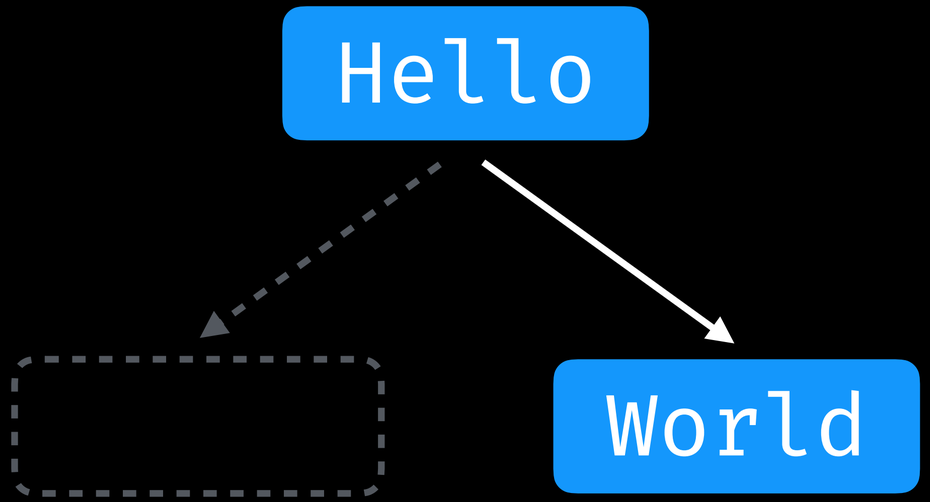
```
enum LinkedList<T> {
  // …
  func walk(closure: (T) → ()) {
    if case let .cons(head, tail) = self {
      closure(head)
      tail.walk(closure: closure)
    }
  }
  // …
}
```

```
enum LinkedList<T> {
  // …
  func collect() → [T] {
    var array: [T] = []
    self.walk { array.append($0) }
    return array
  }
  // …
}
```

```
let empty = LinkedList<Int>()
let list1 = LinkedList(head: 1, tail: empty)
let list2 = LinkedList(head: 2, tail: list1)
let list3 = LinkedList(head: 3, tail: list2)

print(empty.collect()) // []
print(list1.collect()) // [1]
print(list2.collect()) // [2, 1]
print(list3.collect()) // [3, 2, 1]
```


World
Hello


Hello
World

```
enum BinaryTree<T> {
  indirect case branch(
    BinaryTree, T, BinaryTree
  )
  case leaf

  // …
}
```

```
enum BinaryTree<T> {
  // …
  init() { self = .leaf }
  init(_ value: T) {
    self.init(.leaf, value, .leaf)
  }
  init(_ left: BinaryTree, _ value: T, _
right: BinaryTree) {
    self = .branch(left, value, right)
  }
  // …
}
```

```
enum BinaryTree<T> {
  // …
  init() { self = .leaf }
  init(_ value: T) {
    self.init(.leaf, value, .leaf)
  }
  init(_ left: BinaryTree, _ value: T, _
right: BinaryTree) {
    self = .branch(left, value, right)
  }
  // …
}
```

```
enum BinaryTree<T> {
  // …
  init() { self = .leaf }
  init(_ value: T) {
    self.init(.leaf, value, .leaf)
  }
  init(_ left: BinaryTree, _ value: T, _
right: BinaryTree) {
    self = .branch(left, value, right)
  }
  // …
}
```

```
enum BinaryTree<T> {
  // …
  init() { self = .leaf }
  init(_ value: T) {
    self.init(.leaf, value, .leaf)
  }
  init(_ left: BinaryTree, _ value: T, _
right: BinaryTree) {
    self = .branch(left, value, right)
  }
  // …
}
```

Protocol-oriented Programming

```
public protocol BinaryTreeType {
  associatedtype Element

  init()

  func analysis<U>(branch: (Self, Element,
Self) → U, leaf: () → U) → U
}
```

```
enum BinaryTree<T> {
  // …
  func analysis<U>(_ branch: (BinaryTree, T,
BinaryTree) → U, leaf: () → U) → U {
    switch self {
    case .leaf:
      return leaf()
    case let .branch(left, element, right):
      return branch(left, element, right)
    }
  }
  // …
}
```

```
enum BinaryTree<T> {
  // …
  func analysis<U>(_ branch: (BinaryTree, T,
BinaryTree) → U, leaf: () → U) → U {
    switch self {
    case .leaf:
      return leaf()
    case let .branch(left, element, right):
      return branch(left, element, right)
    }
  }
  // …
}
```

```
enum BinaryTree<T> {
  // …
  func analysis<U>(_ branch: (BinaryTree, T,
BinaryTree) → U, leaf: () → U) → U {
    switch self {
    case .leaf:
      return leaf()
    case let .branch(left, element, right):
      return branch(left, element, right)
    }
  }
  // …
}
```

```
extension BinaryTreeType {
  func inorder(_ closure: (Element) → ()) {
        self.analysis({ (l, e, r) → () in
            l.inorder(closure)
            closure(e)
            r.inorder(closure)
        }, leaf: {})
    }
}
```

```
public protocol BinarySearchTreeType :
BinaryTreeType {
  associatedtype Element: Comparable

  init<S: Sequence>(sortedSequence: S) where
S.Iterator.Element == Element
}
```

```
extension BinarySearchTreeType {
  func get(_ element: Element) → Element? {
    return analysis(
      branch: { l, e, r in
        if element < e {
          return l.get(element)
        } else if element > e {
          return r.get(element)
        } else { return e }
      },
      leaf: { return nil }
    )
  }
}
```

```
extension BinarySearchTreeType {
  func get(_ element: Element) → Element? {
    return analysis(
      branch: { l, e, r in
        if element < e {
          return l.get(element)
        } else if element > e {
          return r.get(element)
        } else { return e }
      },
      leaf: { return nil }
    )
  }
}
```

```
extension BinarySearchTreeType {
  func get(_ element: Element) → Element? {
    return analysis(
      branch: { l, e, r in
        if element < e {
          return l.get(element)
        } else if element > e {
          return r.get(element)
        } else { return e }
      },
      leaf: { return nil }
    )
  }
}
```

```
extension BinarySearchTreeType {
  func get(_ element: Element) → Element? {
    return analysis(
      branch: { l, e, r in
        if element < e {
          return l.get(element)
        } else if element > e {
          return r.get(element)
        } else { return e }
      },
      leaf: { return nil }
    )
  }
}
```

```
extension BinarySearchTreeType {
  func get(_ element: Element) → Element? {
    return analysis(
      branch: { l, e, r in
        if element < e {
          return l.get(element)
        } else if element > e {
          return r.get(element)
        } else { return e }
      },
      leaf: { return nil }
    )
  }
}
```

```
extension BinarySearchTreeType {
  func get(_ element: Element) → Element? {
    return analysis(
      branch: { l, e, r in
        if element < e {
          return l.get(element)
        } else if element > e {
          return r.get(element)
        } else { return e }
      },
      leaf: { return nil }
    )
  }
}
```

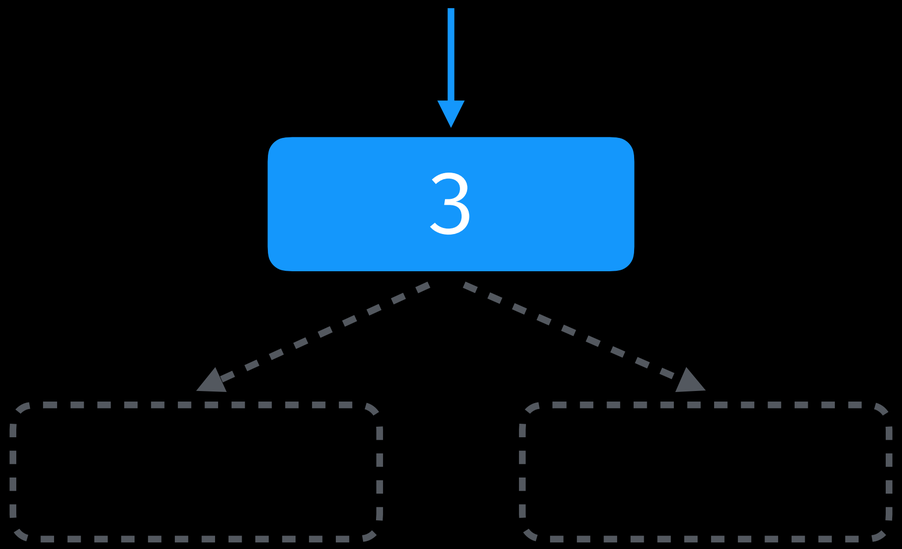
```
protocol GrowableBinarySearchTreeType :
BinarySearchTreeType {
  func insertAndReturnExisting(_ element:
Element) → (Self, Element?)
}
protocol PrunableBinarySearchTreeType :
BinarySearchTreeType {
  func removeAndReturnExisting(_ element:
Element) → (Self, Element?)
}
protocol MutableBinarySearchTreeType :
GrowableBinarySearchTreeType,
PrunableBinarySearchTreeType {}
```

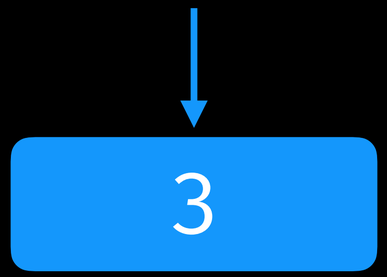
```
protocol GrowableBinarySearchTreeType :
BinarySearchTreeType {
  func insertAndReturnExisting(_ element:
Element) → (Self, Element?)
}
protocol PrunableBinarySearchTreeType :
BinarySearchTreeType {
  func removeAndReturnExisting(_ element:
Element) → (Self, Element?)
}
protocol MutableBinarySearchTreeType :
GrowableBinarySearchTreeType,
PrunableBinarySearchTreeType {}
```

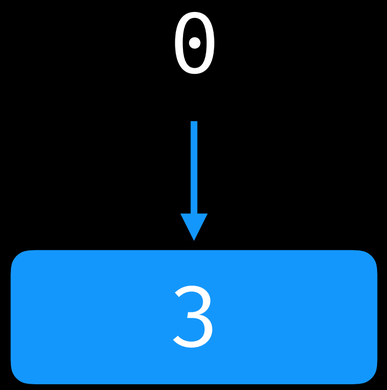
```
protocol GrowableBinarySearchTreeType :
BinarySearchTreeType {
  func insertAndReturnExisting(_ element:
Element) → (Self, Element?)
}
protocol PrunableBinarySearchTreeType :
BinarySearchTreeType {
  func removeAndReturnExisting(_ element:
Element) → (Self, Element?)
}
protocol MutableBinarySearchTreeType :
GrowableBinarySearchTreeType,
PrunableBinarySearchTreeType {}
```

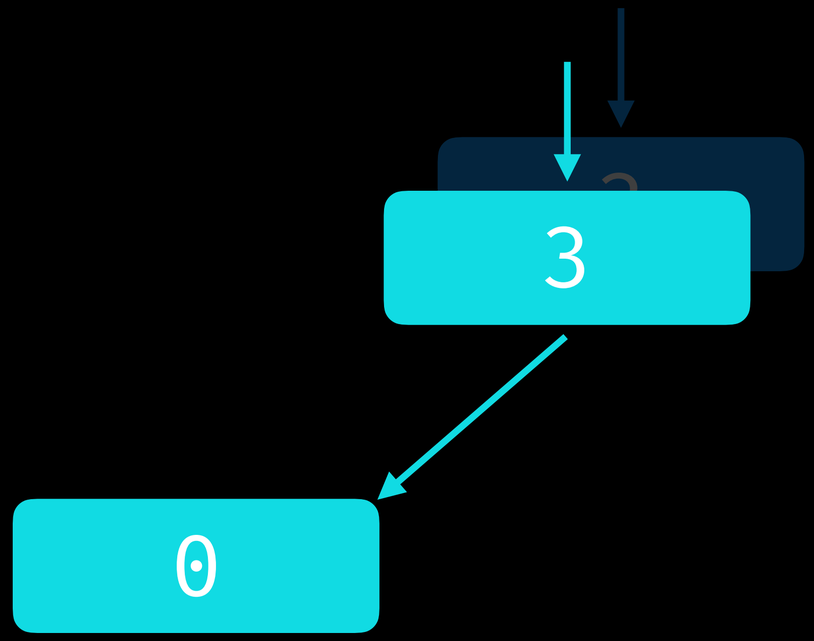
```
protocol GrowableBinarySearchTreeType :
BinarySearchTreeType {
  func insertAndReturnExisting(_ element:
Element) → (Self, Element?)
}
protocol PrunableBinarySearchTreeType :
BinarySearchTreeType {
  func removeAndReturnExisting(_ element:
Element) → (Self, Element?)
}
protocol MutableBinarySearchTreeType :
GrowableBinarySearchTreeType,
PrunableBinarySearchTreeType {}
```

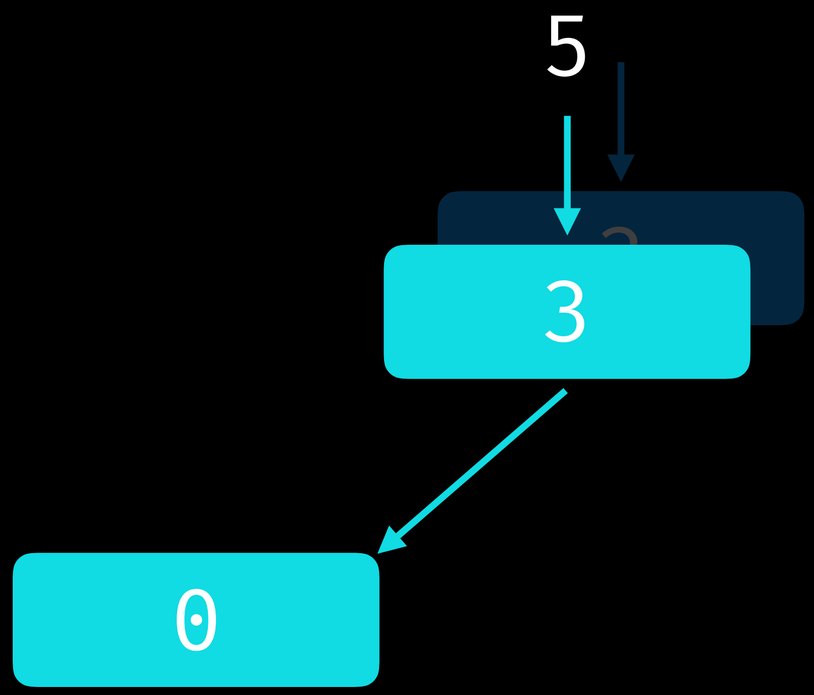
3

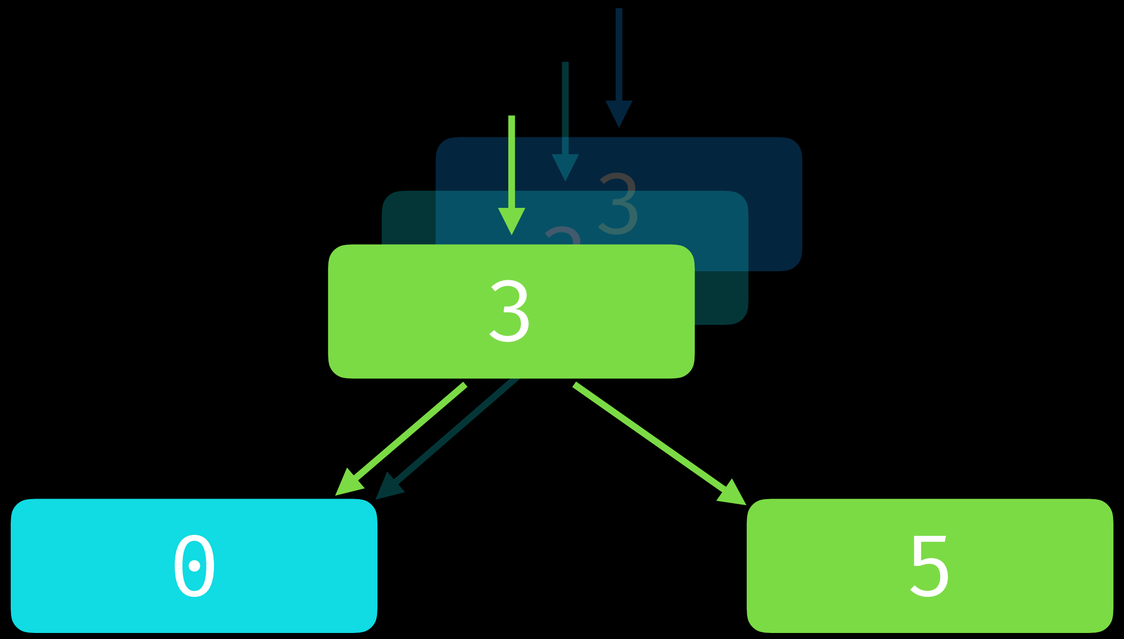

3

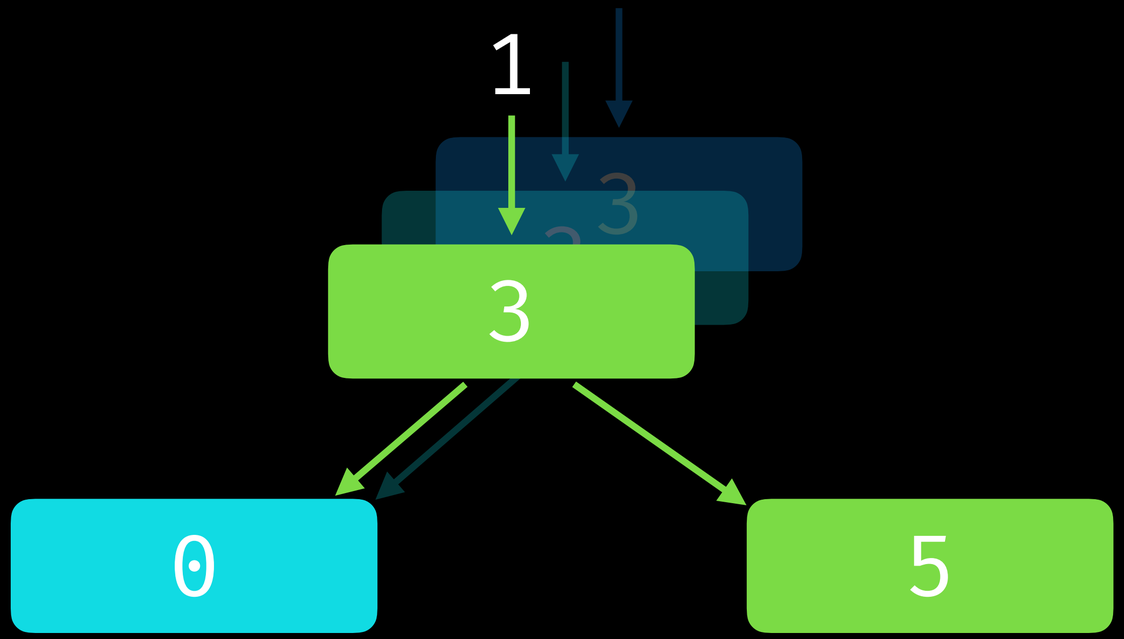

3

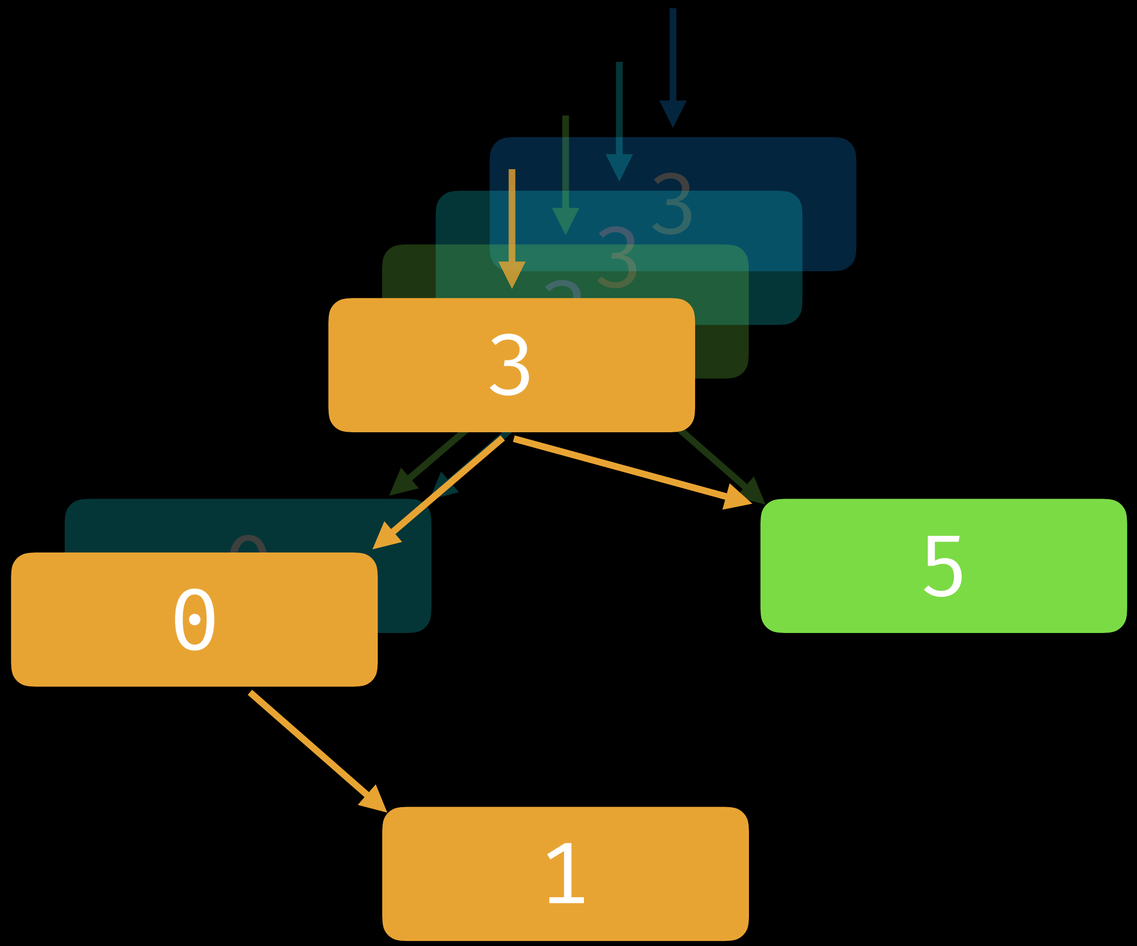

0
3

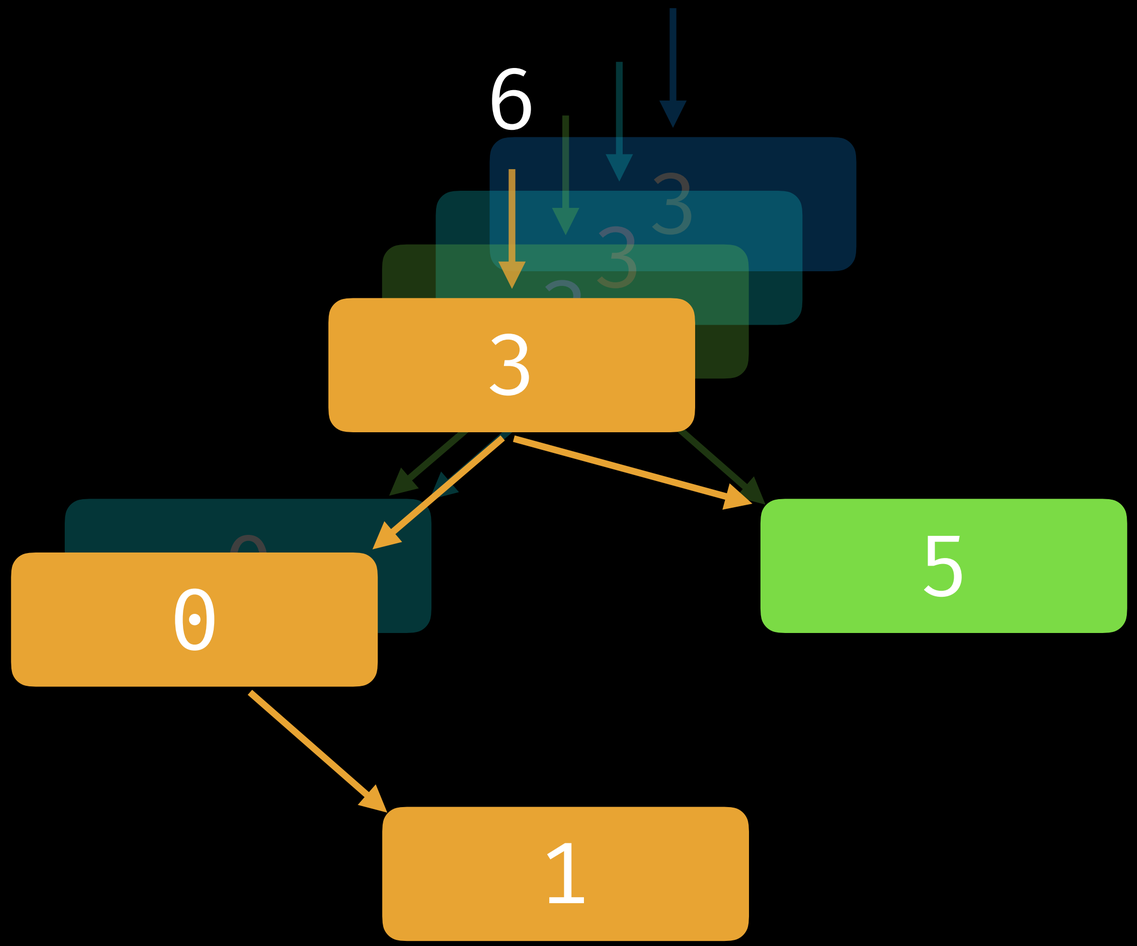

3
0

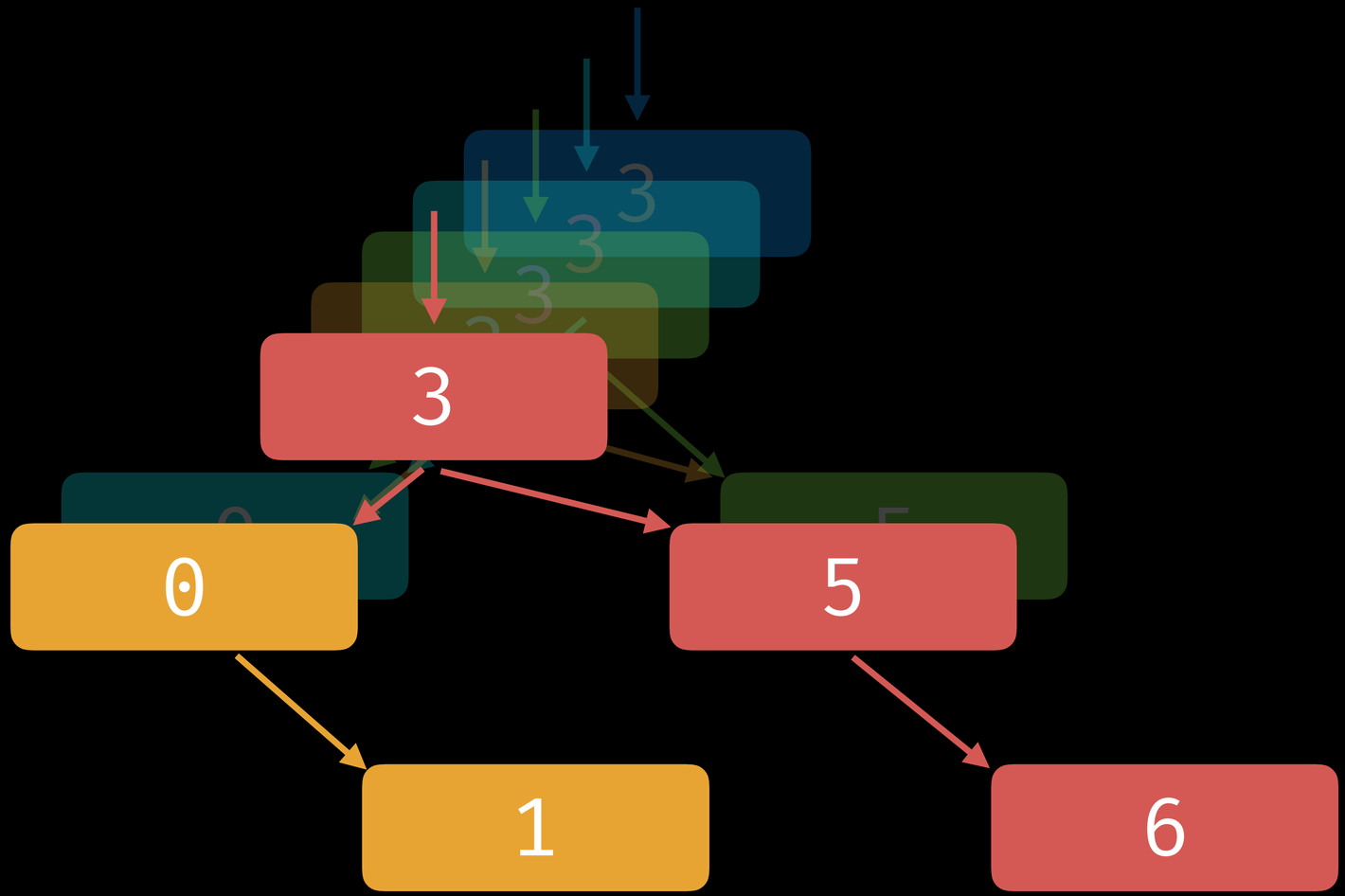

5
3
0


3
3
3
0
5


1
3
3
0
5


3
0
5
1


6
3
3
3
3
0
5
1


3
0
5
1
6

Demo!

https://github.com/regexident/Forest

https://github.com/regexident/Forest

Vincent Esche | @regexident
blog.definiteloops.com

Thanks!

Questions?

https://github.com/regexident/Forest

Vincent Esche | @regexident
blog.definiteloops.com