# containers: Maps, Sets, and more

In order to solve almost any problem, that requires manipulation of data, the very first question should be: What is the most suitable data structure that we can use for the problem at hand?

## Preface

Haskell is a pure functional language, thus, by it's nature, most of the available containers are immutable and, without a doubt, the most common one is a list `[a]`. Certainly, it is not efficiency that made it so popular, but rather its simplicity, consequently, it is also the first da structure, that you get introduced to while learning Haskell. Although, lists are perfectly suitable for some problems, more often than not, we need something that is tailored to how we are trying to use our data.

Here are some situations that containers package can be of help. It provides efficient implementation of some of the most commonly use containers used in programming:

- `Data.Set` - you care about uniqueness and possibly the order of elements.
- `Data.Map` - you need a mapping from unique keys to values and operations you perform can take advantage of ordering of keys.
- `Data.IntSet` and `Data.IntMap` - just as above, but when elements and keys respectively are `Int`s.
- `Data.Sequence` - can be of use when a linear structure is required for a finite number of elements with fast access from both of it sides and a fast concatenation with other sequences.
- `Data.Tree` and `Data.Graph` - for describing more complicated relations of elements.

## Map

`Map` is one of the most interesting and commonly used abstractions from the package, so most of the examples will be based on it. Moreover, interface provided for `Map`, `IntMap`, `Set` and `IntSet` is very similar, so analogous examples can be easily derived for all of the above.

### Setup a problem.

One of the common mappings in real life, that we encounter, is a person's identification number, that maps a unique number to an actual human being. Social Security Number (SSN) is what normally used for that purpose in the USA and despite that it is not totally unique, fo demonstration purpose, we can assume it actually is. Although it is a 9 digit number and using `IntMap` would be more efficient, it does have some structure to it and we will take advantage of it, so we will use a custom data type `SSN` as a key.

```haskell
import qualified Data.Map as Map
import qualified Data.Set as Set
import Data.List as List
import Data.Monoid
import Text.Printf

-- | Social Security Number. Commonly used as a unique identification number of a
-- person.
data SSN = SSN
  { ssnPrefix :: Int
  , ssnInfix :: Int
  , ssnSuffix :: Int
  } deriving (Eq, Ord)

instance Show SSN where
  show (SSN p i s) = printf "%03d-%02d-%04d" p i s

data Gender = Male | Female deriving (Eq, Show)

data Person = Person
  { firstName :: String
  , lastName :: String
  , gender :: Gender
  } deriving (Eq)

instance Show Person where
  show (Person fName lName g) = fName ++ ' ':lName ++ " (" ++ show g ++ ")"

type Employees = Map.Map SSN Person
```

I would like to stress how important `Eq` and `Ord` instances of a data type used as a key actually are. Because they are essential for underlying representation of a `Map`, providing incomplete or incorrect instances for these classes will lead to some strange behavior of your data mappings, therefore, either make sure you know what you are doing when creating custom instances, or use derived instances, as they are always safe.

Because Social Security Numbers have a specific structure we would like to enforce it by providing a constructor function that performs certain validations.

```haskell
mkSSN :: Int -> Int -> Int -> SSN
mkSSN p i s
  | p <= 0 || p == 666 || p >= 900 = error $ "Invalid SSN prefix: " ++ show p
  | i <= 0 || i > 99 = error $ "Invalid SSN infix: " ++ show i
  | s <= 0 || s > 9999 = error $ "Invalid SSN suffix: " ++ show s
  | otherwise = SSN p i s
```

## Converting maps

Let's go ahead and create our mapping of employees using `Map.fromList`:

```
employees :: Employees
employees =
  Map.fromList
    [ (mkSSN 525 21 5423, Person "John" "Doe" Male)
    , (mkSSN 521 01 8756, Person "Mary" "Jones" Female)
    , (mkSSN 585 11 1234, Person "William" "Smith" Male)
    , (mkSSN 525 15 5673, Person "Maria" "Gonzalez" Female)
    , (mkSSN 524 34 1234, Person "Bob" "Jones" Male)
    , (mkSSN 522 43 9862, Person "John" "Doe" Male)
    , (mkSSN 527 75 1035, Person "Julia" "Bloom" Female)
    ]
```

As you can see above, there is no particular order to our data as we defined it, which results in creation of a `Map` in *O(n*log n)* time complexity, but, if we were sure ahead of time, that our list was sorted and unique with respect to the first element of a tuple, it would be more efficient to use `Map.fromAscList`, which would run in *O(n)* complexity instead.

## Operate on data.

Now that we have our program properly set up, most of available functions will correspond directly to functions that we might try to use our data, e.g.:

```
lookupEmployee :: SSN -> Employees -> Maybe Person
lookupEmployee = Map.lookup
```

which does exactly what is expected of it:

```
λ> lookupEmployee (mkSSN 524 34 1234) employees
Just Bob Jones (Male)
λ> lookupEmployee (mkSSN 555 12 3456) employees
Nothing
```

In order to refrain from redefining functions which trivially correspond to existing ones, let's go through some of them:

- Checking presence of an employee by the social security number:

```
λ> mkSSN 585 11 1234 `Map.member` employees
True
λ> mkSSN 621 24 8736 `Map.member` employees
False
```

- Looking up an employee with a default name:

```
λ> Map.findWithDefault (Person "Bill" "Smith" Male) (mkSSN 585 11 1234) employees
William Smith (Male)
λ> Map.findWithDefault (Person "Anthony" "Richardson" Male) (mkSSN 621 24 8736) employees
Anthony Richardson (Male)
```

- Getting total number of employees.

```
λ> Map.size employees
7
```

- Deleting an employee:

```
λ> Map.size $ Map.delete (mkSSN 585 11 1234) employees
6
λ> Map.size $ Map.delete (mkSSN 621 24 8736) employees
7
```

- Adding an employee:

```
λ> Map.size $ Map.insert (mkSSN 621 24 8736) (Person "Anthony" "Richardson" Male) employees
8
```

## Folding

It would be useful to present our `Employees` in a user friendly format, so let's define a `showMap` function by converting our `Map` to a printable string:

```
showMap :: (Show k, Show v) => Map.Map k v -> String
showMap = List.intercalate "\n" . map show . Map.toList
```

Let's give it a try:

```
λ> putStrLn $ showMap employees
(521-01-8756,Mary Jones (Female))
(522-43-9862,John Doe (Male))
(524-34-1234,Bob Jones (Male))
(525-15-5673,Maria Gonzalez (Female))
(525-21-5423,John Doe (Male))
(527-75-1035,Julia Bloom (Female))
(585-11-1234,William Smith (Male))
```

As you can see, all employees are sorted by their SSN, so conversion to a list is done in ascending order, but it is worth noting, that if ther is a desire to guarantee this behavior, `Map.toAscList` should be used instead, or `Map.toDescList` to get it in a reverse order.

Conversion is nice an simple, but how about using folding in a way that is native to a `Map`? While we are at it, we should probably improve formatting as well.

```
showEmployee :: (SSN, Person) -> String
showEmployee (social, person) =
  concat [show social, ": ", show person]

showEmployees :: Employees -> String
showEmployees es
  | Map.null es = ""
  | otherwise = showE ssn0 person0 ++ Map.foldrWithKey prepender "" rest
  where
    showE = curry showEmployee
    ((ssn0, person0), rest) = Map.deleteFindMin es
    prepender key person acc = '\n' : showE key person ++ acc

printEmployees :: Employees -> IO ()
printEmployees = putStrLn . showEmployees
```

Now that looks slightly better:

```
λ> printEmployees employees
521-01-8756: Mary Jones (Female)
522-43-9862: John Doe (Male)
524-34-1234: Bob Jones (Male)
525-15-5673: Maria Gonzalez (Female)
525-21-5423: John Doe (Male)
527-75-1035: Julia Bloom (Female)
585-11-1234: William Smith (Male)
```

**Exercise**: Using the fact that List is an instance of `Monoid` and implement `showEmployees` with the help of `Map.foldMapWithKey`.

**Exercise**: Implement `showEmployeesReversed` using `Map.foldlWithKey` (*hint*: use `Map.deleteFindMax`).

## Mapping

There is a collection of mapping functions available, which range from a simple `Map.map` to a more complex `Map.mapAccumRWithKey`.

Because `Map` is an instance of `Functor` we can use `fmap` for mapping a function over it's values, but for this example we will use it's native equivalent `Map.map` to retain only last names of employees and `Map.elems` to retrieve a list of new elements:

```
λ> Map.elems $ Map.map lastName employees
["Jones","Doe","Jones","Gonzalez","Doe","Bloom","Smith"]
```

If for some reason, we would like to map a function over keys, we can use `Map.mapKeys` for this purpose, for instance getting a list of all SSN prefixes:

```
λ> Map.keys $ Map.mapKeys (show . ssnPrefix) employees
["521","522","524","525","527","585"]
```

We need to pay some extra attention to usage of `Map.mapKeys`, because whenever a function that is being mapped over is not 1-to-1, it possible that some values will be lost. Although, sometimes discarding some elements maybe desired or simply irrelevant, here is an example of how we can mistakenly lose an employee if we assume that last four numbers of a social are unique:

```
λ> putStrLn $ showMap $ Map.mapKeys ssnSuffix employees
(1035,Julia Bloom)
(1234,William Smith)
(5423,John Doe)
(5673,Maria Gonzalez)
(8756,Mary Jones)
(9862,John Doe)
```

If we are sure that our function is not only 1-to-1, but it is also monotonic (i.e. it doesn't change the order of the resulting keys) we could use a more efficient mapping function `Map.mapKeysMonotonic`. Say a `show` function on `SSN` would be safe to use, since ordering woul be preserved. One of the simplest examples of a non-monotonic function would be `negate` and a strictly-monotonic one: `succ`.

# Filtering

Let's start with a couple of simple examples:

```
λ> printEmployees (Map.filter (("Jones"==) . lastName) employees)
521-01-8756: Mary Jones (Female)
524-34-1234: Bob Jones (Male)
```

Partitioning by gender:

```
λ> let (men, women) = Map.partition ((Male==) . gender) employees
λ> printEmployees men
522-43-9862: John Doe (Male)
524-34-1234: Bob Jones (Male)
525-21-5423: John Doe (Male)
585-11-1234: William Smith (Male)
λ> printEmployees women
521-01-8756: Mary Jones (Female)
525-15-5673: Maria Gonzalez (Female)
527-75-1035: Julia Bloom (Female)
```

Prior to June 25th, 2011, Social Security prefixes, were restricted to states where they were issued in. Let's assume this is still the case an use this information to figure out which states our employees received their Social Security Cards in.

First, we need to define a function, that retrieves employees within a prefix range, so a naïve approach would be to use `Map.filterWithKey`:

```
withinPrefixRangeNaive :: Int -> Int -> Employees -> Employees
withinPrefixRangeNaive prefixLow prefixHigh = Map.filterWithKey ssnInRange where
  ssnInRange (SSN prefix _ _) _ = prefix >= prefixLow && prefix <= prefixHigh
```

which runs in *O(n)*, but we can do better than that, simply by taking advantage of ordering of keys:

```
withinPrefixRange :: Int -> Int -> Employees -> Employees
withinPrefixRange prefixLow prefixHigh =
  fst . Map.split (SSN (prefixHigh + 1) 0 0) . snd . Map.split (SSN prefixLow 0 0)

employeesFromColorado :: Employees -> Employees
employeesFromColorado = withinPrefixRange 521 524
```

Naturally, this function will give us all employees that got their Social Security Card in Colorado:

```
λ> printEmployees $ employeesFromColorado employees
521-01-8756: Mary Jones
522-43-9862: John Doe
524-34-1234: Bob Jones
```

That worked well for Colorado state, but some states have noncontiguous groups of area numbers, which means we need to join togethe results from a couple of ranges:

```
employeesFromNewMexico :: Employees -> Employees
employeesFromNewMexico es =
  withinPrefixRange 525 525 es `Map.union` withinPrefixRange 585 585 es
```

# Sets and Maps

Until previous example, we've looked at functions that deal only with a single `Map`, but most of the familiar functions from set theory are also made available to us, so we can easily operate on more than one `Map`/`Set`. In order to provide some meaningful examples let's define geographic regions of the USA and their SSN prefix ranges:

## Creation

```
data State =
  Arizona | California | Colorado | NewMexico | Nevada | Oklahoma | Texas | Utah | ...
  deriving (Show, Eq, Ord, Enum)

statePrefixRangeMap :: Map.Map State [(Int, Int)]
statePrefixRangeMap =
  Map.fromList
    [ (Arizona, [(526, 527)])
    , (California, [(545, 573)])
    , (Colorado, [(521, 524)])
    , (NewMexico, [(525, 525), (585, 585)])
    , (Nevada, [(530, 530), (680, 680)])
    , (Oklahoma, [(440, 448)])
    , (Texas, [(449, 467)])
    , (Utah, [(528, 529)])
      ...
    ]

allStates :: Set.Set State
allStates = Set.fromDistinctAscList [toEnum 0 ..]
```

For compactness, only states that are considered the South West are included, complete source code can be found as a gist.

Note, that because we are using `toEnum` and `enumFrom` - (`..`), we are guaranteed that all States will be unique and in a proper ascendin order, thus we are safe to use `Set.fromDistinctAscList` instead of a less efficient `Set.fromList` or even `Set.fromAscList`.

## Binary operations

Exclude employees from New Mexico:

```
λ> printEmployees (employees Map.\\ employeesFromNewMexico employees)
521-01-8756: Mary Jones
522-43-9862: John Doe
524-34-1234: Bob Jones
527-75-1035: Julia Bloom
```

Some common manipulations on `Set`s:

```
λ> let fourCorners = Set.fromList [Arizona, NewMexico, Colorado, Utah]
λ> let borderStates = Set.fromList [California, Arizona, NewMexico, Texas]
λ> Set.union fourCorners borderStates
fromList [Arizona,California,Colorado,NewMexico,Texas,Utah]
λ> Set.intersection fourCorners borderStates
fromList [Arizona,NewMexico]
λ> Set.difference fourCorners borderStates
fromList [Colorado,Utah]
λ> Set.difference borderStates fourCorners
fromList [California,Texas]
λ> let symmetricDifference a b = Set.union a b Set.\\ Set.intersection a b
λ> symmetricDifference fourCorners borderStates
fromList [California,Colorado,Texas,Utah]
```

## Conversion

Map a State to a set of SSN prefixes that correspond to it:

```
statePrefixMap :: Map.Map State (Set.Set Int)
statePrefixMap =
  Map.fromSet (Set.fromList . concatMap (uncurry enumFromTo) . (statePrefixRangeMap Map.!))
allStates
```

Inverse of what we have above: Map from prefix to `State`:

```
prefixStateMap :: Map.Map Int State
prefixStateMap = Map.foldlWithKey addPrefixes Map.empty statePrefixMap where
  addPrefixes spm state = Map.union spm . Map.fromSet (const state)
```

## Transformation

Using above Map we can list all employees we have per state:

```
-- | Transform `Map` to another `Map`
statePersonsMap :: Employees -> Map.Map State [Person]
statePersonsMap = Map.foldlWithKey updateState Map.empty
  where updateState sm ssn p =
          case Map.lookup (ssnPrefix ssn) prefixStateMap of
            Nothing    -> sm
            Just state -> Map.alter (consPerson p) state sm
        consPerson p Nothing = Just [p]
        consPerson p (Just ps) = Just (p:ps)
```

And create a `Set` of all Social Security Numbers of our employees per State:

```
-- Transform `Map` to `Set` to `Map`
stateSocialsMap :: Employees -> Map.Map State (Set.Set SSN)
stateSocialsMap = Set.foldl updateState Map.empty . Map.keysSet
  where updateState sm ssn =
          case Map.lookup (ssnPrefix ssn) prefixStateMap of
            Nothing    -> sm
            Just state -> Map.alter (addSSN ssn) state sm
        addSSN ssn Nothing = Just $ Set.singleton ssn
        addSSN ssn (Just ssnSet) = Just $ Set.insert ssn ssnSet
```

Here is a trial run on our `employees` data base:

```
λ> putStrLn $ showMap $ statePersonsMap employees
(Arizona,[Julia Bloom (Female)])
(Colorado,[Mary Jones (Female),John Doe (Male),Bob Jones (Male)])
(NewMexico,[Maria Gonzalez (Female),John Doe (Male),William Smith (Male)])
λ> putStrLn $ showMap $ stateSocialsMap employees
(Arizona,fromList [527-75-1035])
(Colorado,fromList [521-01-8756,522-43-9862,524-34-1234])
(NewMexico,fromList [525-15-5673,525-21-5423,585-11-1234])
```

**Alternative approach**

Although desired result is achieved, we can do better in terms of performance, since above implementation does not take advantage of tl ordering of Social Security Numbers that is made available to us by `Map` interface.

First, we want to generalize our `employeesFrom*` functions and then use it to partition our `Employees`:

```
employeesFrom :: State -> Employees -> Employees
employeesFrom state es = Map.unions $ map fromRange (statePrefixRangeMap Map.! state)
  where fromRange (low, high) = withinPrefixRange low high es

allStateEmployeesMap :: Employees -> Map.Map State Employees
allStateEmployeesMap es = Map.fromSet (`employeesFrom` es) allStates
```

Calling `allStateEmployeesMap` will produce undesired empty Maps of `Employees` for some States, but there is a reason behind it, it will help us demonstrate similarities between `Map.filter` and `Map.mapMaybe`:

```
statePersonsMap' :: Employees -> Map.Map State [Person]
statePersonsMap' = Map.map Map.elems . Map.filter (not . Map.null) . allStateEmployeesMap

stateSocialsMap' :: Employees -> Map.Map State (Set.Set SSN)
stateSocialsMap' = Map.mapMaybe nonEmptyElems . allStateEmployeesMap
  where nonEmptyElems sem | Map.null sem = Nothing
                          | otherwise = Just $ Map.keysSet sem
```

# Subset and Submap.

Here are two equivalent approaches of how we would check if we are missing a `State` from our `Map`:

```
λ> Set.isProperSubsetOf (Map.keysSet $ allStateEmployeesMap employees) allStates
False
λ> Set.isProperSubsetOf (Map.keysSet $ statePersonsMap employees) allStates
True
λ> Map.isProperSubmapOfBy (const . const True) (allStateEmployeesMap employees) statePrefixMap
False
λ> Map.isProperSubmapOfBy (const . const True) (statePersonsMap employees) statePrefixMap
True
```

# Conclusion

In this tutorial we looked at very useful `Map` and `Set` containers, but there are plenty of other packages that provide implementations of a sorts of data structures that might be a better match for your needs. In particular, if keys and elements for `Map` and `Set` respectively are hashable and their ordering is of no importance, it is recommended to use `HashMap` and `HashSet` from unordered-containers instead. That package provides very similar interface to one from containers library, but it has much smaller memory impact and better performance.

Here is a list of some packages and corresponding tutorials, that can help you choose and get started with an appropriate data structure for your problem:

- vector: Efficient Packed-Memory Data Representations - vector library
- bytestring and text: String Types

# Strict vs lazy values

Summary: unless you really know what you're doing, always import the `.Strict` modules.

Maps are always strict in their keys, meaning that forcing a map always forcing all of the keys. That's because it's necessary to analyze th keys themselves to put them into the hash table or binary tree appropriately. However, it's not necessary to be strict in the values. The default modules (`Data.Map` and `Data.IntMap`) are *lazy*. Use the `.Strict` modules unless you have strong reason to do otherwise.

# Mutability

Unlike other languages, Maps are immutable. This means you can safely pass them to other threads and functions, you never worry abou data races, etc. The functions that "mutate" actually return brand new map values. If you need mutation across threads, you need to use a mutable variable like `IORef` or `TVar` (which we haven't covered yet).

Downside of immutability: Maps do have a performance overhead. There's a mutable hashtables library, but it's not nearly as well used as containers and unordered-containers.

# Exercises

Calculate the frequency of each byte available on standard input. Example usage:

```
$ echo hello world | ./Main.hs
(10,1)
(32,1)
(100,1)
(101,1)
(104,1)
(108,3)
(111,2)
(114,1)
(119,1)
```

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import qualified Data.ByteString.Lazy as BL
import qualified Data.Map.Strict as Map

main :: IO ()
main = do
  lbs <- BL.getContents
  let add m w = Map.insertWith (+) w 1 m
  mapM_ print $ Map.toList $ BL.foldl' add Map.empty lbs
```

Implement a `MultiMap` based on `Map` and `Set` with the following signature and provides instances for `Show`, `Eq`, `Foldable`, `Semigrou` and `Monoid`.

```
newtype MultiMap k v
insert :: (Ord k, Ord v) => k -> v -> MultiMap k v -> MultiMap k v
delete :: (Ord k, Ord v) => k -> v -> MultiMap k v -> MultiMap k v
deleteAll :: Ord k => k -> MultiMap k v -> MultiMap k v
lookup :: Ord k => k -> MultiMap k v -> Set v
member :: (Ord k, Ord v) => k -> v -> MultiMap k v -> Bool
```

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE DeriveFoldable #-}
import qualified Data.Map.Strict as Map
import Data.Map.Strict (Map)
import qualified Data.Set as Set
import Data.Set (Set)
import Data.Semigroup
import Data.Maybe (fromMaybe)
import Test.Hspec
import Prelude hiding (lookup)

newtype MultiMap k v = MultiMap
  { toMap :: Map k (Set v)
  }
  deriving (Show, Eq, Foldable)

instance (Ord k, Ord v) => Semigroup (MultiMap k v) where
  MultiMap l <> MultiMap r = MultiMap $ Map.unionWith Set.union l r

instance (Ord k, Ord v) => Monoid (MultiMap k v) where
  mempty = MultiMap mempty
  mappend = (<>)

insert :: (Ord k, Ord v) => k -> v -> MultiMap k v -> MultiMap k v
insert k v (MultiMap m) =
  MultiMap $ Map.insertWith Set.union k (Set.singleton v) m

delete :: (Ord k, Ord v) => k -> v -> MultiMap k v -> MultiMap k v
delete k v (MultiMap m) =
  MultiMap $ Map.update fixSet k m
  where
    fixSet set0
      | Set.null set1 = Nothing
      | otherwise = Just set1
      where
        set1 = Set.delete v set0

deleteAll :: Ord k => k -> MultiMap k v -> MultiMap k v
deleteAll k (MultiMap m) = MultiMap $ Map.delete k m

lookup :: Ord k => k -> MultiMap k v -> Set v
lookup k (MultiMap m) = fromMaybe Set.empty $ Map.lookup k m

member :: (Ord k, Ord v) => k -> v -> MultiMap k v -> Bool
member k v (MultiMap m) = maybe False (v `Set.member`) $ Map.lookup k m

main :: IO ()
main = hspec $ do
  it "member on empty fails" $ member () () mempty `shouldBe` False
  it "insert/member" $ member () () (insert () () mempty) `shouldBe` True
  it "insert/lookup"
    $ lookup () (insert () () mempty) `shouldBe` Set.singleton ()
```

```
  it "deleteAll" $ deleteAll () (insert () () mempty) `shouldBe` mempty
  it "delete" $ delete () () (insert () () mempty) `shouldBe` mempty
  it "<>" $
    lookup () (insert () False mempty <> insert () True mempty) `shouldBe`
    Set.fromList [False, True]
```

# More exercises!

## Exercise 1

Implement the `printScore` function:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map

printScore :: Map String Int -> String -> IO ()
printScore = _

main :: IO ()
main =
    mapM_ (printScore scores) ["Alice", "Bob", "David"]
  where
    scores :: Map String Int
    scores = Map.fromList
      [ ("Alice", 95)
      , ("Bob", 90)
      , ("Charlies", 85)
      ]
```

## Exercise 2

We're going to write a program to figure out how much money people have after a number of transactions. Fill in the implementation of `addMoney`:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Data.HashMap.Strict (HashMap)
import qualified Data.HashMap.Strict as HashMap
import Control.Monad.State

addMoney :: (String, Int) -> State (HashMap String Int) ()
addMoney = _

main :: IO ()
main =
    print $ execState (mapM_ addMoney transactions) HashMap.empty
  where
    transactions :: [(String, Int)]
    transactions =
      [ ("Alice", 5)
      , ("Bob", 12)
      , ("Alice", 20)
      , ("Charles", 3)
      , ("Bob", -7)
      ]
```

The result should be:

```
fromList [("Bob",5),("Alice",25),("Charles",3)]
```

## Exercise 3

The final output from the previous program is kind of ugly. Instead, I want it to say:

```
Alice: 25
Bob: 5
Charles: 3
```

Modify the program above to get that result.

BONUS: If you really want to experience real-world Haskell code, go for better performance and use this module: https://www.stackage.or /haddock/lts-12.21/bytestring-0.10.8.1/Data-ByteString-Builder.html.

## Exercise 4

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Data.HashMap.Strict (HashMap)
import qualified Data.HashMap.Strict as HashMap
import Data.List (sort)

type Name = String
type StudentId = Int
type Score = Double

students :: HashMap Name StudentId
students = HashMap.fromList
  [ ("Alice", 1)
  , ("Bob", 2)
  , ("Charlie", 3)
  ]

scores :: HashMap Name Score
scores = HashMap.singleton "Bob" 90.4

noTestScore :: [Name]
noTestScore = _

main :: IO ()
main = do
  putStrLn "The following students have not taken the test"
  mapM_ putStrLn $ sort noTestScore
```

Implement `noTestScore` such that the output is:

```
The following students have not taken the test
Alice
Charlie
```

NOTE: hard-coding `["Alice", "Charlie"]` is cheating! :)

## Exercise 5

I want to drop the bottom 20% of test scores. Fill in the helper function.

**NOTE** I'm switching this exercise from lts-12.21 to lts-12.21. There's a new helper library function available that makes this much more straightforward to implement. Browse the docs at:

https://www.stackage.org/haddock/lts-12.21/containers-0.5.10.2/Data-Set.html

You'll also want to use the `div` function, which does integration division.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Data.Set (Set)
import qualified Data.Set as Set

scores :: Set Int
scores = Set.fromList [1..10]

dropBottom20Percent :: Ord a => Set a -> Set a
dropBottom20Percent = _

main :: IO ()
main = print $ dropBottom20Percent scores
```

CHALLENGE: Do it for a `HashSet` instead. Why is this different?

# Exercise solutions

## Exercise 1

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map

printScore :: Map String Int -> String -> IO ()
printScore scores name =
  case Map.lookup name scores of
    Just score -> putStrLn $ concat
      [ "Score for "
      , name
      , ": "
      , show score
      ]
    Nothing -> putStrLn $ "No score for " ++ name

main :: IO ()
main =
    mapM_ (printScore scores) ["Alice", "Bob", "David"]
  where
    scores :: Map String Int
    scores = Map.fromList
      [ ("Alice", 95)
      , ("Bob", 90)
      , ("Charlies", 85)
      ]
```

## Exercise 2

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Data.HashMap.Strict (HashMap)
import qualified Data.HashMap.Strict as HashMap
import Control.Monad.State

addMoney :: (String, Int) -> State (HashMap String Int) ()
addMoney (name, amt) = modify $ HashMap.insertWith (+) name amt

main :: IO ()
main =
    print $ execState (mapM_ addMoney transactions) HashMap.empty
  where
    transactions :: [(String, Int)]
    transactions =
      [ ("Alice", 5)
      , ("Bob", 12)
      , ("Alice", 20)
      , ("Charles", 3)
      , ("Bob", -7)
      ]
```

## Exercise 3

The first thing to note is that the easiest way to sort the people is to switch from a `HashMap` to a `Map`. A simple find-replace on the file w work for this.

There are a few different solutions here. Perhaps the most obvious looks like this:

```
import Data.Foldable (for_)

for_ (Map.toList $ execState (mapM_ addMoney transactions) Map.empty)
  $ \(name, total) -> putStrLn $ name ++ ": " ++ show total
```

However, this is arguably *bad*: we're doing more inside `IO` than really necessary. Instead, we should stick to pure code as much as possibl and instead build up a `String` value:

```
    putStr $ mapToString $ execState (mapM_ addMoney transactions) Map.empty
  where
    transactions :: [(String, Int)]
    transactions = ...

    pairToString :: (String, Int) -> String
    pairToString (name, total) = name ++ ": " ++ show total

    pairsToString :: [(String, Int)] -> String
    pairsToString = unlines . map pairToString

    mapToString :: Map String Int -> String
    mapToString = pairsToString . Map.toList
```

This reduces the surface area of code that can do dangerous things, and makes it easier to test our pure functions. One downside of this that string concatenation isn't particularly efficient. Using a bytestring builder approach is even better.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import           Control.Monad.State
import           Data.ByteString.Builder (Builder, hPutBuilder, intDec)
import           Data.Map.Strict         (Map)
import qualified Data.Map.Strict         as Map
import           Data.Monoid             ((<>))
import           Data.Text               (Text)
import           Data.Text.Encoding      (encodeUtf8Builder)
import           System.IO               (stdout)

addMoney :: (Text, Int) -> State (Map Text Int) ()
addMoney (name, amt) = modify $ Map.insertWith (+) name amt

main :: IO ()
main =
    hPutBuilder stdout $
    mapToBuilder $ execState (mapM_ addMoney transactions) Map.empty
  where
    transactions :: [(Text, Int)]
    transactions =
      [ ("Alice", 5)
      , ("Bob", 12)
      , ("Alice", 20)
      , ("Charles", 3)
      , ("Bob", -7)
      ]

    pairToBuilder :: (Text, Int) -> Builder
    pairToBuilder (name, total) =
      encodeUtf8Builder name <> ": " <> intDec total <> "\n"

    pairsToBuilder :: [(Text, Int)] -> Builder
    pairsToBuilder = foldMap pairToBuilder

    mapToBuilder :: Map Text Int -> Builder
    mapToBuilder = pairsToBuilder . Map.toList
```

This is more involved than the way most people would write this solution, but provides great performance. Note that it assumes your output will be UTF8 encoded.

## Exercise 4

```
noTestScore :: [Name]
noTestScore = HashMap.keys $ students `HashMap.difference` scores
```

Also: we can bypass the `sort` if we move over to a `Map` instead.

## Exercise 5

```
dropBottom20Percent :: Ord a => Set a -> Set a
dropBottom20Percent s = Set.drop (Set.size s `div` 5) s
```

## Contact Us

Corporate Office
10130 Perimeter
Parkway
Suite 200
Charlotte, NC 28216

+1 858-617-0430

sales@fpcomplete.com

## Services

- Custom Software
- Development
- DevSecOps
- Blockchain
- Rust
- Haskell
- Training
- All Services

## Products

- Kube360®
- Zehut
- Konsole360
- Idiom
- Kafka Library

## Resources

- Blog Posts
- Video Library
- Case Studies
- White Papers

## Our Company

- Our Journey
- Our Mission
- Our Leadership
- Our Engineers
- Our Clients
- Jobs