Other Collections

Vectors

- We have seen that lists are linear. Access to the first element is much faster than access to the middle or end of a list
- The Scala library also defines an alternative sequence implementation – Vector
- This one has even more evenly balanced access patterns than List
- The idea with a vector is that it's essentially a tree with a very high branch out factor
- If the vector is small up to 32 elements, then it's just an array
- If the vector grows beyond 32 elements, then it becomes an array of arrays, each of which has 32 elements. We have 32 x 32 = 1024 elements in the array
- If the vector grows beyond that, then each of these upper race will again spawn 32 children of 32 children each, and so on
- The vector could have a maximum of five levels which would give you $2^{5 \times 5} = 2^{25}$ elements. That's the maximum size of a vector
- Let's say you want to change an element. What you need to do is essentially create a new array of 32 elements which contains the changed element, and then its parents need to change as well
- It's not free to change your single element functionally; in general, you have to modify as many arrays as in the depth of your tree

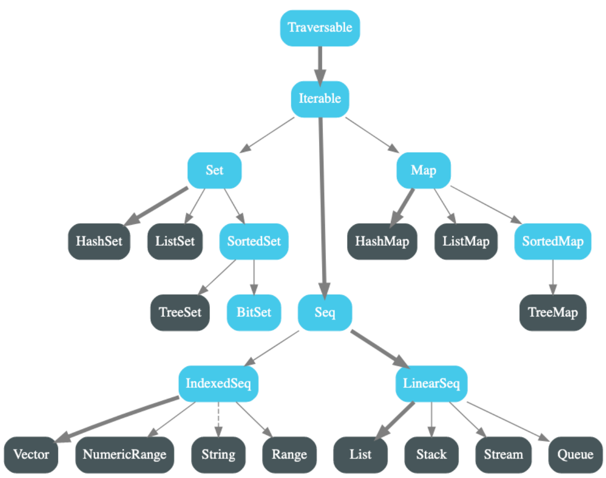
Operations on Vectors

- Vectors are created analogously to lists:

  val nums = Vector(1, 2, 3, -88)

  val people = Vector("Bob", "James", "Alice")
- They support the same operations as lists, with the exception of ::
- Instead of ***x :: xs***, there is:
  - ***x +: xs*** – create a new vector with leading element x, followed by all elements of xs
  - ***xs :+ x*** – create a new vector with trailing element x, preceded by all elements of xs
  - Note that the : always points to the sequence

Collections Hierarchy



Arrays and Strings

- Arrays and Strings support the same operations as ***Seq*** and can implicitly be converted to sequences where needed
- They cannot be subclasses of ***Seq*** because they come from Java

Ranges

- Another simple kind of sequence is the ***Range***
- It represents a sequence of evenly spaced integers
- Three operations: ***to***(inclusive), ***until***(exclusive), ***by***(to determine step value):

  val r: Range = 1 until 5 --> 1, 2, 3, 4
  val s: Range = 1 to 5 --> 1, 2, 3, 4, 5
  1 to 10 by 3 --> 1, 4, 7, 10
  6 to 1 by -2 --> 6, 4, 2
- A ***Range*** is represented as a single object with three fields: lower bound, upper bound and step value

Some more Sequence Operations

- ***xs.exists(p)*** – true if there is an element x of xs such that p(x) holds, false otherwise
- ***xs.forall(p)*** – true if p(x) holds for all elements x of xs, false otherwise
- ***xs.zip(ys)*** – a sequence of pairs drawn from corresponding elements of sequences xs and ys; if one of them is longer than the other, then it's truncated to make them fit
- ***xs.unzip*** – splits a sequence of pairs xs into two sequences consisting of the first, respectively second halves of all pairs
- ***xs.flatMap(f)*** – applies collection-valued function f to all elements of xs and concatenates the results
- ***xs.sum*** – the sum of all elements of this numeric collection
- ***xs.product*** – the product of all elements of this numeric collection
- ***xs.max*** – the maximum of all elements of this collection
- ***xs.min*** – the minimum of all elements of this collection

Example: Combinations
To list all combinations of numbers x and y where x is drawn from 1...M and y is drawn from 1…N:

```
(1 to M).flatMap(x => (1 to N).map(y => (x, y)))
```

Example: Scalar Product
To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map((x, y) => x * y).sum
```

Note that there is some automatic decomposition going on here.
Each pair of elements from xs and ys is split into its halves which are then passed as the x and y parameters to the lambda.
If we wanted to be more explicit, we could also write scalar product like this:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map(xy => xy._1 * xy._2).sum
```

On the other hand, if we wanted to be more concise, we could also write it like this:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
  xs.zip(ys).map( _ * _ ).sum
```

Exercise: *A number is prime if the only divisors of n are 1 and n itself. What is a high-level way to write a test for primality of numbers? For once, value conciseness over efficiency.*

```
def isPrime(n: Int): Boolean =
  (2 to n - 1).forall(n % _ != 0)
```