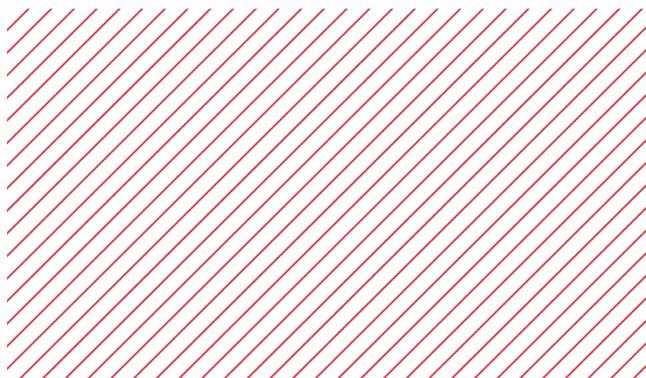


академия  
больших  
данных



# Advanced Scala

Корольков Алексей



# Структура курса

1. Введение в Scala. Основные конструкции языка



2. ООП в Scala. Pattern matching. Функциональные конструкции. Adt

3. Библиотека коллекций в Scala



4. Асинхронные операции, обработка исключений, неявные параметры

5. Параметрический полиморфизм. Имплиситы



6. Основы функционального программирования. Часть 1

7. Основы функционального программирования. Часть 2

8. ZIO, TF и основной ФП-стэк в Scala

9. Функциональные стримы на примере fs2, работа с бд

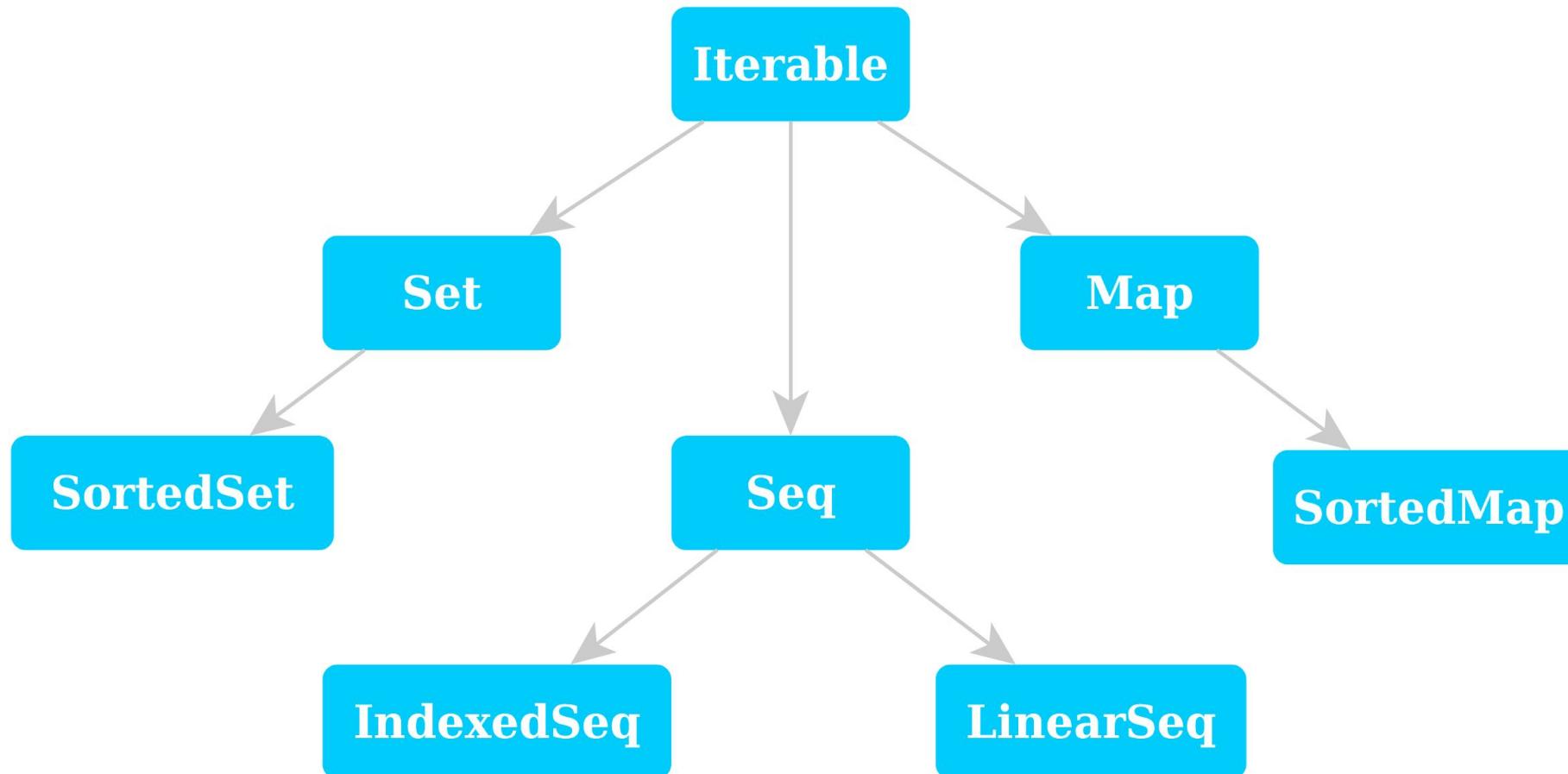
10. Акторы в Scala



иерархия коллекций

# Базовые интерфейсы коллекций

---





# Mutable vs Immutable

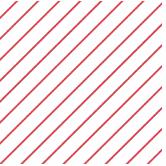
---

## **Неизменяемые (*immutable*)**

- Безопасно при работе из нескольких потоков
- Не нужно задумываться, что коллекция может измениться

## **Изменяемые (*mutable*) :**

- Операции изменения работают быстрее
- Требуется меньше памяти



# *Iterable*

---

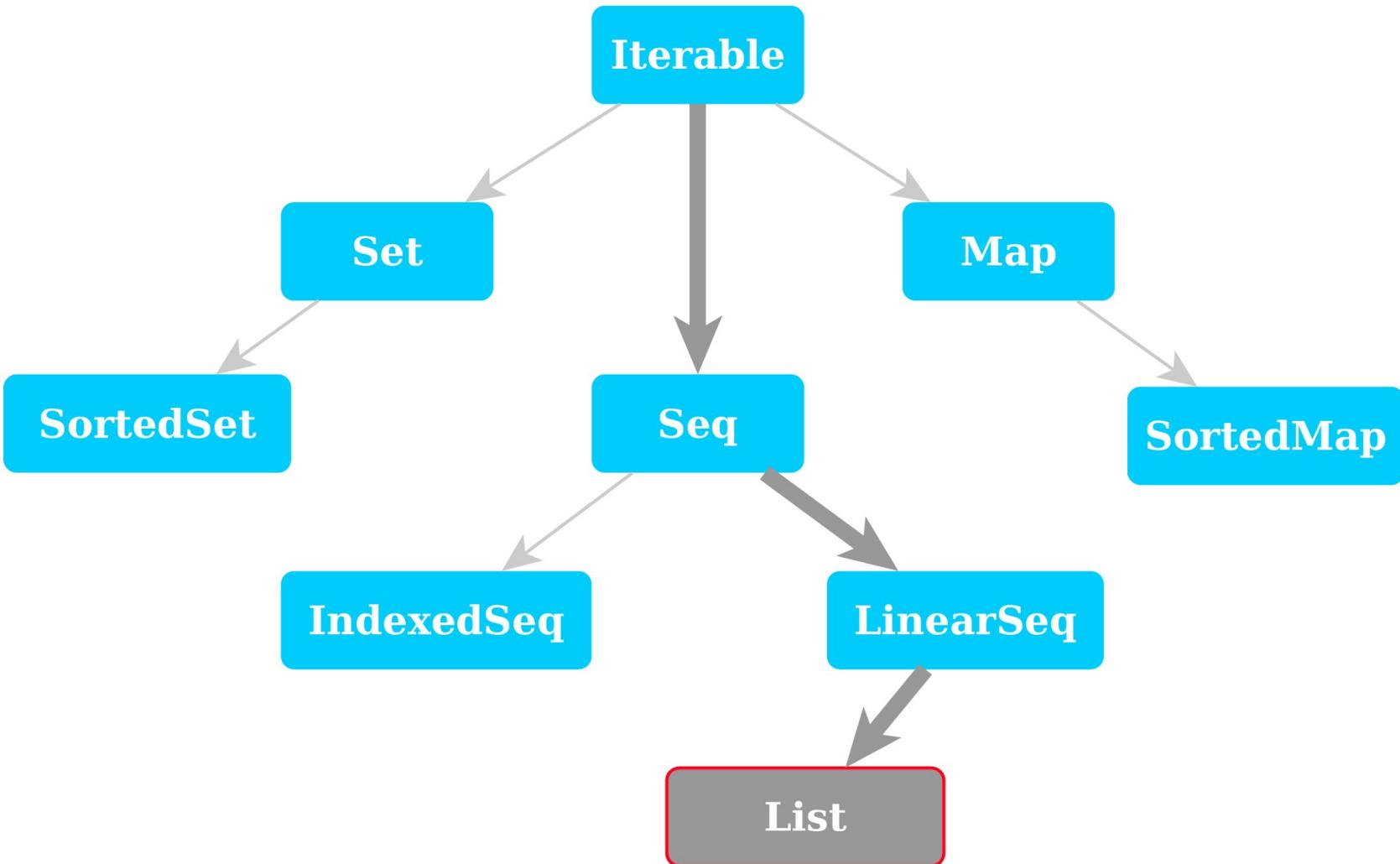
- базовый интерфейс для изменяемых и неизменяемых коллекций
- в нем объявлен метод ***iterator***, с помощью которого можно обойти все элементы коллекции
- практически все основные операции над коллекциями объявлены в этом интерфейсе



List

# Seq. LinearSeq. List

---



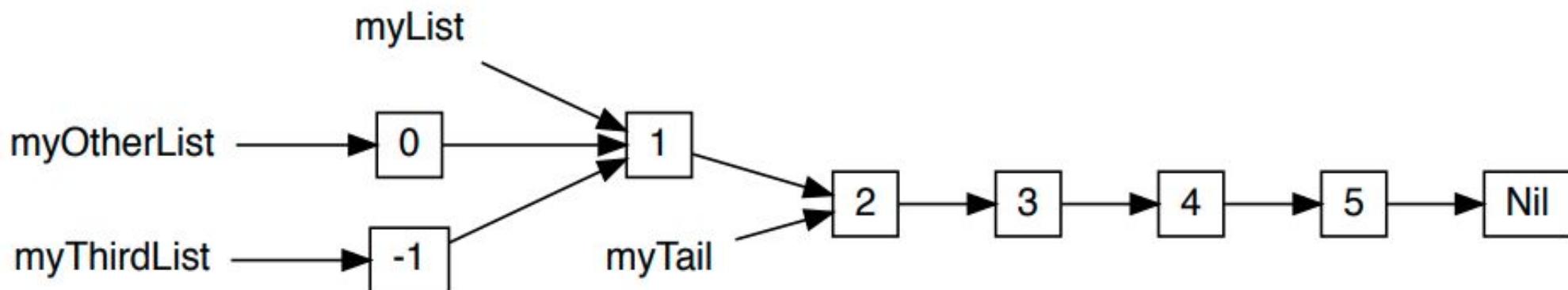
# List

---

**List[A]** – неизменяемый односвязный список элементов некоторого типа **A**

При добавлении новых элементов – новый лист не создается при каждом изменении, а только изменяется ссылка на узел, что позволяет сэкономить память.

Пример:



# List. Constructors

---

**List** можно инициализировать разными способами:

используя оператор `cons ::`

```
val list = 1 :: 2 :: 3 :: Nil
```

используя метод **apply** –

```
val list = List(1, 2, 3)
```

используя метод **range** –

```
val list = List.range(1, 10)
```

используя метод **apply** из компаньона **Seq** –

```
val list = Seq(1, 2, 3)
```

\*В текущей версии языка реализация **Seq** по умолчанию это **List**



# List. Pattern matching

---

**List** часто используется в сопоставлении с образцом:

- `case Nil` – константа обозначающая конец списка
- `case y :: ys` – паттерн обозначающий *List* с одним или более элементами
- `case List(y1, ..., yn)` – эквивалент  
`y1 :: ... :: yn :: Nil`



# List. Pattern matching

---

Рассмотрим образец листа:

**x :: y :: List(xs, ys) :: zs**

какой может быть длина листа соответствующего такому образцу?

*L* == 3

*L* == 4

*L* == 5

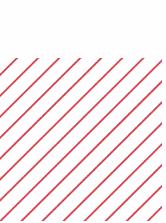
*L* >= 3

*L* >= 4

*L* >= 5



комбинаторы



# List. map

---

Операции модификации элементов обобщенно можно выразить через метод **map**:

*List*[T]

```
def map[U](f: T => U): List[U] = this match {  
  case Nil => this  
  case x :: xs => f(x) :: xs.map(f)  
}
```

\* в реальности метод **map** реализован сложнее из-за оптимизаций и специфики для разных коллекций



# List. filter

---

Другая часто используемая операция это фильтрация элементов по определенному критерию:

```
def positiveElements(xs: List[Int]): List[Int] = xs match {  
    case Nil => xs  
    case y :: ys =>  
        if (y > 0) y :: positiveElements(ys)  
        else positiveElements(ys)  
}
```

# List. filter

---

Если абстрагироваться от конкретного предиката, то реализация метода *filter* может выглядеть так:

```
def filter(p: T => Boolean): List[T] = this match {  
    case Nil => this  
    case x :: xs =>  
        if p(x) x :: xs.filter (p)  
        else xs.filter (p)  
}
```

```
def positiveElements(xs: List[Int]): List[Int] =  
    xs.filter(x => x > 0)
```

# List. collect

Комбинацию операций **filter** + **map** можно заменить операцией **collect**:

```
def collect[B] (pf: PartialFunction[A, B]): List[B]
```

```
val list = List(1, -2, 3, -4)
```

```
list
  .filter(_ / 2 == 0)
  .map(_ + 1) //List(2)
```

```
list.collect {
  case x if x / 2 == 0 => x + 1
} //List(2)
```

# List. flatMap

---

Часто приходится иметь дело с функциями, которые при применении к элементу коллекции возвращают новую коллекцию с элементами:

Рассмотрим задачу формирования троек элементов:

*List(1, 2, 3) => List(1, 1, 1, 2, 2, 2, 3, 3, 3)*

```
def tripleElements[T](ys: List[T]): List[T] = ???
```

# List. flatMap

---

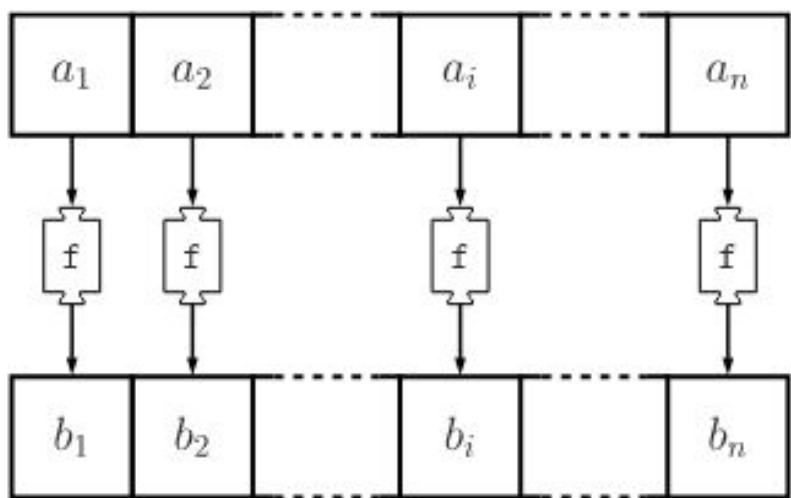
Операции модификации элементов с помощью функции, которая возвращает новый лист обобщенно можно выразить через метод **flatMap**:

```
def flatMap[U](f: T => List[U]): List[U] = this match {  
  case Nil => this  
  case x :: xs => f(x) ++ xs.flatMap(f)  
}
```

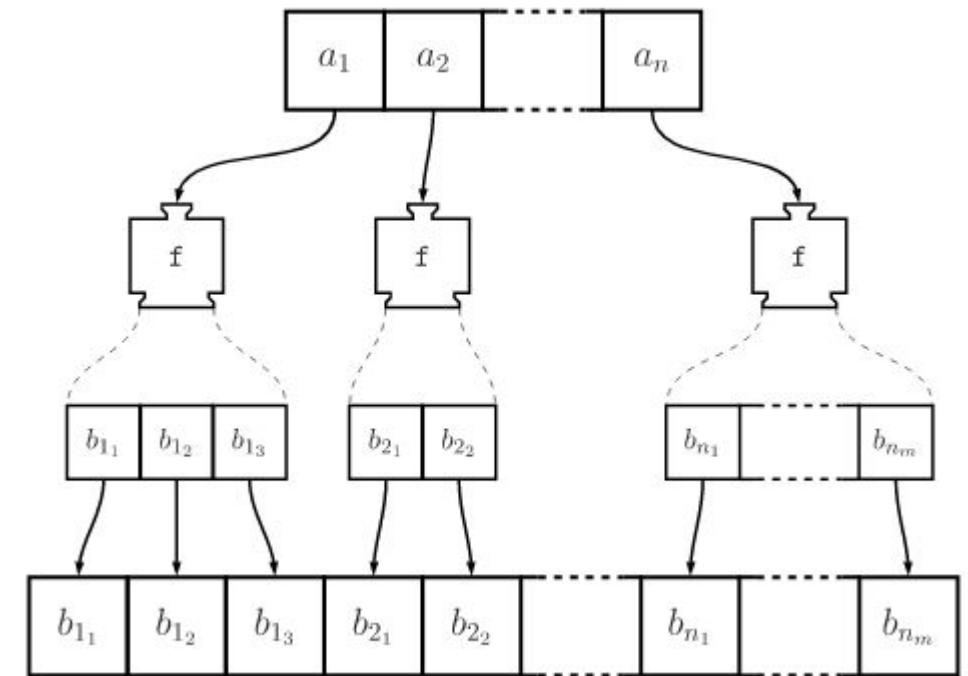
\* в реальности метод **flatMap** реализован сложнее из-за оптимизаций и специфики для разных коллекций

# List. map vs flatMap

```
def map[U](f: T => U): List[U]
```



```
def flatMap[U](f: T => List[U]): List[U]
```



# For comprehension

---

**For comprehension** - это синтаксический сахар, предназначенный для повышения читаемости кода, в случаях, когда необходимо итерироваться по одной или более коллекциям, **его** всегда можно заменить на эквивалентное выражение через функции:

**foreach, map, flatMap, filter, withFilter**

```
case class Person(name: String, age: Int)
```

```
val persons: List[Person] = ...
```

```
persons
  .filter(p => p.age > 20)
  .map(p => p.name)
```

```
for {
  p <- persons if p.age > 20
} yield p.name
```

# For comprehension. Desugaring

```
for (x <- c1; y <- c2; z <- c3) { ... }

c1.foreach(x => c2.foreach(y => c3.foreach(z => { ... })))

for (x <- c1; y <- c2; z <- c3) yield { ... }

c1.flatMap(x => c2.flatMap(y => c3.map(z => { ... })))

for (x <- c; if cond) yield { ... }

c.withFilter(x => cond).map(x => { ... })
```

Если у коллекции не реализован метод `withFilter`, то будет использован метод `filter`

```
c.filter(x => cond).map(x => { ... })
```

# Custom For comprehension

---

В **for comprehension** можно использовать не только коллекции но и другие классы, в частности можно добавить поддержку своим классам:

```
class Foo[R] { ??? }
```

```
val someFoo = new Foo[String]  
val someBar = new Foo[String]
```

```
for {  
    f <- someFoo  
    b <- someBar  
} yield f + b
```



свертки

# Option

---

**Option[T]** - это тип, который отражает факт неопределенности наличия элемента типа *T*

**Option** - очень эффективный метод избавиться от **NPE**

```
trait Option[+A]
case class Some[+A] (value: A) extends Option[A]
object None extends Option[Nothing]
```

- *Some[T]* - говорит о наличии элемента
- *None* - об отсутствии
- *Option("hello") == Some[String]("hello")*
- *Option(null) == None*
- *Some(null) == Some[Null](null)*



# List. queries

---

Основные операции извлечения элементов из коллекции:

- `def head: A` - берем первый элемент  
(небезопасно, если коллекция пустая, вылетит ошибка)
- `def headOption: Option[A]` - берем опционально  
первый элемент
- `def find(p: A => Boolean): Option[A]` - найти  
элемент удовлетворяющий некоторому критерию

# Aggregations

---

Часто при работе с коллекциями нам необходимо получить некоторое агрегированное значение, например, сумму элементов, рекурсивно это можно реализовать след. образом:

```
def sum(xs: List[Int]): Int = xs match {
  case Nil => 0
  case y :: ys => y + sum(ys)
}
```

Для подобных операций агрегирования над коллекциями существуют следующие методы:

```
def foldLeft[B](z: B)(op: (B, A) => B): B
def foldRight[B](z: B)(op: (A, B) => B): B
def reduceLeft[B >: A](op: (B, A) => B): B
def reduceRight[B >: A](op: (B, A) => B): B
```

# Aggregations. foldLeft

---

***foldLeft*** принимает некоторое начальное значение ***z*** (**аккумулятор**) и функцию ***op***, которая применяется к текущему состоянию аккумулятора и следующему значению:

$$List(x_1, \dots, x_n).foldLeft(z)(op) = z.op(x_1).op \dots .op(x_n)$$

```
def foldLeft[U](z: U)(op: (U, T) => U): U = this match {  
  case Nil => z  
  case x :: xs => xs.foldLeft(op(z, x))(op)  
}
```

# Aggregations. reduceLeft

---

**reduceLeft** то же самое что и **foldLeft**, но в качестве начального значения **z** выступает первый элемент коллекции:

*List(x<sub>1</sub>, ..., x<sub>n</sub>).reduceLeft(op) = x<sub>1</sub>.op(x<sub>2</sub>). ... .op(x<sub>n</sub>)*

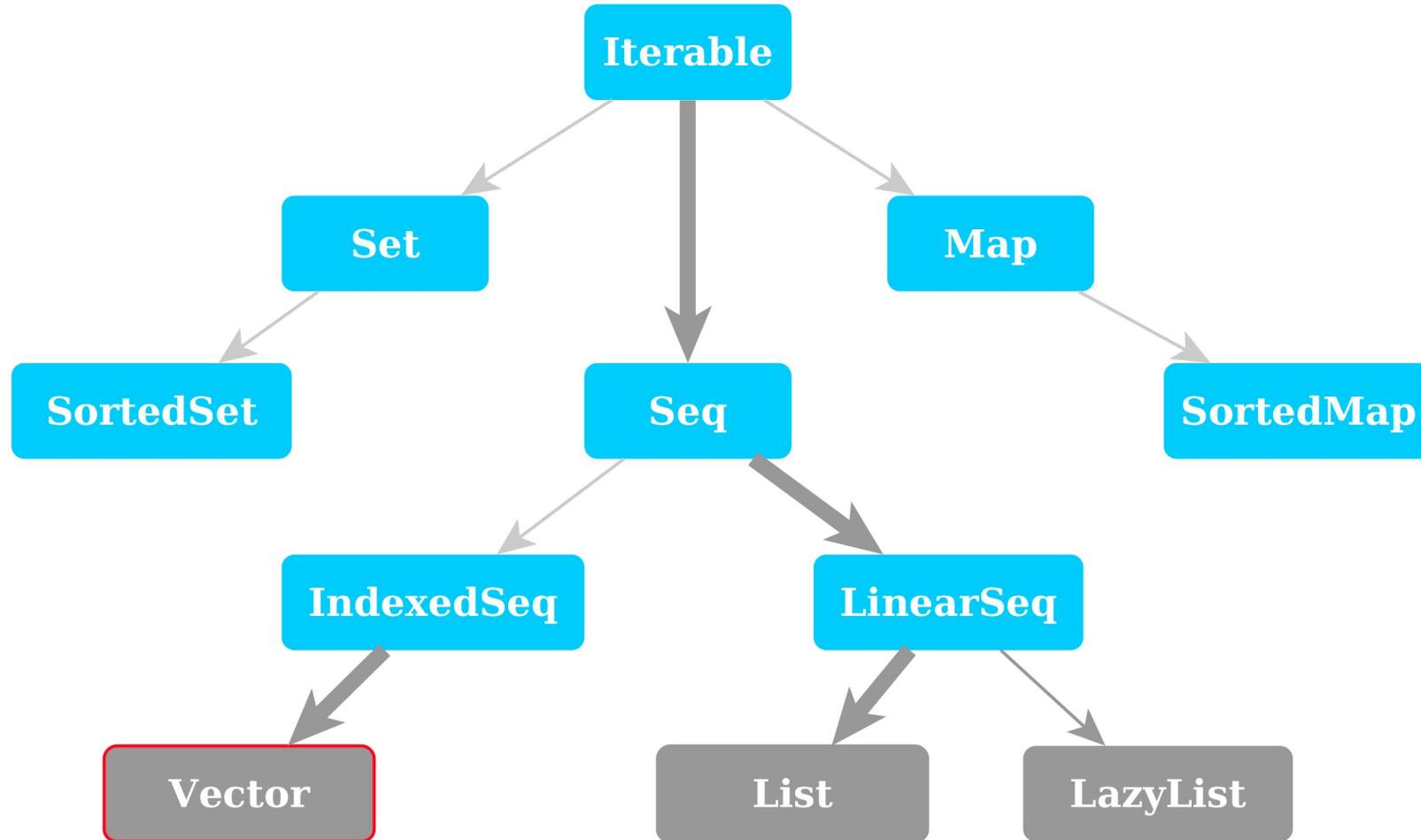
```
def reduceLeft(op: (T, T) => T): T = this match {  
  case Nil => throw new Error("Nil.reduceLeft")  
  case x :: xs => xs.foldLeft(x)(op)  
}
```



коллекции

# Indexed Seq. Vector

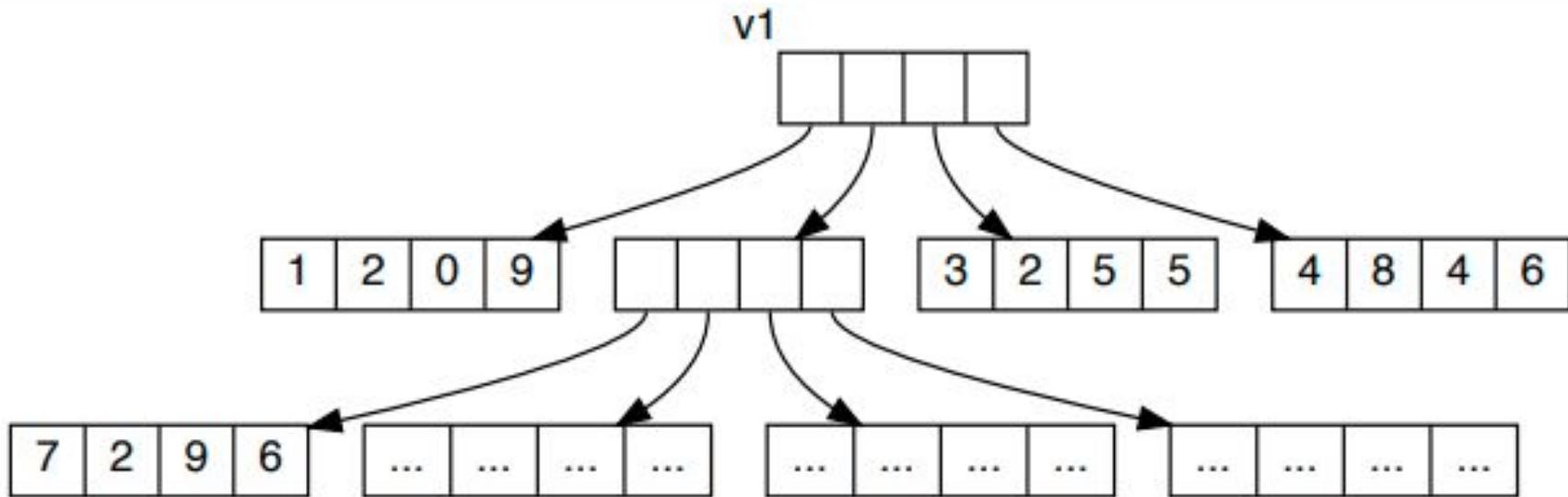
---



# Vector

**Vector** - неизменяемая коллекция, которая имеет сложность  $O(\log n)$  для большинства операций, в том числе при операциях изменения, префиксное дерево [Trie](#)

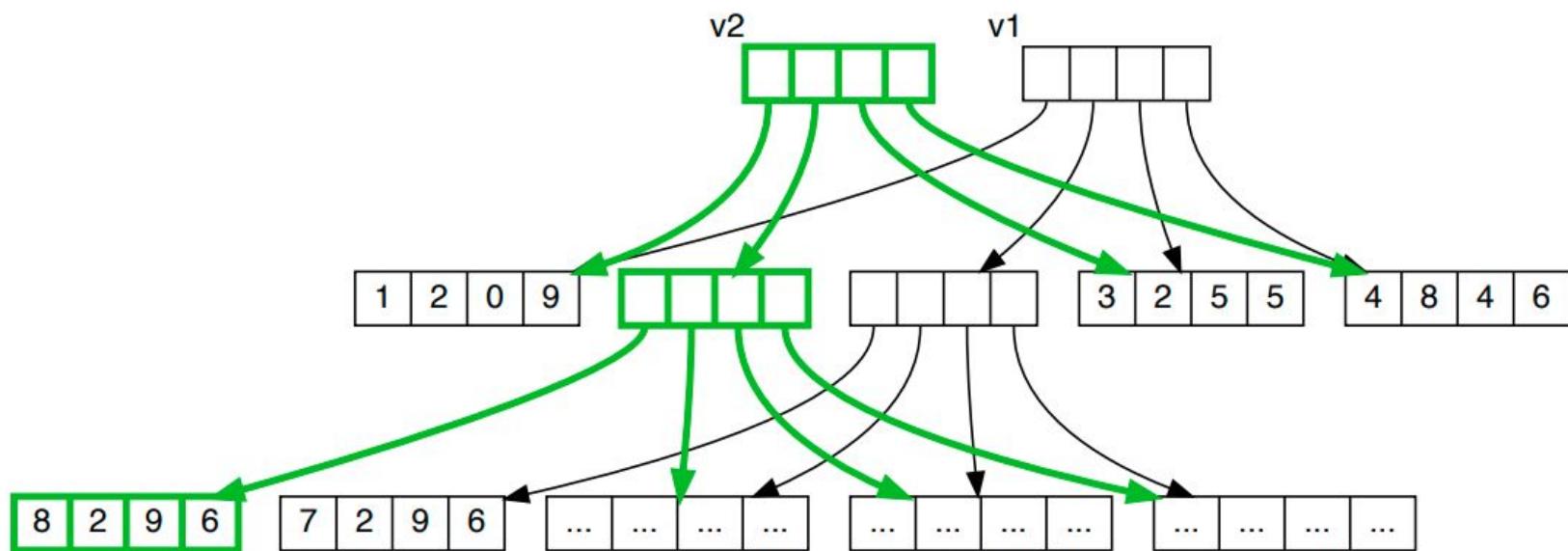
```
val v1 = Vector(1, 2, 0, 9, 7, 2, 9, 6, ..., 3, 2, 5, 5, 4, 8, 4, 6)
```



\* В действительности в каждом узле вектора содержится не 4 элемента, а 32

# Vector. Structural sharing

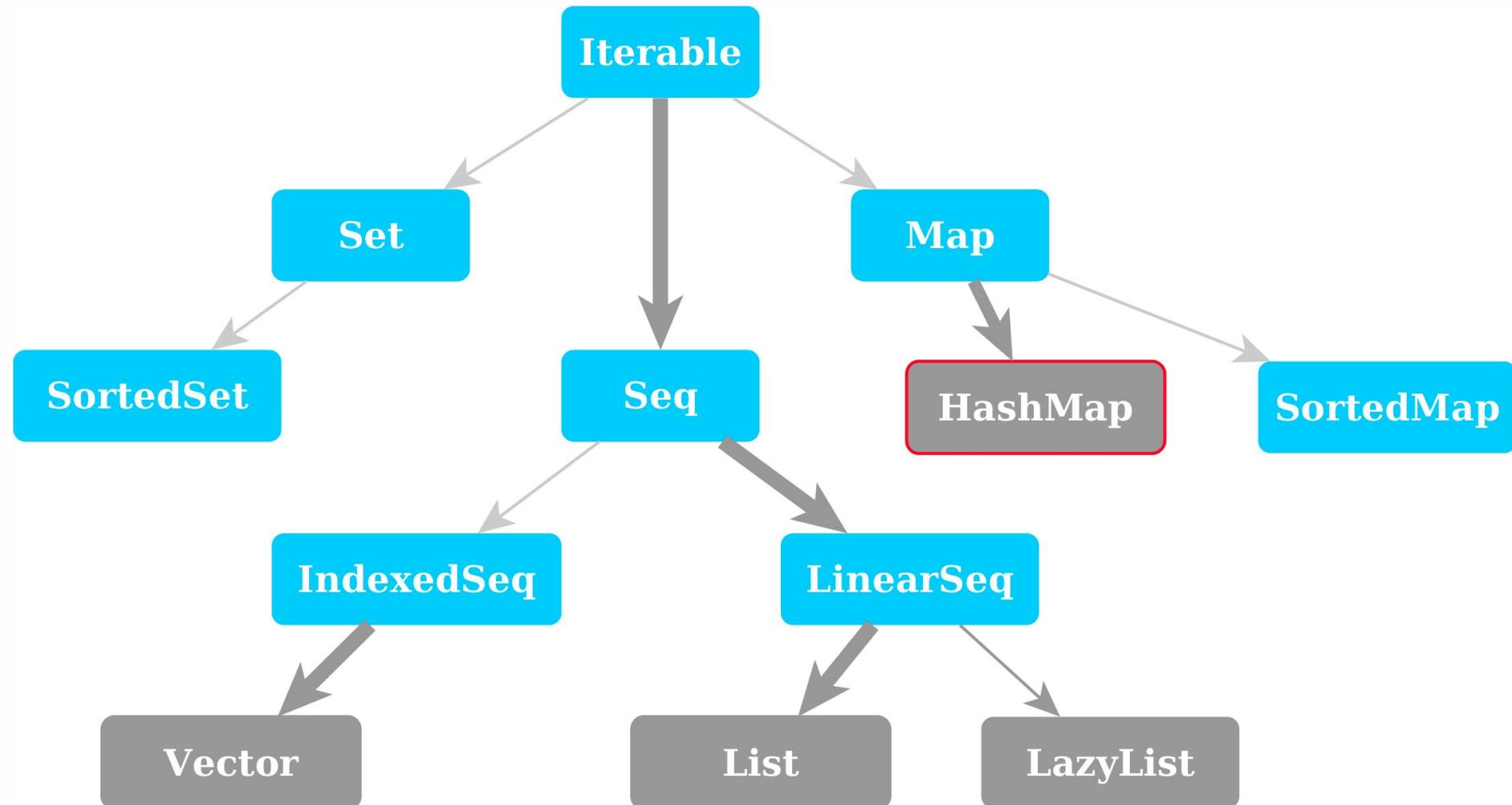
```
val v2 = v1.updated(4, 8)
```



Количество узлов, которые необходимо копировать, пропорционально высоте дерева в то время как остальные узлы будут переиспользованы. Данная процедура позволяет произвести изменение Vector за  $O(\log n)$ . Что намного более эффективно, чем стандартный алгоритм копирования массива  $O(n)$

# Map

---



# Map

---

Словарь *Map*[Key, Value] – также наследник *Iterable*[(Key, Value)], на нем определены рассмотренные ранее функции:

```
val capitalOfCountry = Map("US" -> "Washington", "Switzerland" -> "Bern")  
  
val countryOfCapital = capitalOfCountry.map { case (x, y) => (x, y) }  
// Map("Washington" -> "US", "Bern" -> "Switzerland")
```

*Map*[Key, Value] также является наследником функции Key => Value, т.е. *Map* можно использовать как функцию:

```
capitalOfCountry("US") // "Washington"
```

# Map

---

При этом использование *Map* как функции над несуществующим ключом выбрасывает исключение:

```
capitalOfCountry("Andorra")
// java.util.NoSuchElementException: key not found: Andorra
```

Поэтому для получения элемента из *Map* предпочтительно использовать метод *get*:

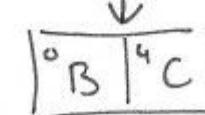
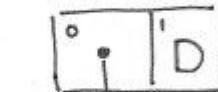
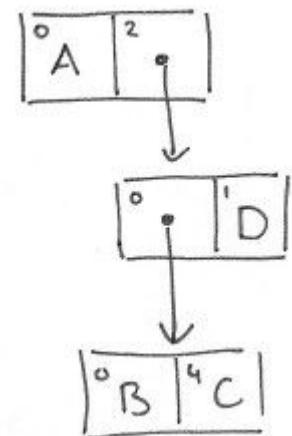
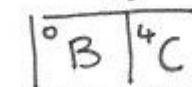
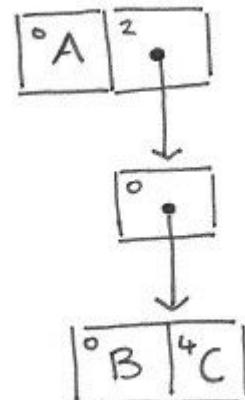
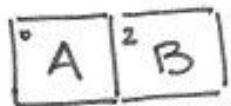
```
capitalOfCountry.get("US") // Some("Washington")
capitalOfCountry.get("Andorra") // None
```

# Map. Impl

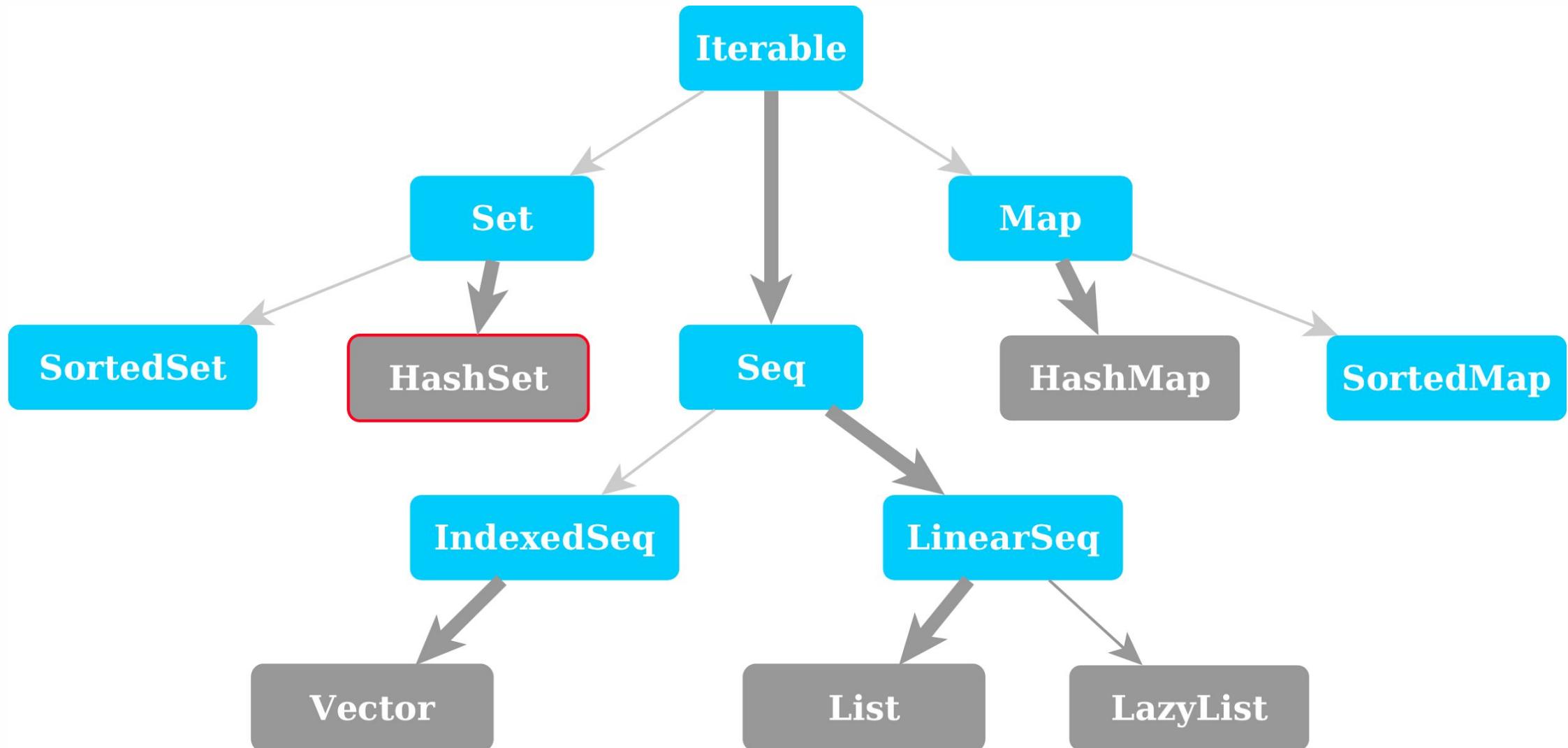
Дефолтной реализацией *Map* является *Compressed Hash-Array Mapped Prefix-trees (CHAMP)*, благодаря которой можно переиспользовать память при изменениях, подробнее: [CHAMP](#)

Пример построения такой структуры:

hash(A) = 010...      hash(B) = 200...      hash(C) = 204...      hash(D) = 210...



# Set



# Set

---

**Set[A]** (Множество) - еще одна фундаментальная коллекция

```
val fruit = Set("apple", "banana", "pear")  
val s = (1 to 6).toSet
```

Принципиальные различия между Set и Seq:

- Set - неотсортированное множество, элементы в ней не имеют порядка
- Set не имеет дубликатов:

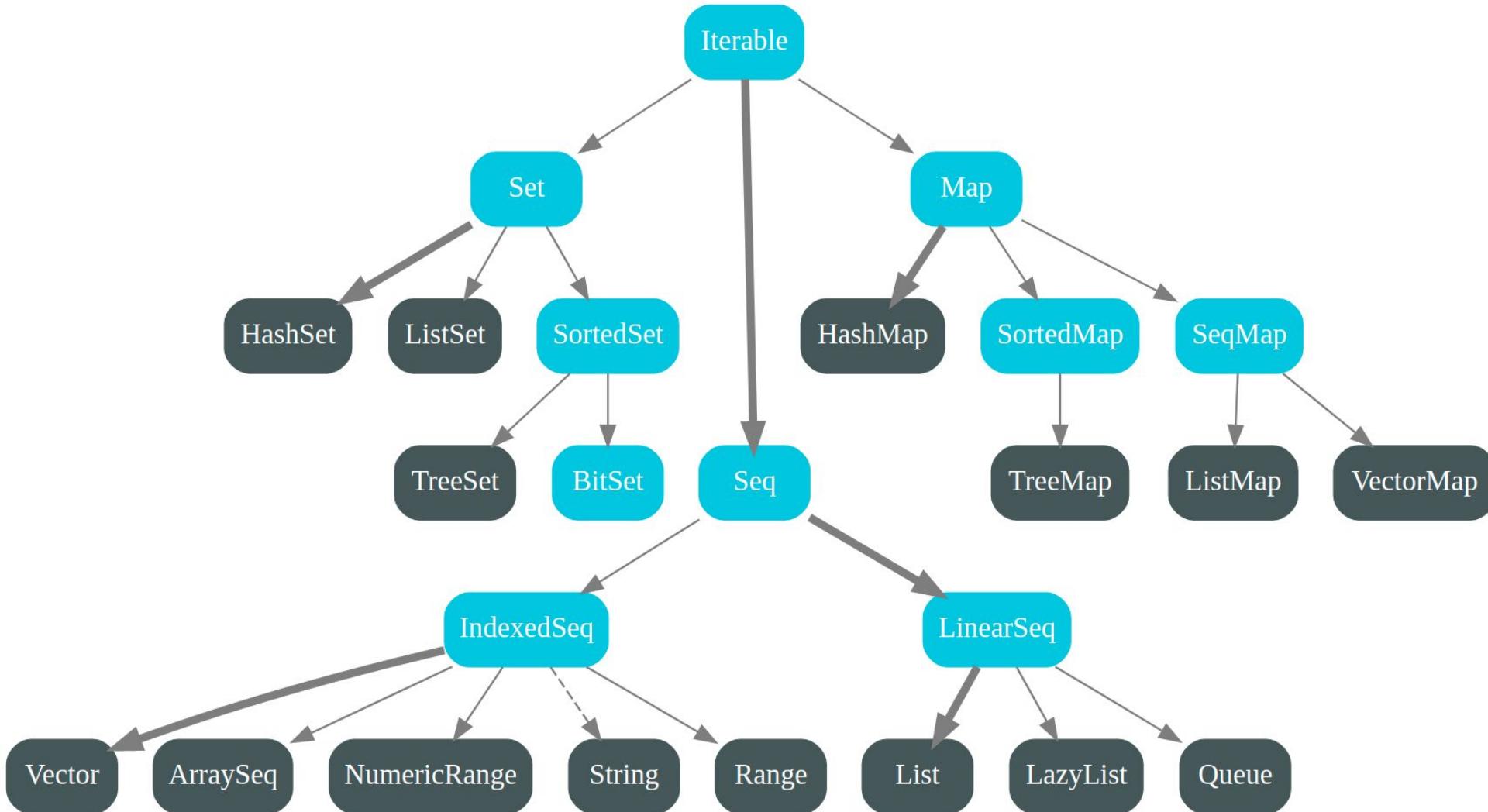
```
s.map(_ / 2) // Set(2, 0, 3, 1)
```

- Главная и эффективная операции на множество - это проверка на вхождение:

```
s.contains(5) // true
```

# Immutable collections

---



Спасибо за внимание