Lecture 5 – Immutable Collections

SWS121: Secure Programming

Jihyeok Park

PLRG

2024 Spring

Recall

- **Functions**
  - Methods vs Functions
  - Eta Expansion
  - Recursive Functions
  - Tail-Call Optimization
  - Default Parameter Values
  - Nested Methods
  - Multiple Parameter Lists
- **Pattern Matching**
  - Sealed Types
  - Regular Expression Patterns
  - Extractor Objects
- **Functional Error Handling**
  - Option Type
  - Try Type
  - Either Type

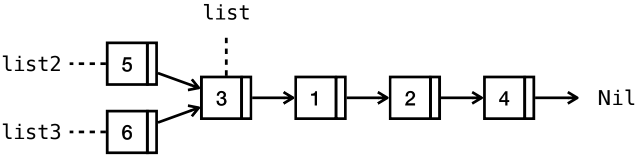
Contents

Contents

Lists

Lists are **immutable** sequences of elements of the same type

```scala
val list: List[Int]  = 3 :: 1 :: 2 :: 4 :: Nil
val list2: List[Int] = 5 :: list
val list3: List[Int] = 6 :: list
```



and support various **methods**:

```scala
list.size                        // 4                          : Int
list.map(_ * 2)                  // List(6, 2, 4, 8)           : List[Int]
list.filter(_ % 2 == 1)          // List(3, 1)                 : List[Int]
list.flatMap(x => List(x, -x))   // List(3, -3, ..., 4, -4)    : List[Int]
list.foldLeft(0)(_ + _)          // 0 + 3 + 1 + 2 + 4 = 10     : Int
```

Options, Maps, and Sets

We learned other basic immutable collections:

- `Option[T]`: represents **optional** values
- `Map[K, V]`: represents a collection of **key-value** pairs
- `Set[T]`: represents a collection of **unique** elements

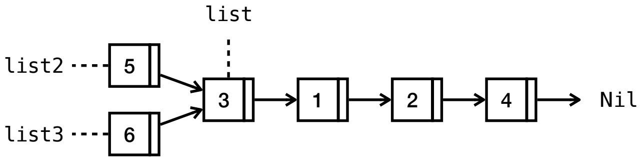
```
val opt: Option[Int]         = Some(5)
val map: Map[String, Int] = Map("one" -> 1, "two" -> 2)
val set: Set[Int]            = Set(1, 2, 3, 4)
```

and support similar **methods**:

```
opt.size                    // 1                         : Int
set.map(_ * 2)              // Set(2, 4, 6, 8)           : Set[Int]
map.filter((k, v) => v < 2) // Map("one" -> 1)           : Map[String, Int]
opt.flatMap(x => Some(x * 2)) // Some(10)                : Option[Int]
set.foldLeft(1)(_ * _)      // 1 * 1 * 2 * 3 * 4 = 24 : Int
```

Contents

Why Immutable Collections?



Why we should use **immutable** collections?

- **Thread Safety:** Since immutable collections cannot be modified once created, they are inherently thread-safe (e.g., no race conditions).
- **Security:** We can avoid bugs caused by unintended modifications from external libraries or other parts of the code.
- **Easier Debugging**: There is no need to trace changes in the code that might have altered the value of an immutable object.
- **Memory Efficiency**: Immutable collections are more memory-efficient as they can share common parts of their structure instead.

Contents

Collections Hierarchy

- All collection classes are found in the package `scala.collection` divided into **mutable** and **immutable** collections.
- By default, Scala always picks immutable collections.
- For example, `List` is an alias of the following:

```
List           // scala.collection.immutable.List
```

- `Set` without a prefix refers to an immutable collection, whereas `mutable.Set` refers to the mutable counterpart.

```
Set            // scala.collection.immutable.Set
mutable.Set    // scala.collection.mutable.Set
```

- Let's explore the collections hierarchy in Scala.

Trait Iterable

The **Iterable** trait is the root trait of all collection classes.
It defines the following **concrete methods**:

Category	Methods
Addition	`concat` (`++`)
Map	`map`, `flatMap`, `collect`
Conversions	`to`, `toList`, `toVector`, `toMap`, `toSet`, `toSeq`, `toIndexedSeq`, `toBuffer`, `toArray`
Copying	`copyToArray`
Size Info	`isEmpty`, `nonEmpty`, `size`, `knownSize`, `sizeIs`
Element	`head`, `last`, `headOption`, `lastOption`, `find`
Sub-collection	`tail`, `init`, `slice`, `take`, `drop`, `takeWhile`, `dropWhile`, `filter`, `filterNot`, `withFilter`
Subdivision	`splitAt`, `span`, `partition`, `partitionMap`, `groupBy`, `groupMap`, `groupMapReduce`
Element Tests	`exists`, `forall`, `count`
Folds	`foldLeft`, `foldRight`, `reduceLeft`, `reduceRight`
Specific Folds	`sum`, `product`, `min`, `max`
String Operations	`mkString`, `addString`
View	`view`

Trait Iterable

To support previous concrete methods, we need to implement the following **abstract method** called **iterator**:

```
def iterator: Iterator[A]
```

We need to implement following **abstract method** for the **Iterator** object:

```
def hasNext: Boolean  // Check if there is a next element available
def next(): A         // Return the next element and advance iterator
```

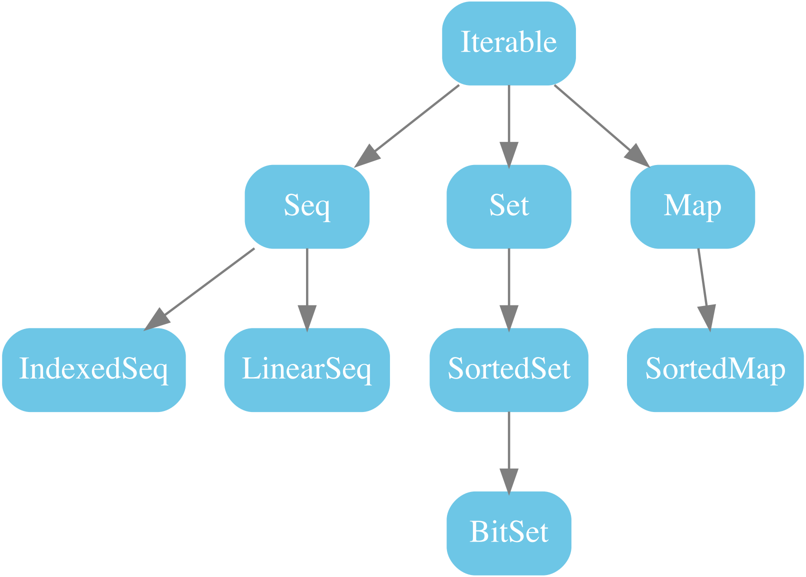
For example, the `headOption` method is implemented as follows:

```
def headOption: Option[A] =
  val it = iterator
  if (it.hasNext) Some(it.next()) else None
```

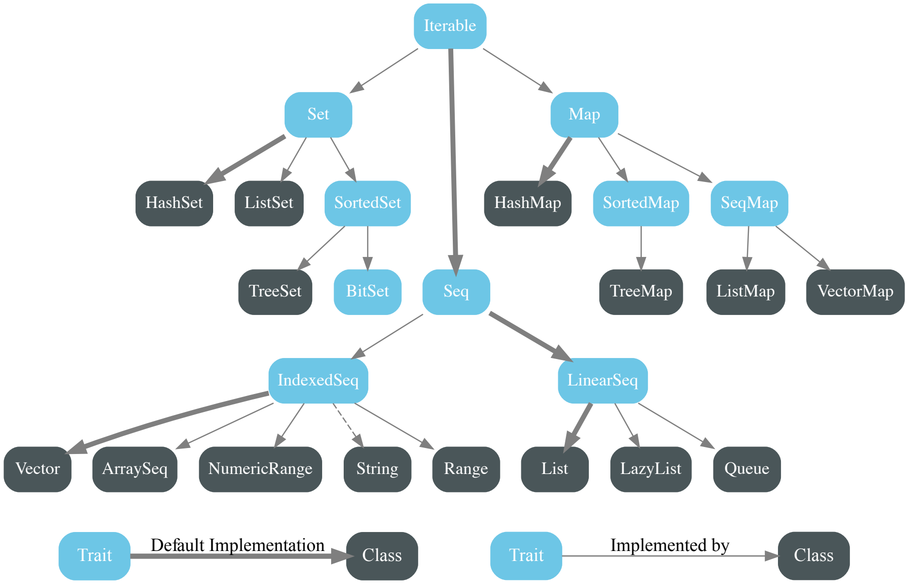
Then, we can use the `headOption` method as follows:

```
Nil.headOption            // None
List(1, 2, 3).headOption  // Some(1)
```

Collections Hierarchy



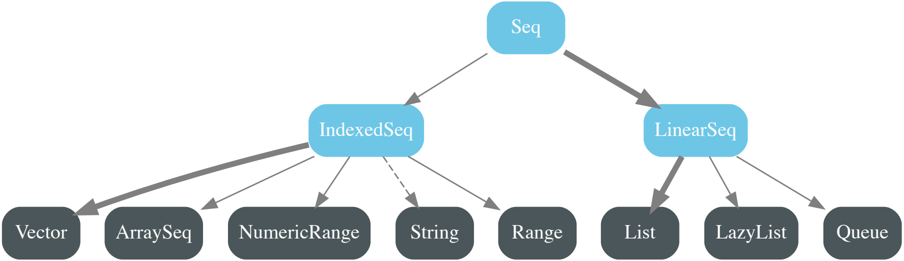
Immutable Collections Hierarchy



Contents

Sequences

PLRG



- **IndexedSeq**: A sequence of elements with efficient **random access**.

```
val seq: IndexedSeq[Int] = ArraySeq(0, 1, 2, 3, 4)
seq(3)                  // 3                (constant time)
```

- **LinearSeq**: A sequence of elements with efficient **linear access**.

```
val list: LinearSeq[Int] = List(0, 1, 2, 3, 4)
list.head               // 0                (constant time)
list.tail               // List(1, 2, 3, 4) (constant time)
```

Indexed Sequence – ArraySeq

ArraySeq is an **indexed sequence** backed by an **array**.

In memory, the elements are stored in a **contiguous block** of memory.

Consider the following example:

```
val arraySeq: ArraySeq[Int] = ArraySeq(3, 7, 1, 4, 2, 8, 5, 6, 9, 0)
```

Then, the elements are stored as follows:

3	7	1	4	2	8	5	6	9	0
0	1	2	3	4	5	6	7	8	9

Thus, the time complexity of `apply` is **constant time**.

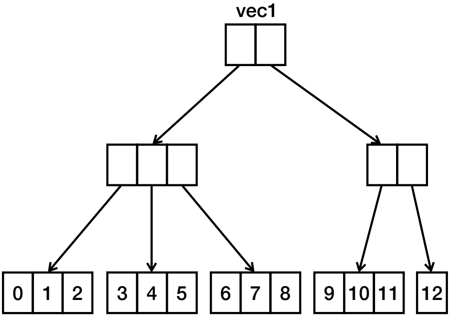
However, the time complexity of `prepend`, `update`, `prepend`, and `append` is **linear time** because we need to **copy all elements**.

Indexed Sequence – Vector

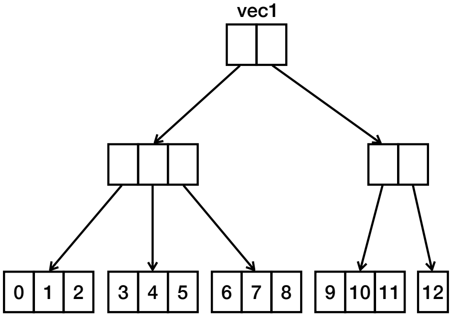
Vector is a indexed sequence collection type that provides good performance for all its operations.

Vectors are represented as *m*-**wide trees**. For example, a vector with 3-wide trees is shown below:

```
val vec1: Vector[Int] = Vector(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
```



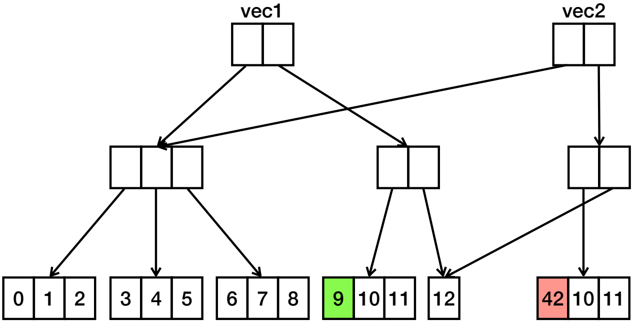
Indexed Sequence – Vector



The indexing operation is **effectively constant time** because the depth of the tree is **logarithmic** in the number of elements.

```
vec1(9)  // 9
```

Indexed Sequence – Vector



The update operation is also **effectively constant time** because the depth of the tree is **logarithmic** in the number of elements.

It is memory-efficient because it **shares common parts** of the tree.

```
val vec2 = vec1.updated(9, 42)
```

Indexed Sequence – Vector

PLRG

- In fact, vectors are represented as variants of 32-wide trees.[1]
- Vectors with up to 32 elements can be represented in a single node, and vectors with up to 32 * 32 = 1024 elements can be represented with a single indirection (hop).
- **Five hops** for vectors with up to $2^{30} \approx 1$ billion elements.
- So for all vectors of reasonable size, an element selection involves up to 5 primitive array selections.
- This is why the time complexity of element access is **effectively constant time**.

[1] At the first time, the **relaxed radix balanced (RRB) trees** (ICFP 2015) were used, but now they are replaced by the **radix-balanced finger (RBF) trees** (2019).

Indexed Sequence – Range

Range is a collection of **equally spaced** integers.

For example, consider the following range:

```scala
val range: Range = Range(2, 28, 3)
```

It represents the range **starting** from 2 and **ending** at 28 with a **step** of 3:

$$2, 5, 8, 11, 14, 17, 20, 23, 26$$

We can define ranges also using the methods (`to`, `until`, and `by`):

```scala
0 to 10         // Range(0, 1, 2, ..., 10)
0 to 10 by 2    // Range(0, 2, 4, 6, 8, 10)
0 until 10      // Range(0, 1, 2, ..., 9)
0 until 10 by 2 // Range(0, 2, 4, 6, 8)
```

The time complexity of `apply` is **constant time** because we can compute the element using the formula:

$$\texttt{start} + \texttt{step} \times \texttt{index}$$

Linear Sequence – Queue

We can treat a **List** as a **stack** by using the `::` operator for **pop** and `head`/`tail` methods for **push**.

By combining **two lists**, we can implement a **queue**:

- The **enqueue** operation is implemented by **pushing** the elements to the `in` list.
- The **dequeue** operation is implemented by 1) **moving** the elements from the `in` list to the `out` list only when the `out` list is empty and 2) **poping** an element from the `out` list.

Linear Sequence – Queue

```scala
val q1 = Queue().enqueue(1).enqueue(2).enqueue(3)
```

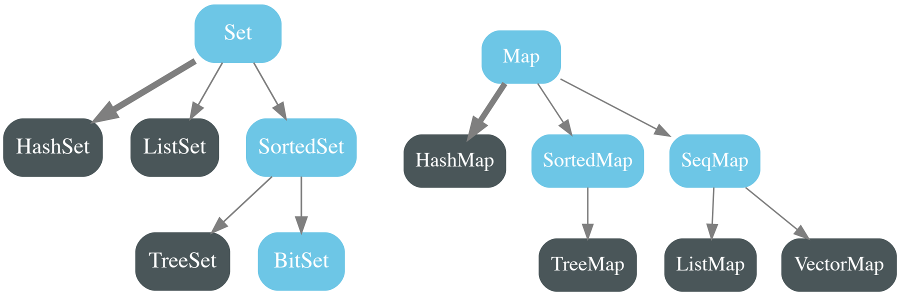

```scala
val (x, q2) = q1.dequeue       // x == 1
```


```scala
val q3 = q2.enqueue(4).enqueue(5).enqueue(6)
```


```scala
val (y, q4) = q3.dequeue       // y == 2
```

Contents

Sets and Maps



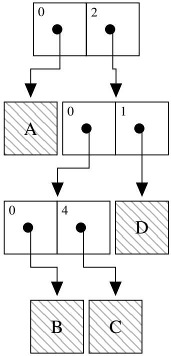
- **HashSet/HashMap**: A set/map of elements with **no order**.
- **TreeSet/TreeMap**: A set/map of elements with **sorted order**
- **BitSet**: A set of bits with **dense packing**.
- **VectorMap**: A map of elements with **insertion order**.

HashSet and HashMap

HashSet and **HashMap** are sets and maps of elements with **no order** using a **compressed hash-array mapped prefix-tree (CHAMP)**[2], which is a variant of the **hash-array mapped trie (HAMT)**.

Following shows an example of a HAMT with 32-ary nodes:

$$
\begin{aligned}
hash(A) &= 32_{10} &= \; 0 \;\; 1 \;\; 0 \;\; {}_{32} \\
hash(B) &= 2_{10} &= \; 2 \;\; 0 \;\; 0 \;\; {}_{32} \\
hash(C) &= 4098_{10} &= \; 2 \;\; 0 \;\; 4 \;\; {}_{32} \\
hash(C) &= 34_{10} &= \; 2 \;\; 1 \;\; 0 \;\; {}_{32}
\end{aligned}
$$



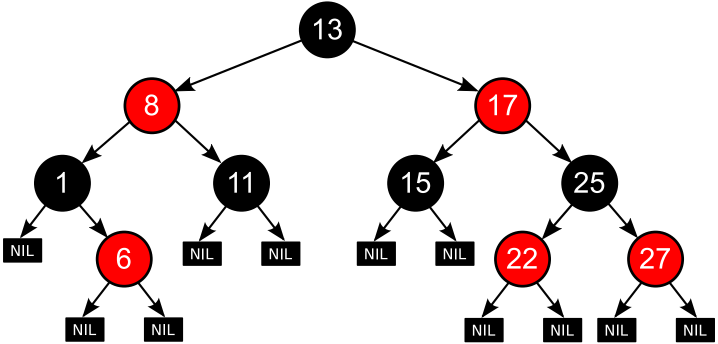
[2]The **CHAMP** (OOPSLA 2015) data structure is a variant of the **HAMT**.

TreeSet and TreeMap

TreeSet and **TreeMap** are sets and maps of elements with **sorted order** using **red-black trees**.

For example, the following set is represented as a red-black tree:

```
val set = Set(1, 6, 8, 11, 13, 15, 17, 25, 22, 27)
```



Contents

Performance Characteristics

	`head`	`tail`	`apply`	`update`	`prepend`	`append`
List	C	C	L	L	C	L
ArraySeq	C	L	C	L	L	L
Vector	eC	eC	eC	eC	eC	eC
Queue	aC	aC	L	L	L	C
Range	C	C	C	-	-	-
String	C	L	C	L	L	L

	`lookup`	`add`	`remove`	`min`
HashSet/HashMap	eC	eC	eC	L
TreeSet/TreeMap	Log	Log	Log	Log
BitSet	C	L	L	eC[3]
VectorMap	eC	eC	aC	L
ListMap	L	L	L	L

where **L** = linear time, **Log** = logarithmic time, **C** = constant time, **eC** = effectively constant time, and **aC** = amortized constant time.

[3] Assuming bits are densely packed.

Summary

1. Recall: Basic Immutable Collections
   - Lists, Options, Maps, and Sets
2. Why Immutable Collections?
3. Collections Hierarchy
4. Sequences
   - ArraySeq
   - Vector
   - Range
   - Queue
5. Sets and Maps
   - HashSet and HashMap
   - TreeSet and TreeMap
6. Performance Characteristics

Next Lecture

- For Comprehensions

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr