FOR NUM IN NUMBERS { ... }

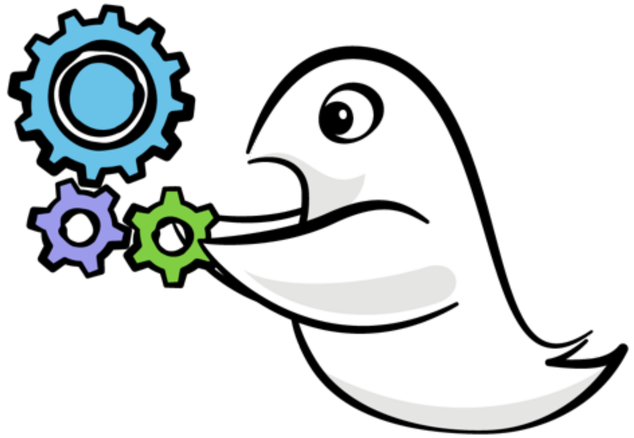
NUMBERS[4]

NUMBERS.FILTER { ... }

NUMBERS.MIN

NUMBERS.FIRST { ... }

Swift Collection Protocols



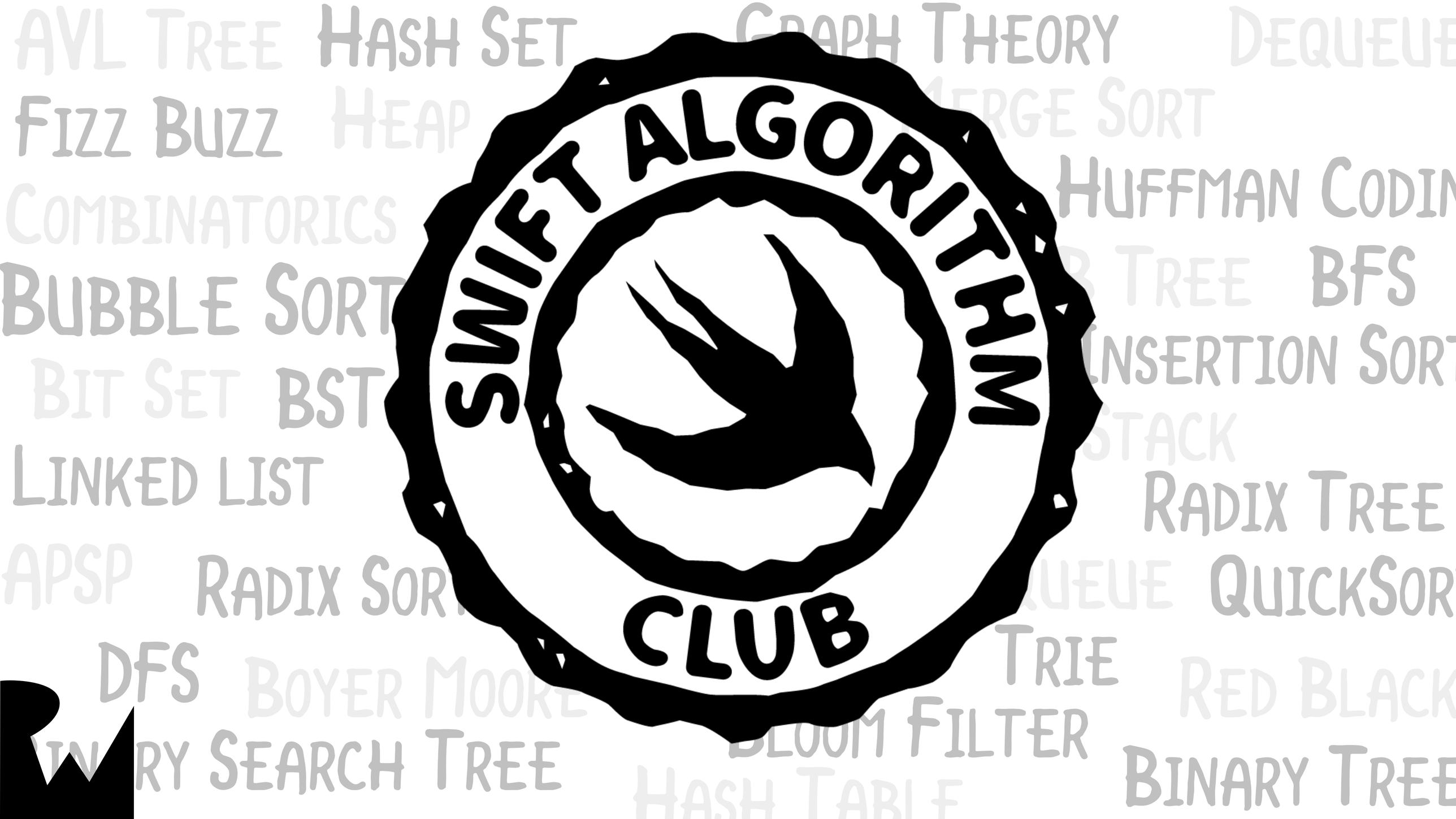
STRINGS.JOINED(SEPARATOR:)

NUMBERS.REDUCE(0) { ... }

NUMBERS.MAP { ... }

NUMBERS.MAX

SWIFT ALGORITHM WORKSHOP


AVL Tree
Hash Set
Graph Theory
Dequeue
Fizz Buzz
Heap
Merge Sort
Huffman Codin
Combinatorics
B Tree
BFS
Bubble Sort
Insertion Sort
Bit Set
BST
Stack
Linked list
Radix Tree
APSP
Radix Sort
Queue
QuickSor
DFS
Boyer Moore
Trie
Red Black
Bloom Filter
Binary Search Tree
Hash Table
SWIFT ALGORITHM
CLUB

Swift Algorithm Club

The Swift Algorithm Club is an open source project to implement popular algorithms and data structures in Swift.

Every month, we write a tutorial showing you how to make a cool algorithm or data structure from the project. Enjoy!

- Join the Swift Algorithm Club!
- Swift Algorithm Club: Swift Tree Data Structure
- Swift Algorithm Club: Swift Trie Data Structure
- Swift Algorithm Club: Swift Binary Search Tree Data Structure
- Swift Algorithm Club: Swift Linked List Data Structure
- Swift Algorithm Club: Swift Queue Data Structure
- Swift Algorithm Club: Swift Stack Data Structure
- Swift Algorithm Club: Graphs with Adjacency List
- Swift Algorithm Club: Swift Merge Sort
- Swift Algorithm Club: Swift Breadth First Search
- Swift Algorithm Club: Swift Depth First Search
- Swift Algorithm Club: Heap and Priority Queue Data Structure
- Swift Algorithm Club: Boyer Moore String Search Algorithm
- Swift Algorithm Club: June Digest 2017
- Swift Algorithm Club: Minimum Spanning Tree with Prim's Algorithm
- Swift Algorithm Club: Swift Dijkstra's Algorithm
- Swift Algorithm Club: Hash Tables

BidirectionalCollection

RandomAccessCollection

Sequence

Collection

RangeReplaceableCollection

MutableCollection

1. Foundation of Knowledge

Adopted By

- AnyIterator
- AnySequence
- AVCaptureSynchronizedDataCollection
- Character.UnicodeScalarView
- CountablePartialRangeFrom
- Dictionary
- DispatchDataIterator
- EmptyIterator
- EnumeratedIterator
- EnumeratedSequence
- FileManager.DirectoryEnumerator
- FlattenIterator
- FlattenSequence
- IndexingIterator
- IteratorOverOne
- IteratorSequence
- JoinedSequence
- LazyCollection
- LazyDropWhileIterator
- LazyFilterCollection
- LazyFilterIterator
- LazyMapIterator
- LazyPrefixWhileIterator
- NSArray
- NSCountedSet
- NSDictionary
- NSEnumerator
- NSIndexSet
- NSMutableArray
- NSMutableDictionary
- NSMutableIndexSet
- NSMutableOrderedSet
- NSMutableSet
- NSOrderedSet
- NSSet
- ReversedCollection.Iterator
- SBElementArray
- Set
- StrideThrough
- StrideTo
- UnfoldSequence
- UnsafeBufferPointerIterator
- UnsafeMutableRawBufferPointer
- UnsafeMutableRawBufferPointer.Iterator
- UnsafeRawBufferPointer
- UnsafeRawBufferPointer.Iterator
- Zip2Sequence

2. Promoting Reusability

```
func algorithm(for array: [Int])
```

[1, 2, 3]

```
func algorithm(for array: [String])
```

["yo", "RWDC"]

2. Promoting Reusability

```
func algorithm<T: Equatable>(for array: [T])
```

[1, 2, 3]

["YO", "RWDC"]

```
func algorithm<T>(for set: Set<T>)
```

[1, 2, 3]

["YO", "RWDC"]

2. Promoting Reusability

[1, 2, 3]

["Yo", "RWDC"]

```
func algorithm<Elements: Sequence>(for elements: Elements)
```

["Marco": "Polo"]

YourCustomSequence

3. USEFUL FUNCTIONS!

FOR NUM IN NUMBERS { ... }

NUMBERS[4]

NUMBERS.FIRST { ... }

NUMBERS.FILTER { ... }

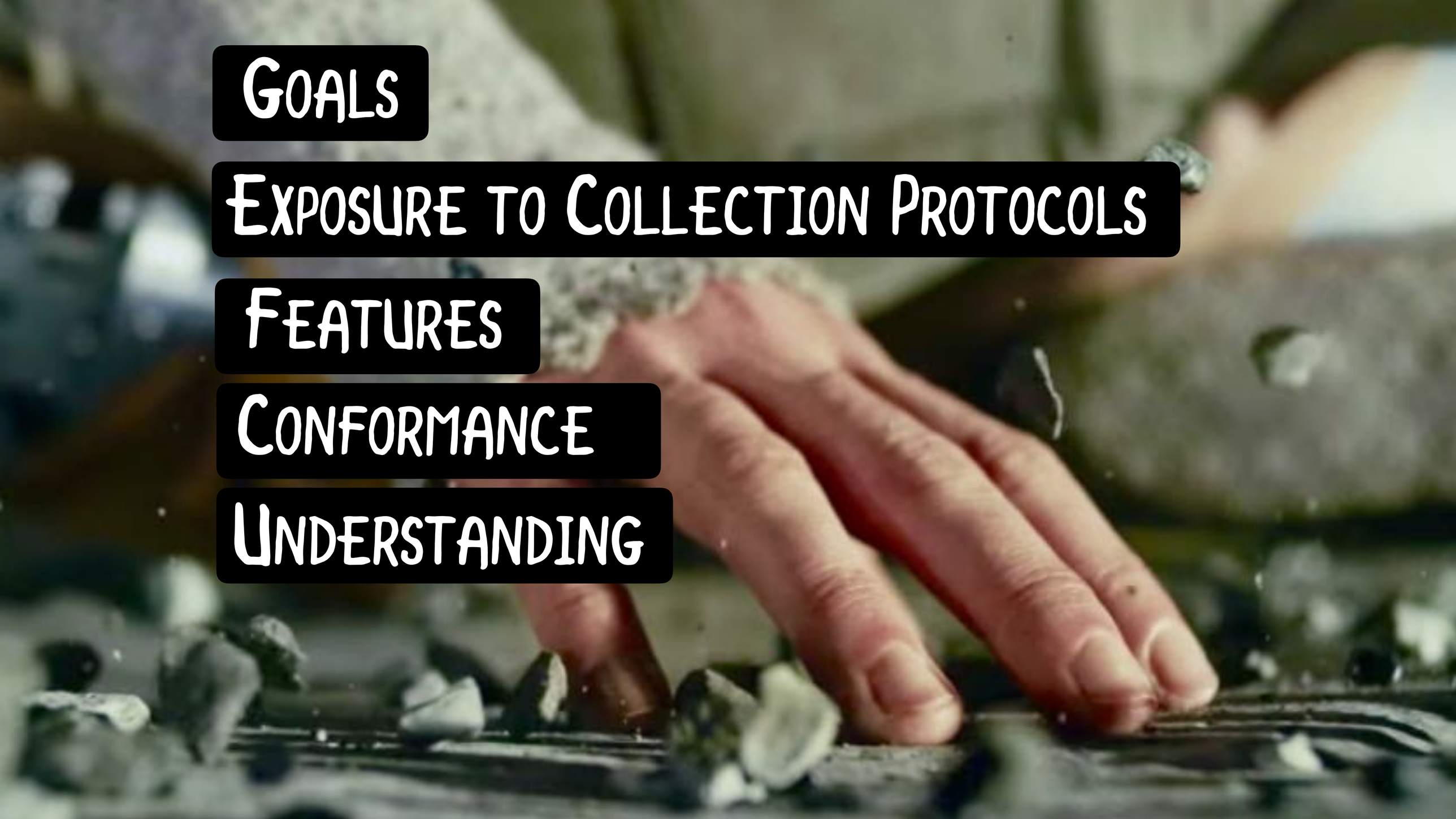
NUMBERS.MIN

NUMBERS.MAX

STRINGS.JOINED(SEPARATOR:)

NUMBERS.REDUCE(0) { ... }

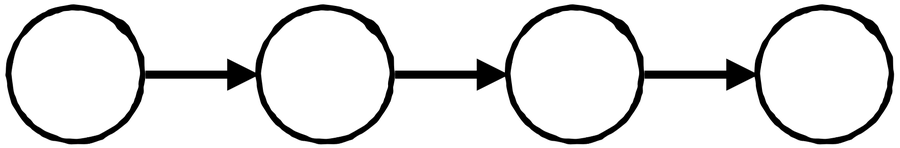
NUMBERS.MAP { ... }


Goals
Exposure to Collection Protocols
Features
Conformance
Understanding

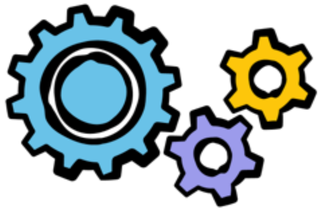

With great protocols
comes great reusability.

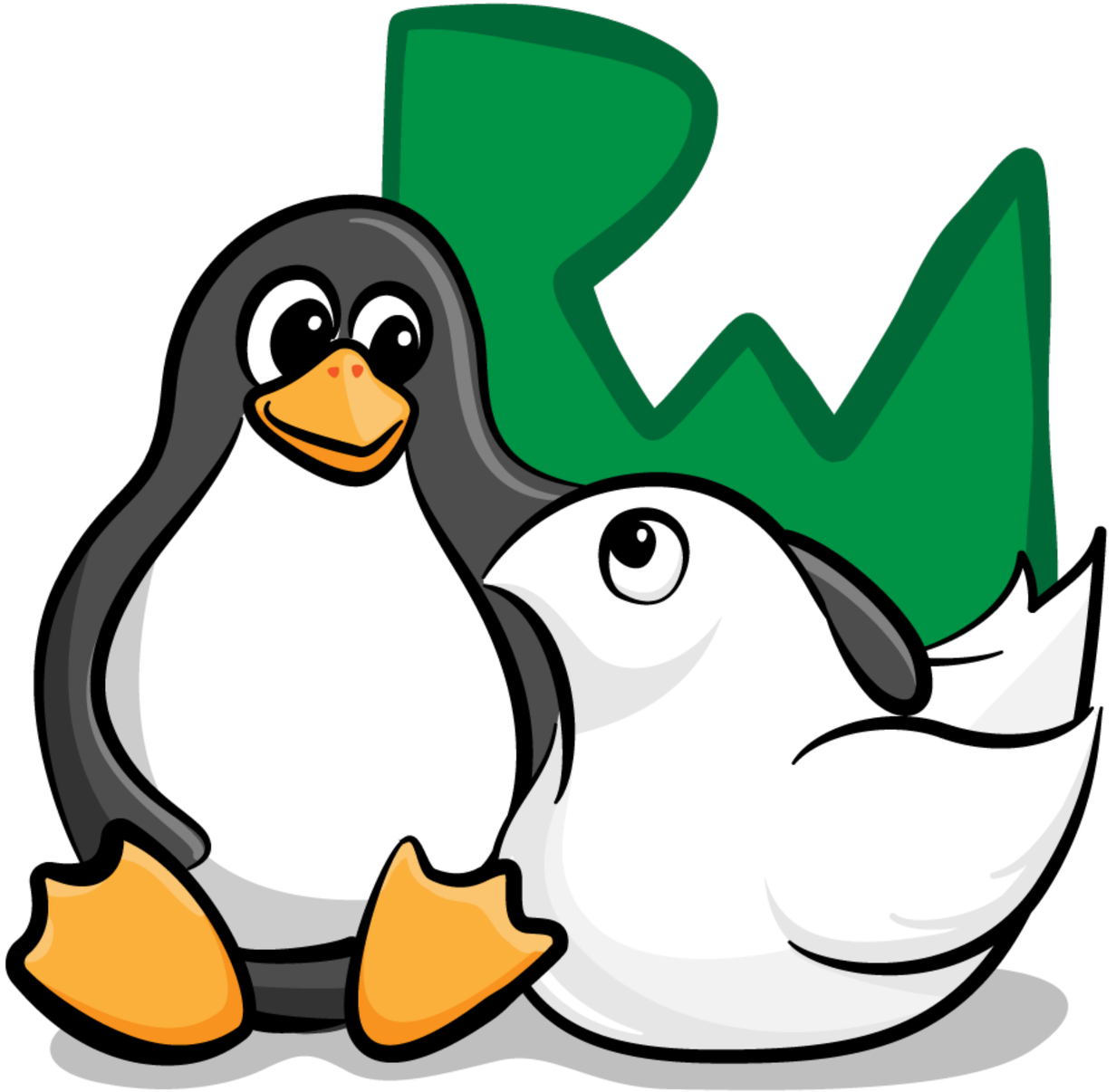
Agenda

Linked List Data Structure



Swift collection protocols





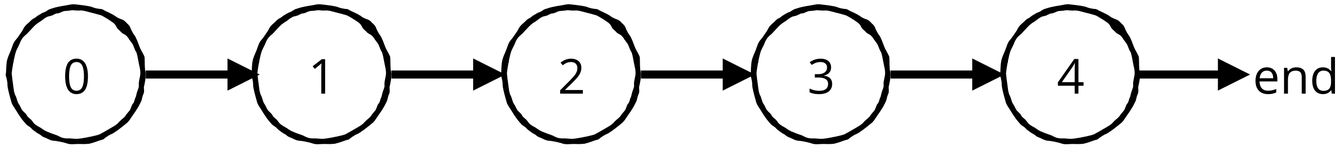
Linked List

What is a Linked List?

Linked List Data Structure

push(0)

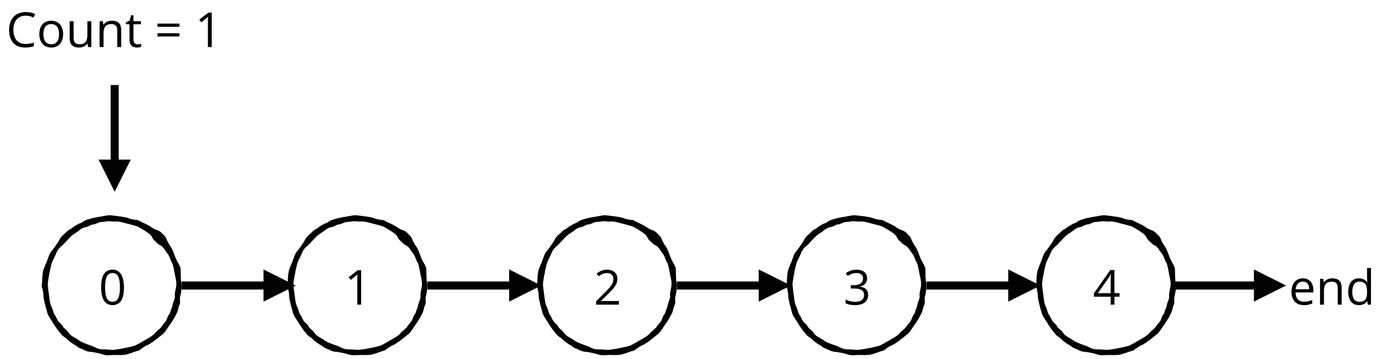
add value 0 to LL

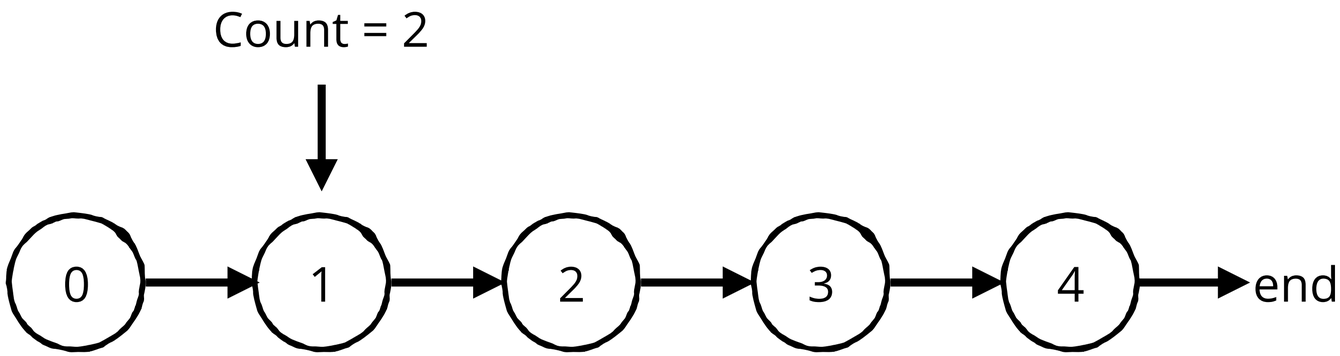


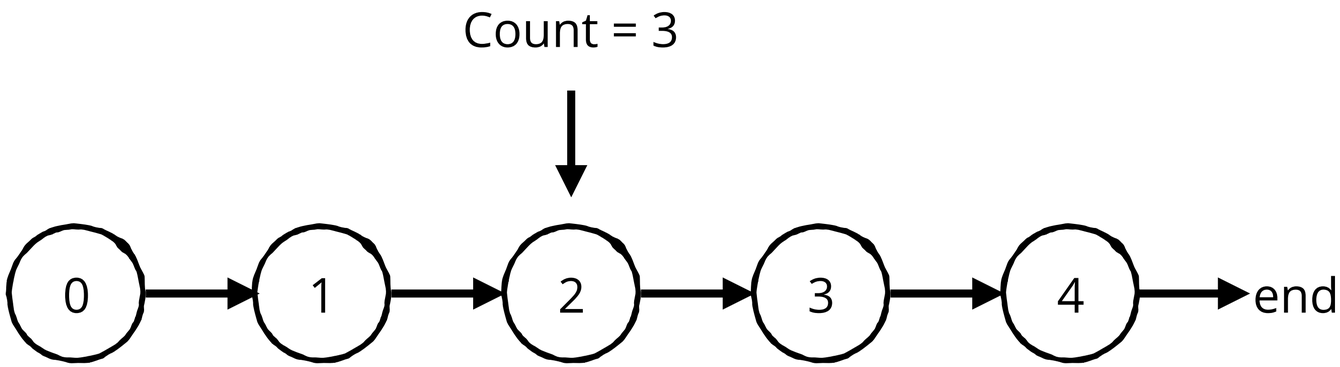
Demo 1
Modelling a Linked List using Enums

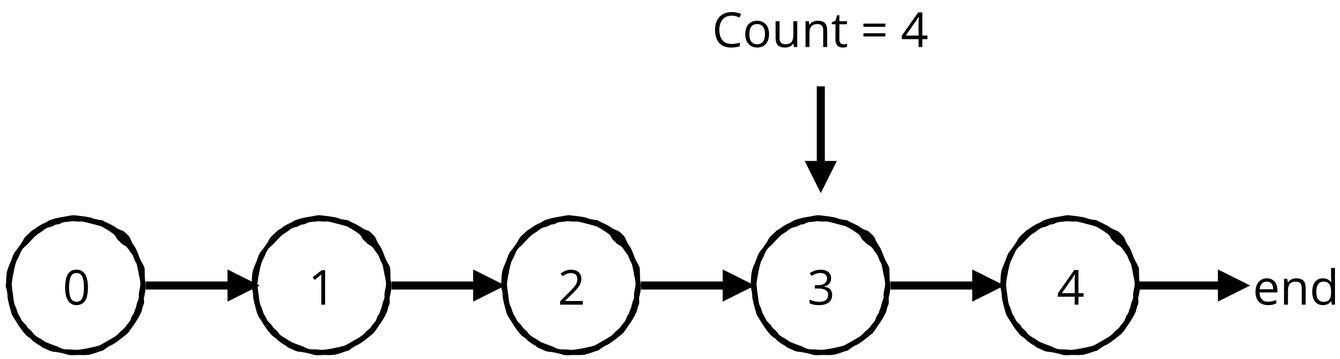
Warming up

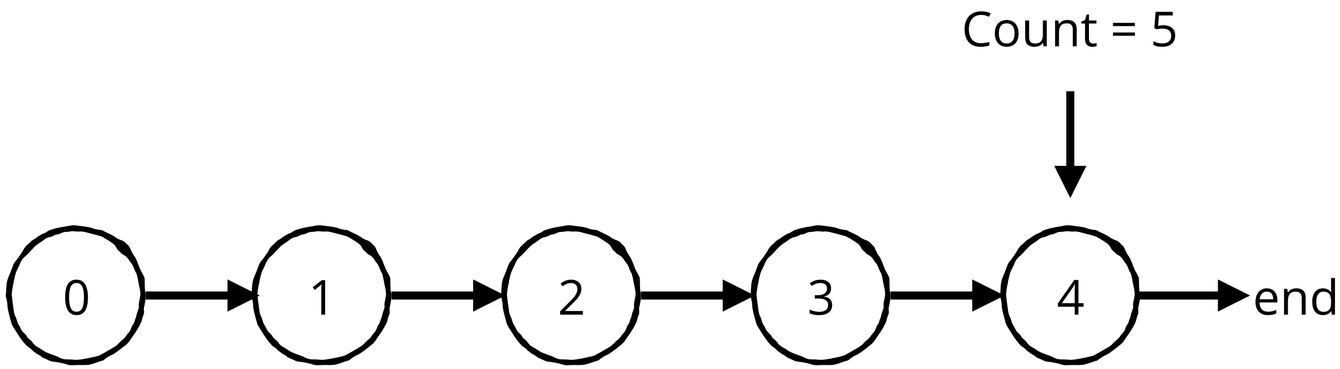
Create a computed property that returns the count of the Linked List

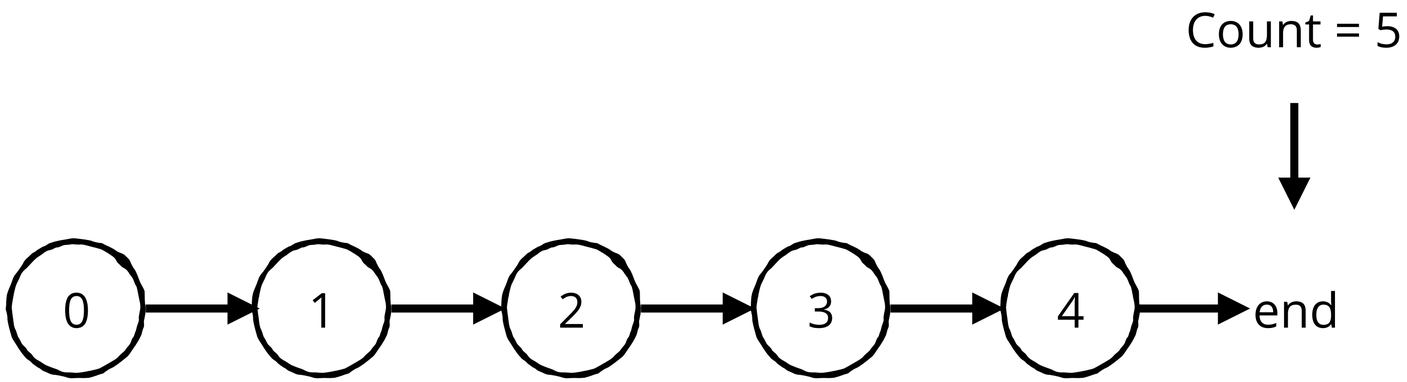

Count = 1
0
1
2
3
4
end


Count = 2
0
1
2
3
4
end


Count = 3
0
1
2
3
4
end


Count = 4
0
1
2
3
4
end


Count = 5
0
1
2
3
4
end


Count = 5
0
1
2
3
4
end

```swift
var count: Int {
  var count = 0
  var current = self

  while case .node(_, let next) = current {
    count += 1
    current = next
  }
  return count
}
```

Bad stuff

```
example(of: "Extracting even numbers from list") {
  var current = list
  var evenNumbers: [Int] = []

  while case let .node(data, next) = current {
    current = next
    if data % 2 == 0 {
      evenNumbers.append(data)
    }
  }

  print(evenNumbers)
}
```

```
example(of: "Summing the elements in list") {
  var current = list
  var sum = 0

  while case let .node(data, next) = current {
    current = next
    sum += data
  }

  print(sum)
}
```

RandomAccessCollection

BidirectionalCollection

Swift Collection Protocols

Sequence

Collection

MutableCollection

RangeReplaceableCollection

Tier 1

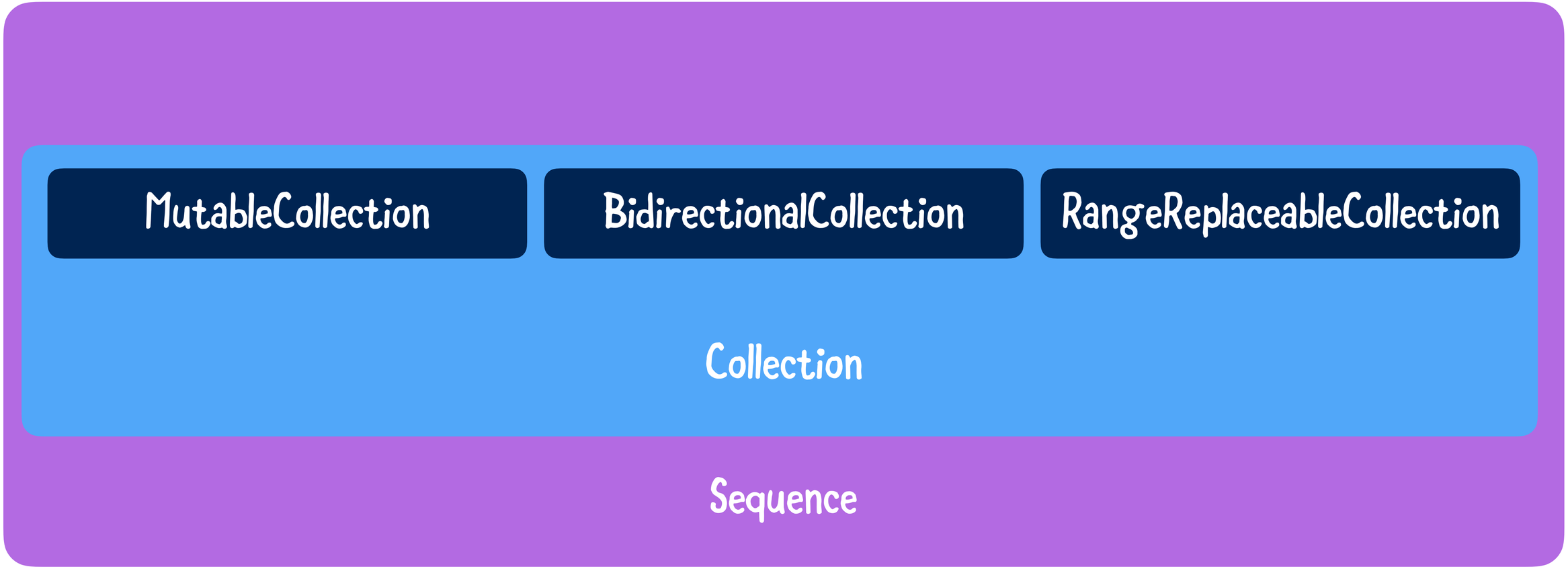
Sequence

Tier 2

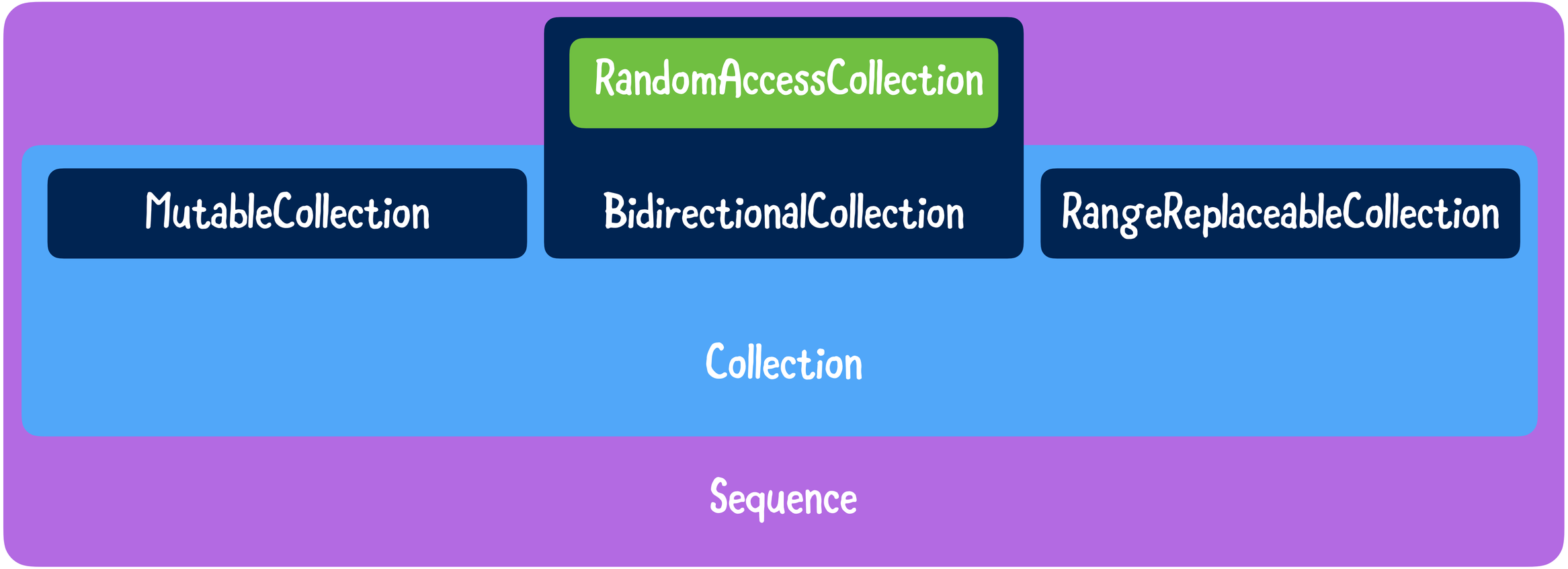
Collection

Sequence

Tier 3


MutableCollection
BidirectionalCollection
RangeReplaceableCollection
Collection
Sequence

Tier 4


RandomAccessCollection
MutableCollection
BidirectionalCollection
RangeReplaceableCollection
Collection
Sequence

SEQUENCE

Tier 1 - Sequence

map

Higher ordered functions

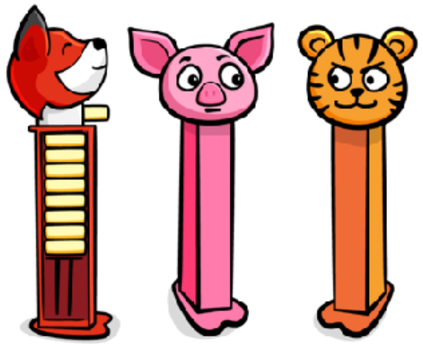
filter reduce

for num in numbers

Improved for loop

enumerated

forEach



Iterator

sort / sorted

Helpful things

min max

reversed

split

fill

clip

And more...

join

lexicographicallyPrecedes

Iterators

The goal: Dispense values 1 at a time

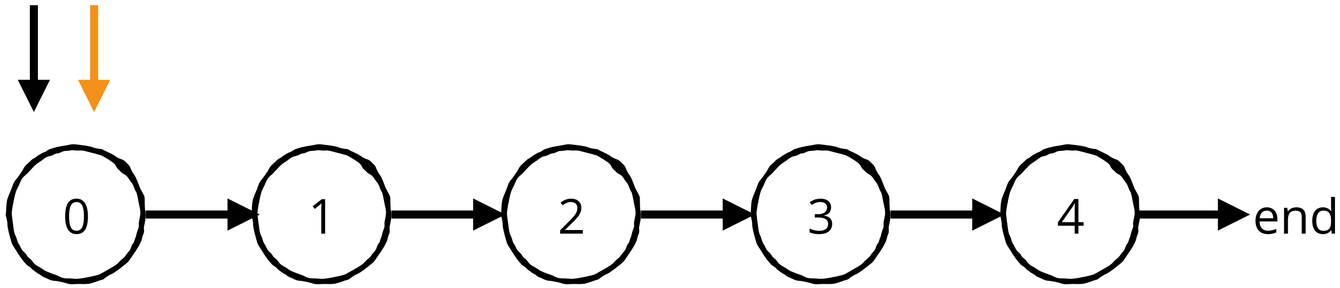
And adopts the `IteratorProtocol` protocol

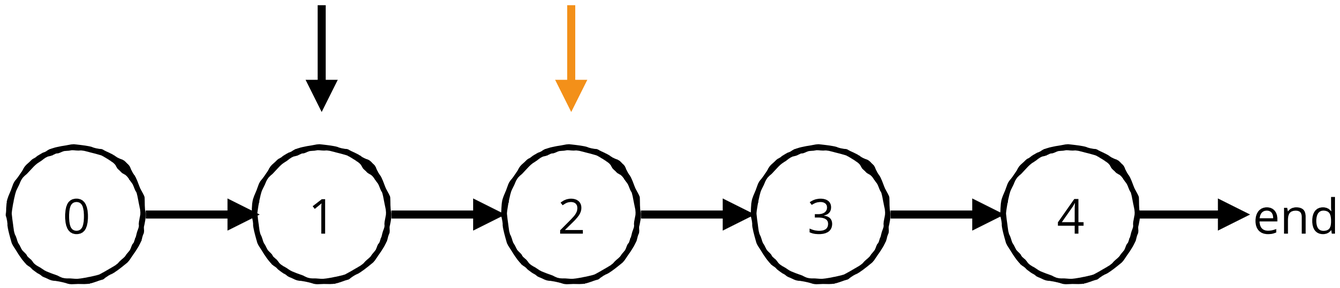
Demo 2
Conforming to Sequence

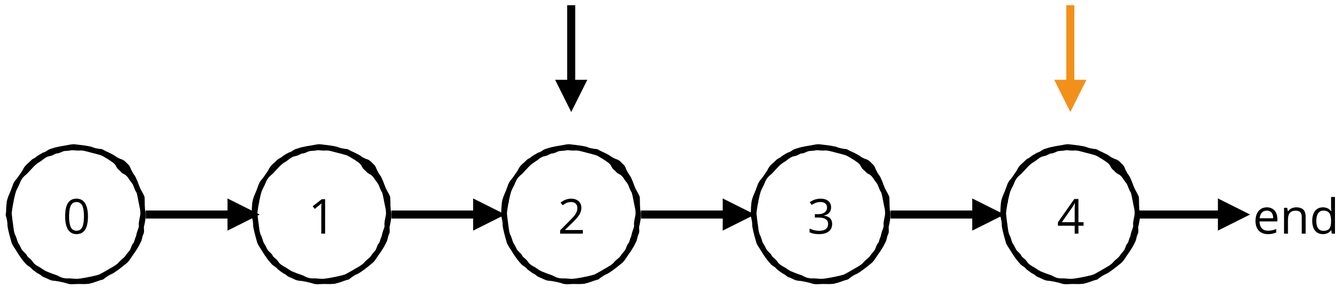
Challenge 1

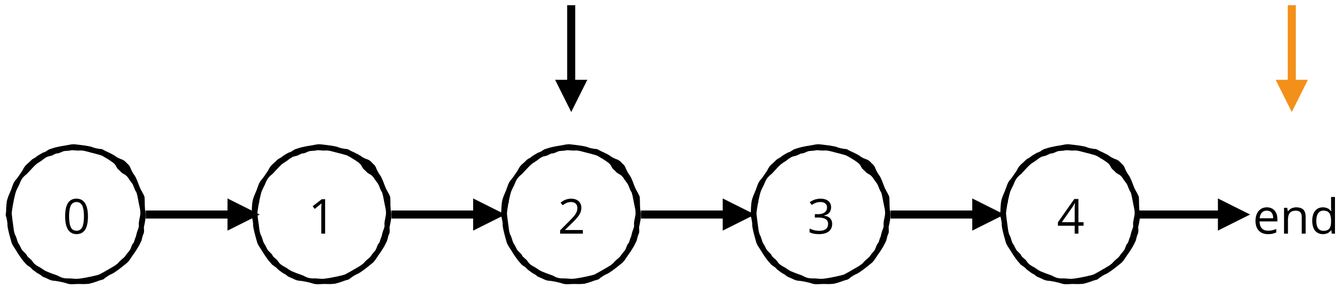
Given a linked list, find the middle value of the list.

Bonus: Can you implement it as an extension of sequence protocol?


0
1
2
3
4
end

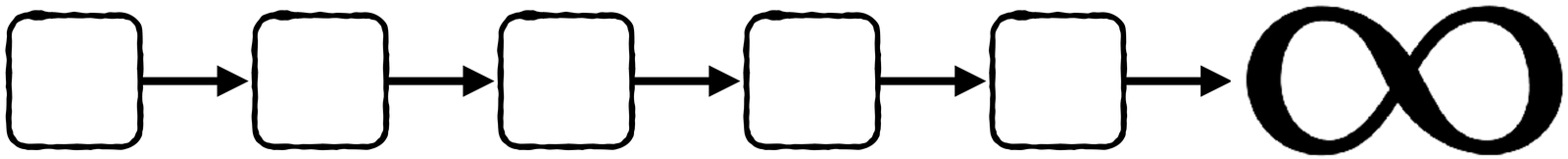

0
1
2
3
4
end


0
1
2
3
4
end

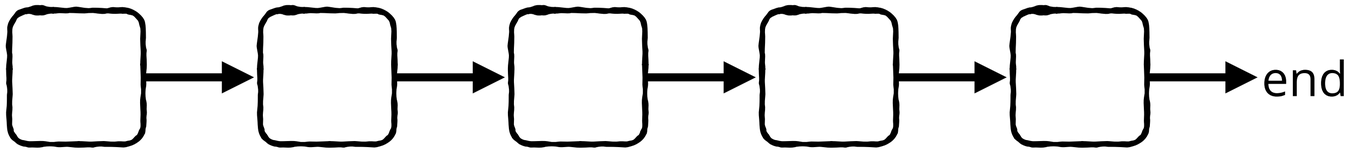

0
1
2
3
4
end

```
extension Sequence {
  var middle: Element? {
    var iterator1 = makeIterator()
    var iterator2 = makeIterator()

    while iterator2.next() != nil {
      guard iterator2.next() != nil else { break }
      iterator1.next()
    }

    return iterator1.next()
  }
}
```

A **Sequence** can be ***infinite***



A **Collection** is ***finite***



Tier 2 - Collection

map

Higher ordered functions

index(of:)

subscript [ ]

for num in numbers

Improved for loop

filter

reduce

forEach

enumerated

Beginnings
Indices [ ] Finite-ness
End

isEmpty

first

split

clip

And more...

fill

join

lexicographicallyPrecedes

count

sort / sorted

min

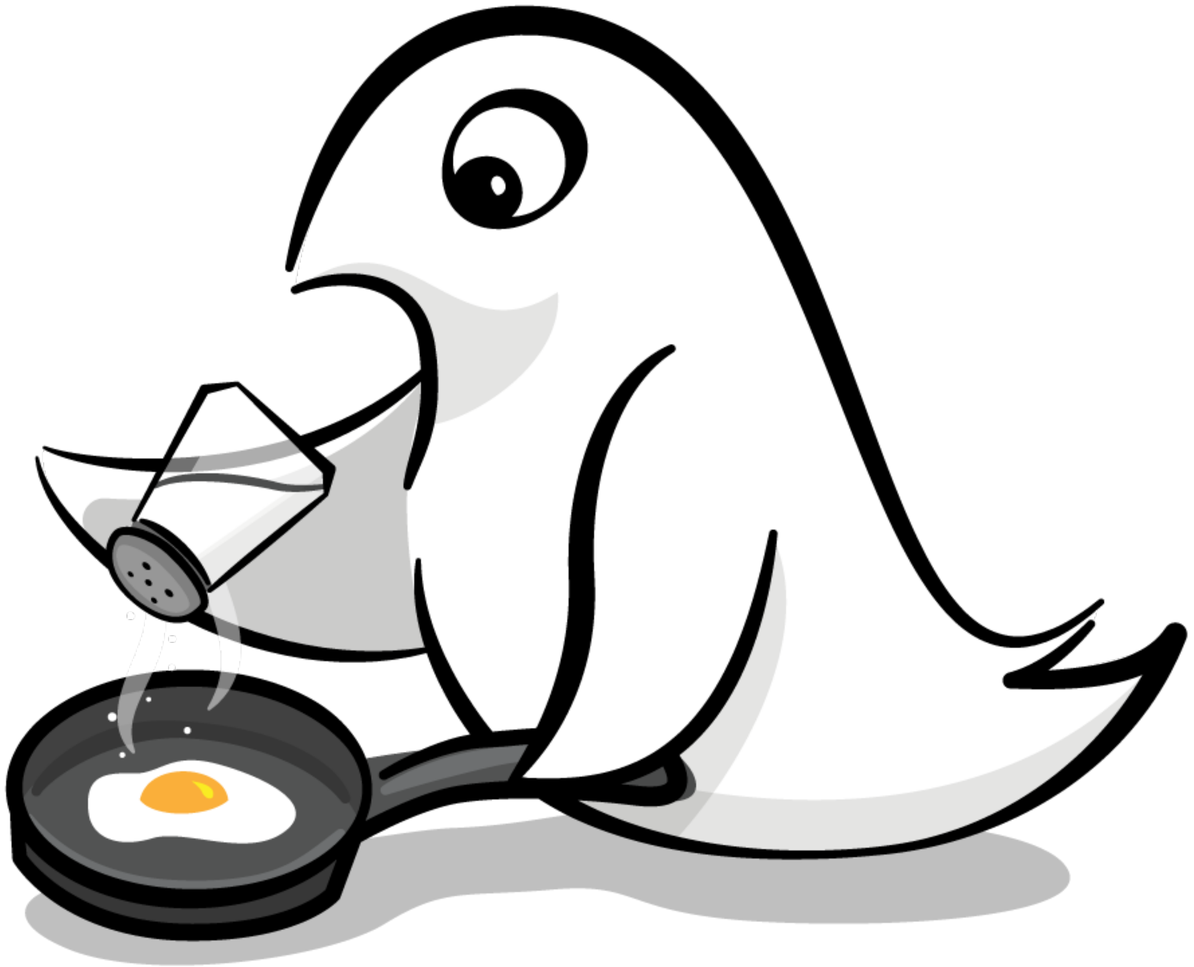
Helpful things

max

reversed

Demo 3
Conforming to Collection

(Audience participation required)

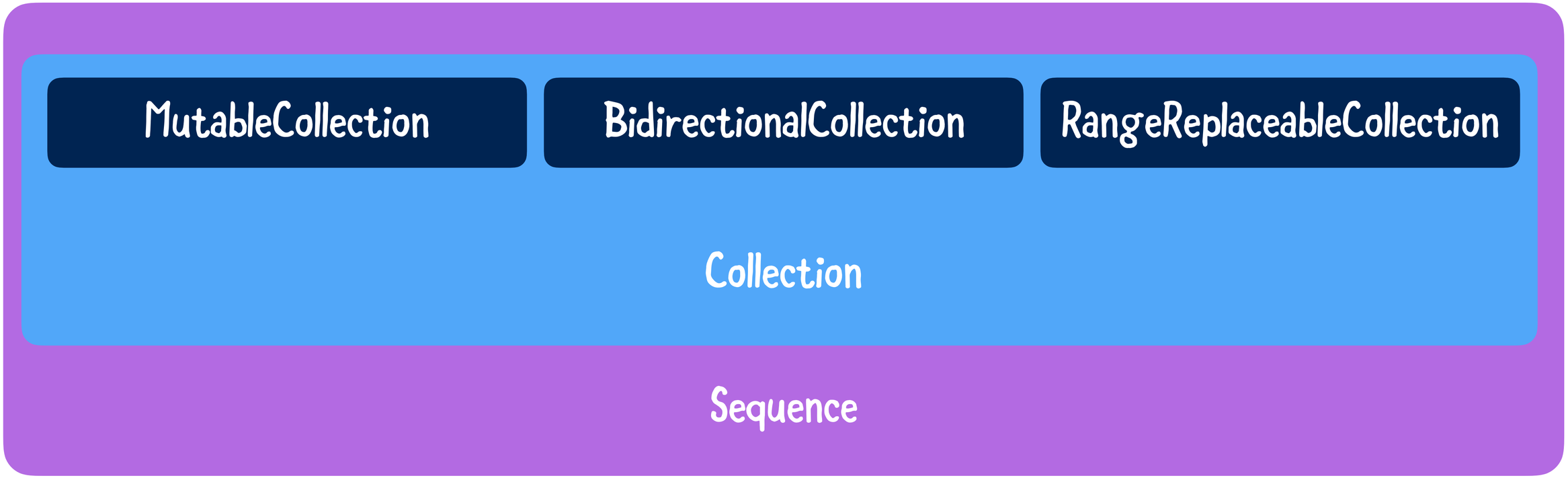


Challenge 2

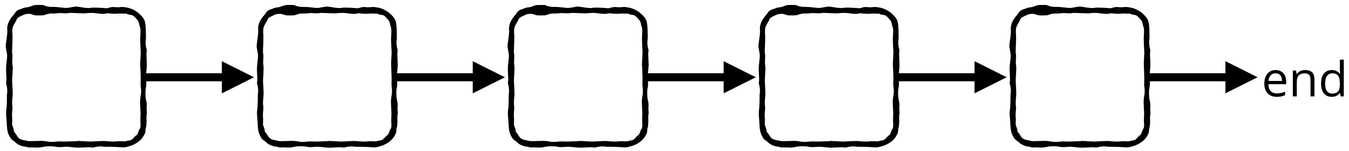
Create a function that checks if a collection is in another collection. If it is, return the start index of the matching collection.

```swift
extension Collection where Element: Equatable {

  func index<Elements: Collection>(for elements: Elements)
    -> Index? where Elements.Element == Element {

    first: for index in indices {
      var current = index
      for element in elements {
        guard element == self[current] else { continue first }
        formIndex(after: &current)
      }
      return index
    }
    return nil
  }
}
```

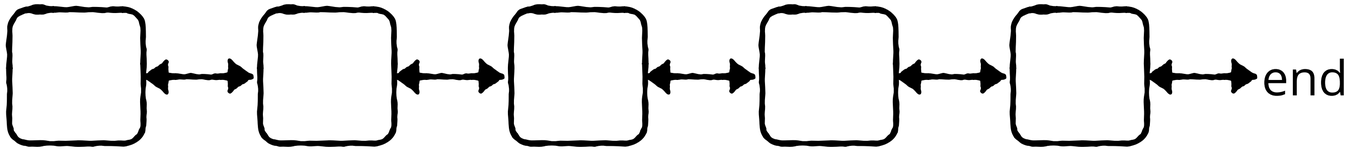
Tier 3


MutableCollection
BidirectionalCollection
RangeReplaceableCollection
Collection
Sequence

A **Collection** is ***finite***



A **BidirectionalCollection** can traverse **backwards.**



Tier 3 - BidirectionalCollection

map

Higher ordered functions

filter reduce

index(of:)

last

for num in numbers

Improved for loop

forEach enumerated

count

reversed

subscript [ ]

suffix

first

index(before:)

Beginnings

Indices [ ] Finite-ness

End

sort / sorted

min Helpful things max

reversed

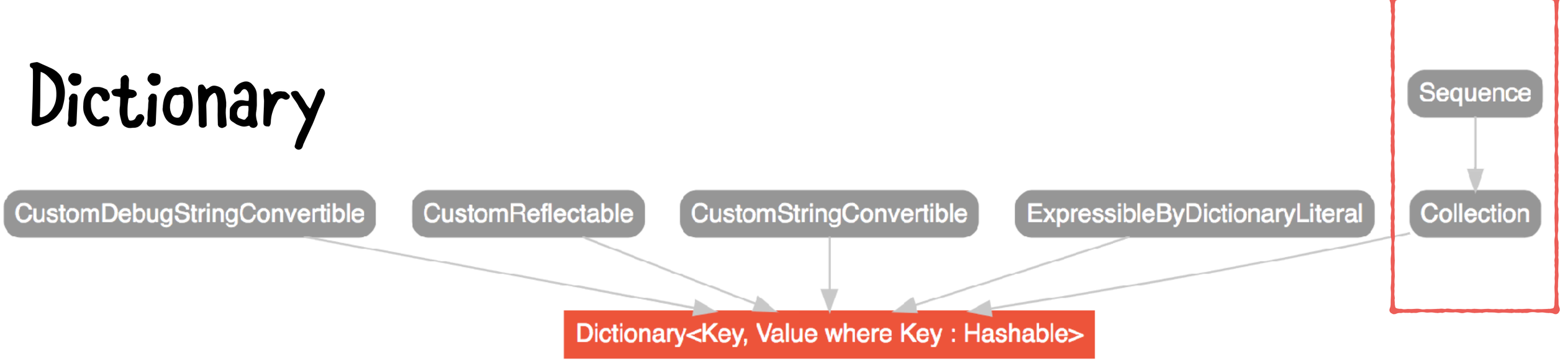
split

clip And more... fill

join lexicographicallyPrecedes

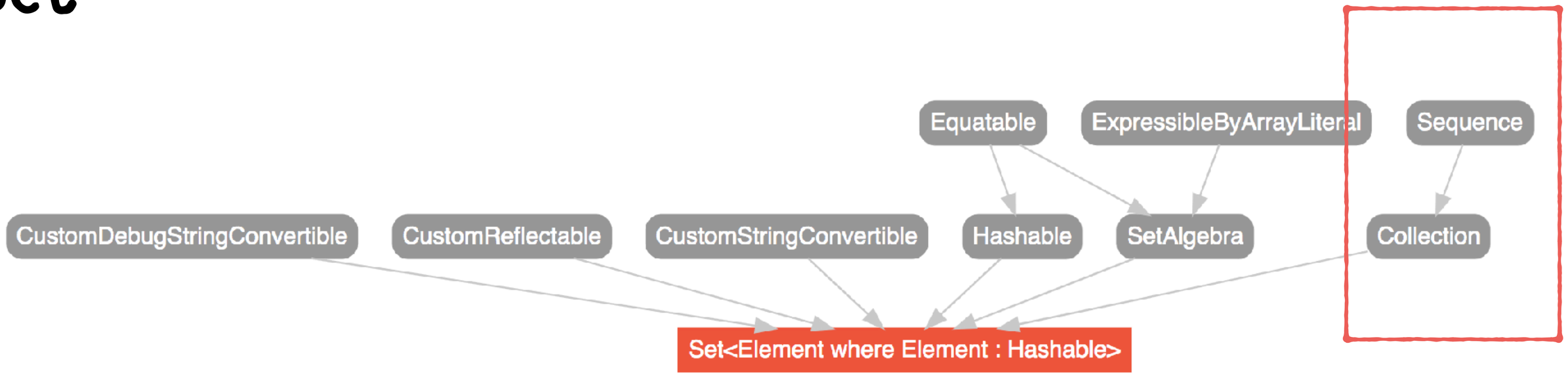
```
extension ReversedCollection: BidirectionalCollection {

  @_inlineable
  public var startIndex: Index {
    return Index(_base.endIndex)
  }

  @_inlineable
  public var endIndex: Index {
    return Index(_base.startIndex)
  }

  @_inlineable
  public func index(after i: Index) -> Index {
    return Index(_base.index(before: i.base))
  }

  @_inlineable
  public func index(before i: Index) -> Index {
    return Index(_base.index(after: i.base))
  }
 …
}
```

Dictionary



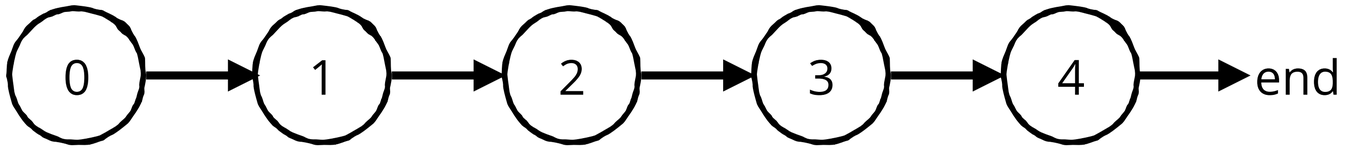
Unordered collections

Set

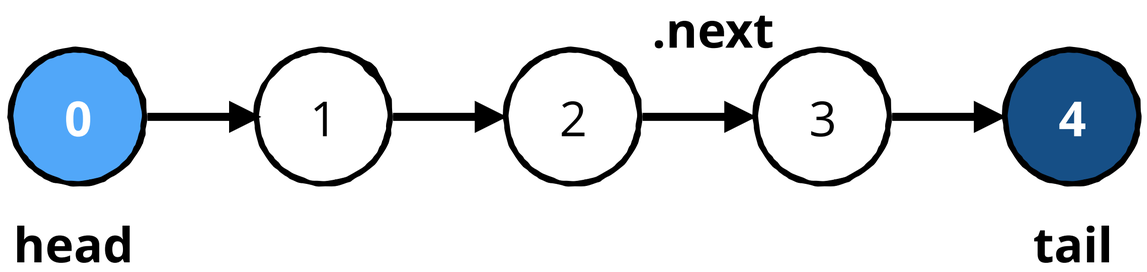


index(before: Index) -> Index

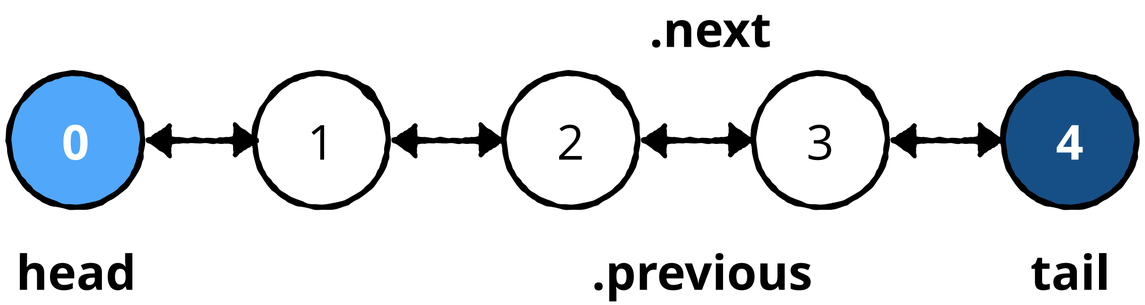
Enum-based Linked List



Singly Linked List



Doubly Linked List



Demo 4
Conforming to BidirectionalCollection

Challenge 3

Given a word, find out if it is a palindrome.

Bonus: Can you implement it as an extension of the BidirectionalCollection protocol?

RADAR

palindrome

RWDEVCON

not palindrome

```swift
extension BidirectionalCollection where Element: Equatable {
    var isPalindrome: Bool {
        // Your solution here!
    }
}
```

Hints

```swift
func formIndex(before i: inout Int)
```

```swift
func formIndex(after i: inout Int)
```

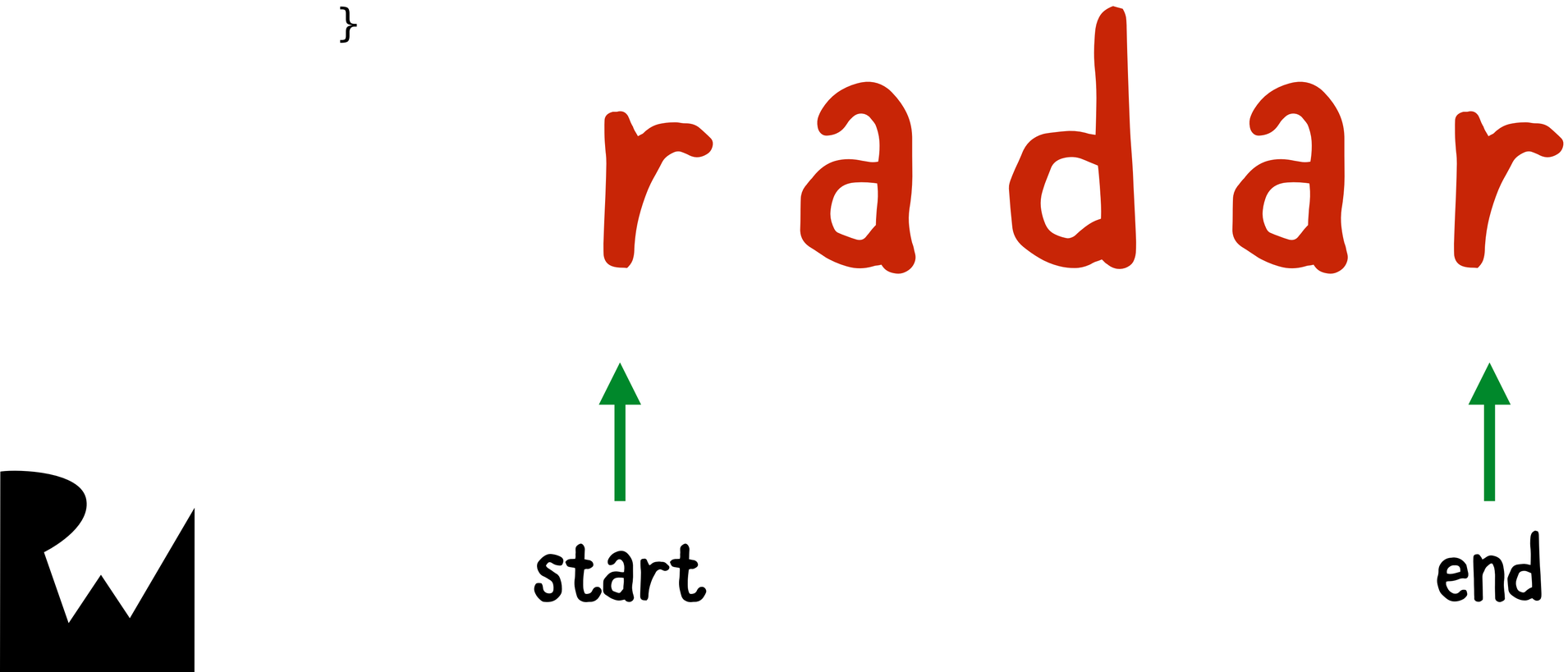
```swift
func index(before i: Index) -> Index
```

```swift
var index = 10
[].formIndex(after: &index)
print(index) // 11

var index2 = 10
[].formIndex(before: &index2)
print(index2) // 9

let index3 = [].index(before: -1)
print(index3) // -2
```

Go over solution

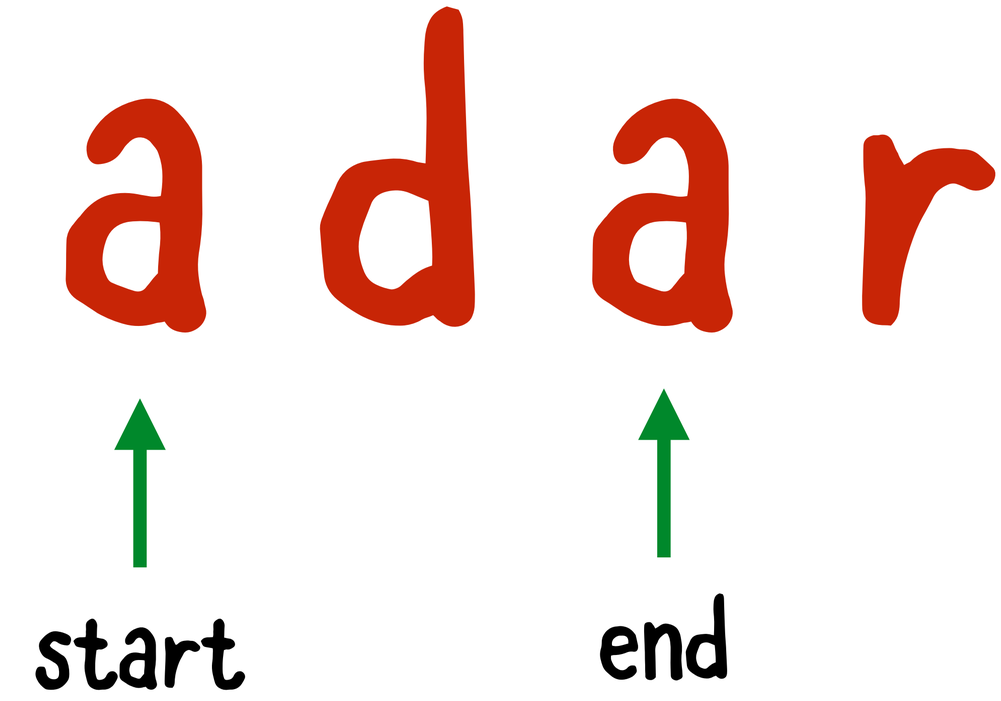
```
extension BidirectionalCollection where Element: Equatable {
  var isPalindrome: Bool {
    var start = startIndex
    var end = index(before: endIndex)
    while start < end {
      guard self[start] == self[end] else { return false }
      formIndex(after: &start)
      formIndex(before: &end)
    }
    return true
  }
}
```

```
extension BidirectionalCollection where Element: Equatable {
  var isPalindrome: Bool {
    var start = startIndex
    var end = index(before: endIndex)
    while start < end {
      guard self[start] == self[end] else { return false }
      formIndex(after: &start)
      formIndex(before: &end)
    }
    return true
  }
}
```

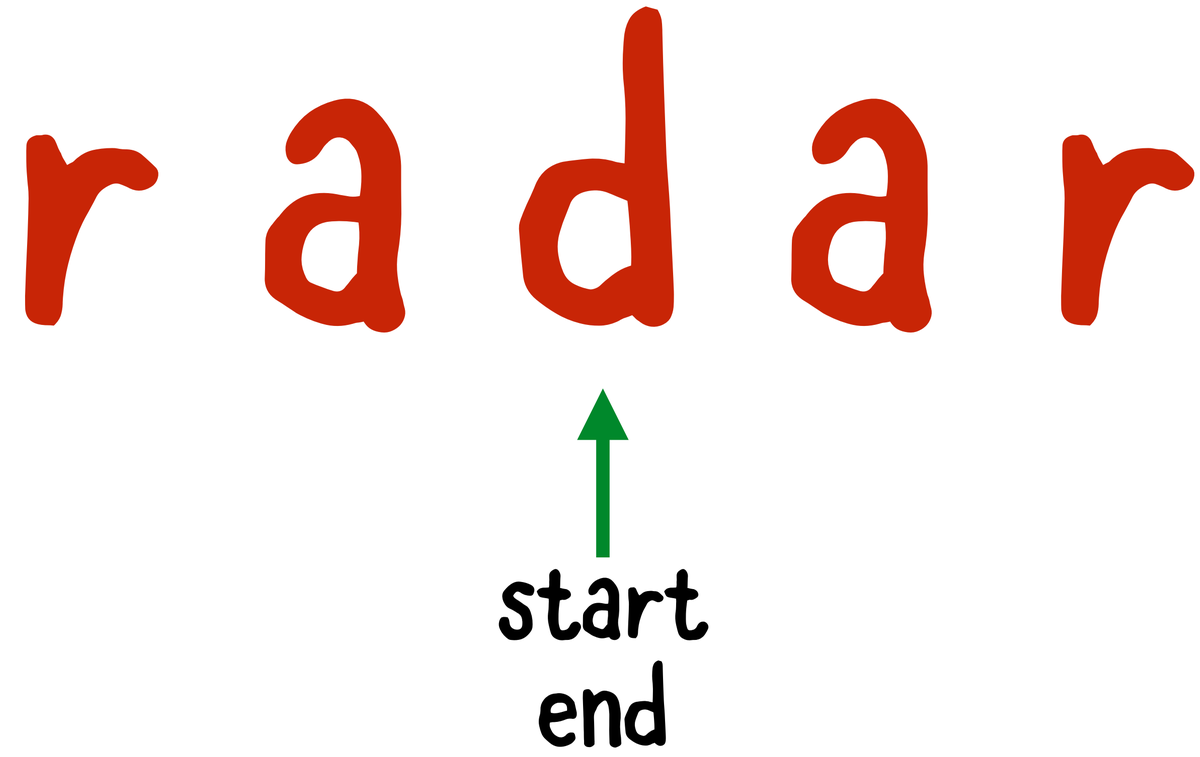


```
extension BidirectionalCollection where Element: Equatable {
  var isPalindrome: Bool {
    var start = startIndex
    var end = index(before: endIndex)
    while start < end {
      guard self[start] == self[end] else { return false }
      formIndex(after: &start)
      formIndex(before: &end)
    }
    return true
  }
}
```

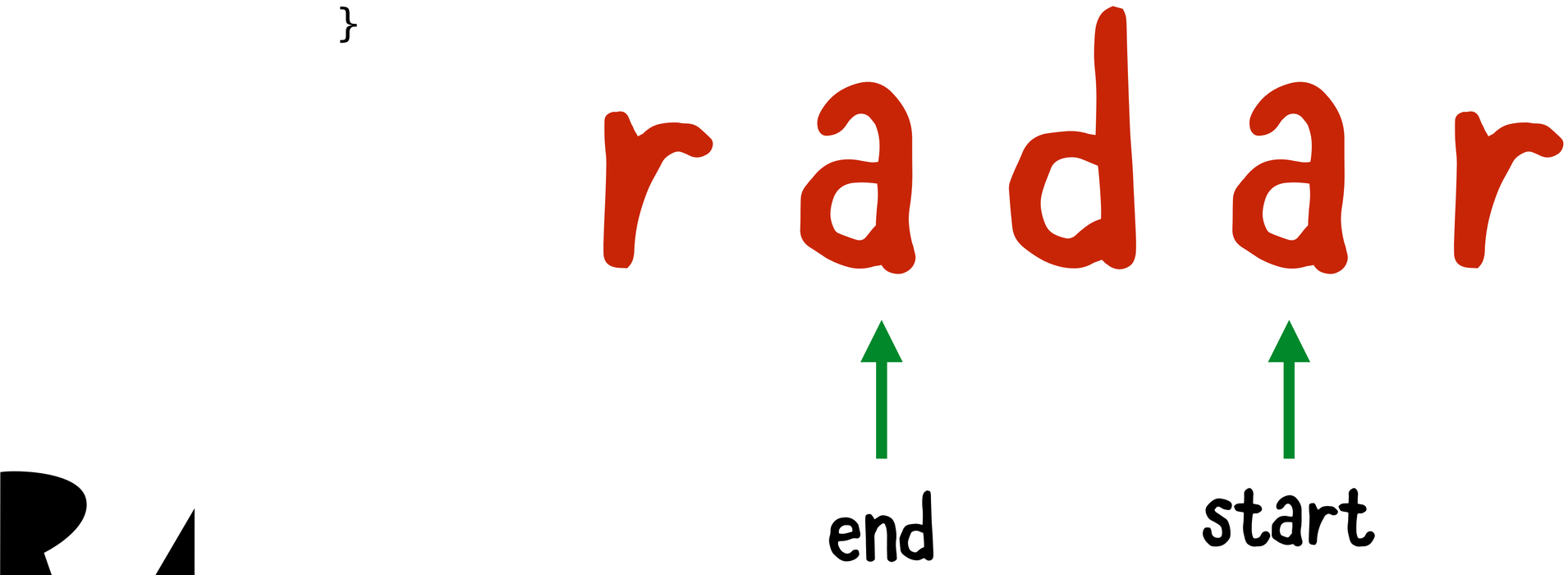
```
extension BidirectionalCollection where Element: Equatable {
  var isPalindrome: Bool {
    var start = startIndex
    var end = index(before: endIndex)
    while start < end {
      guard self[start] == self[end] else { return false }
      formIndex(after: &start)
      formIndex(before: &end)
    }
    return true
  }
}
```



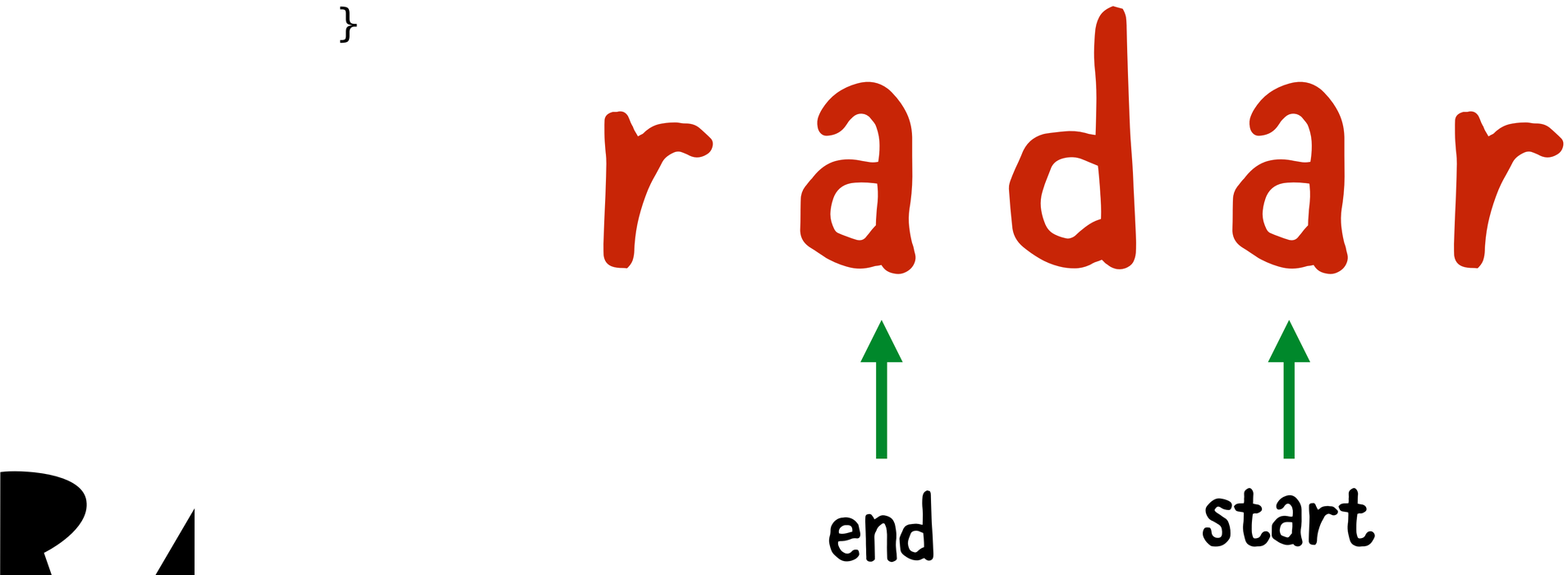
```
extension BidirectionalCollection where Element: Equatable {
  var isPalindrome: Bool {
    var start = startIndex
    var end = index(before: endIndex)
    while start < end {
      guard self[start] == self[end] else { return false }
      formIndex(after: &start)
      formIndex(before: &end)
    }
    return true
  }
}
```



```
extension BidirectionalCollection where Element: Equatable {
  var isPalindrome: Bool {
    var start = startIndex
    var end = index(before: endIndex)
    while start < end {
      guard self[start] == self[end] else { return false }
      formIndex(after: &start)
      formIndex(before: &end)
    }
    return true
  }
}
```

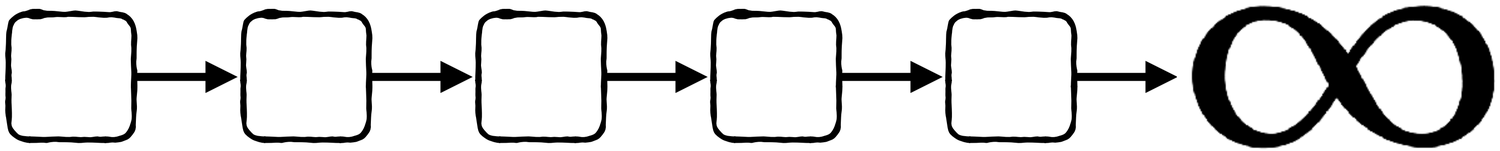


```
extension BidirectionalCollection where Element: Equatable {
  var isPalindrome: Bool {
    var start = startIndex
    var end = index(before: endIndex)
    while start < end {
      guard self[start] == self[end] else { return false }
      formIndex(after: &start)
      formIndex(before: &end)
    }
    return true
  }
}
```

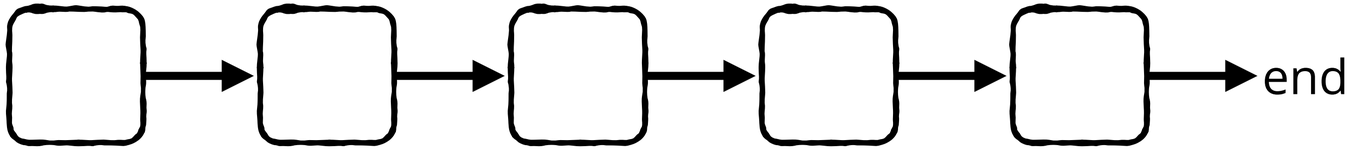


RECAP

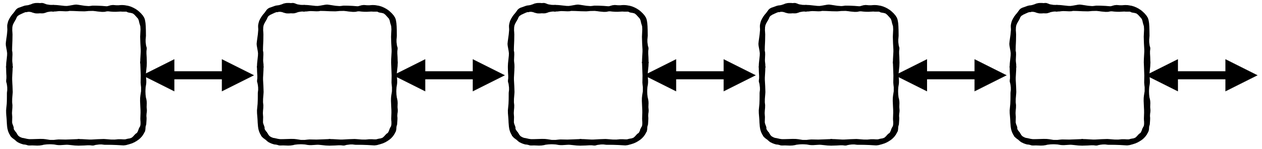
A **Sequence** can be



A **Collection** is ***finite***

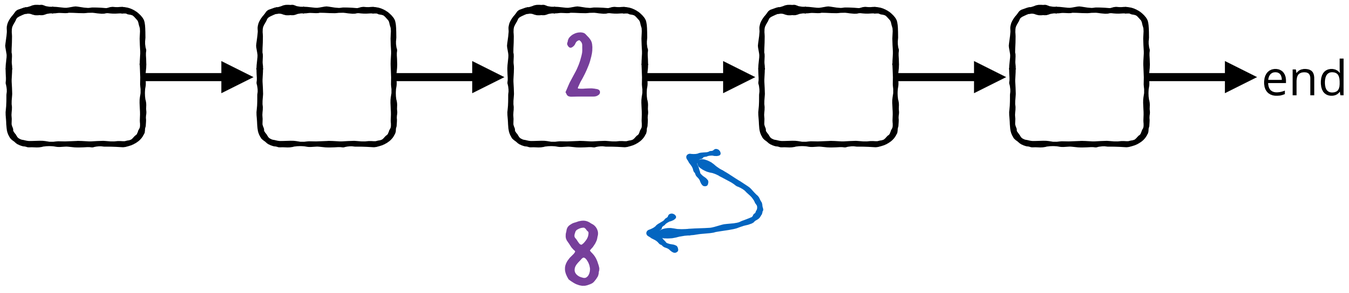


A **BidirectionalCollection** supports backwards traversal




MOVIECLIPS.com

Tier 3 - MutableCollection



Change the value of elements in a collection.

swapAt(_:)

sort()

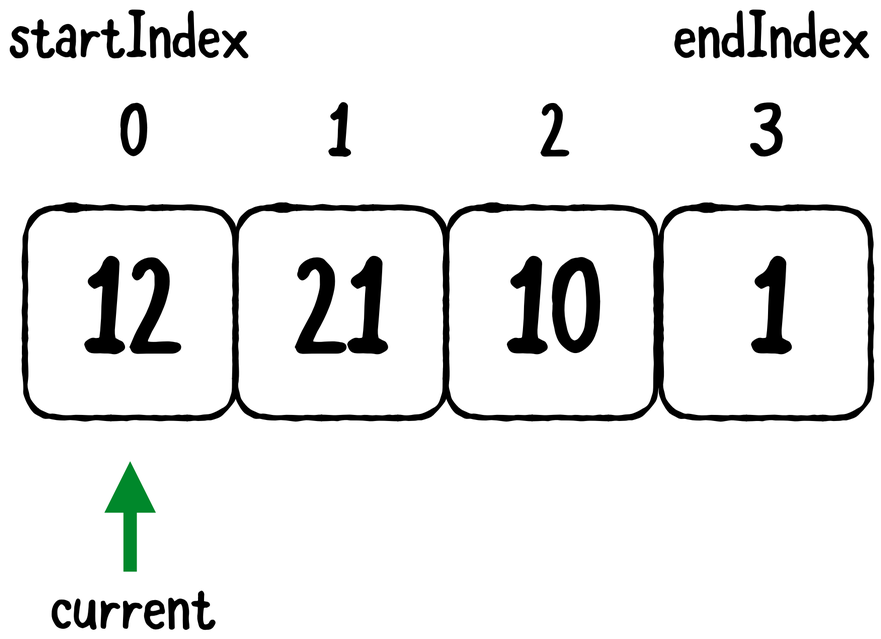
subscript [ ]

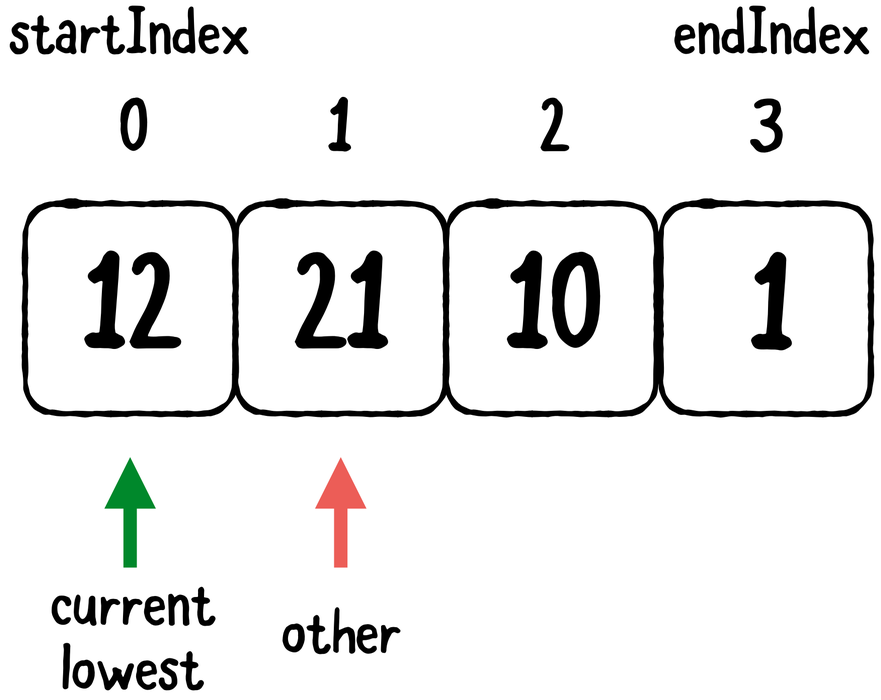
student[i] = “Vincent”

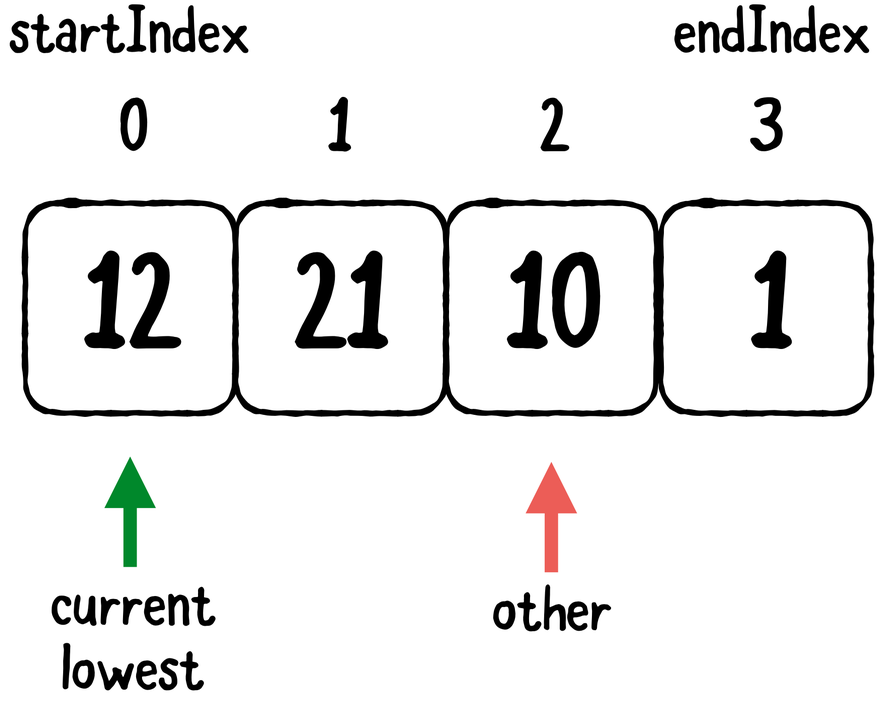
Demo 5
Conforming to MutableCollection

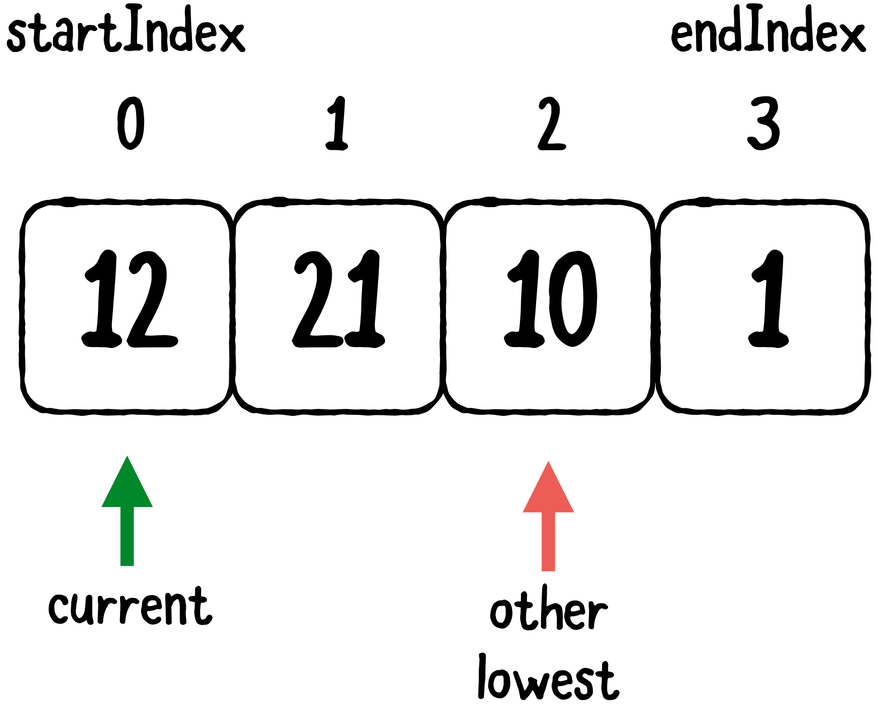
Challenge 4

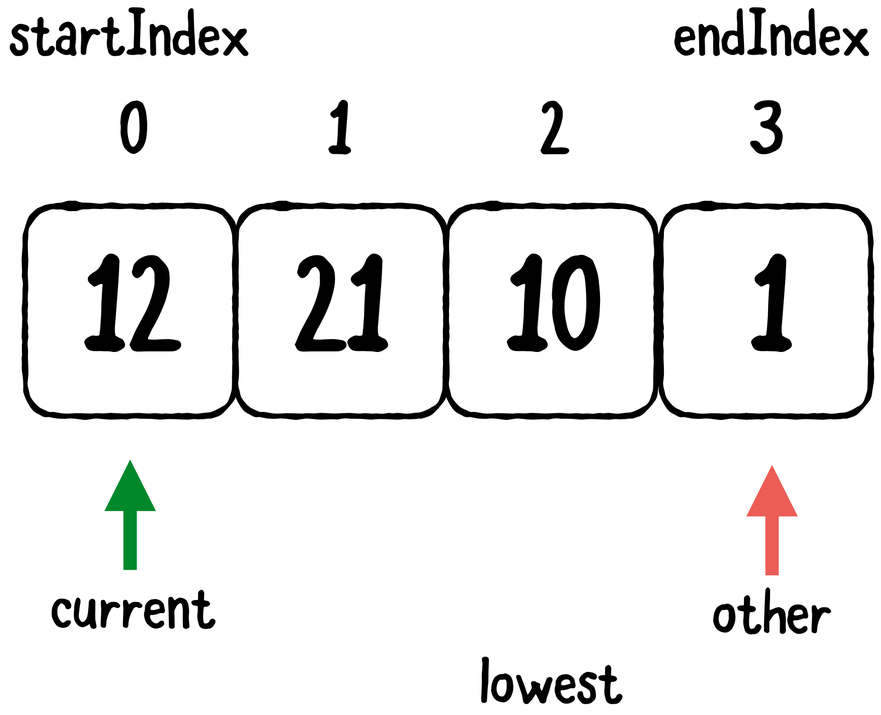
Generalize Selection Sort as an extension of MutableCollection

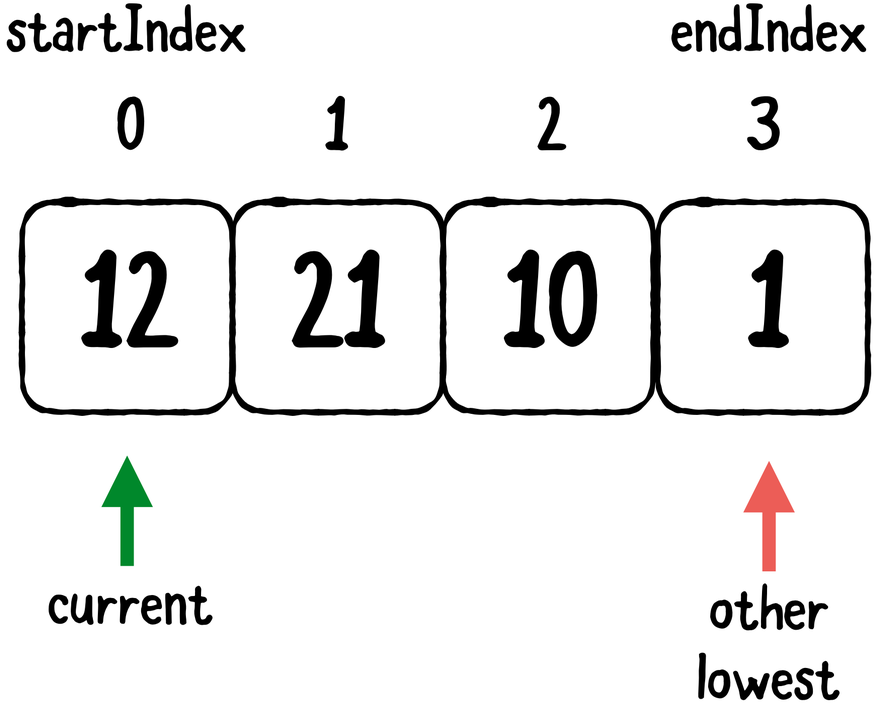

startIndex
endIndex
0
1
2
3
12
21
10
1
current

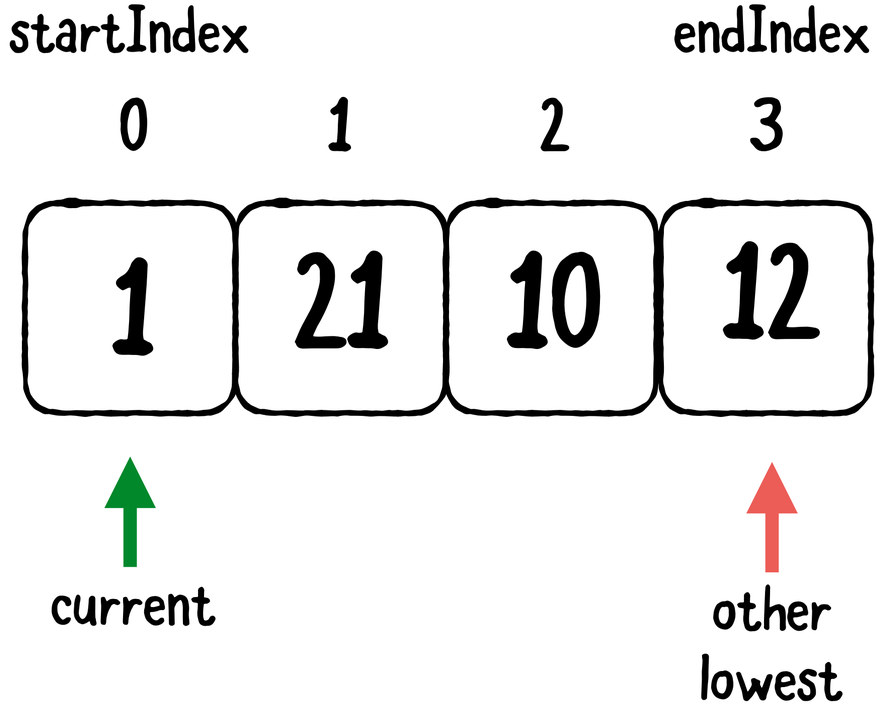

startIndex
endIndex
0
1
2
3
12
21
10
1
current
lowest
other

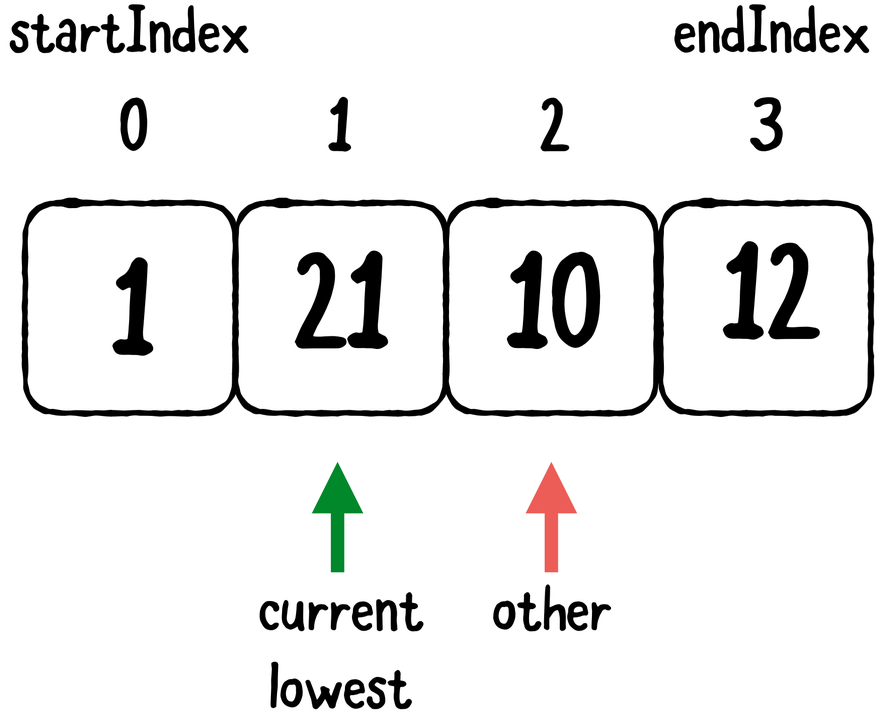

startIndex
endIndex
0
1
2
3
12
21
10
1
current
lowest
other

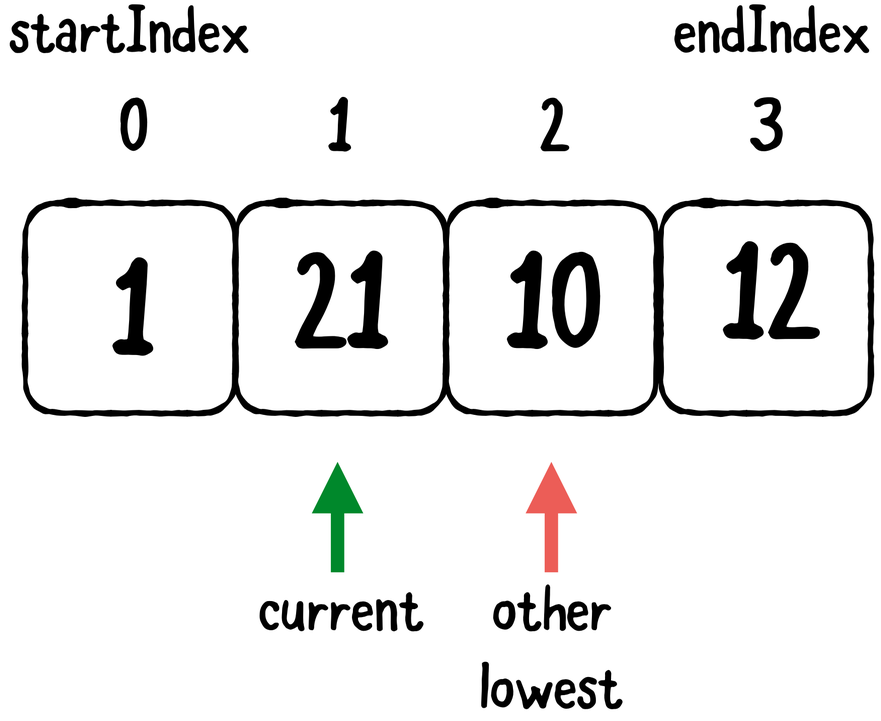

startIndex
endIndex
0
1
2
3
12
21
10
1
current
other
lowest

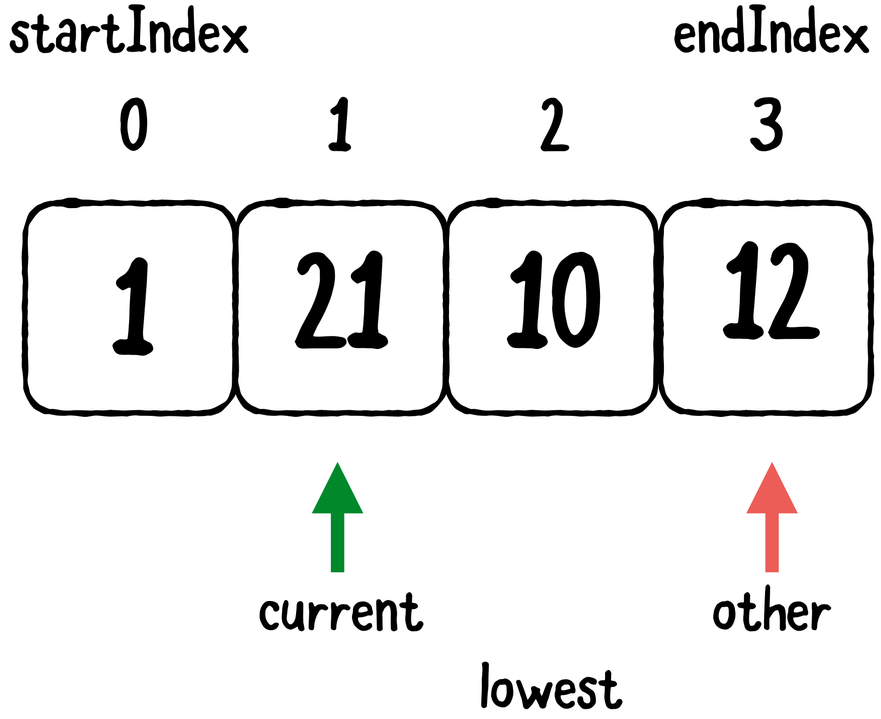

startIndex
endIndex
0
1
2
3
12
21
10
1
current
other
lowest

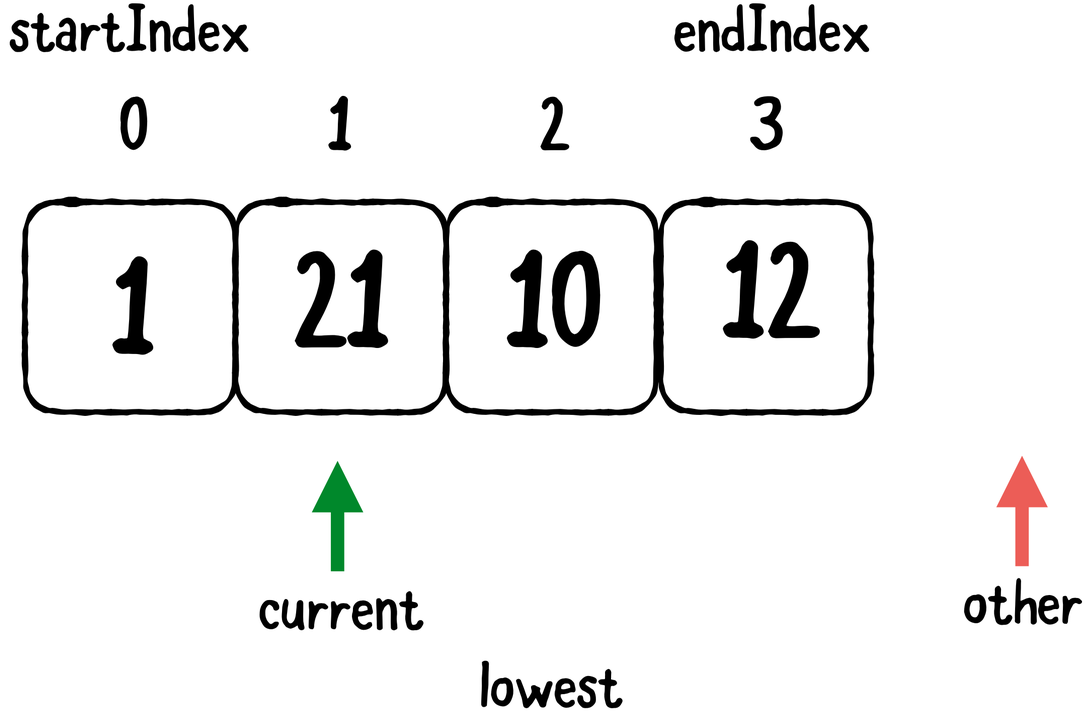

startIndex
endIndex
0
1
2
3
12
21
10
1
current
other
lowest

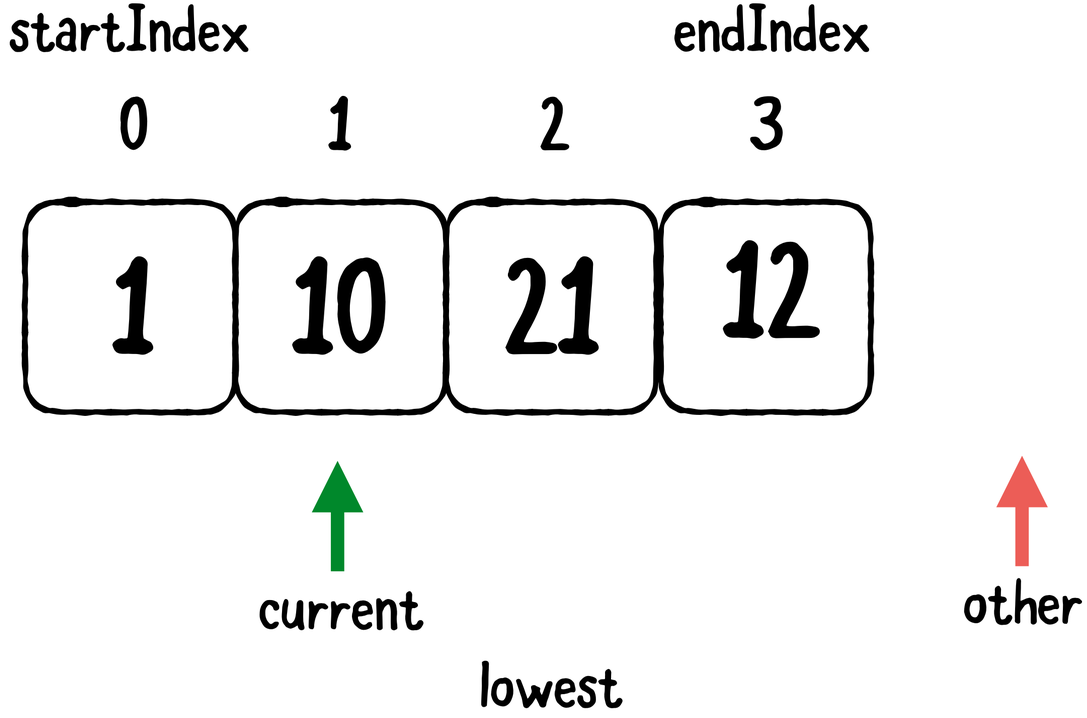

startIndex
endIndex
0
1
2
3
1
21
10
12
current
other
lowest

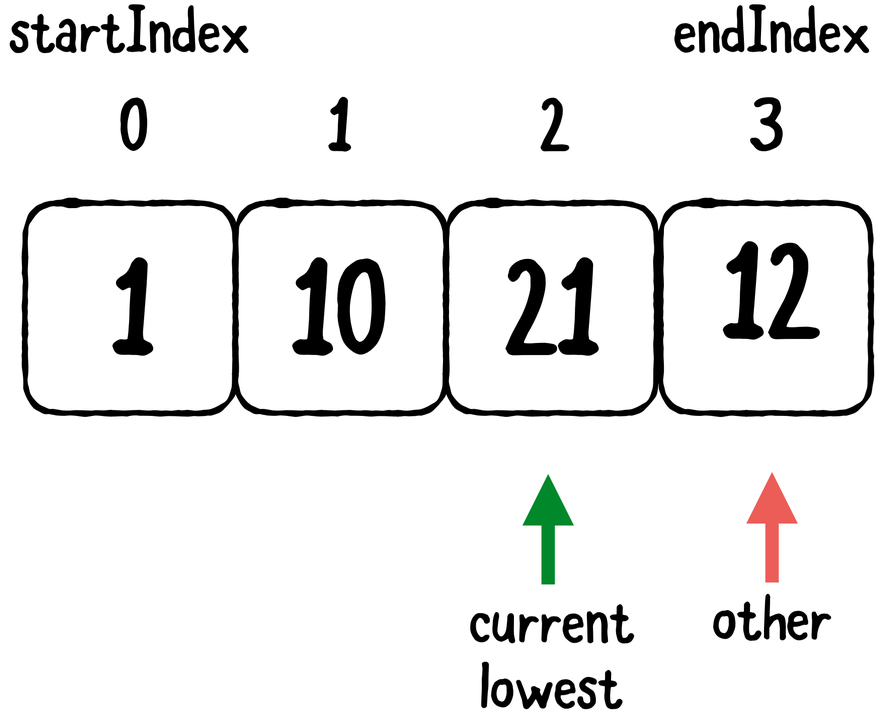

startIndex
endIndex
0
1
2
3
1
21
10
12
current
lowest
other

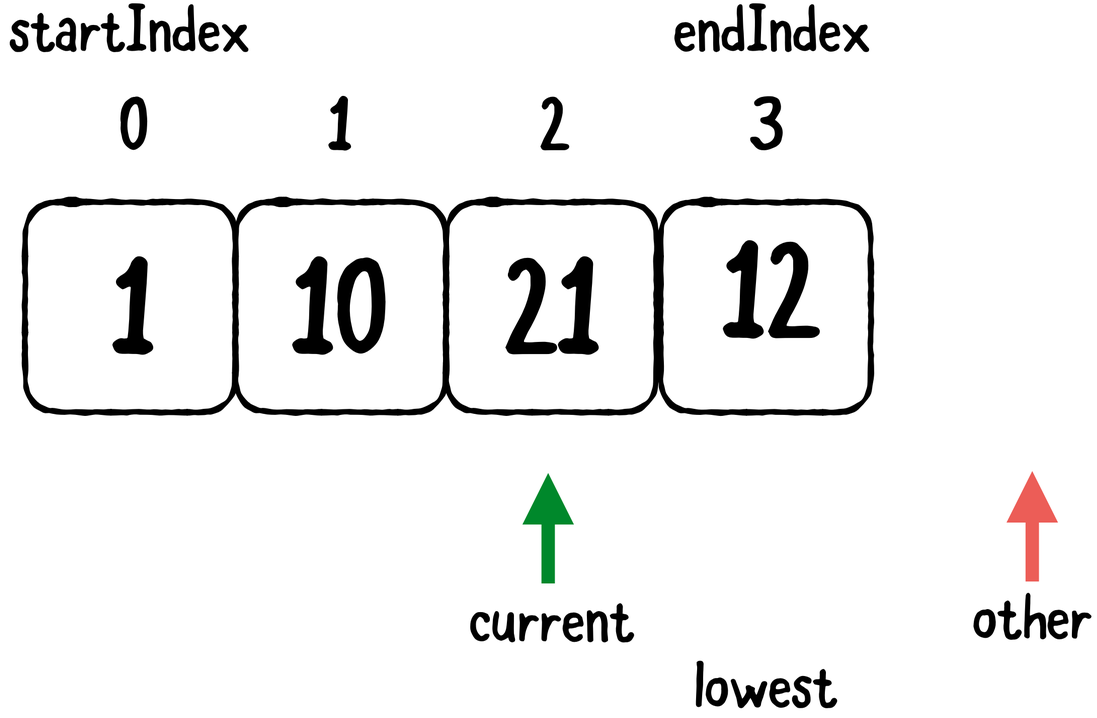

startIndex
endIndex
0
1
2
3
1
21
10
12
current
other
lowest

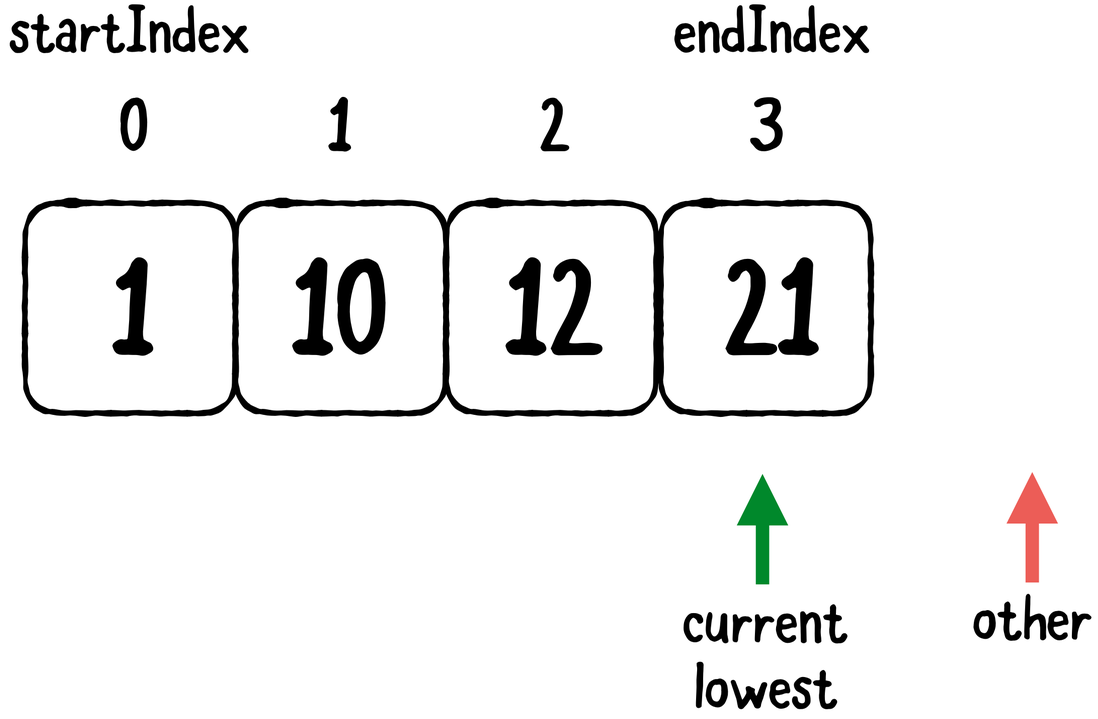

startIndex
endIndex
0
1
2
3
1
21
10
12
current
other
lowest


startIndex
endIndex
0
1
2
3
1
21
10
12
current
other
lowest


startIndex
endIndex
0
1
2
3
1
10
21
12
current
other
lowest


startIndex
endIndex
0
1
2
3
1
10
21
12
current
lowest
other


startIndex
endIndex
0
1
2
3
1
10
21
12
current
lowest
other


startIndex
endIndex
0
1
2
3
1
10
12
21
current
lowest
other

```
extension MutableCollection where Element: Comparable {
  mutating func selectionSort() {
    //...
    for current in indices {
      // ...
    }
  }
}
```

Hints

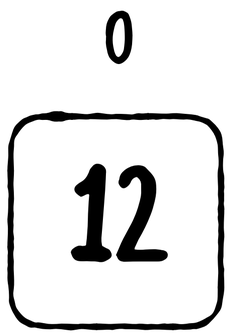
```
mutating func swapAt(_ i: Self.Index, _ j: Self.Index)
```

```
func index(after i: Index) -> Index
```

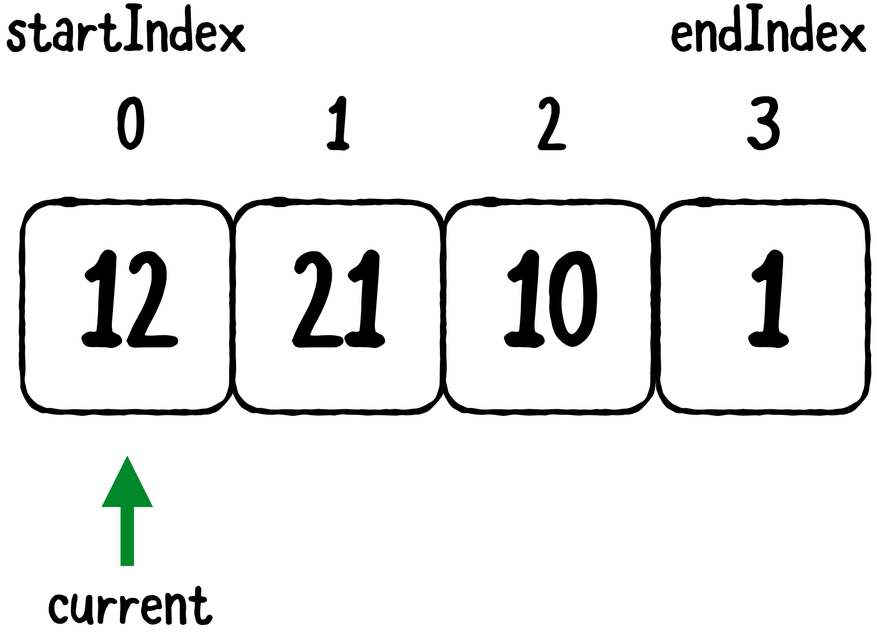
```
var indices: Self.Indices { get }
```

Go over solution

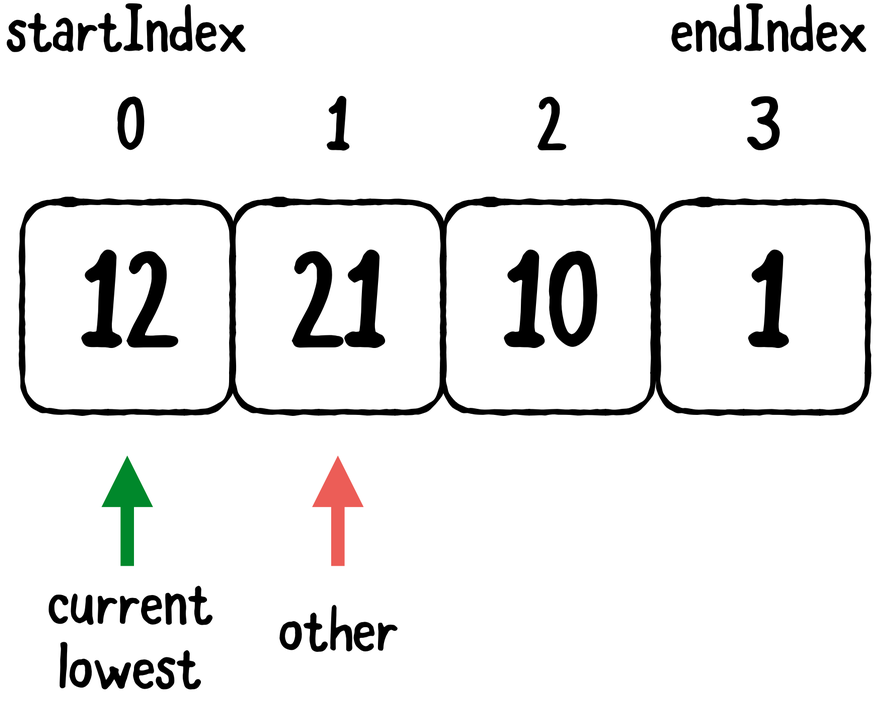
```
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



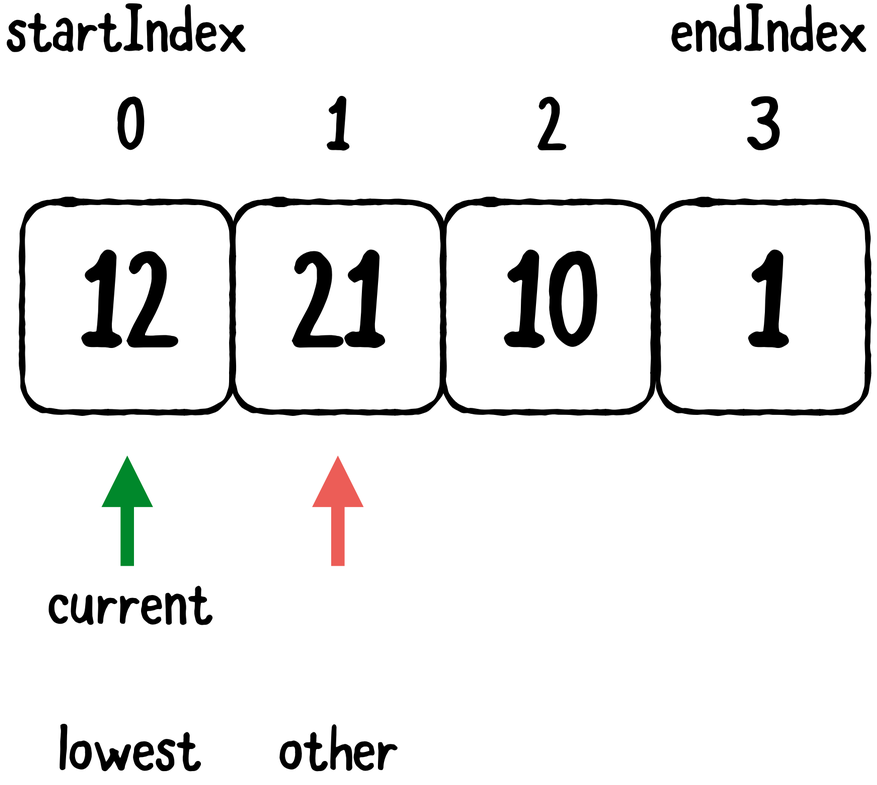
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



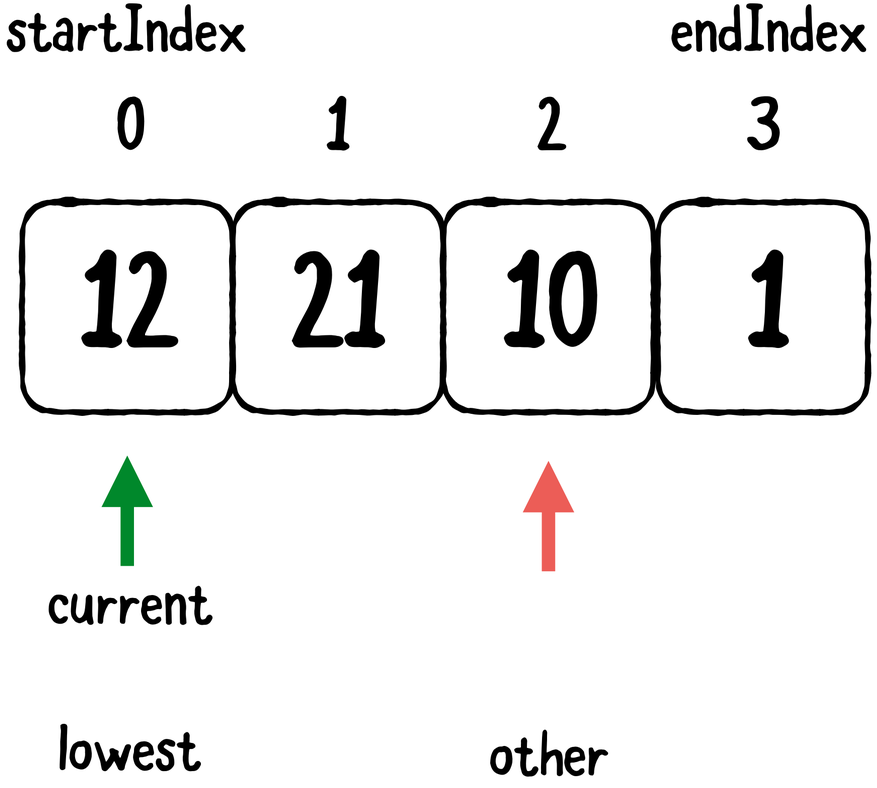
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



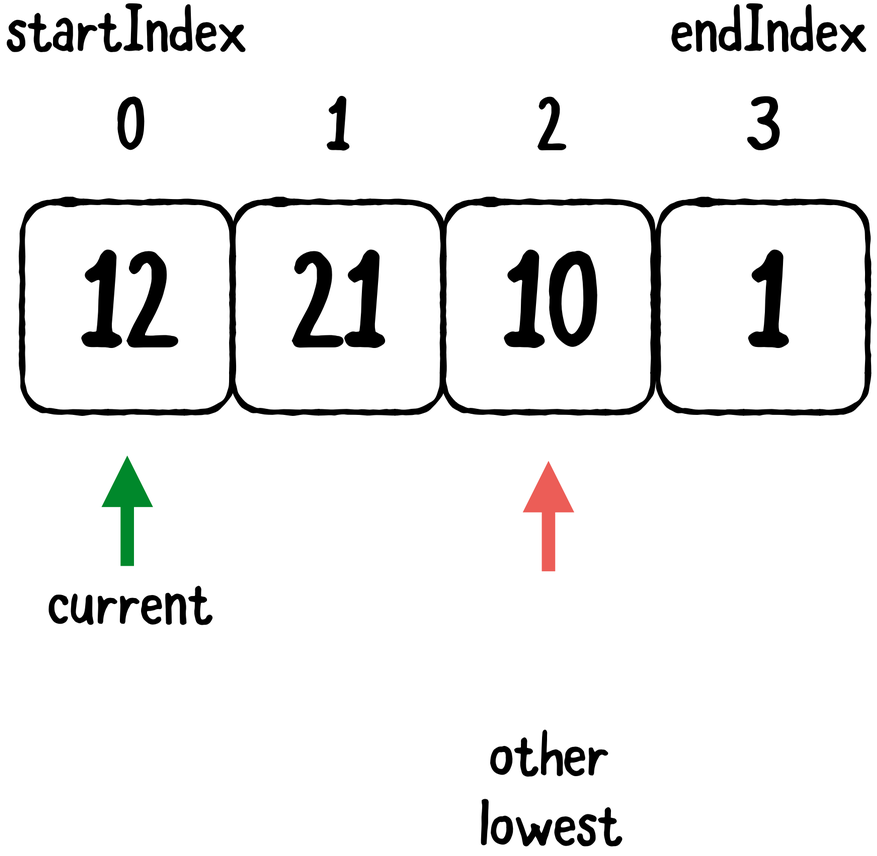
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



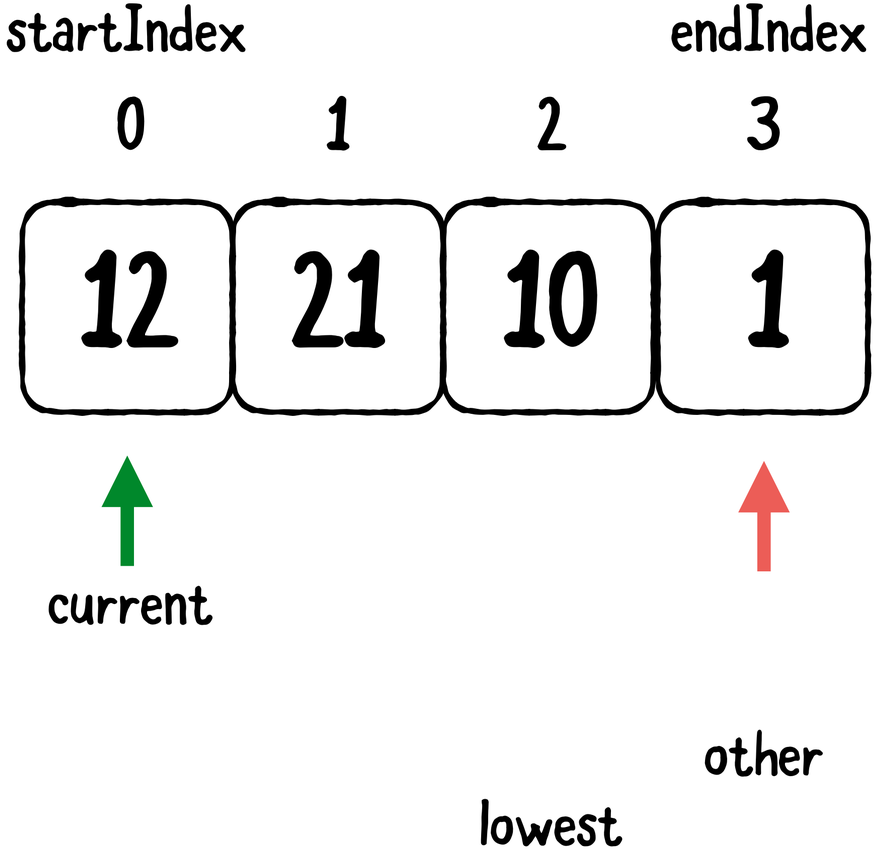
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



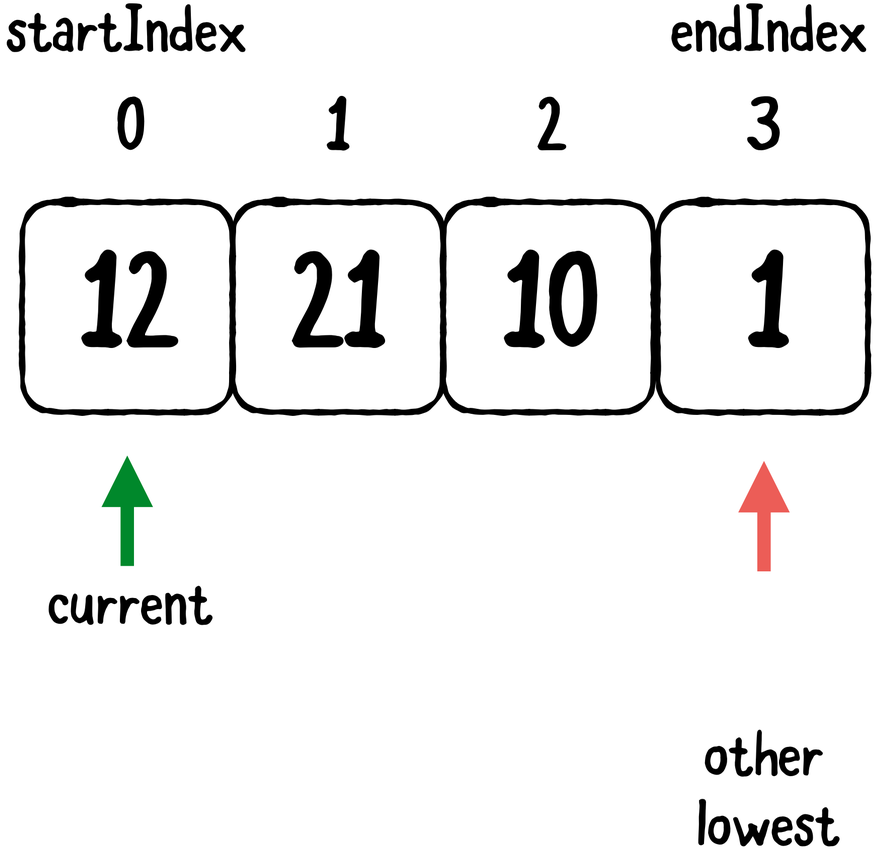
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



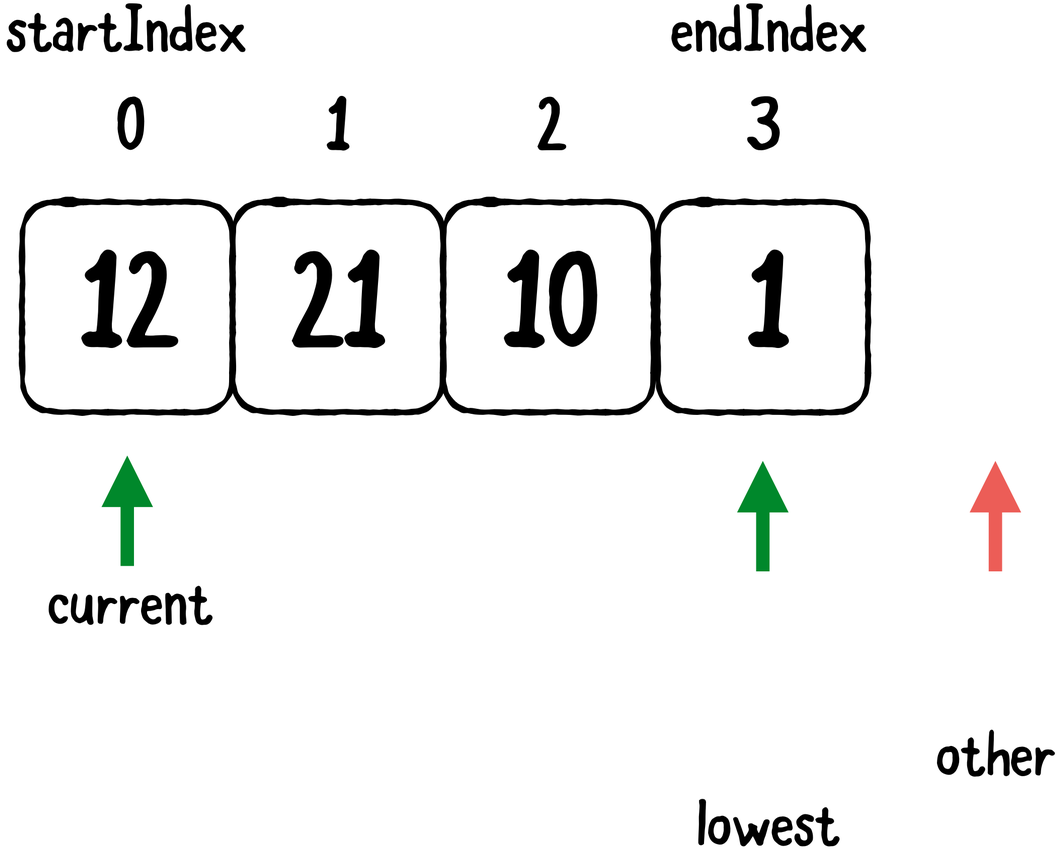
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



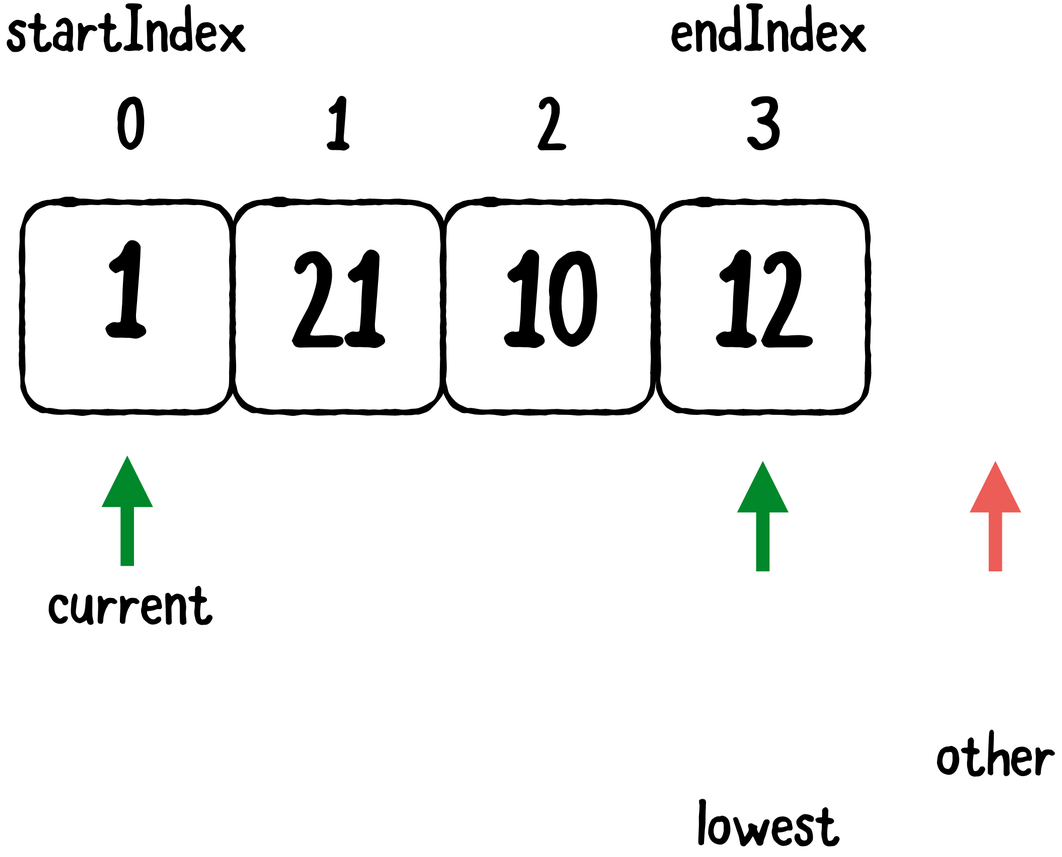
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



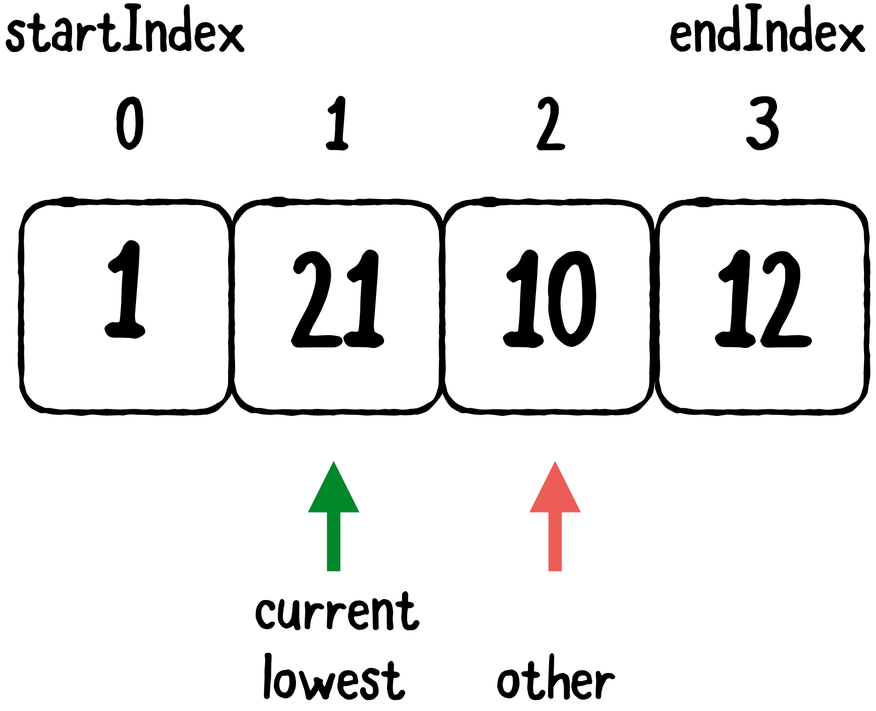
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



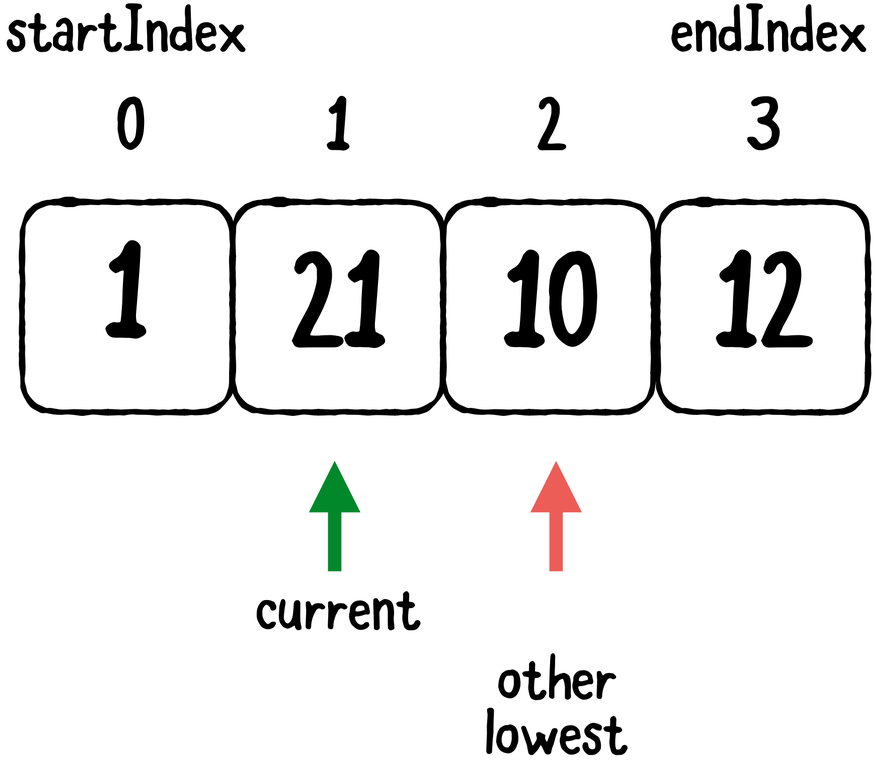
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



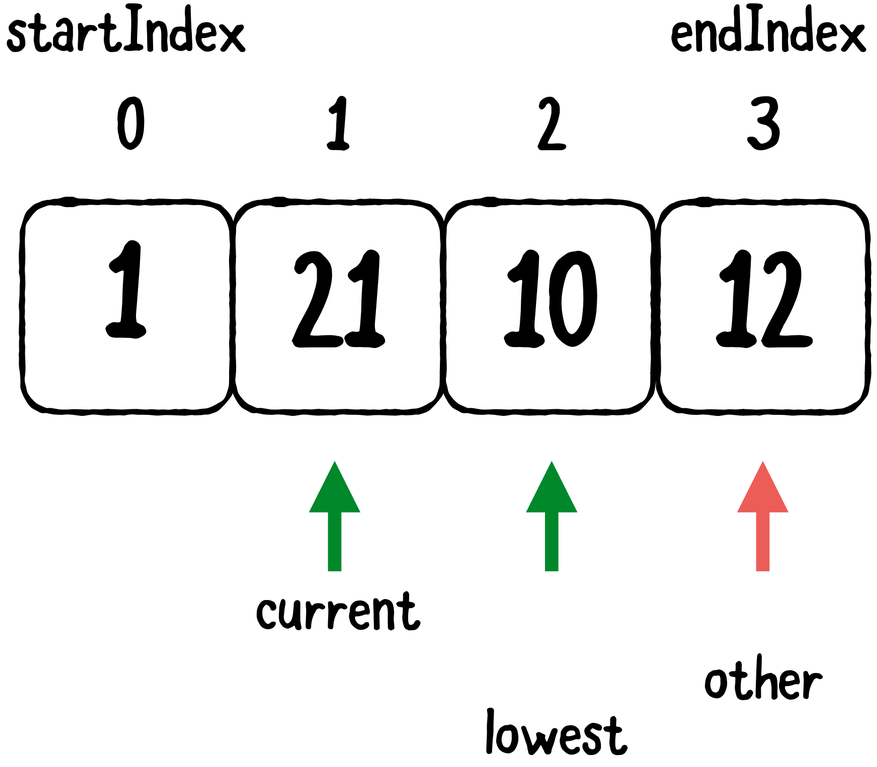
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



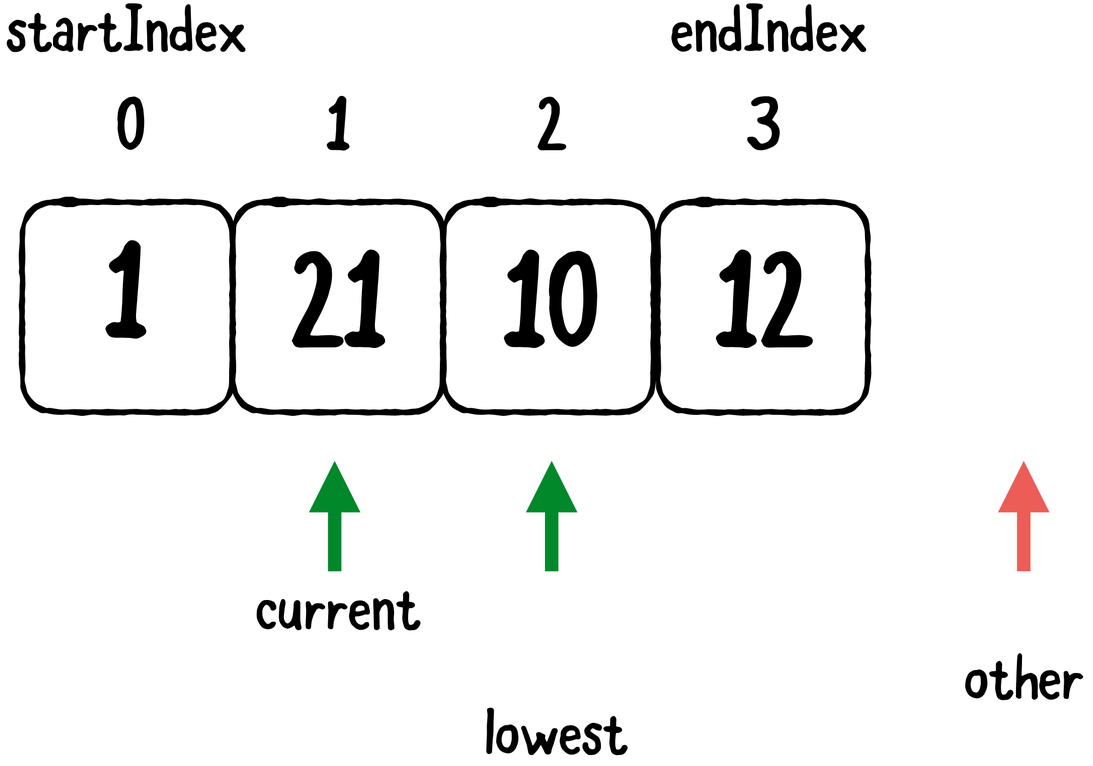
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



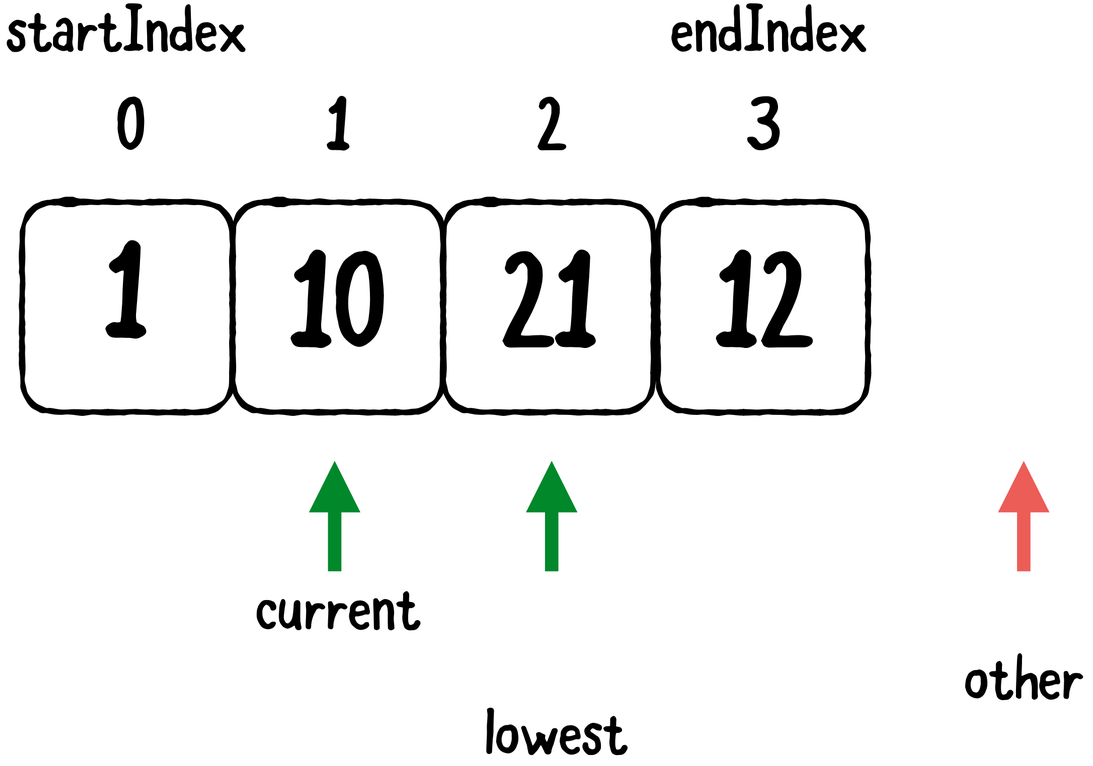
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



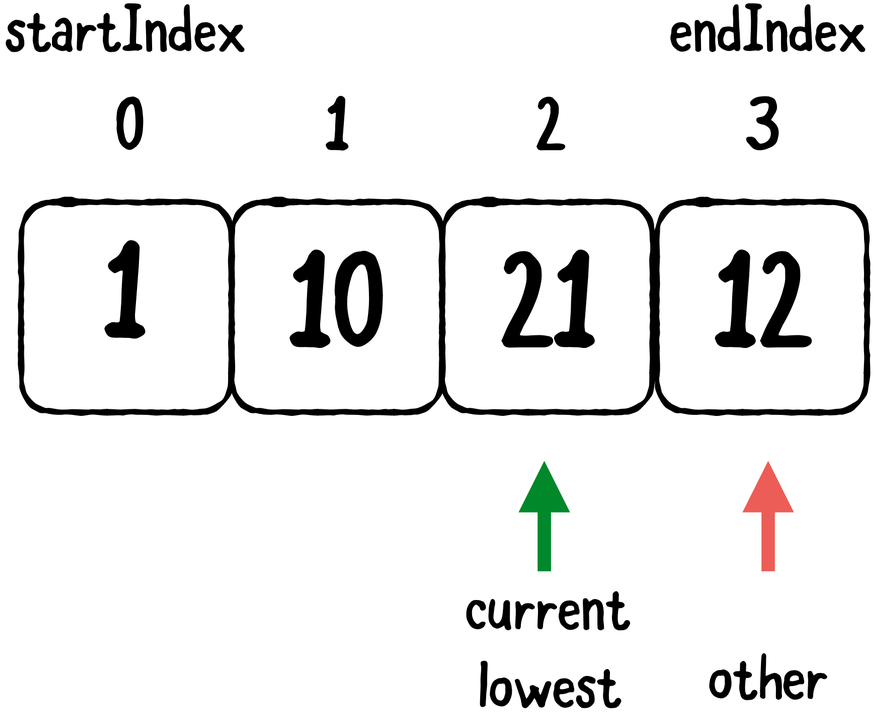
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



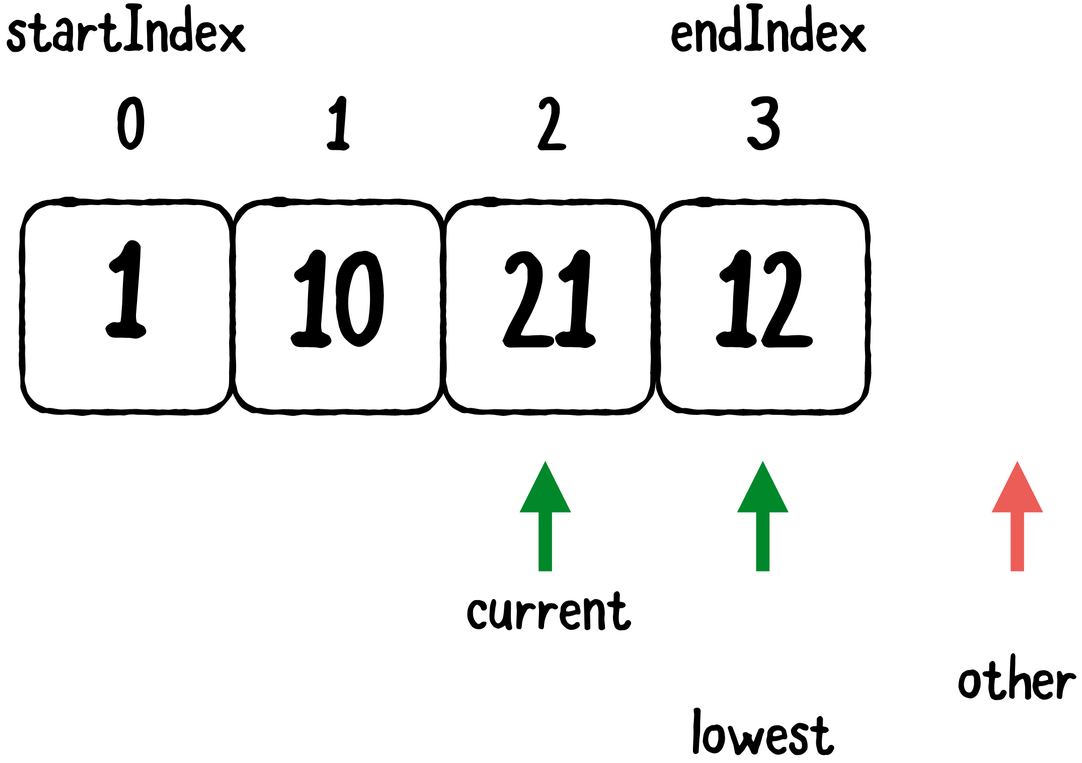
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



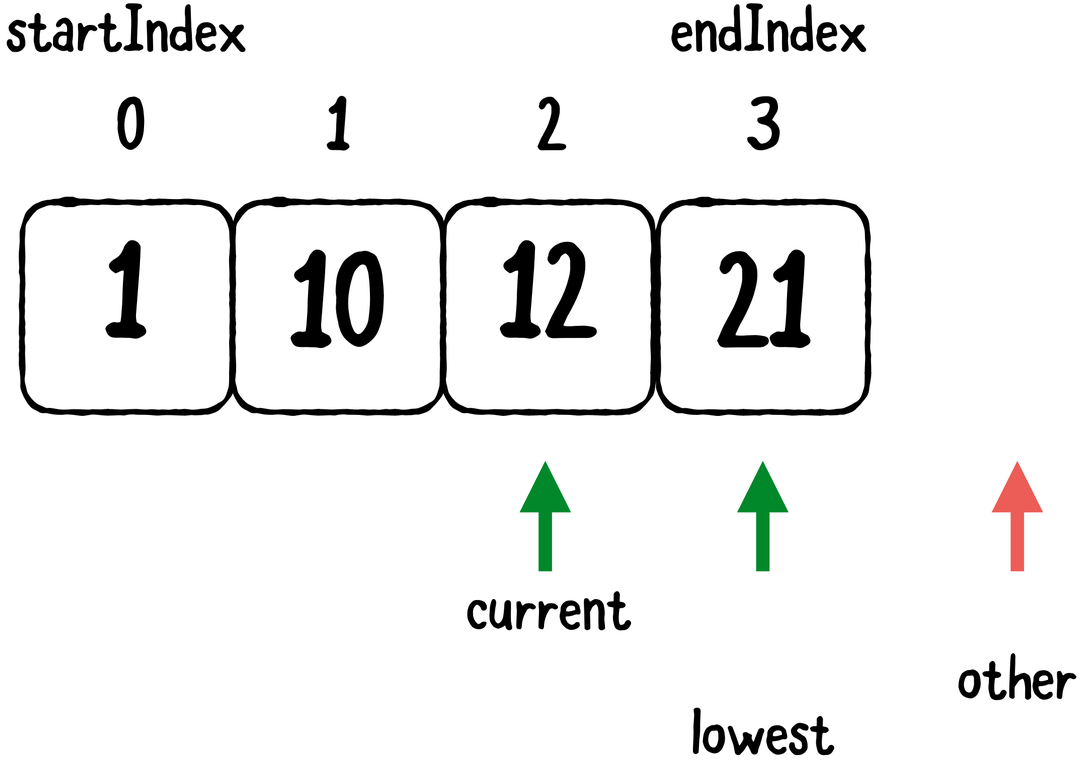
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



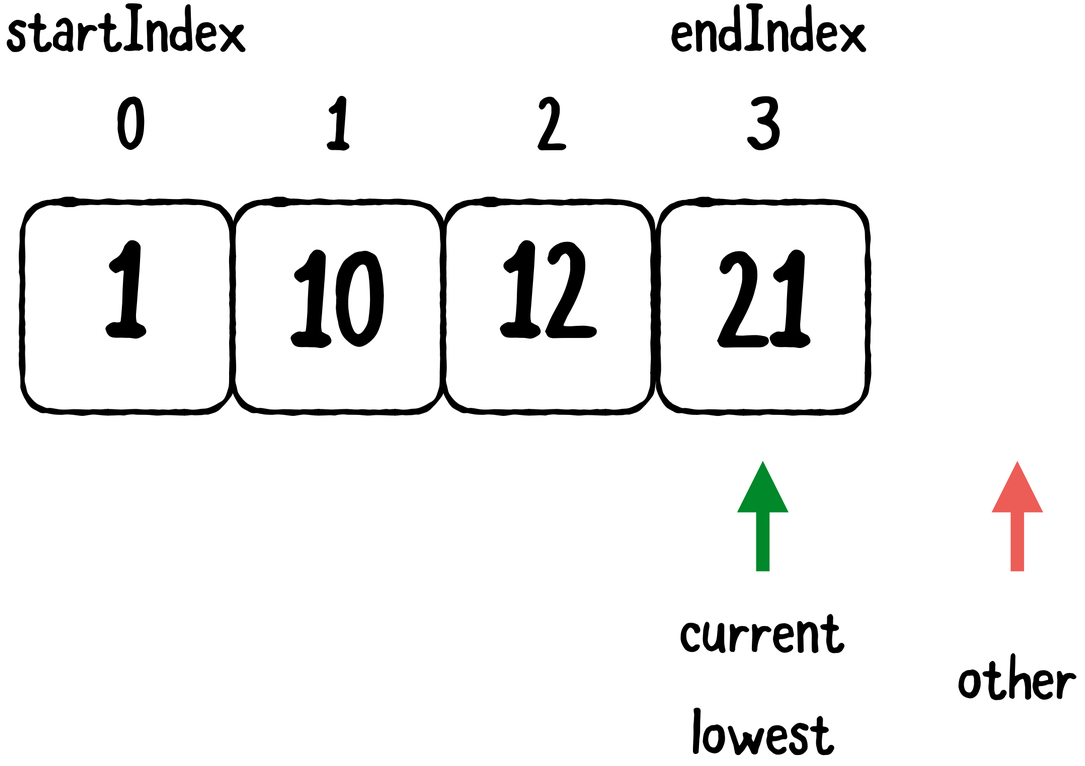
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



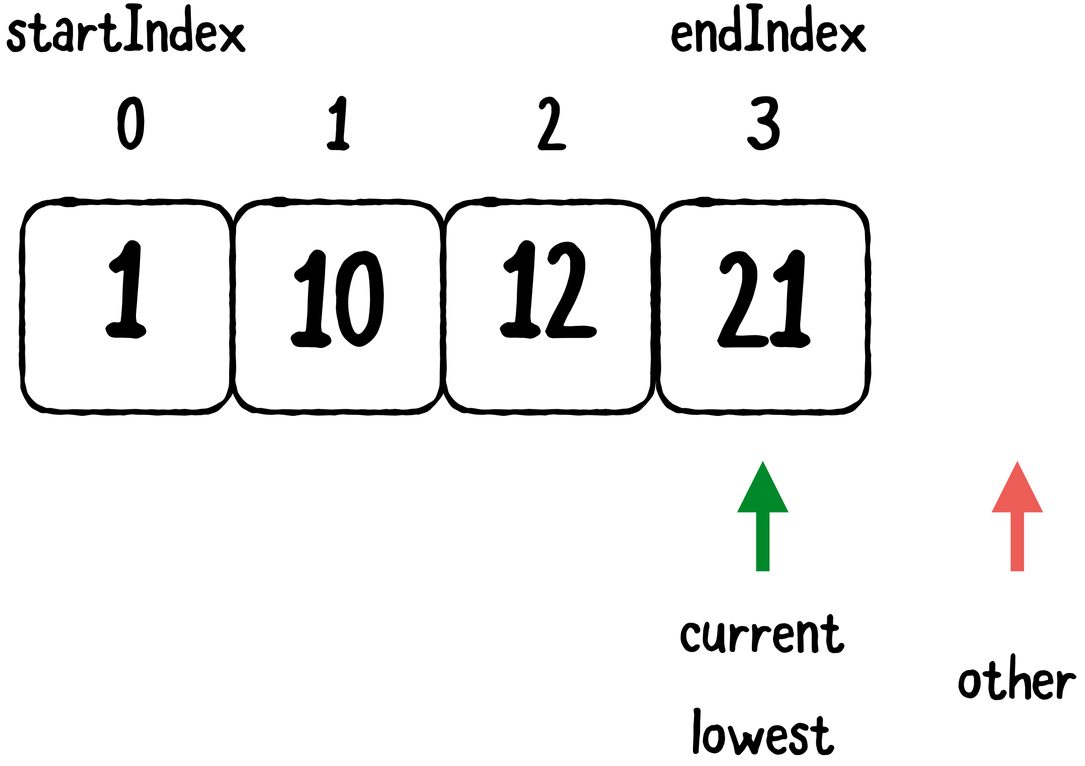
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



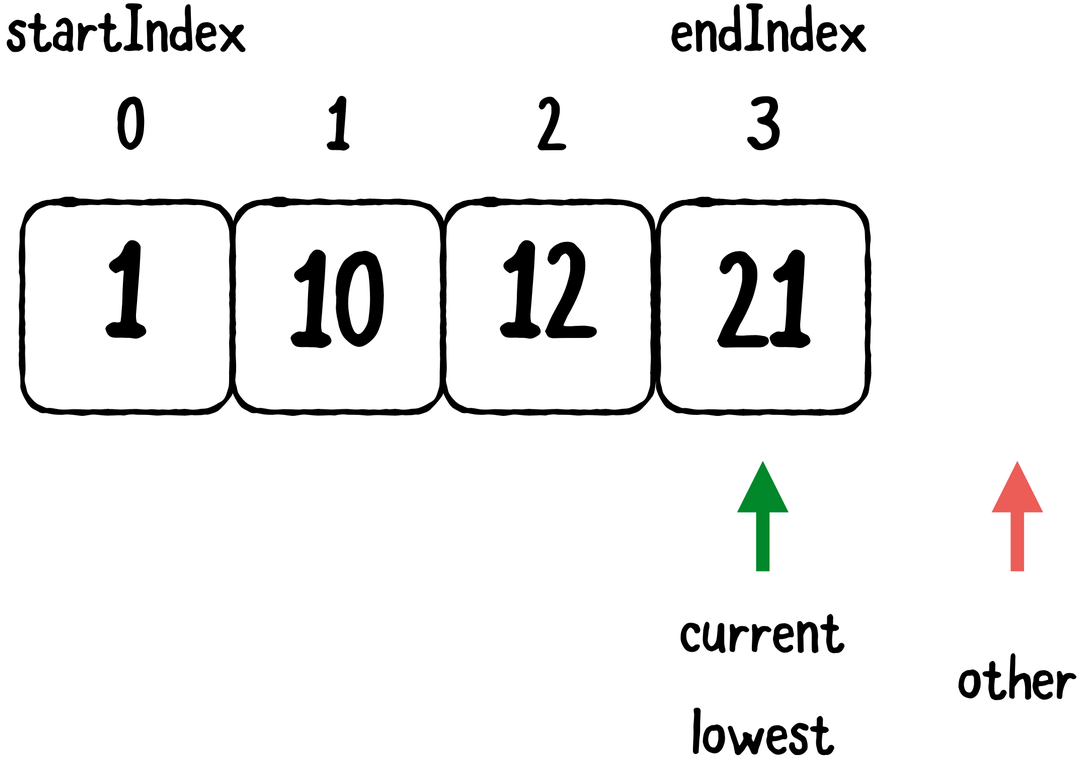
```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```



```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```

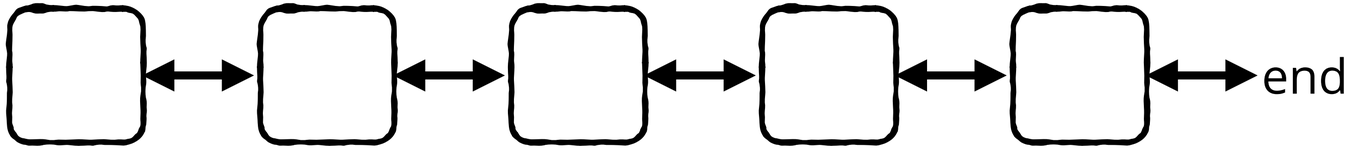


```swift
extension MutableCollection where Self.Element: Comparable {
  mutating func selectionSort() {
    guard self.count >= 2 else { return }

    for current in indices {
      var lowest = current
      var other = index(after: current)
      while other < endIndex {
        if self[lowest] > self[other] {
          lowest = other
        }

        other = index(after: other)
      }

      if lowest != current {
        swapAt(lowest, current)
      }
    }
  }
}
```

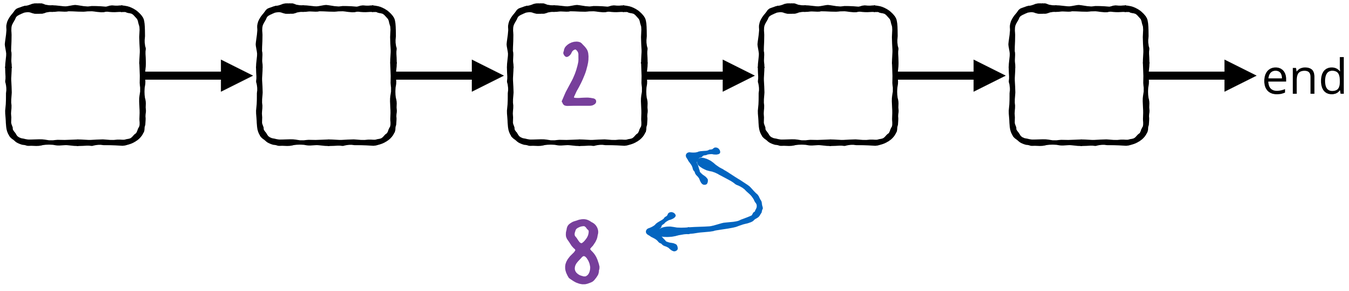


RECAP

A **BidirectionalCollection** supports backwards traversal



A **MutableCollection** enables you to change value of elements.



Tier 3 - RangeReplaceableCollection

supports insertion and removal of elements at arbitrary subrange and indices.

replaceSubrange(subrange:newElements:)

removeAll()

removeFirst()

removeFirst(n)

insert

append

insert(contentsOf:at:)

append(contentsOf:)

```
mutating func replaceSubrange<C>(_ subrange: Range<Self.Index>,
                                 with newElements: C)
        where C : Collection, Self.Element == C.Element
```

RangeReplacableCollection - Removing

```
replaceSubrange<C>(_ subrange: Range<Self.Index>,
                   with newElements: C)
```

removeFirst() removeFirst(n) removeAll() removeSubRange(...)

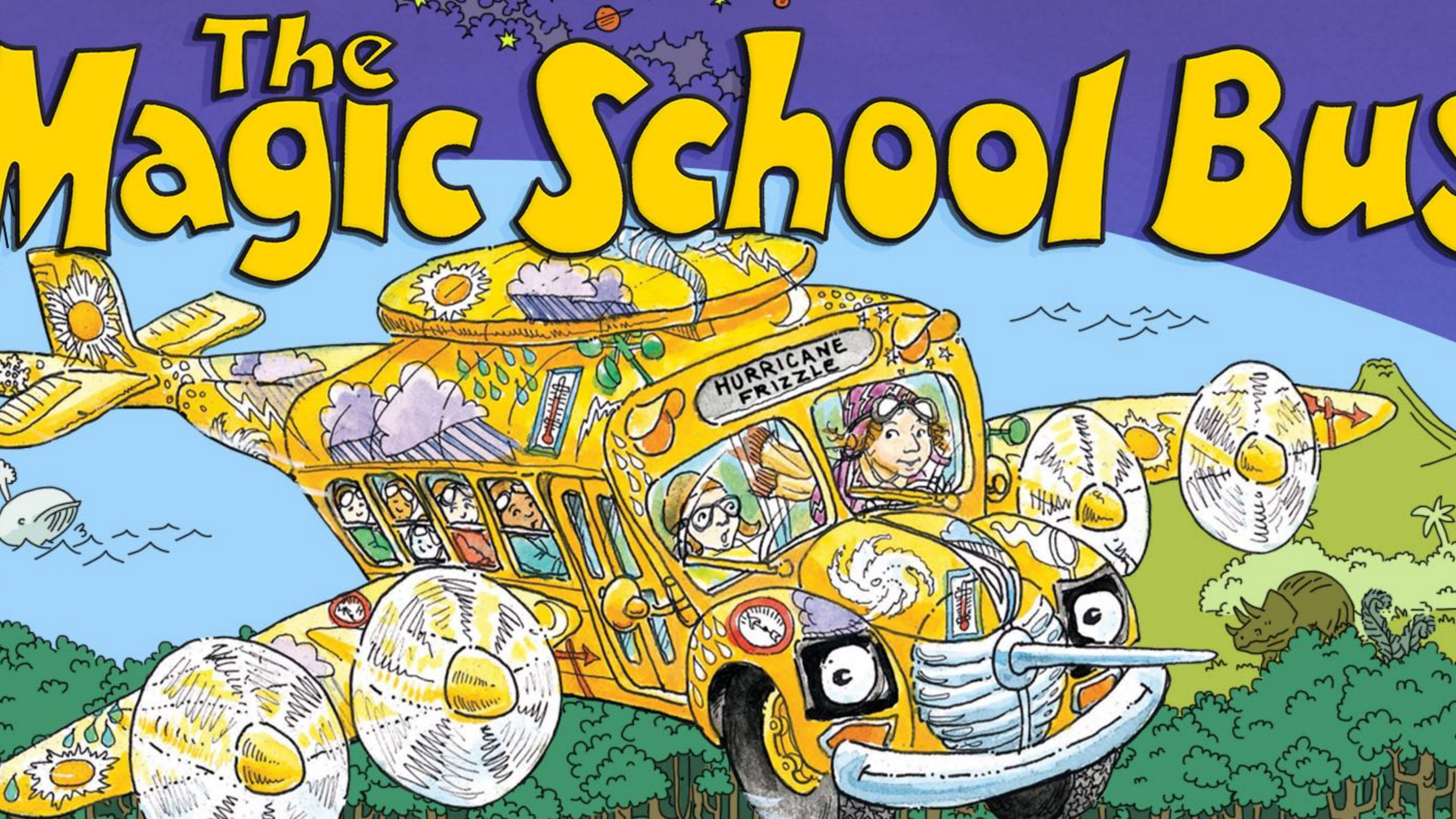
EmptyCollection

RangeReplacableCollection - Inserting

```
replaceSubrange<C>(_ subrange: Range<Self.Index>,
                   with newElements: C)
```

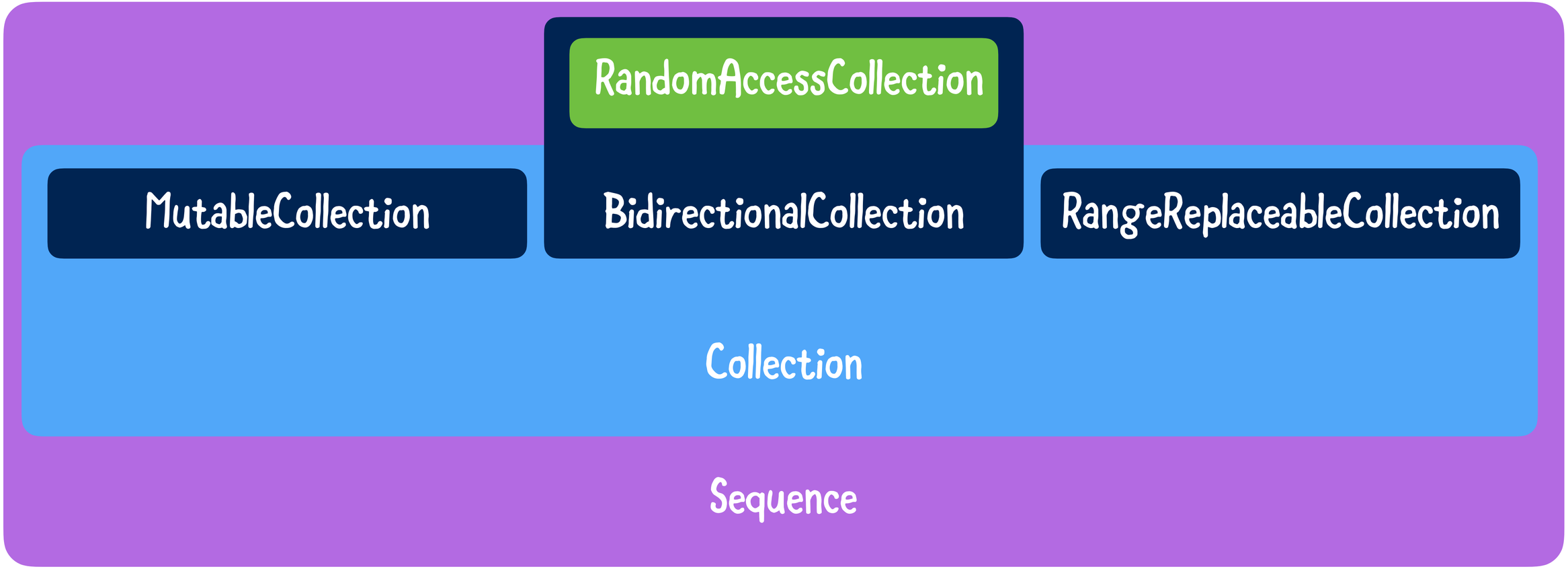
insert append append(contentsOf:) insert(contentsOf:at:)

CollectionOfOne
Collection


The Magic School Bus
HURRICANE FRIZZLE

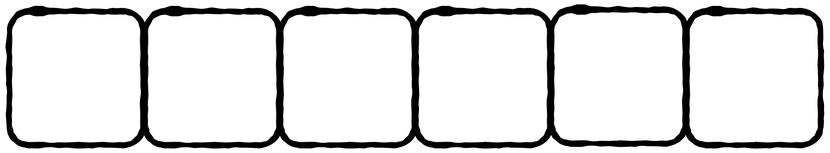
Demo 5
Conforming to RangeReplacableCollection

Tier 4


RandomAccessCollection
MutableCollection
BidirectionalCollection
RangeReplaceableCollection
Collection
Sequence

Tier 4 - RandomAccessCollection

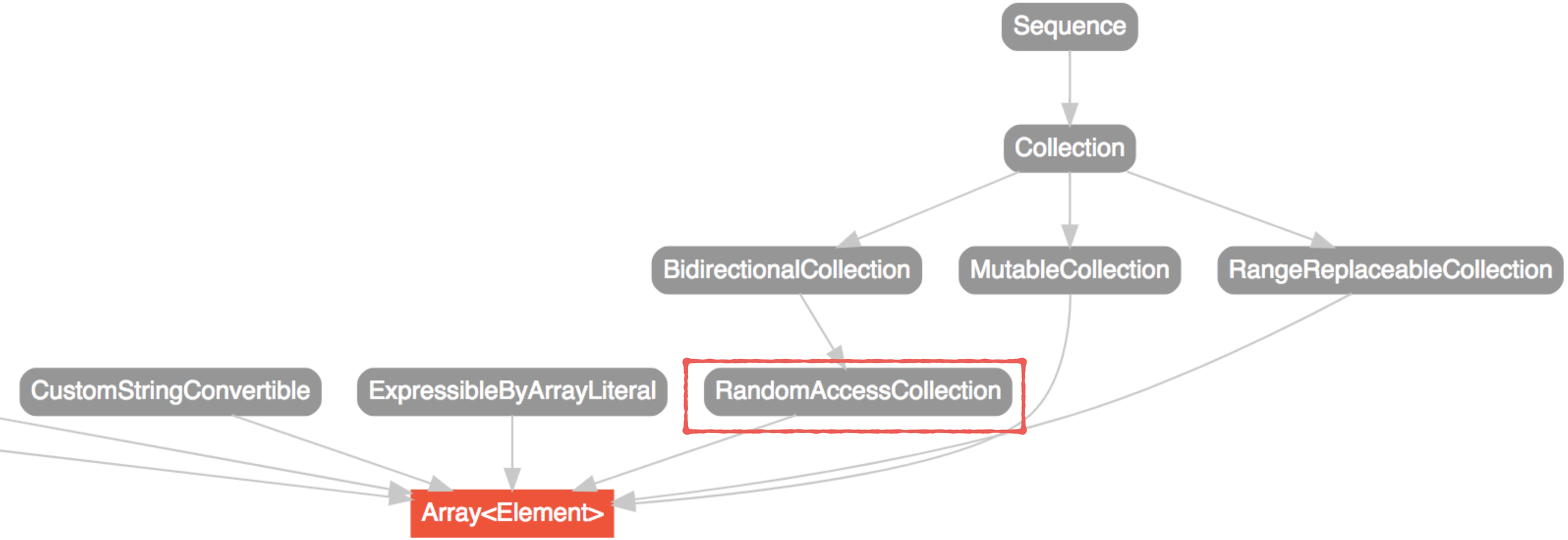
A **RandomAccessCollection** can access elements anywhere!



distance(from:to:)

index(_:offsetBy:)

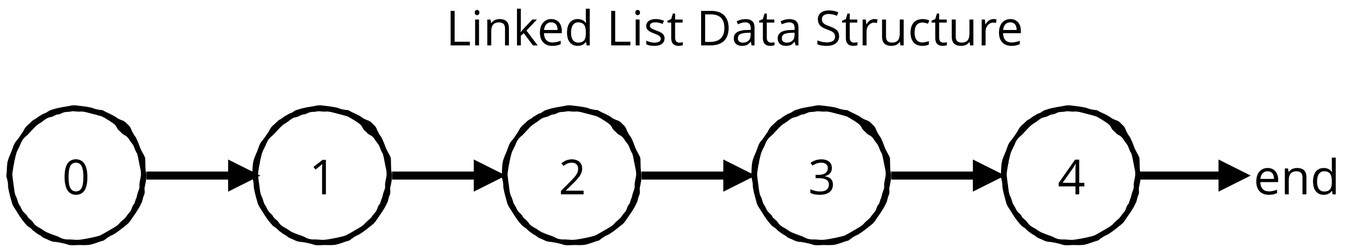
Strideable



#WeLoveArrays

LINKED LIST ~~CONFORMS~~ TO RANDOMACCESSCOLLECTION

distance(from:to:)
index(_:offsetBy:)
} O(1)



Collection.swift

```swift
@_inlineable
public var count: Int {
  return distance(from: startIndex, to: endIndex)
}
```

```swift
@_inlineable
public func distance(from start: Index, to end: Index) -> Int {
  _precondition(start <= end,
    "Only BidirectionalCollections can have end come before start")

  var start = start
  var count = 0
  while start != end {
    count = count + 1
    formIndex(after: &start)
  }
  return count
}
```

https://github.com/apple/swift/blob/master/stdlib/public/core/Collection.swift

BidirectionalCollection.swift

```
@_inlineable
public var count: Int {
  return distance(from: startIndex, to: endIndex)
}
```

```
@_inlineable // FIXME(sil-serialize-all)
public func distance(from start: Index, to end: Index) -> Int {
  var start = start
  var count = 0

  if start < end {
    while start != end {
      count += 1
      formIndex(after: &start)
    }
  }
  else if start > end {
    while start != end {
      count -= 1
      formIndex(before: &start)
    }
  }

  return count
}
}
```

RandomAccessCollection.swift

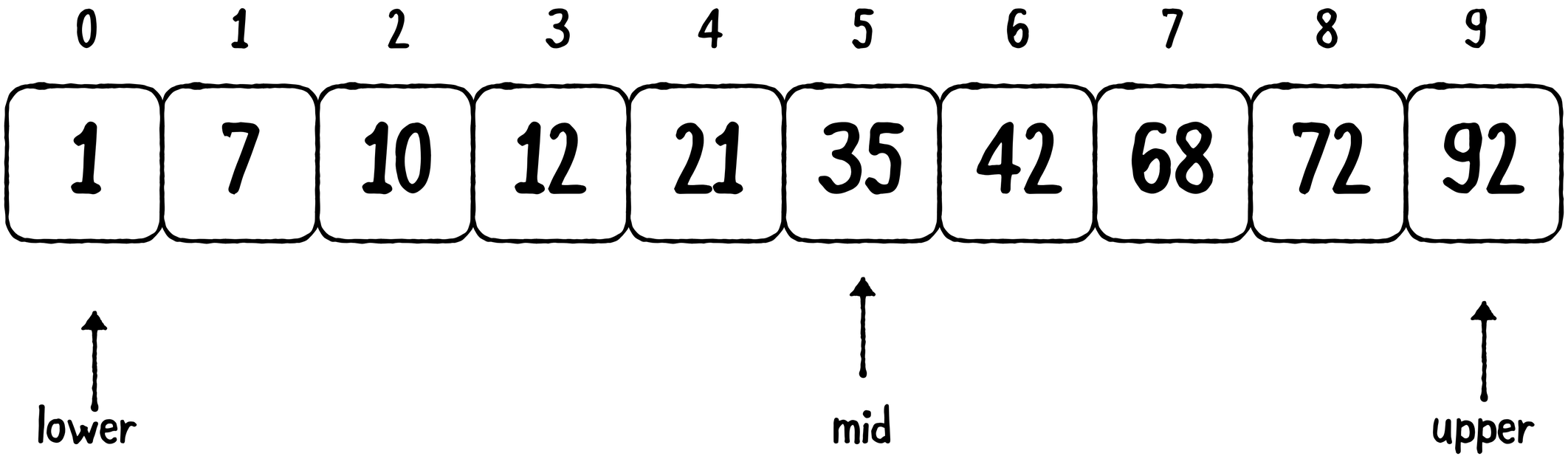
```swift
@_inlineable
public var count: Int {
  return distance(from: startIndex, to: endIndex)
}
```

```swift
  /// Returns the distance between two indices.
  ///
  /// - Parameters:
  ///   - start: A valid index of the collection.
  ///   - end: Another valid index of the collection. If `end` is equal to
  ///     `start`, the result is zero.
  /// - Returns: The distance between `start` and `end`.
  ///
  /// - Complexity: O(1)
  @_inlineable
  public func distance(from start: Index, to end: Index) -> Index.Stride {
    // FIXME: swift-3-indexing-model: tests for traps.
    _failEarlyRangeCheck(
      start, bounds: ClosedRange(uncheckedBounds: (startIndex, endIndex)))
    _failEarlyRangeCheck(
      end, bounds: ClosedRange(uncheckedBounds: (startIndex, endIndex)))
    return start.distance(to: end)
  }
}
```

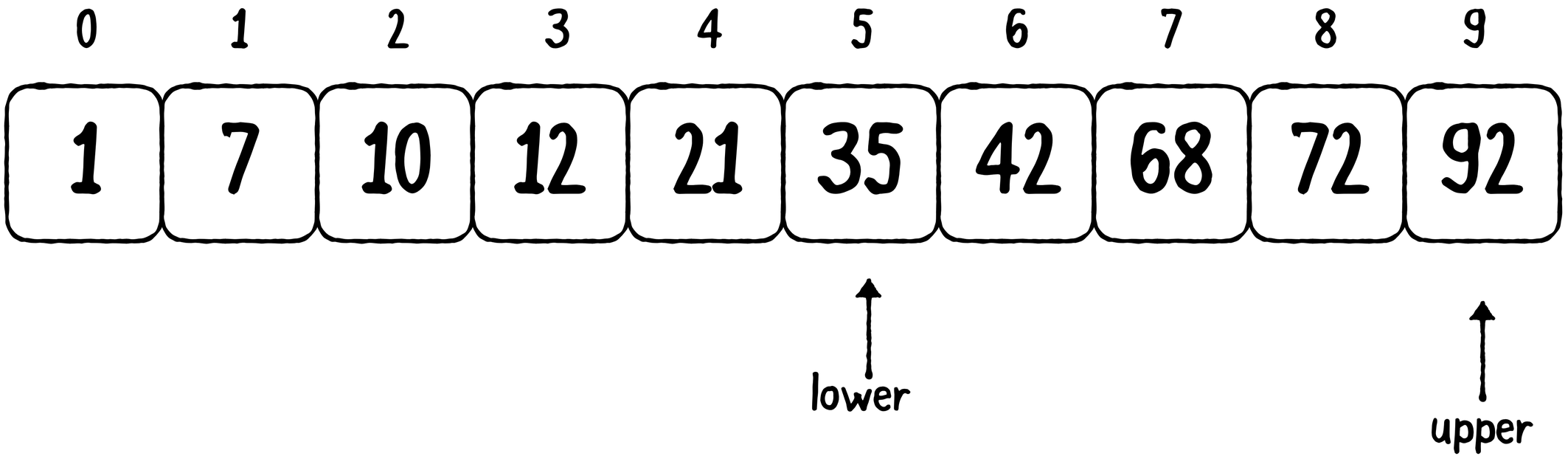
Challenge 6

Given the binary search algorithm, implement it as an extension of RandomAccessCollection protocol.

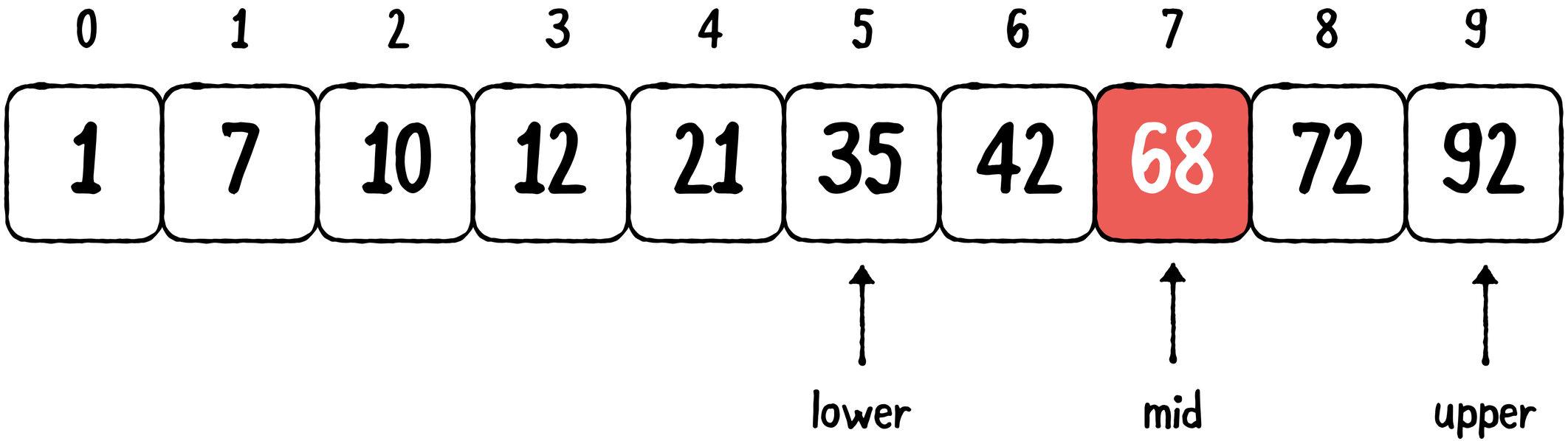
Find 68



Find 68



Find 68



```swift
func binarySearch<T: Comparable>(_ a: [T], value: T) -> Int? {
  var lowerBound = 0
  var upperBound = a.count
  while lowerBound < upperBound {
    let mid = lowerBound + (upperBound - lowerBound) / 2
    let candidate = a[mid]
    if candidate == value {
      return mid
    } else if candidate < value {
      lowerBound = mid + 1
    } else {
      upperBound = mid
    }
  }
  return nil
}
```

```swift
func binarySearch<T: Comparable>(_ a: [T], value: T) -> Int? {
  var lowerBound = 0
  var upperBound = a.count
  while lowerBound < upperBound {
    let mid = lowerBound + (upperBound - lowerBound) / 2
    let candidate = a[mid]
    if candidate == value {
      return mid
    } else if candidate < value {
      lowerBound = mid + 1
    } else {
      upperBound = mid
    }
  }
  return nil
}
```

```swift
extension RandomAccessCollection where Element: Comparable {
  func binarySearch(for value: Element) -> Index? {

  }
}
```

Go over solution

```
extension RandomAccessCollection where Element: Comparable {
    func binarySearch<T: Comparable>(_ a: [T], value: T) -> Int? {
      var lowerBound = 0
      var upperBound = a.count
     while lowerBound < upperBound {
        let mid = lowerBound + (upperBound - lowerBound) / 2
        let candidate = a[mid]
        if candidate == value {
           return mid
         } else if candidate < value {
           lowerBound = mid + 1
         } else {
           upperBound = mid
         }
       }
      return nil
    }
}
```

```
extension RandomAccessCollection where Element: Comparable {
    func binarySearch(for value: Element) -> Index? {
        var lowerBound = 0
        var upperBound = a.count
        while lowerBound < upperBound {
            let mid = lowerBound + (upperBound - lowerBound) / 2
            let candidate = a[mid]
            if candidate == value {
                return mid
            } else if candidate < value {
                lowerBound = mid + 1
            } else {
                upperBound = mid
            }
        }
        return nil
    }
}
```

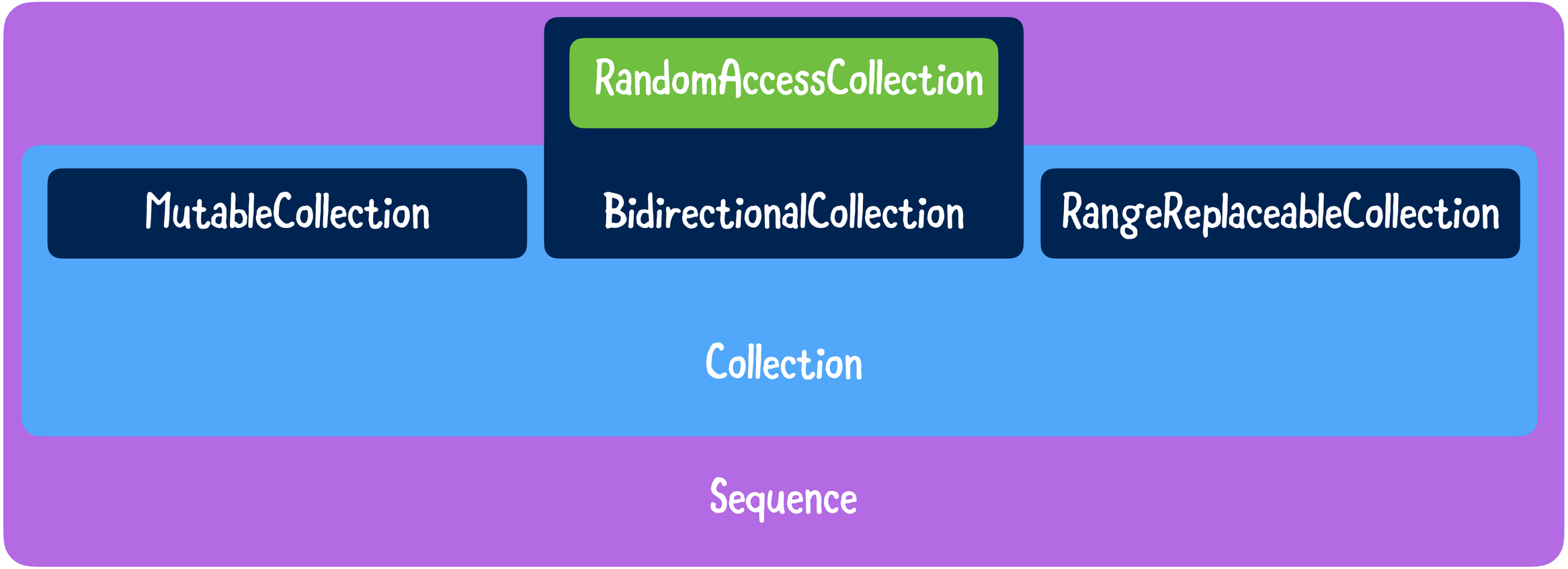
```
extension RandomAccessCollection where Element: Comparable {
    func binarySearch(for value: Element) -> Index? {
      if isEmpty { return nil }
        var lowerBound = 0
        var upperBound = a.count
       while lowerBound < upperBound {
          let mid = lowerBound + (upperBound - lowerBound) / 2
          let candidate = a[mid]
          if candidate == value {
             return mid
           } else if candidate < value {
             lowerBound = mid + 1
           } else {
             upperBound = mid
           }
         }
        return nil
    }
}
```

```
extension RandomAccessCollection where Element: Comparable {
    func binarySearch(for value: Element) -> Index? {
        if isEmpty { return nil }
        var lowerBound = startIndex
        var upperBound = index(before: endIndex)
        while lowerBound < upperBound {
            let mid = lowerBound + (upperBound - lowerBound) / 2
            let candidate = a[mid]
            if candidate == value {
                return mid
            } else if candidate < value {
                lowerBound = mid + 1
            } else {
                upperBound = mid
            }
        }
        return nil
    }
}
```

```
extension RandomAccessCollection where Element: Comparable {
  func binarySearch(for value: Element) -> Index? {
    if isEmpty { return nil }
    var lowerBound = startIndex
    var upperBound = index(before: endIndex)
    while lowerBound < upperBound {
      let dist = distance(from: lowerBound, to: upperBound)
      let mid = index(lowerBound, offsetBy: dist/2)
      let candidate = self[mid]
       if candidate == value {
          return mid
       } else if candidate < value {
          lowerBound = mid + 1
       } else {
          upperBound = mid
       }
    }
    return nil
  }
}
```

```
extension RandomAccessCollection where Element: Comparable {
  func binarySearch(for value: Element) -> Index? {
    if isEmpty { return nil }
    var lowerBound = startIndex
    var upperBound = index(before: endIndex)
    while lowerBound < upperBound {
      let dist = distance(from: lowerBound, to: upperBound)
      let mid = index(lowerBound, offsetBy: dist/2)
      let candidate = self[mid]
       if candidate == value {
          return mid
        } else if candidate < value {
          lowerBound = index(after: mid)
        } else {
          upperBound = index(before: mid)
        }
    }
    return nil
  }
}
```

Tier 4


RandomAccessCollection
MutableCollection
BidirectionalCollection
RangeReplaceableCollection
Collection
Sequence

What You Learned

- **Demo 1**: Enum based Linked List
- **Demo 2**: Sequence Protocol
- **Demo 3:** Collection Protocol
- **Demo 4:** BidirectionalCollection
- **Demo 5:** MutableCollection
- **Demo 6:** RangeReplaceableCollection
- **Demo 7:** RandomAccessCollection

Where To Go From Here?

raywenderlich / **swift-algorithm-club**

Algorithms and data structures in Swift, with explanations!

Swift ★ 13,871 2,083 Updated 20 hours ago

http://bit.ly/29ISE0n

Optimizing Collections

Write custom collections in Swift with a strong focus on performance

by Károly Lőrentey

http://bit.ly/2wB38L4

Advanced Swift

A deep dive into Swift's features, from low-level programming to high-level abstractions.

by Chris Eidhof, Ole Begemann, and Airspeed Velocity

http://bit.ly/10SII2K

@KelvinlauKI

@VincentNgo2