

containers-0.8: Assorted concrete container types

[Instances](#) · [Quick Jump](#) · [Source](#) · [Contents](#) · [Index](#)

Data.Map.Strict

Contents

[Map type](#)

[Construction](#)

[From Unordered Lists](#)

[From Ascending Lists](#)

[From Descending Lists](#)

[Insertion](#)

[Deletion/Update](#)

[Query](#)

[Lookup](#)

[Size](#)

[Combine](#)

[Union](#)

[Difference](#)

[Intersection](#)

[Symmetric difference](#)

[Disjoint](#)

[Compose](#)

[General combining functions](#)

[Deprecated general combining function](#)

| | |
|---------------------|---|
| Copyright | (c) Daan Leijen 2002 (c) Andriy Palamarchuk 2008 |
| License | BSD-style |
| Maintainer | libraries@haskell.org |
| Portability | portable |
| Safe Haskell | Safe |
| Language | Haskell2010 |

Traversal

Map

Folds

Strict folds

Conversion

Lists

Ordered lists

Filter

Submap

Indexed

Min/Max

Debugging

Finite Maps (strict interface)

The `Map k v` type represents a finite map (sometimes called a dictionary) from keys of type `k` to values of type `v`.

Each function in this module is careful to force values before installing them in a `Map`. This is usually more efficient when laziness is not necessary. When laziness *is* required, use the functions in [Data.Map.Lazy](#).

In particular, the functions in this module obey the following law:

- If all values stored in all maps in the arguments are in WHNF, then all values stored in all maps in the results will be in WHNF once those maps are evaluated.

When deciding if this is the correct data structure to use, consider:

- If you are using `Int` keys, you will get much better performance for most operations using [Data.IntMap.Strict](#).
- If you don't care about ordering, consider use `Data.HashMap.Strict` from the [unordered-containers](#) package instead.

For a walkthrough of the most commonly used functions see the [maps introduction](#).

This module is intended to be imported qualified, to avoid name clashes with Prelude functions, e.g.

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

Note that the implementation is generally *left-biased*. Functions that take two maps as arguments and combine them, such as `union` and `intersection`, prefer the values in the first argument to those in the second.

Warning

The size of a `Map` must not exceed `maxBound :: Int`. Violation of this condition is not detected and if the size limit is exceeded, its behaviour is undefined.

The `Map` type is shared between the lazy and strict modules, meaning that the same `Map` value can be passed to functions in both modules. This means that the `Functor`, `Traversable` and `Data` instances are the same as for the `Data.Map.Lazy` module, so if they are used the resulting maps may contain suspended values (thunks).

Implementation

The implementation of `Map` is based on *size balanced* binary trees (or trees of *bounded balance*) as described by:

- Stephen Adams, "*Efficient sets—a balancing act*", Journal of Functional Programming 3(4):553-562, October 1993, <https://doi.org/10.1017/S0956796800000885>, <https://groups.csail.mit.edu/mac/users/adams/BB/index.html>.
- J. Nievergelt and E.M. Reingold, "*Binary search trees of bounded balance*", SIAM journal of computing 2(1), March 1973. <https://doi.org/10.1137/0202005>.
- Yoichi Hirai and Kazuhiko Yamamoto, "*Balancing weight-balanced trees*", Journal of Functional Programming 21(3):287-307, 2011, <https://doi.org/10.1017/S0956796811000104>

Bounds for `union`, `intersection`, and `difference` are as given by

- Guy Blelloch, Daniel Ferizovic, and Yihan Sun, "*Parallel Ordered Sets Using Join*", <https://arxiv.org/abs/1602.02120v4>.

Performance information

The time complexity is given for each operation in *big-O notation*, with n referring to the number of entries in the map.

Operations like `lookup`, `insert`, and `delete` take $O(\log n)$ time.

Binary set operations like `union` and `intersection` take $O(m \log(\frac{n}{m} + 1))$ time, where m and n are the sizes of the smaller and larger input maps respectively.

Benchmarks comparing `Data.Map.Strict` with other dictionary implementations can be found at <https://github.com/haskell-perf/dictionaries>.

Map type

data `Map` k a

Source

A Map from keys k to values a.

Instances

- ▷ `Bifoldable` Map # Source *Since: 0.6.3.1*
- ▷ `Eq2` Map # Source *Since: 0.5.9*
- ▷ `Ord2` Map # Source *Since: 0.5.9*
- ▷ `Show2` Map # Source *Since: 0.5.9*
- ▷ `NFData2` Map # Source *Since: 0.8*
- ▷ `(Lift k, Lift a) => Lift (Map k a :: Type)` # Source *Since: 0.6.6*
- ▷ `Foldable` (Map k) # Source Folds in order of increasing key.
- ▷ `Eq k => Eq1 (Map k)` # Source *Since: 0.5.9*
- ▷ `Ord k => Ord1 (Map k)` # Source *Since: 0.5.9*
- ▷ `(Ord k, Read k) => Read1 (Map k)` # Source *Since: 0.5.9*

- ▷ `Show k => Show1 (Map k)` # Source *Since: 0.5.9*
- ▷ `Traversable (Map k)` # Source Traverses in order of increasing key.
- ▷ `Functor (Map k)` # Source
- ▷ `NFData k => NFData1 (Map k)` # Source *Since: 0.8*
- ▷ `(Data k, Data a, Ord k) => Data (Map k a)` # Source
- ▷ `Ord k => Monoid (Map k v)` # Source `mempty = empty`
- ▷ `Ord k => Semigroup (Map k v)` # Source `(<>) = union`
Since: 0.5.7
- ▷ `Ord k => IsList (Map k v)` # Source *Since: 0.5.6.2*
- ▷ `(Ord k, Read k, Read e) => Read (Map k e)` # Source
- ▷ `(Show k, Show a) => Show (Map k a)` # Source
- ▷ `(NFData k, NFData a) => NFData (Map k a)` # Source
- ▷ `(Eq k, Eq a) => Eq (Map k a)` # Source
- ▷ `(Ord k, Ord v) => Ord (Map k v)` # Source
- ▷ `type Item (Map k v)` # Source

Construction

`empty :: Map k a`

Source

$O(1)$. The empty map.

```
empty      == fromList []
size empty == 0
```

singleton :: k -> a -> Map k a

[# Source](#)

$O(1)$. A map with a single element.

```
singleton 1 'a'      == fromList [(1, 'a')]
size (singleton 1 'a') == 1
```

fromSet :: (k -> a) -> Set k -> Map k a

[# Source](#)

$O(n)$. Build a map from a set of keys and a function which for each key computes its value.

```
fromSet (\k -> replicate k 'a') (Data.Set.fromList [3, 5]) == fromList [(5,"aaaaa"), (3,"aaa")]
fromSet undefined Data.Set.empty == empty
```

fromArgSet :: Set (Arg k a) -> Map k a

[# Source](#)

$O(n)$. Build a map from a set of elements contained inside [Args](#).

```
fromArgSet (Data.Set.fromList [Arg 3 "aaa", Arg 5 "aaaaa"]) == fromList [(5,"aaaaa"), (3,"aaa")]
fromArgSet Data.Set.empty == empty
```

From Unordered Lists

fromList :: Ord k => [(k, a)] -> Map k a

[# Source](#)

$O(n \log n)$. Build a map from a list of key/value pairs. See also [fromAscList](#). If the list contains more than one value for the same key, the last value for the key is retained.

If the keys are in non-decreasing order, this function takes $O(n)$ time.

```
fromList [] == empty
fromList [(5,"a"), (3,"b"), (5, "c")] == fromList [(5,"c"), (3,"b")]
fromList [(5,"c"), (3,"b"), (5, "a")] == fromList [(5,"a"), (3,"b")]
```

fromListWith :: Ord k => (a -> a -> a) -> [(k, a)] -> Map k a

[# Source](#)

$O(n \log n)$. Build a map from a list of key/value pairs with a combining function. See also [fromAscListWith](#).

If the keys are in non-decreasing order, this function takes $O(n)$ time.

```
fromListWith (++) [(5,"a"), (5,"b"), (3,"x"), (5,"c")] == fromList [(3, "x"), (5, "cba")]
fromListWith (++) [] == empty
```

Note the reverse ordering of "cba" in the example.

The symmetric combining function `f` is applied in a left-fold over the list, as `f new old`.

Performance

You should ensure that the given `f` is fast with this order of arguments.

Symmetric functions may be slow in one order, and fast in another. For the common case of collecting values of matching keys in a list, as above:

The complexity of `(++) a b` is $O(a)$, so it is fast when given a short list as its first argument. Thus:

```
fromListWith      (++) (replicate 1000000 (3, "x"))  -- 0(n), fast
fromListWith (flip (++) (replicate 1000000 (3, "x"))) -- 0(n2), extremely slow
```

because they evaluate as, respectively:

```
fromList [(3, "x" ++ ("x" ++ "xxxxx..xxxxx"))] -- 0(n)
fromList [(3, ("xxxxx..xxxxx" ++ "x") ++ "x")] -- 0(n2)
```

Thus, to get good performance with an operation like `(++)` while also preserving the same order as in the input list, reverse the input:

```
fromListWith (++) (reverse [(5,"a"), (5,"b"), (5,"c")]) == fromList [(5, "abc")]
```

and it is always fast to combine singleton-list values `[v]` with `fromListWith (++)`, as in:

```
fromListWith (++) $ reverse $ map (\(k, v) -> (k, [v])) someListOfTuples
```

fromListWithKey :: Ord k => (k -> a -> a -> a) -> [(k, a)] -> Map k a

[# Source](#)

$O(n \log n)$. Build a map from a list of key/value pairs with a combining function. See also [fromAscListWithKey](#).

If the keys are in non-decreasing order, this function takes $O(n)$ time.

```
let f key new_value old_value = show key ++ ":" ++ new_value ++ "|" ++ old_value
fromListWithKey f [(5,"a"), (5,"b"), (3,"b"), (3,"a"), (5,"c")] == fromList [(3, "3:a|b"), (5, "5:c|5:b|a")]
fromListWithKey f [] == empty
```

Also see the performance note on [fromListWith](#).

From Ascending Lists

fromAscList :: Eq k => [(k, a)] -> Map k a

[# Source](#)

$O(n)$. Build a map from an ascending list in linear time.

Warning: This function should be used only if the keys are in non-decreasing order. This precondition is not checked. Use `fromList` if the precondition may not hold.

```
fromAscList [(3,"b"), (5,"a")]           == fromList [(3, "b"), (5, "a")]
fromAscList [(3,"b"), (5,"a"), (5,"b")] == fromList [(3, "b"), (5, "b")]
valid (fromAscList [(3,"b"), (5,"a"), (5,"b")]) == True
valid (fromAscList [(5,"a"), (3,"b"), (5,"b")]) == False
```

fromAscListWith :: `Eq k => (a -> a -> a) -> [(k, a)] -> Map k a`

[# Source](#)

$O(n)$. Build a map from an ascending list in linear time with a combining function for equal keys.

Warning: This function should be used only if the keys are in non-decreasing order. This precondition is not checked. Use `fromListWith` if the precondition may not hold.

```
fromAscListWith (++) [(3,"b"), (5,"a"), (5,"b")] == fromList [(3, "b"), (5, "ba")]
valid (fromAscListWith (++) [(3,"b"), (5,"a"), (5,"b")]) == True
valid (fromAscListWith (++) [(5,"a"), (3,"b"), (5,"b")]) == False
```

Also see the performance note on `fromListWith`.

fromAscListWithKey :: `Eq k => (k -> a -> a -> a) -> [(k, a)] -> Map k a`

[# Source](#)

$O(n)$. Build a map from an ascending list in linear time with a combining function for equal keys.

Warning: This function should be used only if the keys are in non-decreasing order. This precondition is not checked. Use `fromListWithKey` if the precondition may not hold.

```
let f k a1 a2 = (show k) ++ ":" ++ a1 ++ a2
fromAscListWithKey f [(3,"b"), (5,"a"), (5,"b"), (5,"b")] == fromList [(3, "b"), (5, "5:b5:ba")]
valid (fromAscListWithKey f [(3,"b"), (5,"a"), (5,"b"), (5,"b")]) == True
valid (fromAscListWithKey f [(5,"a"), (3,"b"), (5,"b"), (5,"b")]) == False
```

Also see the performance note on [fromListWith](#).

fromDistinctAscList :: [(k, a)] -> Map k a

[# Source](#)

$O(n)$. Build a map from an ascending list of distinct elements in linear time.

Warning: This function should be used only if the keys are in strictly increasing order. This precondition is not checked. Use [fromList](#) if the precondition may not hold.

```
fromDistinctAscList [(3,"b"), (5,"a")] == fromList [(3, "b"), (5, "a")]
valid (fromDistinctAscList [(3,"b"), (5,"a")]) == True
valid (fromDistinctAscList [(3,"b"), (5,"a"), (5,"b")]) == False
```

From Descending Lists

fromDescList :: Eq k => [(k, a)] -> Map k a

[# Source](#)

$O(n)$. Build a map from a descending list in linear time.

Warning: This function should be used only if the keys are in non-increasing order. This precondition is not checked. Use [fromList](#) if the precondition may not hold.

```
fromDescList [(5,"a"), (3,"b")] == fromList [(3, "b"), (5, "a")]
fromDescList [(5,"a"), (5,"b"), (3,"a")] == fromList [(3, "b"), (5, "b")]
valid (fromDescList [(5,"a"), (5,"b"), (3,"b")]) == True
valid (fromDescList [(5,"a"), (3,"b"), (5,"b")]) == False
```

fromDescListWith :: Eq k => (a -> a -> a) -> [(k, a)] -> Map k a

[# Source](#)

$O(n)$. Build a map from a descending list in linear time with a combining function for equal keys.

Warning: This function should be used only if the keys are in non-increasing order. This precondition is not checked. Use [fromListWith](#) if the precondition may not hold.

```
fromDescListWith (++) [(5,"a"), (5,"b"), (3,"b")] == fromList [(3, "b"), (5, "ba")]
valid (fromDescListWith (++) [(5,"a"), (5,"b"), (3,"b")]) == True
valid (fromDescListWith (++) [(5,"a"), (3,"b"), (5,"b")]) == False
```

Also see the performance note on [fromListWith](#).

fromDescListWithKey :: [Eq](#) k => (k -> a -> a -> a) -> [(k, a)] -> Map k a

[# Source](#)

$O(n)$. Build a map from a descending list in linear time with a combining function for equal keys.

Warning: This function should be used only if the keys are in non-increasing order. This precondition is not checked. Use [fromListWithKey](#) if the precondition may not hold.

```
let f k a1 a2 = (show k) ++ ":" ++ a1 ++ a2
fromDescListWithKey f [(5,"a"), (5,"b"), (5,"b"), (3,"b")] == fromList [(3, "b"), (5, "5:b5:ba")]
valid (fromDescListWithKey f [(5,"a"), (5,"b"), (5,"b"), (3,"b")]) == True
valid (fromDescListWithKey f [(5,"a"), (3,"b"), (5,"b"), (5,"b")]) == False
```

Also see the performance note on [fromListWith](#).

fromDistinctDescList :: [(k, a)] -> Map k a

[# Source](#)

$O(n)$. Build a map from a descending list of distinct elements in linear time.

Warning: This function should be used only if the keys are in strictly decreasing order. This precondition is not checked. Use [fromList](#) if the precondition may not hold.

```
fromDistinctDescList [(5,"a"), (3,"b")] == fromList [(3, "b"), (5, "a")]
valid (fromDistinctDescList [(5,"a"), (3,"b")]) == True
valid (fromDistinctDescList [(5,"a"), (3,"b"), (3,"a")]) == False
```

Insertion

insert :: Ord k => k -> a -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Insert a new key and value in the map. If the key is already present in the map, the associated value is replaced with the supplied value. `insert` is equivalent to `insertWith const`.

```
insert 5 'x' (fromList [(5,'a'), (3,'b')]) == fromList [(3, 'b'), (5, 'x')]
insert 7 'x' (fromList [(5,'a'), (3,'b')]) == fromList [(3, 'b'), (5, 'a'), (7, 'x')]
insert 5 'x' empty                          == singleton 5 'x'
```

insertWith :: Ord k => (a -> a -> a) -> k -> a -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Insert with a function, combining new value and old value. `insertWith f key value mp` will insert the pair (key, value) into mp if key does not exist in the map. If the key does exist, the function will insert the pair (key, f new_value old_value).

```
insertWith (++) 5 "xxx" (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "xxxa")]
insertWith (++) 7 "xxx" (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a"), (7, "xxx")]
insertWith (++) 5 "xxx" empty                          == singleton 5 "xxx"
```

Also see the performance note on `fromListWith`.

insertWithKey :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Insert with a function, combining key, new value and old value. `insertWithKey f key value mp` will insert the pair (key, value) into mp if key does not exist in the map. If the key does exist, the function will insert the pair (key, f key new_value old_value). Note that the key passed to f is the same key passed to `insertWithKey`.

```
let f key new_value old_value = (show key) ++ ":" ++ new_value ++ "|" ++ old_value
insertWithKey f 5 "xxx" (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "5:xxx|a")]
insertWithKey f 7 "xxx" (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a"), (7, "xxx")]
insertWithKey f 5 "xxx" empty                          == singleton 5 "xxx"
```

Also see the performance note on [fromListWith](#).

insertLookupWithKey :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> (Maybe a, Map k a)

[# Source](#)

$O(\log n)$. Combines insert operation with old value retrieval. The expression (`insertLookupWithKey f k x map`) is a pair where the first element is equal to (`lookup k map`) and the second element equal to (`insertWithKey f k x map`).

```
let f key new_value old_value = (show key) ++ ":" ++ new_value ++ "|" ++ old_value
insertLookupWithKey f 5 "xxx" (fromList [(5,"a"), (3,"b")]) == (Just "a", fromList [(3, "b"), (5, "5:xxx|a")])
insertLookupWithKey f 7 "xxx" (fromList [(5,"a"), (3,"b")]) == (Nothing, fromList [(3, "b"), (5, "a"), (7, "xxx")])
insertLookupWithKey f 5 "xxx" empty == (Nothing, singleton 5 "xxx")
```

This is how to define `insertLookup` using `insertLookupWithKey`:

```
let insertLookup kx x t = insertLookupWithKey (\_ a _ -> a) kx x t
insertLookup 5 "x" (fromList [(5,"a"), (3,"b")]) == (Just "a", fromList [(3, "b"), (5, "x")])
insertLookup 7 "x" (fromList [(5,"a"), (3,"b")]) == (Nothing, fromList [(3, "b"), (5, "a"), (7, "x")])
```

Also see the performance note on [fromListWith](#).

Deletion/Update

delete :: Ord k => k -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Delete a key and its value from the map. When the key is not a member of the map, the original map is returned.

```
delete 5 (fromList [(5,"a"), (3,"b")]) == singleton 3 "b"
delete 7 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a")]
delete 5 empty == empty
```

adjust :: Ord k => (a -> a) -> k -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Update a value at a specific key with the result of the provided function. When the key is not a member of the map, the original map is returned.

```
adjust ("new " ++) 5 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "new a")]
adjust ("new " ++) 7 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a")]
adjust ("new " ++) 7 empty                        == empty
```

adjustWithKey :: Ord k => (k -> a -> a) -> k -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Adjust a value at a specific key. When the key is not a member of the map, the original map is returned.

```
let f key x = (show key) ++ ":new " ++ x
adjustWithKey f 5 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "5:new a")]
adjustWithKey f 7 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a")]
adjustWithKey f 7 empty                        == empty
```

update :: Ord k => (a -> Maybe a) -> k -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. The expression (`update f k map`) updates the value `x` at `k` (if it is in the map). If (`f x`) is `Nothing`, the element is deleted. If it is (`Just y`), the key `k` is bound to the new value `y`.

```
let f x = if x == "a" then Just "new a" else Nothing
update f 5 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "new a")]
update f 7 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a")]
update f 3 (fromList [(5,"a"), (3,"b")]) == singleton 5 "a"
```

updateWithKey :: Ord k => (k -> a -> Maybe a) -> k -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. The expression (`updateWithKey f k map`) updates the value `x` at `k` (if it is in the map). If (`f k x`) is `Nothing`, the element is deleted. If it is (`Just y`),

the key `k` is bound to the new value `y`.

```
let f k x = if x == "a" then Just ((show k) ++ ":new a") else Nothing
updateWithKey f 5 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "5:new a")]
updateWithKey f 7 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a")]
updateWithKey f 3 (fromList [(5,"a"), (3,"b")]) == singleton 5 "a"
```

updateLookupWithKey :: Ord k => (k -> a -> Maybe a) -> k -> Map k a -> (Maybe a, Map k a)

[# Source](#)

$O(\log n)$. Look up and update. See also `updateWithKey`. This function returns the changed value, if it is updated. Returns the original key value if the map entry is deleted.

```
let f k x = if x == "a" then Just ((show k) ++ ":new a") else Nothing
updateLookupWithKey f 5 (fromList [(5,"a"), (3,"b")]) == (Just "5:new a", fromList [(3, "b"), (5, "5:new a")])
updateLookupWithKey f 7 (fromList [(5,"a"), (3,"b")]) == (Nothing, fromList [(3, "b"), (5, "a")])
updateLookupWithKey f 3 (fromList [(5,"a"), (3,"b")]) == (Just "b", singleton 5 "a")
```

alter :: Ord k => (Maybe a -> Maybe a) -> k -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. The expression `(alter f k map)` alters the value `x` at `k`, or absence thereof. `alter` can be used to insert, delete, or update a value in a `Map`. In short : `lookup k (alter f k m) = f (lookup k m)`.

```
let f _ = Nothing
alter f 7 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a")]
alter f 5 (fromList [(5,"a"), (3,"b")]) == singleton 3 "b"

let f _ = Just "c"
alter f 7 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a"), (7, "c")]
alter f 5 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "c")]
```

Note that `adjust = alter . fmap`.

alterF :: (Functor f, Ord k) => (Maybe a -> f (Maybe a)) -> k -> Map k a -> f (Map k a)

[# Source](#)

$O(\log n)$. The expression `alterF f k map` alters the value `x` at `k`, or absence thereof. `alterF` can be used to inspect, insert, delete, or update a value in a `Map`. In short: `lookup k <$> alterF f k m = f (lookup k m)`.

Example:

```
interactiveAlter :: Int -> Map Int String -> IO (Map Int String)
interactiveAlter k m = alterF f k m where
  f Nothing = do
    putStrLn $ show k ++
      " was not found in the map. Would you like to add it?"
    getUserResponse1 :: IO (Maybe String)
  f (Just old) = do
    putStrLn $ "The key is currently bound to " ++ show old ++
      ". Would you like to change or delete it?"
    getUserResponse2 :: IO (Maybe String)
```

`alterF` is the most general operation for working with an individual key that may or may not be in a given map. When used with trivial functors like `Identity` and `Const`, it is often slightly slower than more specialized combinators like `lookup` and `insert`. However, when the functor is non-trivial and key comparison is not particularly cheap, it is the fastest way.

Note on rewrite rules:

This module includes GHC rewrite rules to optimize `alterF` for the `Const` and `Identity` functors. In general, these rules improve performance. The sole exception is that when using `Identity`, deleting a key that is already absent takes longer than it would without the rules. If you expect this to occur a very large fraction of the time, you might consider using a private copy of the `Identity` type.

Note: `alterF` is a flipped version of the `at` combinator from `Control.Lens.At`.

Since: 0.5.8

Query

Lookup

lookup :: Ord k => k -> Map k a -> Maybe a

[# Source](#)

$O(\log n)$. Look up the value at a key in the map.

The function will return the corresponding value as (**Just** value), or **Nothing** if the key isn't in the map.

An example of using lookup:

```
import Prelude hiding (lookup)
import Data.Map

employeeDept = fromList([("John","Sales"), ("Bob","IT")])
deptCountry = fromList([("IT","USA"), ("Sales","France")])
countryCurrency = fromList([("USA", "Dollar"), ("France", "Euro")])

employeeCurrency :: String -> Maybe String
employeeCurrency name = do
  dept <- lookup name employeeDept
  country <- lookup dept deptCountry
  lookup country countryCurrency

main = do
  putStrLn $ "John's currency: " ++ (show (employeeCurrency "John"))
  putStrLn $ "Pete's currency: " ++ (show (employeeCurrency "Pete"))
```

The output of this program:

```
John's currency: Just "Euro"
Pete's currency: Nothing
```

(!?) :: Ord k => Map k a -> k -> Maybe a | infixl 9

[# Source](#)

$O(\log n)$. Find the value at a key. Returns `Nothing` when the element can not be found.

```
fromList [(5, 'a'), (3, 'b')] !? 1 == Nothing
```

```
fromList [(5, 'a'), (3, 'b')] !? 5 == Just 'a'
```

Since: 0.5.9

(!) :: Ord k => Map k a -> k -> a | infixl 9 |

[# Source](#)

$O(\log n)$. Find the value at a key. Calls `error` when the element can not be found.

```
fromList [(5,'a'), (3,'b')] ! 1    Error: element not in the map
fromList [(5,'a'), (3,'b')] ! 5 == 'a'
```

findWithDefault :: Ord k => a -> k -> Map k a -> a

[# Source](#)

$O(\log n)$. The expression `(findWithDefault def k map)` returns the value at key `k` or returns default value `def` when the key is not in the map.

```
findWithDefault 'x' 1 (fromList [(5,'a'), (3,'b')]) == 'x'
findWithDefault 'x' 5 (fromList [(5,'a'), (3,'b')]) == 'a'
```

member :: Ord k => k -> Map k a -> Bool

[# Source](#)

$O(\log n)$. Is the key a member of the map? See also `notMember`.

```
member 5 (fromList [(5,'a'), (3,'b')]) == True
member 1 (fromList [(5,'a'), (3,'b')]) == False
```

notMember :: Ord k => k -> Map k a -> Bool

[# Source](#)

$O(\log n)$. Is the key not a member of the map? See also [member](#).

```
notMember 5 (fromList [(5,'a'), (3,'b')]) == False
notMember 1 (fromList [(5,'a'), (3,'b')]) == True
```

lookupLT :: Ord k => k -> Map k v -> Maybe (k, v)

[# Source](#)

$O(\log n)$. Find largest key smaller than the given one and return the corresponding (key, value) pair.

```
lookupLT 3 (fromList [(3,'a'), (5,'b')]) == Nothing
lookupLT 4 (fromList [(3,'a'), (5,'b')]) == Just (3, 'a')
```

lookupGT :: Ord k => k -> Map k v -> Maybe (k, v)

[# Source](#)

$O(\log n)$. Find smallest key greater than the given one and return the corresponding (key, value) pair.

```
lookupGT 4 (fromList [(3,'a'), (5,'b')]) == Just (5, 'b')
lookupGT 5 (fromList [(3,'a'), (5,'b')]) == Nothing
```

lookupLE :: Ord k => k -> Map k v -> Maybe (k, v)

[# Source](#)

$O(\log n)$. Find largest key smaller or equal to the given one and return the corresponding (key, value) pair.

```
lookupLE 2 (fromList [(3,'a'), (5,'b')]) == Nothing
lookupLE 4 (fromList [(3,'a'), (5,'b')]) == Just (3, 'a')
lookupLE 5 (fromList [(3,'a'), (5,'b')]) == Just (5, 'b')
```

lookupGE :: Ord k => k -> Map k v -> Maybe (k, v)

[# Source](#)

$O(\log n)$. Find smallest key greater or equal to the given one and return the corresponding (key, value) pair.

```
lookupGE 3 (fromList [(3,'a'), (5,'b')]) == Just (3, 'a')
lookupGE 4 (fromList [(3,'a'), (5,'b')]) == Just (5, 'b')
lookupGE 6 (fromList [(3,'a'), (5,'b')]) == Nothing
```

Size

null :: Map k a -> Bool

[# Source](#)

$O(1)$. Is the map empty?

```
Data.Map.null (empty)           == True
Data.Map.null (singleton 1 'a') == False
```

size :: Map k a -> Int

[# Source](#)

$O(1)$. The number of elements in the map.

```
size empty           == 0
size (singleton 1 'a') == 1
size (fromList([(1,'a'), (2,'c'), (3,'b')])) == 3
```

Combine

Union

union :: Ord k => Map k a -> Map k a -> Map k a

Source

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. The expression (`union t1 t2`) takes the left-biased union of `t1` and `t2`. It prefers `t1` when duplicate keys are encountered, i.e. (`union == unionWith const`).

```
union (fromList [(5, "a"), (3, "b")]) (fromList [(5, "A"), (7, "C")]) == fromList [(3, "b"), (5, "a"), (7, "C")]
```

unionWith :: Ord k => (a -> a -> a) -> Map k a -> Map k a -> Map k a

Source

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Union with a combining function.

```
unionWith (++) (fromList [(5, "a"), (3, "b")]) (fromList [(5, "A"), (7, "C")]) == fromList [(3, "b"), (5, "aA"), (7, "C")]
```

Also see the performance note on `fromListWith`.

unionWithKey :: Ord k => (k -> a -> a -> a) -> Map k a -> Map k a -> Map k a

Source

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Union with a combining function.

```
let f key left_value right_value = (show key) ++ ":" ++ left_value ++ "|" ++ right_value
unionWithKey f (fromList [(5, "a"), (3, "b")]) (fromList [(5, "A"), (7, "C")]) == fromList [(3, "b"), (5, "5:a|A"), (7, "7:C")]
```

Also see the performance note on `fromListWith`.

unions :: (Foldable f, Ord k) => f (Map k a) -> Map k a

Source

The union of a list of maps: (`unions == foldl union empty`).

```
unions [(fromList [(5, "a"), (3, "b")]), (fromList [(5, "A"), (7, "C")]), (fromList [(5, "A3"), (3, "B3")])]
== fromList [(3, "b"), (5, "a"), (7, "C")]
```

```
unions [(fromList [(5, "A3"), (3, "B3")]), (fromList [(5, "A"), (7, "C")]), (fromList [(5, "a"), (3, "b")])]
  == fromList [(3, "B3"), (5, "A3"), (7, "C")]
```

unionsWith :: (Foldable f, Ord k) => (a -> a -> a) -> f (Map k a) -> Map k a

Source

The union of a list of maps, with a combining operation: (`unionsWith f == foldl (unionWith f) empty`).

```
unionsWith (++) [(fromList [(5, "a"), (3, "b")]), (fromList [(5, "A"), (7, "C")]), (fromList [(5, "A3"), (3, "B3")])]
  == fromList [(3, "bB3"), (5, "AA3"), (7, "C")]
```

Difference

difference :: Ord k => Map k a -> Map k b -> Map k a

Source

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Difference of two maps. Return elements of the first map not existing in the second map.

```
difference (fromList [(5, "a"), (3, "b")]) (fromList [(5, "A"), (7, "C")]) == singleton 3 "b"
```

(\\) :: Ord k => Map k a -> Map k b -> Map k a | infixl 9 |

Source

Same as `difference`.

differenceWith :: Ord k => (a -> b -> Maybe a) -> Map k a -> Map k b -> Map k a

Source

$O(n + m)$. Difference with a combining function. When two equal keys are encountered, the combining function is applied to the values of these keys. If it returns `Nothing`, the element is discarded (proper set difference). If it returns `Just y`, the element is updated with a new value `y`.

```
let f al ar = if al == "b" then Just (al ++ ":" ++ ar) else Nothing
differenceWith f (fromList [(5, "a"), (3, "b")]) (fromList [(5, "A"), (3, "B"), (7, "C")])
```

```
== singleton 3 "b:B"
```

differenceWithKey :: Ord k => (k -> a -> b -> Maybe a) -> Map k a -> Map k b -> Map k a

Source

$O(n + m)$. Difference with a combining function. When two equal keys are encountered, the combining function is applied to the key and both values. If it returns **Nothing**, the element is discarded (proper set difference). If it returns **Just y**, the element is updated with a new value y.

```
let f k al ar = if al == "b" then Just ((show k) ++ ":" ++ al ++ "|" ++ ar) else Nothing
differenceWithKey f (fromList [(5, "a"), (3, "b")]) (fromList [(5, "A"), (3, "B"), (10, "C")])
== singleton 3 "3:b|B"
```

Intersection

intersection :: Ord k => Map k a -> Map k b -> Map k a

Source

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Intersection of two maps. Return data in the first map for the keys existing in both maps. (**intersection** m1 m2 == **intersectionWith const** m1 m2).

```
intersection (fromList [(5, "a"), (3, "b")]) (fromList [(5, "A"), (7, "C")]) == singleton 5 "a"
```

intersectionWith :: Ord k => (a -> b -> c) -> Map k a -> Map k b -> Map k c

Source

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Intersection with a combining function.

```
intersectionWith (++) (fromList [(5, "a"), (3, "b")]) (fromList [(5, "A"), (7, "C")]) == singleton 5 "aA"
```

intersectionWithKey :: Ord k => (k -> a -> b -> c) -> Map k a -> Map k b -> Map k c

Source

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Intersection with a combining function.

```
let f k al ar = (show k) ++ ":" ++ al ++ "|" ++ ar
intersectionWithKey f (fromList [(5, "a"), (3, "b")]) (fromList [(5, "A"), (7, "C")]) == singleton 5 "5:a|A"
```

Symmetric difference

symmetricDifference :: Ord k => Map k a -> Map k a -> Map k a

[# Source](#)

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. The symmetric difference of two maps.

The result contains entries whose keys appear in exactly one of the two maps.

```
symmetricDifference
  (fromList [(0,'q'),(2,'b'),(4,'w'),(6,'o')])
  (fromList [(0,'e'),(3,'r'),(6,'t'),(9,'s')])
==
fromList [(2,'b'),(3,'r'),(4,'w'),(9,'s')]
```

Since: 0.8

Disjoint

disjoint :: Ord k => Map k a -> Map k b -> Bool

[# Source](#)

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Check whether the key sets of two maps are disjoint (i.e., their **intersection** is empty).

```
disjoint (fromList [(2,'a')]) (fromList [(1,()), (3,())]) == True
disjoint (fromList [(2,'a')]) (fromList [(1,'a'), (2,'b')]) == False
disjoint (fromList []) (fromList []) == True
```

```
xs `disjoint` ys = null (xs `intersection` ys)
```

Since: 0.6.2.1

Compose

compose :: Ord b => Map b c -> Map a b -> Map a c

[# Source](#)

Relate the keys of one map to the values of the other, by using the values of the former as keys for lookups in the latter.

Complexity: $O(n \log m)$, where m is the size of the first argument

```
compose (fromList [('a', "A"), ('b', "B")]) (fromList [(1,'a'),(2,'b'),(3,'z')]) = fromList [(1,"A"),(2,"B")]
```

```
(compose bc ab !?) = (bc !?) <=< (ab !?)
```

Note: Prior to v0.6.4, `Data.Map.Strict` exposed a version of `compose` that forced the values of the output `Map`. This version does not force these values.

▷ **Note on complexity**

Since: 0.6.3.1

General combining functions

See [Data.Map.Merge.Strict](#)

Deprecated general combining function

mergeWithKey :: Ord k => (k -> a -> b -> Maybe c) -> (Map k a -> Map k c) -> (Map k b -> Map k c) -> Map k a -> Map k b
-> Map k c

[# Source](#)

$O(n + m)$. An unsafe universal combining function.

Warning: This function can produce corrupt maps and its results may depend on the internal structures of its inputs. Users should prefer `merge` or `mergeA`.

When `mergeWithKey` is given three arguments, it is inlined to the call site. You should therefore use `mergeWithKey` only to define custom combining functions. For example, you could define `unionWithKey`, `differenceWithKey` and `intersectionWithKey` as

```
myUnionWithKey f m1 m2 = mergeWithKey (\k x1 x2 -> Just (f k x1 x2)) id id m1 m2
myDifferenceWithKey f m1 m2 = mergeWithKey f id (const empty) m1 m2
myIntersectionWithKey f m1 m2 = mergeWithKey (\k x1 x2 -> Just (f k x1 x2)) (const empty) (const empty) m1 m2
```

When calling `mergeWithKey combine only1 only2`, a function combining two `Maps` is created, such that

- if a key is present in both maps, it is passed with both corresponding values to the `combine` function. Depending on the result, the key is either present in the result with specified value, or is left out;
- a nonempty subtree present only in the first map is passed to `only1` and the output is added to the result;
- a nonempty subtree present only in the second map is passed to `only2` and the output is added to the result.

The `only1` and `only2` methods *must return a map with a subset (possibly empty) of the keys of the given map*. The values can be modified arbitrarily. Most common variants of `only1` and `only2` are `id` and `const empty`, but for example `map f` or `filterWithKey f` could be used for any `f`.

Traversal

Map

`map :: (a -> b) -> Map k a -> Map k b`

[# Source](#)

$O(n)$. Map a function over all values in the map.

```
map (++ "x") (fromList [(5,"a"), (3,"b")]) == fromList [(3, "bx"), (5, "ax")]
```

`mapWithKey :: (k -> a -> b) -> Map k a -> Map k b`

[# Source](#)

$O(n)$. Map a function over all values in the map.

```
let f key x = (show key) ++ ":" ++ x
mapWithKey f (fromList [(5,"a"), (3,"b")]) == fromList [(3, "3:b"), (5, "5:a")]
```

traverseWithKey :: *Applicative* t => (k -> a -> t b) -> Map k a -> t (Map k b)

Source

O(n). `traverseWithKey` f m == `fromList <$> traverse (\(k, v) -> (v' -> v' `seq` (k,v')) <$> f k v) (toList m)` That is, it behaves much like a regular `traverse` except that the traversing function also has access to the key associated with a value and the values are forced before they are installed in the result map.

```
traverseWithKey (\k v -> if odd k then Just (succ v) else Nothing) (fromList [(1, 'a'), (5, 'e')]) == Just (fromList [
traverseWithKey (\k v -> if odd k then Just (succ v) else Nothing) (fromList [(2, 'c')]) == Nothing
```

traverseMaybeWithKey :: *Applicative* f => (k -> a -> f (Maybe b)) -> Map k a -> f (Map k b)

Source

O(n). Traverse keys/values and collect the `Just` results.

Since: 0.5.8

mapAccum :: (a -> b -> (a, c)) -> a -> Map k b -> (a, Map k c)

Source

O(n). The function `mapAccum` threads an accumulating argument through the map in ascending order of keys.

```
let f a b = (a ++ b, b ++ "X")
mapAccum f "Everything: " (fromList [(5,"a"), (3,"b")]) == ("Everything: ba", fromList [(3, "bX"), (5, "aX")])
```

mapAccumWithKey :: (a -> k -> b -> (a, c)) -> a -> Map k b -> (a, Map k c)

Source

O(n). The function `mapAccumWithKey` threads an accumulating argument through the map in ascending order of keys.

```
let f a k b = (a ++ " " ++ (show k) ++ "-" ++ b, b ++ "X")
```

```
mapAccumWithKey f "Everything:" (fromList [(5,"a"), (3,"b")]) == ("Everything: 3-b 5-a", fromList [(3, "bX"), (5, "aX")])
```

mapAccumRWithKey :: (a -> k -> b -> (a, c)) -> a -> Map k b -> (a, Map k c)

Source

$O(n)$. The function `mapAccumRWithKey` threads an accumulating argument through the map in descending order of keys.

mapKeys :: Ord k2 => (k1 -> k2) -> Map k1 a -> Map k2 a

Source

$O(n \log n)$. `mapKeys f s` is the map obtained by applying `f` to each key of `s`.

If `f` is monotonically non-decreasing, this function takes $O(n)$ time.

The size of the result may be smaller if `f` maps two or more distinct keys to the same new key. In this case the value at the greatest of the original keys is retained.

```
mapKeys (+ 1) (fromList [(5,"a"), (3,"b")]) == fromList [(4, "b"), (6, "a")]
mapKeys (\ _ -> 1) (fromList [(1,"b"), (2,"a"), (3,"d"), (4,"c")]) == singleton 1 "c"
mapKeys (\ _ -> 3) (fromList [(1,"b"), (2,"a"), (3,"d"), (4,"c")]) == singleton 3 "c"
```

mapKeysWith :: Ord k2 => (a -> a -> a) -> (k1 -> k2) -> Map k1 a -> Map k2 a

Source

$O(n \log n)$. `mapKeysWith c f s` is the map obtained by applying `f` to each key of `s`.

If `f` is monotonically non-decreasing, this function takes $O(n)$ time.

The size of the result may be smaller if `f` maps two or more distinct keys to the same new key. In this case the associated values will be combined using `c`. The value at the greater of the two original keys is used as the first argument to `c`.

```
mapKeysWith (++) (\ _ -> 1) (fromList [(1,"b"), (2,"a"), (3,"d"), (4,"c")]) == singleton 1 "cdab"
mapKeysWith (++) (\ _ -> 3) (fromList [(1,"b"), (2,"a"), (3,"d"), (4,"c")]) == singleton 3 "cdab"
```

Also see the performance note on `fromListWith`.

mapKeysMonotonic :: (k1 -> k2) -> Map k1 a -> Map k2 a

[# Source](#)

$O(n)$. `mapKeysMonotonic f s == mapKeys f s`, but works only when `f` is strictly monotonic. That is, for any values `x` and `y`, if `x < y` then `f x < f y`. Semi-formally, we have:

```
and [x < y ==> f x < f y | x <- ls, y <- ls]
    ==> mapKeysMonotonic f s == mapKeys f s
    where ls = keys s
```

This means that `f` maps distinct original keys to distinct resulting keys. This function has better performance than `mapKeys`.

Warning: This function should be used only if `f` is monotonically strictly increasing. This precondition is not checked. Use `mapKeys` if the precondition may not hold.

```
mapKeysMonotonic (\ k -> k * 2) (fromList [(5,"a"), (3,"b")]) == fromList [(6, "b"), (10, "a")]
valid (mapKeysMonotonic (\ k -> k * 2) (fromList [(5,"a"), (3,"b")])) == True
valid (mapKeysMonotonic (\ _ -> 1) (fromList [(5,"a"), (3,"b")])) == False
```

Folds

foldr :: (a -> b -> b) -> b -> Map k a -> b

[# Source](#)

$O(n)$. Fold the values in the map using the given right-associative binary operator, such that `foldr f z == foldr f z . elems`.

For example,

```
elems map = foldr (:) [] map
```

```
let f a len = len + (length a)
foldr f 0 (fromList [(5,"a"), (3,"bbb")]) == 4
```

foldl :: (a -> b -> a) -> a -> Map k b -> a

[# Source](#)

$O(n)$. Fold the values in the map using the given left-associative binary operator, such that `foldl f z == foldl f z . elems`.

For example,

```
elems = reverse . foldl (flip (:)) []
```

```
let f len a = len + (length a)
foldl f 0 (fromList [(5,"a"), (3,"bbb")]) == 4
```

foldrWithKey :: (k -> a -> b -> b) -> b -> Map k a -> b

[# Source](#)

$O(n)$. Fold the keys and values in the map using the given right-associative binary operator, such that `foldrWithKey f z == foldr (uncurry f) z . toAscList`.

For example,

```
keys map = foldrWithKey (\k x ks -> k:ks) [] map
```

```
let f k a result = result ++ "(" ++ (show k) ++ ":" ++ a ++ ")"
foldrWithKey f "Map: " (fromList [(5,"a"), (3,"b")]) == "Map: (5:a)(3:b)"
```

foldlWithKey :: (a -> k -> b -> a) -> a -> Map k b -> a

[# Source](#)

$O(n)$. Fold the keys and values in the map using the given left-associative binary operator, such that `foldlWithKey f z == foldl (\z' (kx, x) -> f z' kx x) z . toAscList`.

For example,

```
keys = reverse . foldlWithKey (\ks k x -> k:ks) []
```

```
let f result k a = result ++ "(" ++ (show k) ++ ":" ++ a ++ ")"
foldlWithKey f "Map: " (fromList [(5,"a"), (3,"b")]) == "Map: (3:b)(5:a)"
```

foldMapWithKey :: Monoid m => (k -> a -> m) -> Map k a -> m

Source

$O(n)$. Fold the keys and values in the map using the given monoid, such that

```
foldMapWithKey f = fold . mapWithKey f
```

This can be an asymptotically faster than `foldrWithKey` or `foldlWithKey` for some monoids.

Since: 0.5.4

Strict folds

foldr' :: (a -> b -> b) -> b -> Map k a -> b

Source

$O(n)$. A strict version of `foldr`. Each application of the operator is evaluated before using the result in the next application. This function is strict in the starting value.

foldl' :: (a -> b -> a) -> a -> Map k b -> a

Source

$O(n)$. A strict version of `foldl`. Each application of the operator is evaluated before using the result in the next application. This function is strict in the starting value.

foldrWithKey' :: (k -> a -> b -> b) -> b -> Map k a -> b

Source

$O(n)$. A strict version of `foldrWithKey`. Each application of the operator is evaluated before using the result in the next application. This function is strict in the starting value.

foldlWithKey' :: (a -> k -> b -> a) -> a -> Map k b -> a

Source

$O(n)$. A strict version of `foldlWithKey`. Each application of the operator is evaluated before using the result in the next application. This function is strict in the starting value.

Conversion

elems :: Map k a -> [a]

Source

$O(n)$. Return all elements of the map in the ascending order of their keys. Subject to list fusion.

```
elems (fromList [(5,"a"), (3,"b")]) == ["b","a"]
elems empty == []
```

keys :: Map k a -> [k]

Source

$O(n)$. Return all keys of the map in ascending order. Subject to list fusion.

```
keys (fromList [(5,"a"), (3,"b")]) == [3,5]
keys empty == []
```

assocs :: Map k a -> [(k, a)]

Source

$O(n)$. An alias for `toAscList`. Return all key/value pairs in the map in ascending key order. Subject to list fusion.

```
assocs (fromList [(5,"a"), (3,"b")]) == [(3,"b"), (5,"a")]
assocs empty == []
```

keySet :: Map k a -> Set k

Source

$O(n)$. The set of all keys of the map.

```
keysSet (fromList [(5,"a"), (3,"b")]) == Data.Set.fromList [3,5]
keysSet empty == Data.Set.empty
```

argSet :: Map k a -> Set (Arg k a)

[# Source](#)

$O(n)$. The set of all elements of the map contained in **Args**.

```
argSet (fromList [(5,"a"), (3,"b")]) == Data.Set.fromList [Arg 3 "b",Arg 5 "a"]
argSet empty == Data.Set.empty
```

Since: 0.6.6

Lists

toList :: Map k a -> [(k, a)]

[# Source](#)

$O(n)$. Convert the map to a list of key/value pairs. Subject to list fusion.

```
toList (fromList [(5,"a"), (3,"b")]) == [(3,"b"), (5,"a")]
toList empty == []
```

Ordered lists

toAscList :: Map k a -> [(k, a)]

[# Source](#)

$O(n)$. Convert the map to a list of key/value pairs where the keys are in ascending order. Subject to list fusion.

```
toAscList (fromList [(5,"a"), (3,"b")]) == [(3,"b"), (5,"a")]
```

toDescList :: Map k a -> [(k, a)]

Source

$O(n)$. Convert the map to a list of key/value pairs where the keys are in descending order. Subject to list fusion.

```
toDescList (fromList [(5,"a"), (3,"b")]) == [(5,"a"), (3,"b")]
```

Filter

filter :: (a -> Bool) -> Map k a -> Map k a

Source

$O(n)$. Filter all values that satisfy the predicate.

```
filter (> "a") (fromList [(5,"a"), (3,"b")]) == singleton 3 "b"
filter (> "x") (fromList [(5,"a"), (3,"b")]) == empty
filter (< "a") (fromList [(5,"a"), (3,"b")]) == empty
```

filterKeys :: (k -> Bool) -> Map k a -> Map k a

Source

$O(n)$. Filter all keys that satisfy the predicate.

```
filterKeys p = filterWithKey (\k _ -> p k)
```

```
filterKeys (> 4) (fromList [(5,"a"), (3,"b")]) == singleton 5 "a"
```

Since: 0.8

filterWithKey :: (k -> a -> Bool) -> Map k a -> Map k a

Source

$O(n)$. Filter all keys/values that satisfy the predicate.

```
filterWithKey (\k _ -> k > 4) (fromList [(5,"a"), (3,"b")]) == singleton 5 "a"
```

restrictKeys :: Ord k => Map k a -> Set k -> Map k a

Source

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Restrict a Map to only those keys found in a Set.

```
m `restrictKeys` s = filterWithKey (\k _ -> k `member` s) m
m `restrictKeys` s = m `intersection` fromSet (const ()) s
```

Since: 0.5.8

withoutKeys :: Ord k => Map k a -> Set k -> Map k a

Source

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Remove all keys in a Set from a Map.

```
m `withoutKeys` s = filterWithKey (\k _ -> k `notMember` s) m
m `withoutKeys` s = m `difference` fromSet (const ()) s
```

Since: 0.5.8

partition :: (a -> Bool) -> Map k a -> (Map k a, Map k a)

Source

$O(n)$. Partition the map according to a predicate. The first map contains all elements that satisfy the predicate, the second all elements that fail the predicate. See also [split](#).

```
partition (> "a") (fromList [(5,"a"), (3,"b")]) == (singleton 3 "b", singleton 5 "a")
partition (< "x") (fromList [(5,"a"), (3,"b")]) == (fromList [(3, "b"), (5, "a")], empty)
partition (> "x") (fromList [(5,"a"), (3,"b")]) == (empty, fromList [(3, "b"), (5, "a")])
```

partitionWithKey :: (k -> a -> Bool) -> Map k a -> (Map k a, Map k a)

[# Source](#)

$O(n)$. Partition the map according to a predicate. The first map contains all elements that satisfy the predicate, the second all elements that fail the predicate. See also [split](#).

```
partitionWithKey (\ k _ -> k > 3) (fromList [(5,"a"), (3,"b")]) == (singleton 5 "a", singleton 3 "b")
partitionWithKey (\ k _ -> k < 7) (fromList [(5,"a"), (3,"b")]) == (fromList [(3, "b"), (5, "a")], empty)
partitionWithKey (\ k _ -> k > 7) (fromList [(5,"a"), (3,"b")]) == (empty, fromList [(3, "b"), (5, "a")])
```

takeWhileAntitone :: (k -> Bool) -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Take while a predicate on the keys holds. The user is responsible for ensuring that for all keys j and k in the map, $j < k \implies p j \geq p k$. See note at [spanAntitone](#).

```
takeWhileAntitone p = fromDistinctAscList . takeWhile (p . fst) . toList
takeWhileAntitone p = filterWithKey (k _ -> p k)
```

Since: 0.5.8

dropWhileAntitone :: (k -> Bool) -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Drop while a predicate on the keys holds. The user is responsible for ensuring that for all keys j and k in the map, $j < k \implies p j \geq p k$. See note at [spanAntitone](#).

```
dropWhileAntitone p = fromDistinctAscList . dropWhile (p . fst) . toList
dropWhileAntitone p = filterWithKey (\k _ -> not (p k))
```

Since: 0.5.8

spanAntitone :: (k -> Bool) -> Map k a -> (Map k a, Map k a)

[# Source](#)

$O(\log n)$. Divide a map at the point where a predicate on the keys stops holding. The user is responsible for ensuring that for all keys j and k in the map, $j < k \implies p\ j \geq p\ k$.

```
spanAntitone p xs = (takeWhileAntitone p xs, dropWhileAntitone p xs)
spanAntitone p xs = partitionWithKey (\k _ -> p k) xs
```

Note: if p is not actually antitone, then `spanAntitone` will split the map at some *unspecified* point where the predicate switches from holding to not holding (where the predicate is seen to hold before the first key and to fail after the last key).

Since: 0.5.8

mapMaybe :: (a -> Maybe b) -> Map k a -> Map k b

[# Source](#)

$O(n)$. Map values and collect the **Just** results.

```
let f x = if x == "a" then Just "new a" else Nothing
mapMaybe f (fromList [(5,"a"), (3,"b")]) == singleton 5 "new a"
```

mapMaybeWithKey :: (k -> a -> Maybe b) -> Map k a -> Map k b

[# Source](#)

$O(n)$. Map keys/values and collect the **Just** results.

```
let f k _ = if k < 5 then Just ("key : " ++ (show k)) else Nothing
mapMaybeWithKey f (fromList [(5,"a"), (3,"b")]) == singleton 3 "key : 3"
```

mapEither :: (a -> Either b c) -> Map k a -> (Map k b, Map k c)

[# Source](#)

$O(n)$. Map values and separate the **Left** and **Right** results.

```
let f a = if a < "c" then Left a else Right a
mapEither f (fromList [(5,"a"), (3,"b"), (1,"x"), (7,"z")])
```

```
== (fromList [(3,"b"), (5,"a")], fromList [(1,"x"), (7,"z")])
mapEither (\ a -> Right a) (fromList [(5,"a"), (3,"b"), (1,"x"), (7,"z")])
== (empty, fromList [(5,"a"), (3,"b"), (1,"x"), (7,"z")])
```

mapEitherWithKey :: (k -> a -> Either b c) -> Map k a -> (Map k b, Map k c)

Source

$O(n)$. Map keys/values and separate the **Left** and **Right** results.

```
let f k a = if k < 5 then Left (k * 2) else Right (a ++ a)
mapEitherWithKey f (fromList [(5,"a"), (3,"b"), (1,"x"), (7,"z")])
== (fromList [(1,2), (3,6)], fromList [(5,"aa"), (7,"zz")])

mapEitherWithKey (\_ a -> Right a) (fromList [(5,"a"), (3,"b"), (1,"x"), (7,"z")])
== (empty, fromList [(1,"x"), (3,"b"), (5,"a"), (7,"z")])
```

split :: Ord k => k -> Map k a -> (Map k a, Map k a)

Source

$O(\log n)$. The expression (**split** k map) is a pair (map1, map2) where the keys in map1 are smaller than k and the keys in map2 larger than k. Any key equal to k is found in neither map1 nor map2.

```
split 2 (fromList [(5,"a"), (3,"b")]) == (empty, fromList [(3,"b"), (5,"a")])
split 3 (fromList [(5,"a"), (3,"b")]) == (empty, singleton 5 "a")
split 4 (fromList [(5,"a"), (3,"b")]) == (singleton 3 "b", singleton 5 "a")
split 5 (fromList [(5,"a"), (3,"b")]) == (singleton 3 "b", empty)
split 6 (fromList [(5,"a"), (3,"b")]) == (fromList [(3,"b"), (5,"a")], empty)
```

splitLookup :: Ord k => k -> Map k a -> (Map k a, Maybe a, Map k a)

Source

$O(\log n)$. The expression (**splitLookup** k map) splits a map just like **split** but also returns **lookup** k map.

```
splitLookup 2 (fromList [(5,"a"), (3,"b")]) == (empty, Nothing, fromList [(3,"b"), (5,"a")])
splitLookup 3 (fromList [(5,"a"), (3,"b")]) == (empty, Just "b", singleton 5 "a")
splitLookup 4 (fromList [(5,"a"), (3,"b")]) == (singleton 3 "b", Nothing, singleton 5 "a")
splitLookup 5 (fromList [(5,"a"), (3,"b")]) == (singleton 3 "b", Just "a", empty)
splitLookup 6 (fromList [(5,"a"), (3,"b")]) == (fromList [(3,"b"), (5,"a")], Nothing, empty)
```

splitRoot :: Map k b -> [Map k b]

[# Source](#)

$O(1)$. Decompose a map into pieces based on the structure of the underlying tree. This function is useful for consuming a map in parallel.

No guarantee is made as to the sizes of the pieces; an internal, but deterministic process determines this. However, it is guaranteed that the pieces returned will be in ascending order (all elements in the first submap less than all elements in the second, and so on).

Examples:

```
splitRoot (fromList (zip [1..6] ['a'..])) ==
  [fromList [(1,'a'),(2,'b'),(3,'c')],fromList [(4,'d')],fromList [(5,'e'),(6,'f')]]
```

```
splitRoot empty == []
```

Note that the current implementation does not return more than three submaps, but you should not depend on this behaviour because it can change in the future without notice.

Since: 0.5.4

Submap

isSubmapOf :: (Ord k, Eq a) => Map k a -> Map k a -> Bool

[# Source](#)

$O\left(m \log\left(\frac{n}{m} + 1\right)\right)$, $0 < m \leq n$. This function is defined as (`isSubmapOf = isSubmapOfBy (==)`).

isSubmapOfBy :: Ord k => (a -> b -> Bool) -> Map k a -> Map k b -> Bool

[# Source](#)

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. The expression (`isSubmapOfBy f t1 t2`) returns `True` if all keys in `t1` are in tree `t2`, and when `f` returns `True` when applied to their respective values. For example, the following expressions are all `True`:

```
isSubmapOfBy (==) (fromList [('a',1)]) (fromList [('a',1),('b',2)])
isSubmapOfBy (<=) (fromList [('a',1)]) (fromList [('a',1),('b',2)])
isSubmapOfBy (==) (fromList [('a',1),('b',2)]) (fromList [('a',1),('b',2)])
```

But the following are all `False`:

```
isSubmapOfBy (==) (fromList [('a',2)]) (fromList [('a',1),('b',2)])
isSubmapOfBy (<) (fromList [('a',1)]) (fromList [('a',1),('b',2)])
isSubmapOfBy (==) (fromList [('a',1),('b',2)]) (fromList [('a',1)])
```

Note that `isSubmapOfBy (_ _ -> True) m1 m2` tests whether all the keys in `m1` are also keys in `m2`.

isProperSubmapOf :: (Ord k, Eq a) => Map k a -> Map k a -> Bool

[# Source](#)

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Is this a proper submap? (ie. a submap but not equal). Defined as (`isProperSubmapOf = isProperSubmapOfBy (==)`).

isProperSubmapOfBy :: Ord k => (a -> b -> Bool) -> Map k a -> Map k b -> Bool

[# Source](#)

$O(m \log(\frac{n}{m} + 1))$, $0 < m \leq n$. Is this a proper submap? (ie. a submap but not equal). The expression (`isProperSubmapOfBy f m1 m2`) returns `True` when `keys m1` and `keys m2` are not equal, all keys in `m1` are in `m2`, and when `f` returns `True` when applied to their respective values. For example, the following expressions are all `True`:

```
isProperSubmapOfBy (==) (fromList [(1,1)]) (fromList [(1,1),(2,2)])
isProperSubmapOfBy (<=) (fromList [(1,1)]) (fromList [(1,1),(2,2)])
```

But the following are all `False`:

```
isProperSubmapOfBy (==) (fromList [(1,1),(2,2)]) (fromList [(1,1),(2,2)])
isProperSubmapOfBy (==) (fromList [(1,1),(2,2)]) (fromList [(1,1)])
isProperSubmapOfBy (<) (fromList [(1,1)]) (fromList [(1,1),(2,2)])
```

Indexed

lookupIndex :: Ord k => k -> Map k a -> Maybe Int

Source

$O(\log n)$. Look up the *index* of a key, which is its zero-based index in the sequence sorted by keys. The index is a number from 0 up to, but not including, the *size* of the map.

```
isJust (lookupIndex 2 (fromList [(5,"a"), (3,"b")])) == False
fromJust (lookupIndex 3 (fromList [(5,"a"), (3,"b")])) == 0
fromJust (lookupIndex 5 (fromList [(5,"a"), (3,"b")])) == 1
isJust (lookupIndex 6 (fromList [(5,"a"), (3,"b")])) == False
```

findIndex :: Ord k => k -> Map k a -> Int

Source

$O(\log n)$. Return the *index* of a key, which is its zero-based index in the sequence sorted by keys. The index is a number from 0 up to, but not including, the *size* of the map. Calls *error* when the key is not a *member* of the map.

```
findIndex 2 (fromList [(5,"a"), (3,"b")]) Error: element is not in the map
findIndex 3 (fromList [(5,"a"), (3,"b")]) == 0
findIndex 5 (fromList [(5,"a"), (3,"b")]) == 1
findIndex 6 (fromList [(5,"a"), (3,"b")]) Error: element is not in the map
```

elemAt :: Int -> Map k a -> (k, a)

Source

$O(\log n)$. Retrieve an element by its *index*, i.e. by its zero-based index in the sequence sorted by keys. If the *index* is out of range (less than zero, greater or equal to

`size` of the map), `error` is called.

```
elemAt 0 (fromList [(5,"a"), (3,"b")]) == (3,"b")
elemAt 1 (fromList [(5,"a"), (3,"b")]) == (5, "a")
elemAt 2 (fromList [(5,"a"), (3,"b")])   Error: index out of range
```

updateAt :: (k -> a -> Maybe a) -> Int -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Update the element at *index*. Calls `error` when an invalid index is used.

```
updateAt (\_ _ -> Just "x") 0 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "x"), (5, "a")]
updateAt (\_ _ -> Just "x") 1 (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "x")]
updateAt (\_ _ -> Just "x") 2 (fromList [(5,"a"), (3,"b")])   Error: index out of range
updateAt (\_ _ -> Just "x") (-1) (fromList [(5,"a"), (3,"b")]) Error: index out of range
updateAt (\_ _ -> Nothing) 0 (fromList [(5,"a"), (3,"b")]) == singleton 5 "a"
updateAt (\_ _ -> Nothing) 1 (fromList [(5,"a"), (3,"b")]) == singleton 3 "b"
updateAt (\_ _ -> Nothing) 2 (fromList [(5,"a"), (3,"b")])   Error: index out of range
updateAt (\_ _ -> Nothing) (-1) (fromList [(5,"a"), (3,"b")]) Error: index out of range
```

deleteAt :: Int -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Delete the element at *index*, i.e. by its zero-based index in the sequence sorted by keys. If the *index* is out of range (less than zero, greater or equal to `size` of the map), `error` is called.

```
deleteAt 0 (fromList [(5,"a"), (3,"b")]) == singleton 5 "a"
deleteAt 1 (fromList [(5,"a"), (3,"b")]) == singleton 3 "b"
deleteAt 2 (fromList [(5,"a"), (3,"b")])   Error: index out of range
deleteAt (-1) (fromList [(5,"a"), (3,"b")]) Error: index out of range
```

take :: Int -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Take a given number of entries in key order, beginning with the smallest keys.

```
take n = fromDistinctAscList . take n . toAscList
```

Since: 0.5.8

drop :: Int -> Map k a -> Map k a

Source

$O(\log n)$. Drop a given number of entries in key order, beginning with the smallest keys.

```
drop n = fromDistinctAscList . drop n . toAscList
```

Since: 0.5.8

splitAt :: Int -> Map k a -> (Map k a, Map k a)

Source

$O(\log n)$. Split a map at a particular index.

```
splitAt !n !xs = (take n xs, drop n xs)
```

Since: 0.5.8

Min/Max

lookupMin :: Map k a -> Maybe (k, a)

Source

$O(\log n)$. The minimal key of the map. Returns **Nothing** if the map is empty.

```
lookupMin (fromList [(5,"a"), (3,"b")]) == Just (3,"b")  
lookupMin empty = Nothing
```

Since: 0.5.9

lookupMax :: Map k a -> Maybe (k, a)

Source

$O(\log n)$. The maximal key of the map. Returns **Nothing** if the map is empty.

```
lookupMax (fromList [(5,"a"), (3,"b")]) == Just (5,"a")
lookupMax empty = Nothing
```

Since: 0.5.9

findMin :: Map k a -> (k, a)

Source

$O(\log n)$. The minimal key of the map. Calls **error** if the map is empty.

```
findMin (fromList [(5,"a"), (3,"b")]) == (3,"b")
findMin empty                          Error: empty map has no minimal element
```

findMax :: Map k a -> (k, a)

Source

$O(\log n)$. The maximal key of the map. Calls **error** if the map is empty.

```
findMax (fromList [(5,"a"), (3,"b")]) == (5,"a")
findMax empty                          Error: empty map has no maximal element
```

deleteMin :: Map k a -> Map k a

Source

$O(\log n)$. Delete the minimal key. Returns an empty map if the map is empty.

```
deleteMin (fromList [(5,"a"), (3,"b"), (7,"c")]) == fromList [(5,"a"), (7,"c")]
```

```
deleteMin empty == empty
```

deleteMax :: Map k a -> Map k a

[# Source](#)

$O(\log n)$. Delete the maximal key. Returns an empty map if the map is empty.

```
deleteMax (fromList [(5,"a"), (3,"b"), (7,"c")]) == fromList [(3,"b"), (5,"a")]
deleteMax empty == empty
```

deleteFindMin :: Map k a -> ((k, a), Map k a)

[# Source](#)

$O(\log n)$. Delete and find the minimal element.

```
deleteFindMin (fromList [(5,"a"), (3,"b"), (10,"c")]) == ((3,"b"), fromList[(5,"a"), (10,"c")])
deleteFindMin empty                                     Error: can not return the minimal element of an empty map
```

deleteFindMax :: Map k a -> ((k, a), Map k a)

[# Source](#)

$O(\log n)$. Delete and find the maximal element.

```
deleteFindMax (fromList [(5,"a"), (3,"b"), (10,"c")]) == ((10,"c"), fromList [(3,"b"), (5,"a")])
deleteFindMax empty                                     Error: can not return the maximal element of an empty map
```

updateMin :: (a -> Maybe a) -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Update the value at the minimal key.

```
updateMin (\ a -> Just ("X" ++ a)) (fromList [(5,"a"), (3,"b")]) == fromList [(3, "Xb"), (5, "a")]
updateMin (\ _ -> Nothing)         (fromList [(5,"a"), (3,"b")]) == singleton 5 "a"
```

updateMax :: (a -> Maybe a) -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Update the value at the maximal key.

```
updateMax (\ a -> Just ("X" ++ a)) (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "Xa")]
updateMax (\ _ -> Nothing)          (fromList [(5,"a"), (3,"b")]) == singleton 3 "b"
```

updateMinWithKey :: (k -> a -> Maybe a) -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Update the value at the minimal key.

```
updateMinWithKey (\ k a -> Just ((show k) ++ ":" ++ a)) (fromList [(5,"a"), (3,"b")]) == fromList [(3,"3:b"), (5,"a")]
updateMinWithKey (\ _ _ -> Nothing)                (fromList [(5,"a"), (3,"b")]) == singleton 5 "a"
```

updateMaxWithKey :: (k -> a -> Maybe a) -> Map k a -> Map k a

[# Source](#)

$O(\log n)$. Update the value at the maximal key.

```
updateMaxWithKey (\ k a -> Just ((show k) ++ ":" ++ a)) (fromList [(5,"a"), (3,"b")]) == fromList [(3,"b"), (5,"5:a")]
updateMaxWithKey (\ _ _ -> Nothing)                (fromList [(5,"a"), (3,"b")]) == singleton 3 "b"
```

minView :: Map k a -> Maybe (a, Map k a)

[# Source](#)

$O(\log n)$. Retrieves the value associated with minimal key of the map, and the map stripped of that element, or **Nothing** if passed an empty map.

```
minView (fromList [(5,"a"), (3,"b")]) == Just ("b", singleton 5 "a")
minView empty == Nothing
```

maxView :: Map k a -> Maybe (a, Map k a)

[# Source](#)

$O(\log n)$. Retrieves the value associated with maximal key of the map, and the map stripped of that element, or **Nothing** if passed an empty map.

```
maxView (fromList [(5,"a"), (3,"b")]) == Just ("a", singleton 3 "b")
maxView empty == Nothing
```

minViewWithKey :: Map k a -> Maybe ((k, a), Map k a)

[# Source](#)

$O(\log n)$. Retrieves the minimal (key,value) pair of the map, and the map stripped of that element, or **Nothing** if passed an empty map.

```
minViewWithKey (fromList [(5,"a"), (3,"b")]) == Just ((3,"b"), singleton 5 "a")
minViewWithKey empty == Nothing
```

maxViewWithKey :: Map k a -> Maybe ((k, a), Map k a)

[# Source](#)

$O(\log n)$. Retrieves the maximal (key,value) pair of the map, and the map stripped of that element, or **Nothing** if passed an empty map.

```
maxViewWithKey (fromList [(5,"a"), (3,"b")]) == Just ((5,"a"), singleton 3 "b")
maxViewWithKey empty == Nothing
```

Debugging

valid :: Ord k => Map k a -> Bool

[# Source](#)

$O(n)$. Test if the internal map structure is valid.

```
valid (fromAscList [(3,"b"), (5,"a")]) == True  
valid (fromAscList [(5,"a"), (3,"b")]) == False
```