# Basic Data Types

## Sequences/Lists/Arrays

consecutive items in memory

    val k = List(1, 2, 3, 4)

-> size - bounded by memory

-> items inserted at end

-> can be sorted

## Sets

no particular order; no repeated values

    val s = Set(1, 2, 3, 4)

-> size - bounded by memory

-> can be queried

-> operations: union, intersection, difference, subset

## Maps/Dictionaries/Associative Arrays

collection of (k,v) pairs; unique k values

    val m = Map((“a” -> 1), (“b” -> 2))

-> one key - one value

-> accessing key - almost O(1)

## Nested Data Types

### Graphs

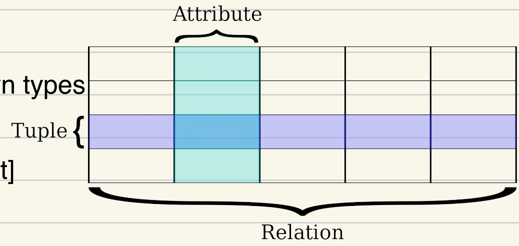
set of vertices/nodes, set of (un)ordered pairs of vertices for (un)directed graph

    Map[Node, List(Edge)]
    case class Node(id: Int, attributes: Map[A, B])
    case class Edge(a: Node, b: Node, directed: Boolean, weight: Double)

### Trees

    book = {“id”: “1”, “title”: “abc”, “author”: {“id”: 2, “name”: “xyz”}}
    Map(“id” -> “1”, “title” -> “abc”, “author -> Map(“id” -> 2, “name” -> “xyz”))

## Tuples




n-tuple - sequence of n elements with known types

val record = (1, “Matt”, “Damon”, 1970)

val record = Tuple4[Int, String, String, Int]

## Relations

set of n-tuples of the same type; one of tuples elements - (unique value) key

val movie1 = (1, “Martian”, 2015, 1)

val movie2 = (2, “Prometheus”, 2012, 1)

val director1 = (1, “Ridley Scott”, 1928)

val movies = Set(movie1, movie2) - cannot add director1 to the relation

## Key/Value Pairs

general relation type; non-unique value keys

val a = (1, (“Martian”, 2015))

val b = (1, (“Prometheus”, 2012))

val kv = List(a,b) - List[(Int, (String, Int))]

val xs = Map(1 -> List((“Martian”, 2015), (“Prometheus”, 2012)))

Map[Int, List[(String, Int)]]

# Functional Programming

programming paradigm; programs constructed by applying and composing functions

-> given an argument, always returns same result irrespective of its environment

-> immutable data structures - functions don’t modify the environment

-> higher-order functions - functions as arguments; parametrised behaviour

-> laziness - waiting to compute until the very end

## Function Signatures

foo(x: [A], y: B) -> C

foo - function name

x, y - function arguments

A,B - types of the arguments

-> - denotes return type

C - type of the returned result

[A] - type A is traversed aka list/array of type A

Function foo takes as arguments an array/list of type A and an argument of type B and returns an argument of type C.

f(x: [A], y: (z: A) -> B) -> [B]

Function f takes as arguments an array/list of type A and a function, which takes an argument of type A and returns an argument of type B, and returns an array/list of type B.

## Pure Functions

depends only on its declared inputs and its internal algorithm to produce its output

-> referential transparency - expression that when replaced by its value, doesn't affect the behaviour of the program

## From IP to FP

loops:

-> recursive functions with mutable iterator variables

-> tail-recursive functions optimised by compiler

## Monads

design pattern defining generic types build

Option[T] - null points -> return Some[T] or None

Try[T] - exceptions -> return Success[T] or Failure[E]

Future[T] - latency in asynchronous actions