containers

Release 0.5.10.2

Mar 22, 2020

Contents

This site contains an introduction and overview of the main features of the `containers` package. For full API documentation see the containers Haddocks.

CHAPTER 1

Introduction

The `containers` package provides implementations of various immutable data structures.

Some of the data structures provided by this package have a very large API surface (for better or worse). The docs here focus on the most common functions which should be more than enough to get you started. Once you know the basics, or if you're looking for a specific function, you can head over to the containers Haddocks to check out the full API documentation!

1.1 Provided Data Structures

- *Sets*: ordered, non-duplicated elements
- *Maps*: ordered maps from keys to values (aka. dictionaries)
- *Sequences*: finite sequence of elements, an efficient alternative to list

Note: You'll need `containers >= 0.5.9` for a few of the examples. See *Version Requirements* for info on how to check which version you have and how to upgrade.

1.2 Related Packages

- unordered-containers - containers using hashing instead of ordering.
- array - mutable and immutable arrays.
- vector - efficient `Int`-indexed arrays (boxed and unboxed).
- bytestring - compact, immutable bytestrings, useful for binary and 8-bit character data.
- dlist - difference lists with *O(1)* append, useful for efficient logging and pretty printing.
- hashtables - mutable hash tables in the ST monad.

1.3 Looking for more resources?

If you've worked your way through the documentation here and you're looking for more examples or tutorials you should check out:

- haskell-lang.org's containers tutorial
- Learn You a Haskell "Modules" chapter

1.4 Installing and using the `containers` package

1.4.1 Version Requirements

For some of the examples you'll need `containers >= 0.5.9` which ships with `GHC >= 8.2`. You can check to see which version you have installed with:

```
ghc --version
> The Glorious Glasgow Haskell Compilation System, version 8.2.2
```

If you have an older version, don't worry about it, the majority of the code works with older versions of the package. If you want, you can get a recent version by from haskell.org, or with Stack using `stack --resolver lts-10.2 ghci`.

1.4.2 Importing modules

All of the modules in `containers` should be imported `qualified` since they use names that conflict with the standard Prelude.

```
import qualified Data.Set as Set
import qualified Data.Map.Strict as Map
import qualified Data.Sequence as Seq
```

1.4.3 In GHCi

Start the GHCi REPL with `ghci` or `stack ghci`. Once the REPL is loaded import the modules you want to use and you're good to go!

1.4.4 In a Cabal or Stack project

Add `containers` to the `build-depends:` stanza for your library, executable, or test-suite:

```
library
    build-depends:
        base >= 4.3 && < 5,
        containers >= 0.5.7 && < 0.6
```

and `import` any modules you need in your Haskell source files.

CHAPTER 2

Sets

Sets allow you to store *unique*, *ordered* elements, providing efficient insertion, lookups, deletions, and set operations. There are two implementations provided by the `containers` package: Data.Set and Data.IntSet. Use `IntSet` if you are storing, well... `Int` s.

```
data Set element = ...

data IntSet = ...
```

Important: `Set` relies on the *element* type having instances of the `Eq` and `Ord` typeclass for its internal representation. These are already defined for builtin types, and if you are using your own data type you can use the deriving mechanism.

All of these implementations are *immutable* which means that any update functions do not modify the set that you passed in, they creates a new set. In order to keep the changes you need to assign it to a new variable. For example:

```
let s1 = Set.fromList ["a", "b"]
let s2 = Set.delete "a" s1
print s1
> fromList ["a","b"]
print s2
> fromList ["b"]
```

2.1 Short Example

The following GHCi session shows some of the basic set functionality:

```
import qualified Data.Set as Set

let dataStructures = Set.fromList ["Set", "Map", "Graph", "Sequence"]
```

(continues on next page)

(continued from previous page)

```
-- Check if "Map" and "Trie" are in the set of data structures.
Set.member "Map" dataStructures
> True

Set.member "Trie" dataStructures
> False


-- Add "Trie" to our original set of data structures.
let moreDataStructures = Set.insert "Trie" dataStructures

Set.member "Trie" moreDataStructures
> True


-- Remove "Graph" from our original set of data structures.
let fewerDataStructures = Set.delete "Graph" dataStructures

Set.toAscList fewerDataStructures
> ["Map","Sequence","Set"]


-- Create a new set and combine it with our original set.
let unorderedDataStructures = Set.fromList ["HashSet", "HashMap"]

Set.union dataStructures unorderedDataStructures
> fromList ["Graph","HashMap","HashSet","Map","Sequence","Set"]
```

Tip: You can use the OverloadedLists extension so you don't need to write `fromList [1, 2, 3]` everywhere. Instead you can just write `[1, 2, 3]` and if the function is expecting a set it will be converted automatically! The code here will continue to use `fromList` for clarity though.

2.2 Importing Set and IntSet

When using `Set` or `IntSet` in a Haskell source file you should always use a `qualified` import because these modules export names that clash with the standard Prelude. You can import the type constructor and addional functions that you care about unqualified.

```
import Data.Set (Set, lookupMin, lookupMax)
import qualified Data.Set as Set

import Data.IntSet (IntSet)
import qualified Data.IntSet as IntSet
```

2.3 Common API Functions

Tip: All of these functions that work for `Set` will also work for `IntSet`, which has the element type `a` specialized to `Int`. Anywhere that you see `Set Int` you can replace it with `IntSet`. This will speed up most operations

tremendously (see *Performance*) with the exception of `size` which is O(1) for `Set` and O(n) for `IntSet`.

Note: `fromList [some,list,elements]` is how a `Set` is printed.

2.3.1 Construction and Conversion

Create an empty set

```
Set.empty :: Set a
Set.empty = ...
```

empty creates a set with zero elements.

```
Set.empty
> fromList []
```

Create a set with one element (singleton)

```
Set.singleton :: a -> Set a
Set.singleton x = ...
```

singleton creates a set with a single element `x` in it.

```
Set.singleton "containers"
> fromList ["containers"]

Set.singleton 1
> fromList [1]
```

Create a set from a list

```
Set.fromList :: Ord a => [a] -> Set a
Set.fromList xs = ...
```

fromList creates a set containing the elements of the list `xs`. Since sets don't contain duplicates, if there are repeated elements in the list they will only appear once.

```
Set.fromList ["base", "containers", "QuickCheck"]
> fromList ["QuickCheck","base","containers"]

Set.fromList [1, 1, 2, 3, 4, 4, 5, 1]
> fromList [1,2,3,4,5]
```

Create a list from a set

```
Set.toAscList, Set.toList, Set.elems :: Set a -> [a]
Set.toAscList s = ...
```

toAscList, toList, and elems return a list containing the elements of the set ‘s‘ in *ascending* order.

Note: These all do the same thing; use `toAscList` because its name indicates the ordering.

```
Set.toDescList :: Set a -> [a]
Set.toDescList s = ...
```

toDescList returns a list containing the elements of the set `s` in *descending* order.

```
Set.toAscList (Set.fromList [0, 2, 4, 6])
> [0,2,4,6]

Set.toDescList (Set.fromList [0, 2, 4, 6]
> [6,4,2,0]
```

2.3.2 Querying

Check if an element is in a set (member)

```
Set.member :: Ord a => a -> Set a -> Bool
Set.member x s = ...
```

member returns `True` if the element `x` is in the set `s`, `False` otherwise.

```
Set.member 0 Set.empty
> False

Set.member 0 (Set.fromList [0, 2, 4, 6])
> True
```

Check if a set is empty

```
Set.null :: Set a -> Bool
Set.null s = ...
```

null returns `True` if the set `s` is empty, `False` otherwise.

```
Set.null Set.empty
> True

Set.null (Set.fromList [0, 2, 4, 6])
> False
```

The number of elements in a set

```
Set.size :: Set a -> Int
Set.size s = ...
```

size returns the number of elements in the set `s`.

```
Set.size Set.empty
> 0

Set.size (Set.fromList [0, 2, 4, 6])
> 4
```

Find the minimum/maximum element in a set

Since version 0.5.9

```
lookupMin, lookupMax :: Set a -> Maybe a
lookupMin s = ...
lookupMax s = ...
```

lookupMin returns the minimum, or maximum respectively, element of the set `s`, or `Nothing` if the set is empty.

```
Set.lookupMin Set.empty
> Nothing

Set.lookupMin (Set.fromList [0, 2, 4, 6])
> Just 0

Set.lookupMax (Set.fromList [0, 2, 4, 6])
> Just 6
```

Warning: Unless you're using an old version of `containers` **DO NOT** use `Set.findMin` or `Set.findMax`. They are partial and throw a runtime error if the set is empty.

2.3.3 Modification

Adding a new element to a set

```
Set.insert :: Ord a => a -> Set a -> Set a
Set.insert x s = ...
```

insert places the element `x` into the set `s`, replacing an existing equal element if it already exists.

```
Set.insert 100 Set.empty
> fromList [100]

Set.insert 0 (Set.fromList [0, 2, 4, 6])
> fromList [0,2,4,6]
```

Removing an element from a set

```
Set.delete :: Ord a => a -> Set a -> Set a
Set.delete x s = ...
```

delete the element `x` from the set `s`. If it's not a member it leaves the set unchanged.

```
Set.delete 0 (Set.fromList [0, 2, 4, 6])
> fromList [2,4,6]
```

Filtering elements from a set

```
Set.filter :: (a -> Bool) -> Set a -> Set a
Set.filter predicate s = ...
```

filter produces a set consisting of all elements of `s` for which the *predicate*' returns `True`.

```
Set.filter (==0) (Set.fromList [0, 2, 4, 6])
> fromList [0]
```

2.3.4 Set Operations

Union

```
Set.union :: Ord a => Set a -> Set a -> Set a
Set.union l r = ...
```

union returns a set containing all elements that are in either of the two sets `l` or `r` (set union).

```
Set.union Set.empty (Set.fromList [0, 2, 4, 6])
> fromList [0,2,4,6]

Set.union (Set.fromList [1, 3, 5, 7]) (Set.fromList [0, 2, 4, 6])
> fromList [0,1,2,3,4,5,6,7]
```

Intersection

```
Set.intersection :: Ord a => Set a -> Set a -> Set a
Set.intersection l r = ...
```

intersection returns a set the elements that are in both sets `l` and `r` (set intersection).

```
Set.intersection Set.empty (Set.fromList [0, 2, 4, 6])
> fromList []

Set.intersection (Set.fromList [1, 3, 5, 7]) (Set.fromList [0, 2, 4, 6])
> fromList []

Set.intersection (Set.singleton 0) (Set.fromList [0, 2, 4, 6])
> fromList [0]
```

Difference

```
Set.difference :: Ord a => Set a -> Set a -> Set a
Set.difference l r = ...
```

difference returns a set containing the elements that are in the first set `l` but not the second set `r` (set difference/relative compliment).

```
Set.difference (Set.fromList [0, 2, 4, 6]) Set.empty
> fromList [0,2,4,6]

Set.difference (Set.fromList [0, 2, 4, 6]) (Set.fromList [1, 3, 5, 7])
> fromList [0,2,4,6]

Set.difference (Set.fromList [0, 2, 4, 6]) (Set.singleton 0)
> fromList [2,4,6]
```

Subset

```
Set.isSubsetOf :: Ord a => Set a -> Set a -> Bool
Set.isSubsetOf l r = ...
```

isSubsetOf returns `True` if all elements in the first set `l` are also in the second set `r` (subset).

Note: We use infix notation so that it reads nicer. These are back-ticks (‘), not single quotes (‘).

```
Set.empty `Set.isSubsetOf` Set.empty
> True

Set.empty `Set.isSubsetOf` (Set.fromList [0, 2, 4, 6])
> True

(Set.singleton 0) `Set.isSubsetOf` (Set.fromList [0, 2, 4, 6])
> True

(Set.singleton 1) `Set.isSubsetOf` (Set.fromList [0, 2, 4, 6])
> False
```

2.4 Serialization

The best way to serialize and deserialize sets is to use one of the many libraries which already support serializing sets. binary, cereal, and store are some common libraries that people use.

Tip: If you are writing custom serialization code use fromDistinctAscList (see #405 for more info).

2.5 Performance

The API docs are annotated with the Big-*O* complexities of each of the set operations. For benchmarks see the haskell-perf/sets page.

2.6 Looking for more?

Didn't find what you're looking for? This tutorial only covered the most common set functions, for a full list of functions see the Set and IntSet API documentation.

CHAPTER 3

Maps

Maps (sometimes referred to as dictionaries in other languages) allow you to store associations between *unique keys* and *values*. There are three implementations provided by the `containers` package: Data.Map.Strict, Data.Map.Lazy, and Data.IntMap. You almost never want the lazy version so use `Data.Map.Strict`, or if your keys are `Int` use `Data.IntMap`.

```
data Map k v = ...

data IntMap v = ...
```

Important: `Map` relies on the key type `k` having instances of the `Eq` and `Ord` typeclass for its internal representation. These are already defined for builtin types, and if you are using your own data type you can use the deriving mechanism.

All of these implementations are *immutable* which means that any update functions do not modify the map that you passed in, they creates a new map. In order to keep the changes you need to assign it to a new variable. For example:

```
let m1 = Map.fromList [("a", 1), ("b", 2)]
let m2 = Map.delete "a" m1
print m1
> fromList [("a",1),("b",2)]
print m2
> fromList [("b",2)]
```

3.1 Short Example

The following GHCi session shows some of the basic map functionality:

```
import qualified Data.Map.Strict as Map
```

(continues on next page)

(continued from previous page)

```
let nums = Map.fromList [(1,"one"), (2,"two"), (3,"three")]

-- Get the English word for the number 3 and 4.
Map.lookup 3 nums
> Just "three"

Map.lookup 4 nums
> Nothing


-- Add (4, "four") to our original map.
let moreNums = Map.insert 4 "four" nums

Map.member moreNums 4
> True


-- Remove the entry for 1 from our original map.
let fewerNums = Map.delete 1 nums

Map.toAscList fewerNums
> [(2,"two"),(3,"three")]


-- Create a new map and combine it with our original map.
-- fromList is right-biased: if a key is repeated the rightmost value is taken.
let newNums = Map.fromList [(3,"new three"), (4,"new four"), (4,"newer four")]

-- union is left-biased: if a key occurs more than once the value from the
-- left map is taken.
Map.union newNums nums
> fromList [(1,"one"),(2,"two"),(3,"new three"),(4,"newer four")]
```

Tip: You can use the OverloadedLists extension so you don't need to write `fromList [1, 2, 3]` everywhere; instead you can just write `[1, 2, 3]` and if the function is expecting a map it will be converted automatically! The code here will continue to use `fromList` for clarity though.

3.2 Importing Map and IntMap

When using `Map` or `IntMap` in a Haskell source file you should always use a `qualified` import because these modules export names that clash with the standard Prelude (you can import the type constructor on its own though!). You should also import `Prelude` and hide `lookup` because if you accidentally leave off the `Map.` qualifier you'll get confusing type errors. You can always import any specific identifiers you want unqualified. Most of the time, that will include the type constructor (`Map`).

```
import Prelude hiding (lookup)

import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map

import Data.IntMap (IntMap)
import qualified Data.IntMap.Strict as IntMap
```

3.3 Common API Functions

Tip: All of these functions that work for `Map` will also work for `IntMap`, which has the key type `k` specialized to `Int`. Anywhere that you see `Map Int v` you can replace it with `IntMap v`. This will speed up most operations tremendously (see *Performance*) with the exception of `size` which is O(1) for `Map` and O(n) for `IntMap`.

Note: A `Map` is printed as an association list preceeded by `fromList`. For example, it might look like `fromList [(Key1,True),(Key2,False)]`.

3.3.1 Construction and Conversion

Create an empty map

```
Map.empty :: Map k v
Map.empty = ...
```

empty creates a map without any entries.

```
Map.empty
> fromList []
```

Create a map with one entry (singleton)

```
Map.singleton :: k -> v -> Map k v
Map.singleton key value = ...
```

singleton creates a map with a single `(key,value)` entry in it.

```
Map.singleton 1 "one"
> fromList [(1,"one")]

Map.singleton "containers" ["base"]
> fromList [("containers",["base"])]
```

Create a map from a list

```
Map.fromList :: Ord k => [(k, v)] -> Map k v
Map.fromList xs = ...
```

fromList creates a map containing the entries of the list `xs` where the keys comes from the first entries of the pairs and the values from the second. If the same key appears more than once then the last value is taken.

```
Map.fromList []
> fromList []

Map.fromList [(1,"uno"), (1,"one"), (2,"two"), (3,"three")]
> fromList [(1,"one"),(2,"two"),(3,"three")]
```

There's another incredibly useful function for constructing a map from a list:

```
Map.fromListWith :: Ord k => (a -> a -> a) -> [(k, a)] -> Map.Map k a
Map.fromListWith f xs = ...
```

fromListWith allows you to build a map from a list `xs` with repeated keys, where `f` is used to "combine" (or "choose") values with the same key.

```
-- Build a map from a list, but only keep the largest value for each key.
Map.fromListWith max [("a", 2), ("a", 1), ("b", 2)]
> fromList [("a",2),("b",2)]

-- Build a histogram from a list of elements.
Map.fromListWith (+) (map (\x -> (x, 1)) ["a", "a", "b", "c", "c", "c"])
> fromList [("a",2),("b",1),("c",3)]

-- Build a map from a list, combining the string values for the same key.
Map.fromListWith (++) [(1, "a"), (1, "b"), (2, "x"), (2, "y")]
> fromList [(1,"ba"),(2,"yx")]
```

Create a list from a map

```
Map.toAscList, Map.toList, Map.assocs :: Map k v -> [(k, v)]
Map.toAscList m = ...
```

Note: These all do the same thing; use `toAscList` because its name indicates the ordering.

Note: `Map.toList` is **not** the same as `Foldable.toList`; the latter is equivalent to `elems`, although is rarely useful for maps. In general, use `toAscList`.

toAscList, toList, and assocs returns a list containing the (key, value) pairs in the map `m` in *ascending* key order.

```
Map.toDescList :: Map k v -> [(k, v)]
Map.toDescList m = ...
```

toDescList returns a list containing the (key, value) pairs in the map `m` in *descending* key order.

```
Map.toAscList (Map.fromList [(1,"one"), (2,"two"), (3,"three")])
> [(1,"one"),(2,"two"),(3,"three")]

Map.toDescList (Map.fromList [(1,"one"), (2,"two"), (3,"three")])
> [(3,"three"),(2,"two"),(1,"one")]
```

3.3.2 Querying

Lookup an entry in the map (lookup)

```
Map.lookup :: Ord k => k -> Map k v -> Maybe v
Map.lookup key m = ...
```

(continues on next page)

(continued from previous page)

```
Map.!? :: Ord k => Map k v -> k -> Maybe v
Map.!? m key = ...
```

lookup the value corresponding to the given `key`, returns `Nothing` if the key is not present; the `!?` operator (*since 0.5.10*) is a flipped version of `lookup` and can often be imported unqualified.

If you want to provide a default value if the key doesn't exist you can do:

```
import Data.Maybe (fromMaybe)

-- fromMaybe :: a -> Maybe a -> a
fromMaybe defaultValue (lookup k m)
```

For example:

```
import Data.Map.Strict ((!?))
import Data.Maybe (fromMaybe)

Map.lookup 1 Map.empty
> Nothing

Map.lookup 1 (Map.fromList [(1,"one"),(2,"two"),(3,"three")])
> Just "one"

> (Map.fromList [(1,"one"),(2,"two"),(3,"three")]) !? 1
> Just "one"

fromMaybe "?" (Map.empty !? 1)
> "?"

fromMaybe "?" (Map.fromList [(1,"one"), (2,"two"), (3,"three")] !? 1)
> "one"
```

Warning: **DO NOT** Use `Map.!`. It is partial and throws a runtime error if the key doesn't exist.

Check if a map is empty

```
Map.null :: Map k v -> Bool
Map.null m = ...
```

null returns `True` if the map `m` is empty and `False` otherwise.

```
Map.null Map.empty
> True

Map.null (Map.fromList [(1,"one")])
> False
```

The number of entries in a map

```
Map.size :: Map k v -> Int
Map.size m = ...
```

size returns the number of entries in the map m.

```
Map.size Map.empty
> 0

Map.size (Map.fromList [(1,"one"), (2,"two"), (3,"three")])
> 3
```

Find the minimum/maximum

Since version 0.5.9

```
Map.lookupMin, Map.lookupMax :: Map k v -> Maybe (k, v)
Map.lookupMin m = ...
Map.lookupMax m = ...
```

lookupMin and lookupMax respectively return the minimum or maximum element of the map m, or Nothing if the map is empty.

```
Map.lookupMin Map.empty
> Nothing

Map.lookupMin (Map.fromList [(1,"one"), (2,"two"), (3,"three")])
> Just (1,"one")

Map.lookupMax (Map.fromList [(1,"one"), (2,"two"), (3,"three")])
> Just (3,"three")
```

Warning: **DO NOT** use Map.findMin or Map.findMax. They are partial and throw a runtime error if the map is empty.

3.3.3 Modification

Adding a new entry to a map

```
Map.insert :: Ord k => k -> v -> Map k v -> Map k v
Map.insert key value m = ...
```

insert adds the value into the map m with the given key, replacing the existing value if the key already exists.

```
Map.insert 1 "one" Map.empty
> Map.fromList [(1,"one")]

Map.insert 4 "four" (Map.fromList [(1,"one"), (2,"two"), (3,"three")])
> fromList [(1,"one"),(2,"two"),(3,"three"),(4,"four")]

Map.insert 1 "uno" (Map.fromList [(1,"one"), (2,"two"), (3,"three")])
> fromList [(1,"uno"),(2,"two"),(3,"three")]
```

Removing an entry from a map

```
Map.delete :: Ord k => k -> Map k v -> Map k v
Map.delete key m = ...
```

delete removes the entry with the specified `key` from the map `m`. If the key doesn't exist it leaves the map unchanged.

```
Map.delete 1 Map.empty
> Map.empty

Map.delete 1 (Map.fromList [(1,"one"),(2,"two"),(3,"three")])
> fromList [(2,"two"),(3,"three")]
```

Filtering map entries

```
Map.filterWithKey :: (k -> v -> Bool) -> Map k v -> Map k v
Map.filterWithKey predicate m = ...
```

filterWithKey produces a map consisting of all entries of `m` for which the `predicate` returns `True`.

```
let f key value = key == 2 || value == "one"
Map.filterWithKey f (Map.fromList [(1,"one"), (2,"two"), (3,"three")])
> fromList [(1,"one"),(2,"two"]
```

Modifying a map entry

```
Map.adjust :: Ord k => (v -> v) -> k -> Map k v -> Map k v
Map.adjust f key m = ...
```

abjust applies the value transformation function `f` to the entry with given `key`. If no entry for that key exists then the map is left unchanged.

```
Map.alter :: Ord k => (Maybe v -> Maybe v) -> k -> Map k v -> Map k v
Map.alter f key m = ...
```

Apply the value transformation function `f` to the entry with given `key`, if no entry for that key exists then the function is passed `Nothing`. If the function returns `Nothing` then the entry is deleted, if the function returns `Just v2` then the value for the `key` is updated to `v2`. In other words, alter can be used to insert, update, or delete a value.

```
import Data.Maybe (isJust)
let addValueIfMissing mv = if isJust mv then mv else (Just 1)
Map.alter addValueIfMissing "key" (Map.fromList [("key", 0)])
> fromList [("key",0)]

let addValueIfMissing mv = if isJust mv then mv else (Just 1)
Map.alter addValueIfMissing "new_key" (Map.fromList [("key", 0)])
> fromList [("key",0),("new_key",1)]
```

The function `doubleIfPositivie` below will need to be placed in a Haskell source file.

```
doubleIfPositive :: Maybe Int -> Maybe Int
doubleIfPositive mv = case mv of
  -- Do nothing if the key doesn't exist.
```

(continues on next page)

(continued from previous page)

```
  Nothing -> Nothing

  -- If the key does exist, double the value if it is positive.
  Just v -> if v > 0 then (Just v*2) else (Just v)

-- In GHCi
Map.alter doubleIfPositive "a" (Map.fromList [("a", 1), ("b", -1)])
> Map.fromList [("a",2), ("b",-1)]

Map.alter doubleIfPositive "b" (Map.fromList [("a", 1), ("b", -1)])
> Map.fromList [("a", 1), ("b",-1)]
```

Modifying all map entries (mapping and traversing)

```
Map.map :: (a -> b) -> Map k a -> Map k v
Map.map f m = ...

Map.mapWithKey :: (k -> a -> b) -> Map.Map k a -> Map.Map k b
Map.mapWithKey g m = ...
```

map creates a new map by applying the transformation function `f` to each entries value. This is how Functor is defined for maps.

mapWithKey does the same as `map` but gives you access to the key in the transformation function `g`.

```
Map.map (*10) (Map.fromList [("haskell", 45), ("idris", 15)])
> fromList [("haskell",450),("idris",150)]

-- Use the Functor instance for Map.
(*10) <$> Map.fromList [("haskell", 45), ("idris", 15)]
> fromList [("haskell",450),("idris",150)]

let g key value = if key == "haskell" then (value * 1000) else value
Map.mapWithKey g (Map.fromList [("haskell", 45), ("idris", 15)])
> fromList [("haskell",45000),("idris",15)]
```

You can also apply a function which performs *actions* (such as printing) to each entry in the map.

```
Map.traverseWithKey :: Applicative t => (k -> a -> t b) -> Map.Map k a -> t (Map.Map␣
↪k b)
Map.traverseWithKey f m = ...
```

traverseWithKey maps each element of the map `m` to an *action* that produces a result of type `b`. The actions are performed and the values of the map are replaced with the results from the function. You can think of this as a `map` with affects.

```
-- | Ask the user how they want to schedule a bunch of tasks
-- that the boss has assigned certain priorities.
makeSchedule :: Map Task Priority -> IO (Map Task DateTime)
makeSchedule = traverseWithKey $ \task priority ->
  do
    putStrLn $ "The boss thinks " ++ show task ++
                 " has priority " ++ show priority ++
                 ". When do you want to do it?"
    readLn
```

3.3.4 Set-like Operations

Union

```
Map.unionWith :: Ord k => (v -> v -> v) -> Map k v -> Map k v -> Map k v
Map.unionWith f l r = ...
```

union returns a map containing all entries that are keyed in either of the two maps. If the same key appears in both maps, the value is determined by calling `f` passing in the left and right value (set union).

```
Map.unionWith (++) Map.empty (Map.fromList [(1,"x"),(2,"y")])
> fromList [(1,"x"),(2,"y")]

let f lv rv = lv
Map.unionWith f (Map.fromList [(1, "a")]) (Map.fromList [(1,"x"),(2,"y")])
> fromList [(1,"a"),(2,"y")]

Map.unionWith (++) (Map.fromList [(1, "a")]) (Map.fromList [(1,"x"),(2,"y")])
> fromList [(1,"ax"),(2,"y")]
```

Intersection

```
Map.intersectionWith :: Ord k => (v -> v -> v) -> Map k v -> Map k v -> Map k v
Map.intersectionWith f l r = ...
```

intersection returns a map containing all entries that have a key in both maps `l` and `r`. The value in the returned map is determined by calling `f` on the values from the left and right map (set intersection).

```
Map.intersectionWith (++) Map.empty (Map.fromList [(1,"x"), (2,"y")])
> fromList []

Map.intersectionWith (++) (Map.fromList [(1, "a")]) (Map.fromList [(1,"x"),(2,"y")])
> fromList [(1,"ax")]
```

Difference

```
Map.difference :: Ord k => Map k v -> Map k v -> Map k v
Map.difference l r = ...
```

difference returns a map containing all entries that have a key in the `l` map but not the `r` map (set difference/relative complement).

```
Map.difference (Map.fromList [(1,"one"), (2,"two"), (3,"three")]) Map.empty
> fromList [(1,"uno"),(2,"two"),(3,"three")]

Map.difference (Map.fromList[(1,"one"), (2,"two")]) (Map.fromList [(1,"uno")])
> fromList [(2,"two")]
```

3.4 Serialization

The best way to serialize and deserialize maps is to use one of the many libraries which already support serializing maps. binary, cereal, and store are some common libraries that people use.

Tip: If you are writing custom serialization code use fromDistinctAscList (see #405 for more info).

3.5 Performance

The API docs are annotated with the Big-*O* complexities of each of the map operations. For benchmarks see the haskell-perf/dictionaries page.

3.6 Looking for more?

Didn't find what you're looking for? This tutorial only covered the most common map functions, for a full list of functions see the Map and IntMap API documentation.

CHAPTER 4

Sequences

Sequences allow you to store a finite number of sequential elements, providing fast access to both ends of the sequence as well as efficient concatenation. The `containers` package provides the Data.Sequence module which defines the `Seq` data type.

4.1 Short Example

The following GHCi session shows some of the basic sequence funcitonality:

```
-- Import the Seq type and operators for combining sequences unqualified.
-- Import the rest of the Sequence module qualified.
import Data.Sequence (Seq(..), (<|), (|>), (><))
import qualified Data.Sequence as Seq

let nums = Seq.fromList [1, 2, 3]


-- Put numbers on the front and back.
0 <| nums
> fromList [0,1,2,3]

nums |> 4
> fromList [1,2,3,4]


-- Reverse a sequence
Seq.reverse (Seq.fromList [0, 1, 2])
> fromList [2,1,0]


-- Put two sequences together.
(Seq.fromList [-2, -1]) >< nums
> fromList [-2,-1,0,1,2]
```

(continues on next page)

(continued from previous page)

```
-- Check if a sequence is empty and check the length.
Seq.null nums
> False

Seq.length nums
> 3


-- Lookup an element at a certain index (since version 0.4.8).
Seq.lookup 2 nums
> Just 3

-- Or the unsafe version, you MUST check length beforehand.
Seq.index 2 nums
> 3


-- Map a function over a sequence (can use fmap or the infix function <$>).
fmap show nums
> fromList ["0","1","2"]

show <$> nums
> fromList ["0","1","2"]


-- Fold a sequence into a summary value.
foldr (+) 0 (Seq.fromList [0, 1, 2])
> 3
```

Tip: You can use the OverloadedLists extension so you don't need to write `fromList [1, 2, 3]` everywhere. Instead you can just write `[1, 2, 3]` and if the function is expecting a sequence it will be converted automatically! The code here will continue to use `fromList` for clarity.

4.2 Importing Sequence

When using `Sequence` in a Haskell source file you should always use a `qualified` import becasue it exports names that clash with the standard Prelude (you can import the type constructor and some operators on their own though!).

```
import Data.Sequence (Seq, (<|), (|>), (><))
import qualified Data.Sequence as Seq
```

4.3 Common API Functions

Note: `fromList [some,sequence,elements]` is how a `Seq` is printed.

4.3.1 Construction and Conversion

Create an empty sequence

```
Seq.empty :: Seq a
Seq.empty = ...
```

empty creates a sequence with zero elements.

```
Seq.empty
> fromList []
```

Create a sequence with one element (singleton)

```
Seq.singleton :: a -> Seq a
Seq.singleton x = ...
```

singleton creates a sequence with the single element `x` in it.

```
Seq.singleton "containers"
> fromList ["containers"]

Seq.singleton 1
> fromList [1]
```

Create a sequence with the same element repeated

```
Seq.replicate :: Int -> a -> Seq a
Seq.replicate n x = ...
```

replicate creates a sequence with same element `x` repeated `n` times.

```
Seq.replicate 0 "hi"
> fromList []

Seq.replicate 3 "hi"
> fromList ["hi","hi","hi"]
```

Create a sequence from a list

```
Seq.fromList :: [a] -> Seq a
Seq.FromList xs = ...
```

fromList creates a sequence containing the elements of the list `xs`. Sequences allow duplicate so all elements will be included in the order given.

```
Seq.fromList ["base", "containers", "QuickCheck"]
> fromList ["base","containers","QuickCheck"]

Seq.fromList [0, 1, 1, 2, 3, 1]
> fromList [0,1,1,2,3,1]
```

Adding to an existing sequence

```
(<|) :: a -> Seq a -> Seq a
x <| xs = ...

(|>) :: Seq a -> a -> Seq a
xs |> x = ...

(><) :: Seq a -> Seq a -> Seq a
l >< r = ...
```

- `x <| xs` places the element `x` at the beginning of the sequence `xs`..
- `xs |> x` places the element `x` at the end of the sequence `xs`.
- `l >< r` combines the two sequences `l` and `r` together.

Create a list from a sequence

```
import qualified Data.Foldable as Foldable
Foldable.toList :: Seq a -> [a]
```

There is no `toList` function in the Sequence module since it can be easily implemented with a fold using `Seq`'s Foldable instance.

```
import qualified Data.Foldable as Foldable
Foldable.toList (Seq.fromList ["base", "containers", "QuickCheck"])
> ["base","containers","QuickCheck"]
```

4.3.2 Pattern Matching

Since 0.5.10

Just like you can pattern match (aka. destructure) a list `[a]`, you can do the same with sequneces. Let's first look at how we do this with lists:

```
case [1, 2, 3] of
  [] -> "empty list"
  (x:xs) -> "first:" ++ show x ++ " rest:" ++ show xs
> "first:1 rest:[2,3]"
```

Let's do the same thing with sequences!

```
-- Imports the patterns to match on.
import Data.Sequence (Seq (Empty, (:<|), (:|>)))

case Seq.fromList [1, 2, 3] of
  Empty -> "empty sequence"
  x :<| xs -> "first:" ++ x ++ " rest:" ++ show xs
> "first:1 rest:fromList [2,3]"
```

Note: You can't copy/paste this into GHCi because it's multiple lines.

You can also take an element off the end:

```
-- Imports the patterns to match on.
import Data.Sequence (Seq (Empty, (:<|), (:|>)))

case Seq.fromList [1, 2, 3] of
  Empty -> "empty sequence"
  xs :|> x -> "last element:" ++ show x
> "last element:3"
```

4.3.3 Querying

Check if a sequence is empty

```
Seq.null :: Seq a -> Bool
Seq.null xs = ...
```

null returns `True` if the sequence `xs` is empty, and `False` otherwise.

```
Seq.null Seq.empty
> True

Seq.null (Seq.fromList [1, 2, 3])
> False
```

The length/size of a sequence

```
Seq.length :: Seq a -> Int
Seq.length xs = ...
```

length returns the length of the sequence `xs`.

```
Seq.length Seq.empty
> 0

Seq.length (Seq.fromList [1, 2, 3])
> 3
```

The element at a given index

```
Seq.lookup :: Int -> Seq a -> Maybe a
Seq.lookup n xs = ...

Seq.!? :: Seq a -> Int -> Maybe a
xs !? n = ...
```

lookup returns the element at the position `n`, or `Nothing` if the index is out of bounds. `!?` is simply a flipped version of `lookup`.

Note: You may need to import `!?` qualified if you're using a `Map`, `IntMap`, or `Vector` in the same file because they all export the same operator.

```
Seq.index :: Seq a -> Int -> a
Seq.index xs n = ...
```

index returns the element at the given position. It throws a runtime error if the index is out of bounds.

Tip: Use `lookup`/`!?` whenever you can and explicitly deal with the `Nothing` case.

```
(Seq.fromList ["base", "containers"]) Seq.!? 0
> Just "base"

Seq.index 0 (Seq.fromList ["base", "containers"])
> "base"

(Seq.fromList ["base", "containers"]) Seq.!? 2
> Nothing

Seq.index (Seq.fromList ["base", "containers"]) 2
> "*** Exception: index out of bounds
```

When working with functions that return a `Maybe v`, use a case expression to deal with the `Just` or `Nothing` value:

```
do
  let firstDependency = Seq.fromList ["base", "containers"] !? 0
  case firstDependency of
    Nothing -> print "Whoops! No dependencies!"
    Just dep -> print "The first dependency is " ++ dep
```

4.3.4 Modification

Inserting an element

```
Seq.insertAt :: Int -> a -> Seq a -> Seq a
Seq.insertAt i x xs = ...
```

insertAt inserts `x` into `xs` at the index `i`, shifting the rest of the sequence over. If `i` is out of range then `x` will be inserted at the beginning or the end of the sequence as appropriate.

```
Seq.insertAt 0 "idris" (Seq.fromList ["haskell", "rust"])
> fromList ["idris","haskell","rust"]

Seq.insertAt (-10) "idris" (Seq.fromList ["haskell", "rust"])
> fromList ["idris","haskell","rust"]

Seq.insertAt 10 "idris" (Seq.fromList ["haskell", "rust"])
> fromList ["haskell","rust","idris"]
```

See also *Adding to an existing sequence*.

Delete an element

```
Seq.deleteAt :: Int -> Seq a -> Seq a
Seq.deleteAt i xs = ...
```

deleteAt removes the element of the sequence at index `i`. If the index is out of bounds then the original sequence is returned.

```
Seq.deleteAt 0 (Seq.fromList [0, 1, 2])
> fromList [1,2]

Seq.deleteAt 10 (Seq.fromList [0, 1, 2])
> fromList [0,1,2]
```

Replace an element

```
Seq.update :: Int -> a -> Seq a -> Seq a
Seq.update i x xs = ...
```

update replaces the element at position `i` in the sequence with `x`. If the index is out of bounds then the original sequence is returned.

```
Seq.update 0 "hello" (Seq.fromList ["hi", "world", "!"])
> fromList ["hello","world","!"]

Seq.update 3 "OUTOFBOUNDS" (Seq.fromList ["hi", "world", "!"])
> fromList ["hi","world","!"]
```

Adjust/modify an element

Since version 0.5.8

```
adjust' :: forall a. (a -> a) -> Int -> Seq a -> Seq a
adjust' f i xs = ...
```

adjust' updates the element at position `i` in the sequence by applying the function `f` to the existing element. If the index is out of bounds then the original sequence is returned.

```
Seq.adjust' (*10) 0 (Seq.fromList [1, 2, 3])
> fromList [10,2,3]

Seq.adjust' (*10) 3 (Seq.fromList [1, 2, 3])
> fromList [1,2,3]
```

Note: If you're using an older version of containers which only has `adjust`, be careful because it can lead to poor performance and space leaks (see adjust docs).

Modifying all elements

```
fmap :: (a -> b) -> Seq a -> Seq b
fmap f xs = ...

Seq.mapWithIndex :: (Int -> a -> b) -> Seq a -> Seq b
Seq.mapWithIndex f xs = ...
```

fmap transform each element of the sequence with the function `f`. `fmap` is provided by the Functor instance for sequences and can also be written infix using the `<$>` operator.

mapWithIndex allows you to do a similar transformation but gives you the index that each element is at.

```
fmap (*10) (Seq.fromList [1, 2, 3])
-- = fromList [1*10, 2*10, 3*10]
> fromList [10,20,30]

(*10) <$> Seq.fromList [1, 2, 3]
-- = fromList [1*10, 2*10, 3*10]
> fromList [10,20,30]

let myMapFunc index val = index * val

Seq.mapWithIndex myMapFunc (Seq.fromList [1, 2, 3])
-- = fromList [0*1, 1*2, 2*3]
> fromList [0,2,6]
```

4.3.5 Sorting

```
Seq.sort :: Ord a => Seq a -> Seq a
Seq.sort xs = ...
```

sort the sequence `xs` using the `Ord` instance.

```
Seq.sort (Seq.fromList ["x", "a", "c", "b"])
> fromList ["a","b","c","x"]
```

4.3.6 Subsequences

Take

```
Seq.take :: Int -> Seq a -> Seq a
Seq.take n xs = ...
```

take returns the first `n` elements of the sequence `xs`. If the length of `xs` is less than `n` then all elements are returned.

```
Seq.take 0 (Seq.fromList [1, 2, 3])
> fromList []

Seq.take 2 (Seq.fromList [1, 2, 3])
> fromList [1,2]

Seq.take 5 (Seq.fromList [1, 2, 3])
> fromList [1,2,3]
```

Drop

```
Seq.drop :: Int -> Seq a -> Seq a
Seq.drop n xs = ...
```

drop the first `n` elements of the sequence `xs`. If the length of `xs` is less than `n` then an empty sequence is returned.

```
Seq.drop 0 (Seq.fromList [1, 2, 3])
> fromList [1,2,3]

Seq.drop 2 (Seq.fromList [1, 2, 3])
> fromList [3]

Seq.drop 5 (Seq.fromList [1, 2, 3])
> fromList []
```

Chunks

```
Seq.chunksOf :: Int -> Seq a -> Seq (Seq a)
Seq.chunksOf k xs = ...
```

chunksOf splits the sequence `xs` into chunks of size `k`. If the length of the sequence is not evenly divisible by `k` then the last chunk will have less than `k` elements.

> **Warning:** `k` can only be `0` when the sequence is empty, otherwise a runtime error is thrown.

```
-- A chunk size of 0 can ONLY be given for an empty sequence.
Seq.chunksOf 0 Seq.empty
> fromList []

Seq.chunksOf 1 (Seq.fromList [1, 2, 3])
> fromList [fromList [1],fromList [2],fromList [3]]

Seq.chunksOf 2 (Seq.fromList [1, 2, 3])
> fromList [fromList [1,2],fromList [3]]

Seq.chunksOf 5 (Seq.fromList [1, 2, 3])
> fromList [fromList [1,2,3]]
```

4.3.7 Folding

```
foldr :: (a -> b -> b) -> b -> Seq a -> b
foldr f init xs = ...

Seq.foldrWithIndex :: (Int -> a -> b -> b) -> b -> Seq a -> b
Seq.foldrWithIndex f init xs = ...
```

foldr collapses the sequence into a summary value by repeatedly applying `f`. `foldr` is provided by the Foldable instance for sequences. foldWithIndex gives you access to the position in the sequence when transforming each element.

```
foldr (+) 0 (Seq.fromList [1, 2, 3])
-- = (1 + (2 + (3 + 0)))
> 6

let myFoldFunction index val accum = (index * val) + accum

Seq.foldrWithIndex myFoldFunction 0 (Seq.fromList [1, 2, 3])
-- = ((0*1) + ((1*2) + ((2*3) + 0)))
> 8
```

4.4 Serialization

The best way to serialize and deserialize sequences is to use one of the many libraries which already support serializing sequences. binary, cereal, and store are some common libraries that people use.

4.5 Performance

The API docs are annotated with the Big-*O* complexities of each of the sequence operations. For benchmarks see the haskell-perf/sequences page.

4.6 Looking for more?

Didn't find what you're looking for? This tutorial only covered the most common sequence functions, for a full list of functions see the Data.Sequence API documentation.