





# Contents

<b>1</b>	<b>Control.DeepSeq</b>	<b>11</b>
<b>2</b>	<b>Data.IntMap.Base</b>	<b>15</b>
2.1	Map type . . . . .	16
2.2	Operators . . . . .	16
2.3	Query . . . . .	16
2.4	Construction . . . . .	18
2.4.1	Insertion . . . . .	18
2.4.2	Delete/Update . . . . .	19
2.5	Combine . . . . .	21
2.5.1	Union . . . . .	21
2.5.2	Difference . . . . .	22
2.5.3	Intersection . . . . .	22
2.5.4	Universal combining function . . . . .	23
2.6	Traversal . . . . .	24
2.6.1	Map . . . . .	24
2.7	Folds . . . . .	26
2.7.1	Strict folds . . . . .	27
2.8	Conversion . . . . .	27

2.8.1	Lists . . . . .	28
2.8.2	Ordered lists . . . . .	29
2.9	Filter . . . . .	30
2.10	Submap . . . . .	32
2.11	Min/Max . . . . .	33
2.12	Debugging . . . . .	35
2.13	Internal types . . . . .	35
2.14	Utility . . . . .	35
<b>3</b>	<b>Data.IntMap.Strict</b>	<b>37</b>
3.1	Strictness properties . . . . .	38
3.2	Map type . . . . .	39
3.3	Operators . . . . .	39
3.4	Query . . . . .	40
3.5	Construction . . . . .	41
3.5.1	Insertion . . . . .	42
3.5.2	Delete/Update . . . . .	43
3.6	Combine . . . . .	44
3.6.1	Union . . . . .	44
3.6.2	Difference . . . . .	45
3.6.3	Intersection . . . . .	46
3.6.4	Universal combining function . . . . .	46
3.7	Traversal . . . . .	47
3.7.1	Map . . . . .	47
3.8	Folds . . . . .	49
3.8.1	Strict folds . . . . .	50

<i>CONTENTS</i>	5
3.9 Conversion . . . . .	50
3.9.1 Lists . . . . .	51
3.9.2 Ordered lists . . . . .	52
3.10 Filter . . . . .	53
3.11 Submap . . . . .	55
3.12 Min/Max . . . . .	56
3.13 Debugging . . . . .	58
<b>4 Data.IntSet</b>	<b>59</b>
4.1 Strictness properties . . . . .	60
4.2 Set type . . . . .	60
4.3 Operators . . . . .	61
4.4 Query . . . . .	61
4.5 Construction . . . . .	62
4.6 Combine . . . . .	62
4.7 Filter . . . . .	63
4.8 Map . . . . .	63
4.9 Folds . . . . .	64
4.9.1 Strict folds . . . . .	64
4.9.2 Legacy folds . . . . .	64
4.10 Min/Max . . . . .	65
4.11 Conversion . . . . .	65
4.11.1 List . . . . .	65
4.11.2 Ordered list . . . . .	66
4.12 Debugging . . . . .	66

<b>5</b>	<b>Data.IntSet.Base</b>	<b>67</b>
5.1	Set type . . . . .	68
5.2	Operators . . . . .	68
5.3	Query . . . . .	69
5.4	Construction . . . . .	70
5.5	Combine . . . . .	70
5.6	Filter . . . . .	71
5.7	Map . . . . .	71
5.8	Folds . . . . .	71
5.8.1	Strict folds . . . . .	72
5.8.2	Legacy folds . . . . .	72
5.9	Min/Max . . . . .	72
5.10	Conversion . . . . .	73
5.10.1	List . . . . .	73
5.10.2	Ordered list . . . . .	74
5.11	Debugging . . . . .	74
5.12	Internals . . . . .	74
<b>6</b>	<b>Data.Map.Base</b>	<b>75</b>
6.1	Map type . . . . .	76
6.2	Operators . . . . .	77
6.3	Query . . . . .	77
6.4	Construction . . . . .	79
6.4.1	Insertion . . . . .	79
6.4.2	Delete/Update . . . . .	81
6.5	Combine . . . . .	82

<i>CONTENTS</i>	7
6.5.1 Union . . . . .	82
6.5.2 Difference . . . . .	83
6.5.3 Intersection . . . . .	84
6.5.4 Universal combining function . . . . .	85
6.6 Traversal . . . . .	85
6.6.1 Map . . . . .	85
6.7 Folds . . . . .	87
6.7.1 Strict folds . . . . .	88
6.8 Conversion . . . . .	89
6.8.1 Lists . . . . .	90
6.8.2 Ordered lists . . . . .	90
6.9 Filter . . . . .	92
6.10 Submap . . . . .	94
6.11 Indexed . . . . .	95
6.12 Min/Max . . . . .	96
6.13 Debugging . . . . .	98
<b>7 Data.Map.Strict</b>	<b>101</b>
7.1 Strictness properties . . . . .	102
7.2 Map type . . . . .	103
7.3 Operators . . . . .	103
7.4 Query . . . . .	103
7.5 Construction . . . . .	106
7.5.1 Insertion . . . . .	106
7.5.2 Delete/Update . . . . .	107
7.6 Combine . . . . .	109

7.6.1	Union . . . . .	109
7.6.2	Difference . . . . .	110
7.6.3	Intersection . . . . .	111
7.6.4	Universal combining function . . . . .	111
7.7	Traversal . . . . .	112
7.7.1	Map . . . . .	112
7.8	Folds . . . . .	114
7.8.1	Strict folds . . . . .	115
7.9	Conversion . . . . .	116
7.9.1	Lists . . . . .	116
7.9.2	Ordered lists . . . . .	117
7.10	Filter . . . . .	118
7.11	Submap . . . . .	120
7.12	Indexed . . . . .	121
7.13	Min/Max . . . . .	122
7.14	Debugging . . . . .	125
<b>8</b>	<b>Data.Set.Base</b>	<b>127</b>
8.1	Set type . . . . .	128
8.2	Operators . . . . .	128
8.3	Query . . . . .	128
8.4	Construction . . . . .	130
8.5	Combine . . . . .	130
8.6	Filter . . . . .	131
8.7	Map . . . . .	131
8.8	Folds . . . . .	132



<i>CONTENTS</i>	9
8.8.1 Strict folds . . . . .	132
8.8.2 Legacy folds . . . . .	132
8.9 Min/Max . . . . .	133
8.10 Conversion . . . . .	133
8.10.1 List . . . . .	133
8.10.2 Ordered list . . . . .	134
8.11 Debugging . . . . .	134
<b>9 Data.StrictPair</b>	<b>137</b>
<b>10 Dpfs.DavisPutnamFiniteSets</b>	<b>139</b>
10.0.1 Datatype . . . . .	139
10.1 Davis-Putnam procedures . . . . .	140
10.2 Rules: 1st Extension . . . . .	140
10.3 Rules: 2nd Extension . . . . .	141
10.4 Utilities . . . . .	142
<b>11 Dpfs.Parser</b>	<b>143</b>
<b>12 Dpfs.Simp</b>	<b>145</b>
<b>13 Dpfs.SimpCnf</b>	<b>149</b>



# Chapter 1

## Control.DeepSeq

---

```
module Control.DeepSeq (
    deepseq, ($!!), force, NFData(rnf)
) where
```

---

This module provides an overloaded function, `deepseq`, for fully evaluating data structures (that is, evaluating to "Normal Form").

A typical use is to prevent resource leaks in lazy IO programs, by forcing all characters from a file to be read. For example:

```
import System.IO
import Control.DeepSeq

main = do
    h <- openFile "f" ReadMode
    s <- hGetContents h
    s 'deepseq' hClose h
    return s
```

`deepseq` differs from `seq` as it traverses data structures deeply, for example, `seq` will evaluate only to the first constructor in the list:

```
> [1,2,undefined] 'seq' 3
3
```

While `deepseq` will force evaluation of all the list elements:

```
> [1,2,undefined] 'deepseq' 3
*** Exception: Prelude.undefined
```

Another common use is to ensure any exceptions hidden within lazy fields of a data structure do not leak outside the scope of the exception handler, or to force evaluation of a data structure in one thread, before passing to another thread (preventing work moving to the wrong threads).

```
deepseq :: NFData a => a -> b -> b
```

`deepseq`: fully evaluates the first argument, before returning the second.

The name `deepseq` is used to illustrate the relationship to `seq`: where `seq` is shallow in the sense that it only evaluates the top level of its argument, `deepseq` traverses the entire data structure evaluating it completely.

`deepseq` can be useful for forcing pending exceptions, eradicating space leaks, or forcing lazy I/O to happen. It is also useful in conjunction with parallel Strategies (see the `parallel` package).

There is no guarantee about the ordering of evaluation. The implementation may evaluate the components of the structure in any order or in parallel. To impose an actual order on evaluation, use `pseq` from `Control.Parallel` in the `parallel` package.

```
($!!) :: NFData a => (a -> b) -> a -> b
```

the deep analogue of `$!`. In the expression `f $!! x`, `x` is fully evaluated before the function `f` is applied to it.

```
force :: NFData a => a -> a
```

a variant of `deepseq` that is useful in some circumstances:

```
force x = x 'deepseq' x
```

`force x` fully evaluates `x`, and then returns it. Note that `force x` only performs evaluation when the value of `force x` itself is demanded, so essentially it turns shallow evaluation into deep evaluation.

```
class NFData a where
```

A class of types that can be fully evaluated.

**Methods**

```
rnf :: a -> ()
```

`rnf` should reduce its argument to normal form (that is, fully evaluate all sub-components), and then return `'()'`.

The default implementation of `rnf` is

```
rnf a = a 'seq' ()
```

which may be convenient when defining instances for data types with no unevaluated fields (e.g. enumerations).

```
instance NFData Bool
instance NFData Char
instance NFData Double
instance NFData Float
instance NFData Int
instance NFData Int8
instance NFData Int16
instance NFData Int32
instance NFData Int64
instance NFData Integer
instance NFData Word
instance NFData Word8
instance NFData Word16
instance NFData Word32
instance NFData Word64
instance NFData ()
instance NFData Version
instance NFData IntSet
instance NFData a => NFData [a]
instance (Integral a, NFData a) => NFData (Ratio a)
instance NFData a => NFData (Maybe a)
instance (RealFloat a, NFData a) => NFData (Complex a)
instance NFData (Fixed a)
instance NFData a => NFData (Set a)
instance NFData a => NFData (IntMap a)
```

```
instance NFData (a -> b)
```

This instance is for convenience and consistency with `seq`. This assumes that WHNF is equivalent to NF for functions.

```
instance (NFData a, NFData b) => NFData (Either a b)
instance (NFData a, NFData b) => NFData (a, b)
instance (Ix a, NFData a, NFData b) => NFData (Array a b)
instance (NFData k, NFData a) => NFData (Map k a)
instance (NFData a, NFData b, NFData c) => NFData (a, b, c)
instance (NFData a, NFData b, NFData c, NFData d) => NFData (a, b, c, d)
instance (NFData a1, NFData a2, NFData a3, NFData a4, NFData a5) => NFData (a1, a2, a3, a4, a5)
instance (NFData a1, NFData a2, NFData a3, NFData a4, NFData a5, NFData a6) => NFData (a1, a2, a3, a4, a5, a6)
instance (NFData a1, NFData a2, NFData a3, NFData a4, NFData a5, NFData a6, NFData a7) => NFData (a1, a2, a3, a4, a5, a6, a7)
instance (NFData a1, NFData a2, NFData a3, NFData a4, NFData a5, NFData a6, NFData a7, NFData a8) => NFData (a1, a2, a3, a4, a5, a6, a7, a8)
instance (NFData a1, NFData a2, NFData a3, NFData a4, NFData a5, NFData a6, NFData a7, NFData a8, NFData a9) => NFData (a1, a2, a3, a4, a5, a6, a7, a8, a9)
```

## Chapter 2

### Data.IntMap.Base

---

```
module Data.IntMap.Base (
  IntMap(Bin, Tip, Nil), Key, (!), (\\), null, size, member,
  notMember, lookup, findWithDefault, lookupLT, lookupGT, lookupLE,
  lookupGE, empty, singleton, insert, insertWith, insertWithKey,
  insertLookupWithKey, delete, adjust, adjustWithKey, update,
  updateWithKey, updateLookupWithKey, alter, union, unionWith,
  unionWithKey, unions, unionsWith, difference, differenceWith,
  differenceWithKey, intersection, intersectionWith, intersectionWithKey,
  mergeWithKey, mergeWithKey', map, mapWithKey, traverseWithKey,
  mapAccum, mapAccumWithKey, mapAccumRWithKey, mapKeys, mapKeysWith,
  mapKeysMonotonic, foldr, foldl, foldrWithKey, foldlWithKey, foldr',
  foldl', foldrWithKey', foldlWithKey', elems, keys, assocs, keysSet,
  fromSet, toList, fromList, fromListWith, fromListWithKey, toAscList,
  toDescList, fromAscList, fromAscListWith, fromAscListWithKey,
  fromDistinctAscList, filter, filterWithKey, partition,
  partitionWithKey, mapMaybe, mapMaybeWithKey, mapEither,
  mapEitherWithKey, split, splitLookup, isSubmapOf, isSubmapOfBy,
  isProperSubmapOf, isProperSubmapOfBy, findMin, findMax, deleteMin,
  deleteMax, deleteFindMin, deleteFindMax, updateMin, updateMax,
  updateMinWithKey, updateMaxWithKey, minView, maxView, minViewWithKey,
  maxViewWithKey, showTree, showTreeWith, Mask, Prefix, Nat,
  natFromInt, intFromNat, shiftRL, shiftLL, join, bin, zero, nomatch,
  match, mask, maskW, shorter, branchMask, highestBitMask, foldlStrict
) where
```

---

This defines the data structures and core (hidden) manipulations on representations.

## 2.1 Map type

```
data IntMap a
    =  Bin !Prefix !Mask !(IntMap a) !(IntMap a)
    |  Tip !Key a
    |  Nil
```

A map of integers to values a.

```
instance Functor IntMap
instance Typeable1 IntMap
instance Foldable IntMap
instance Traversable IntMap
instance Eq a => Eq (IntMap a)
instance Data a => Data (IntMap a)
instance Ord a => Ord (IntMap a)
instance Read e => Read (IntMap e)
instance Show a => Show (IntMap a)
instance NFData a => NFData (IntMap a)
instance Monoid (IntMap a)
```

```
type Key = Int
```

## 2.2 Operators

```
(!) :: IntMap a -> Key -> a
```

$O(\min(n, W))$ . 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'
```

```
(\\) :: IntMap a -> IntMap b -> IntMap a
```

Same as difference.

## 2.3 Query

```
null :: IntMap a -> Bool
```

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

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



`size :: IntMap a -> Int`

$O(n)$ . Number of elements in the map.

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

`member :: Key -> IntMap a -> Bool`

$O(\min(n, W))$ . Is the key a member of the map?

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

`notMember :: Key -> IntMap a -> Bool`

$O(\min(n, W))$ . Is the key not a member of the map?

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

`lookup :: Key -> IntMap a -> Maybe a`

$O(\min(n, W))$ . Lookup the value at a key in the map. See also `lookup`.

`findWithDefault :: a -> Key -> IntMap a -> a`

$O(\min(n, W))$ . The expression `(findWithDefault def k map)` returns the value at key `k` or returns `def` when the key is not an element of the map.

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

`lookupLT :: Key -> IntMap a -> Maybe (Key, a)`

$O(\log n)$ . 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 :: Key -> IntMap a -> Maybe (Key, a)`

$O(\log n)$ . 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 :: Key -> IntMap a -> Maybe (Key, a)`

$O(\log n)$ . 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 :: Key -> IntMap a -> Maybe (Key, a)`

$O(\log n)$ . 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
```

## 2.4 Construction

`empty :: IntMap a`

$O(1)$ . The empty map.

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

`singleton :: Key -> a -> IntMap a`

$O(1)$ . A map of one element.

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

### 2.4.1 Insertion

`insert :: Key -> a -> IntMap a -> IntMap a`

$O(\min(n, W))$ . Insert a new key/value pair in the map. If the key is already present in the map, the associated value is replaced with the supplied value, i.e. `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 :: (a -> a -> a) -> Key -> a -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . Insert with a combining function. `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 `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"
```

```
insertWithKey :: (Key -> a -> a -> a)
               -> Key -> a -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . Insert with a combining function. `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 `f key new_value old_value`.

```
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"
```

```
insertLookupWithKey :: (Key -> a -> a -> a)
                    -> Key -> a -> IntMap a -> (Maybe a, IntMap a)
```

$O(\min(n, W))$ . 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")])
```

## 2.4.2 Delete/Update

```
delete :: Key -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . 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 :: (a -> a) -> Key -> IntMap a -> IntMap a`

$O(\min(n, W))$ . Adjust a value at a specific key. 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 :: (Key -> a -> a) -> Key -> IntMap a -> IntMap a`

$O(\min(n, W))$ . 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 :: (a -> Maybe a) -> Key -> IntMap a -> IntMap a`

$O(\min(n, W))$ . 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 :: (Key -> a -> Maybe a) -> Key -> IntMap a -> IntMap a`

$O(\min(n, W))$ . The expression `(update 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 :: (Key -> a -> Maybe a)
                    -> Key -> IntMap a -> (Maybe a, IntMap a)
```

$O(\min(n, W))$ . Lookup and update. The function returns original value, if it is updated. This is different behavior than `updateLookupWithKey`. 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 "a", fromList [(3, "b"), (5, "5:
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 :: (Maybe a -> Maybe a) -> Key -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . 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 an `IntMap`. In short: `lookup k (alter f k m) = f (lookup k m)`.

## 2.5 Combine

### 2.5.1 Union

```
union :: IntMap a -> IntMap a -> IntMap a
```

$O(n+m)$ . The (left-biased) union of two maps. It prefers the first map 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,
```

```
unionWith :: (a -> a -> a) -> IntMap a -> IntMap a -> IntMap a
```

$O(n+m)$ . The union with a combining function.

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

```
unionWithKey :: (Key -> a -> a -> a)
              -> IntMap a -> IntMap a -> IntMap a
```

$O(n+m)$ . The 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, "
```

```
unions :: [IntMap a] -> IntMap a
```

The union of a list of maps.

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

```
unionsWith :: (a -> a -> a) -> [IntMap a] -> IntMap a
```

The union of a list of maps, with a combining operation.

```
unionsWith (++) [(fromList [(5, "a"), (3, "b")]), (fromList [(5, "A"), (7, "C")]), (f
    == fromList [(3, "bB3"), (5, "aAA3"), (7, "C")])
```

### 2.5.2 Difference

```
difference :: IntMap a -> IntMap b -> IntMap a
```

$O(n+m)$ . Difference between two maps (based on keys).

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

```
differenceWith :: (a -> b -> Maybe a)
               -> IntMap a -> IntMap b -> IntMap a
```

$O(n+m)$ . Difference with a combining function.

```
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"
```

[illegible]

$O(n+m)$ . 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"), (1, "C")])
== singleton 3 "3:b|B"
```

### 2.5.3 Intersection

```
intersection :: IntMap a -> IntMap b -> IntMap a
```

$O(n+m)$ . The (left-biased) intersection of two maps (based on keys).

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

```
intersectionWith :: (a -> b -> c)
                 -> IntMap a -> IntMap b -> IntMap c
```

$O(n+m)$ . The intersection with a combining function.

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

```
intersectionWithKey :: (Key -> a -> b -> c)
                   -> IntMap a -> IntMap b -> IntMap c
```

$O(n+m)$ . The 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
```

## 2.5.4 Universal combining function

```
mergeWithKey :: (Key -> a -> b -> Maybe c)
              -> (IntMap a -> IntMap c)
              -> (IntMap b -> IntMap c) -> IntMap a -> IntMap b -> IntMap c
```

$O(n+m)$ . A high-performance universal combining function. Using `mergeWithKey`, all combining functions can be defined without any loss of efficiency (with exception of `union`, `difference` and `intersection`, where sharing of some nodes is lost with `mergeWithKey`).

Please make sure you know what is going on when using `mergeWithKey`, otherwise you can be surprised by unexpected code growth or even corruption of the data structure.

When `mergeWithKey` is given three arguments, it is inlined to the call site. You should therefore use `mergeWithKey` only to define your 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
```

When calling `mergeWithKey combine only1 only2`, a function combining two `IntMaps` 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`.

```
mergeWithKey' :: (Prefix                      -> Mask -> IntMap c -> IntMap c -> IntMap c)
```

## 2.6 Traversal

### 2.6.1 Map

```
map :: (a -> b) -> IntMap a -> IntMap b
```

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

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

```
mapWithKey :: (Key -> a -> b) -> IntMap a -> IntMap b
```

$O(n)$ . 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 => (Key -> a -> t b)
                    -> IntMap a -> t (IntMap b)
```

$O(n)$ . `traverseWithKey f s == fromList $ traverse ((k, v) -> (,) k $ f k v) (toList m)` That is, behaves exactly like a regular `traverse` except that the traversing function also has access to the key associated with a value.

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

```
mapAccum :: (a -> b -> (a, c)) -> a -> IntMap b -> (a, IntMap c)
```

$O(n)$ . 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", fromLis
```



```
mapAccumWithKey :: (a -> Key -> b -> (a, c))
                 -> a -> IntMap b -> (a, IntMap c)
```

$O(n)$ . 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
```

```
mapAccumRWithKey :: (a -> Key -> b -> (a, c))
                  -> a -> IntMap b -> (a, IntMap c)
```

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

```
mapKeys :: (Key -> Key) -> IntMap a -> IntMap a
```

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

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 :: (a -> a -> a)
              -> (Key -> Key) -> IntMap a -> IntMap a
```

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

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`.

```
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"
```

```
mapKeysMonotonic :: (Key -> Key) -> IntMap a -> IntMap a
```

$O(n * \min(n, W))$ . `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`. *The precondition is not checked.* 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 slightly better performance than `mapKeys`.

```
mapKeysMonotonic (\ k -> k * 2) (fromList [(5,"a"), (3,"b")]) == fromList [(6, "b"),
```

## 2.7 Folds

```
foldr :: (a -> b -> b) -> b -> IntMap a -> b
```

$O(n)$ . 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 -> IntMap b -> a
```

$O(n)$ . 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 :: (Int -> a -> b -> b) -> b -> IntMap a -> b
```

$O(n)$ . 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 -> Int -> b -> a) -> a -> IntMap b -> a
```

$O(n)$ . 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)"

```

### 2.7.1 Strict folds

**foldr'** :: (a -> b -> b) -> b -> IntMap a -> b

$O(n)$ . 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 -> IntMap b -> a

$O(n)$ . 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'** :: (Int -> a -> b -> b) -> b -> IntMap a -> b

$O(n)$ . 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 -> Int -> b -> a) -> a -> IntMap b -> a

$O(n)$ . 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.

## 2.8 Conversion

**elems** :: IntMap a -> [a]

$O(n)$ . 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** :: IntMap a -> [Key]

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

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

```
assocs :: IntMap a -> [(Key, a)]
```

$O(n)$ . An alias for `toAscList`. Returns 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 == []
```

```
keysSet :: IntMap a -> IntSet
```

$O(n * \min(n, W))$ . The set of all keys of the map.

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

```
fromSet :: (Key -> a) -> IntSet -> IntMap a
```

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

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

### 2.8.1 Lists

```
toList :: IntMap a -> [(Key, a)]
```

$O(n)$ . 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 == []
```

```
fromList :: [(Key, a)] -> IntMap a
```

$O(n * \min(n, W))$ . Create a map from a list of key/value pairs.

```
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 :: (a -> a -> a) -> [(Key, a)] -> IntMap a
```

$O(n * \min(n, W))$ . Create a map from a list of key/value pairs with a combining function. See also `fromAscListWith`.

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

```
fromListWithKey :: (Key -> a -> a -> a) -> [(Key, a)] -> IntMap a
```

$O(n \cdot \min(n, W))$ . Build a map from a list of key/value pairs with a combining function. See also `fromAscListWithKey`.

```
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:a|b"), (5, "5:b|a")]
fromListWithKey f [] == empty
```

## 2.8.2 Ordered lists

```
toAscList :: IntMap a -> [(Key, a)]
```

$O(n)$ . 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 :: IntMap a -> [(Key, a)]
```

$O(n)$ . 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")]
```

```
fromAscList :: [(Key, a)] -> IntMap a
```

$O(n)$ . Build a map from a list of key/value pairs where the keys are in ascending order.

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

```
fromAscListWith :: (a -> a -> a) -> [(Key, a)] -> IntMap a
```

$O(n)$ . Build a map from a list of key/value pairs where the keys are in ascending order, with a combining function on equal keys. *The precondition (input list is ascending) is not checked.*

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

```
fromAscListWithKey :: (Key -> a -> a -> a)
-> [(Key, a)] -> IntMap a
```

$O(n)$ . Build a map from a list of key/value pairs where the keys are in ascending order, with a combining function on equal keys. *The precondition (input list is ascending) is not checked.*

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

```
fromDistinctAscList :: [(Key, a)] -> IntMap a
```

$O(n)$ . Build a map from a list of key/value pairs where the keys are in ascending order and all distinct. *The precondition (input list is strictly ascending) is not checked.*

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

## 2.9 Filter

```
filter :: (a -> Bool) -> IntMap a -> IntMap a
```

$O(n)$ . Filter all values that satisfy some 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
```

```
filterWithKey :: (Key -> a -> Bool) -> IntMap a -> IntMap a
```

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

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

```
partition :: (a -> Bool) -> IntMap a -> (IntMap a, IntMap a)
```

$O(n)$ . Partition the map according to some 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 :: (Key -> a -> Bool)
-> IntMap a -> (IntMap a, IntMap a)
```

$O(n)$ . Partition the map according to some 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", empty)
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")])
```

`mapMaybe :: (a -> Maybe b) -> IntMap a -> IntMap b`

$O(n)$ . 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 :: (Key -> a -> Maybe b) -> IntMap a -> IntMap b`

$O(n)$ . 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) -> IntMap a -> (IntMap b, IntMap c)`

$O(n)$ . 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 :: (Key -> a -> Either b c) -> IntMap a -> (IntMap b, IntMap c)`

$O(n)$ . 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 :: Key -> IntMap a -> (IntMap a, IntMap a)`

$O(\min(n, W))$ . The expression `(split k map)` is a pair `(map1, map2)` where all keys in `map1` are lower than `k` and all 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 :: Key -> IntMap a -> (IntMap a, Maybe a, IntMap a)`

$O(\min(n, W))$ . Performs a `split` but also returns whether the pivot key was found in the original 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)
```

## 2.10 Submap

`isSubmapOf :: Eq a => IntMap a -> IntMap a -> Bool`

$O(n+m)$ . Is this a submap? Defined as (`isSubmapOf = isSubmapOfBy (==)`).

`isSubmapOfBy :: (a -> b -> Bool) -> IntMap a -> IntMap b -> Bool`

$O(n+m)$ . The expression (`isSubmapOfBy f m1 m2`) returns `True` if 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`:

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

But the following are all `False`:

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

`isProperSubmapOf :: Eq a => IntMap a -> IntMap a -> Bool`

$O(n+m)$ . Is this a proper submap? (ie. a submap but not equal). Defined as (`isProperSubmapOf = isProperSubmapOfBy (==)`).

`isProperSubmapOfBy :: (a -> b -> Bool) -> IntMap a -> IntMap b -> Bool`

$O(n+m)$ . Is this a proper submap? (ie. a submap but not equal). The expression (`isProperSubmapOfBy f m1 m2`) returns `True` when `m1` and `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)])
```

## 2.11 Min/Max

```
findMin :: IntMap a -> (Key, a)
```

$O(\min(n, W))$ . The minimal key of the map.

```
findMax :: IntMap a -> (Key, a)
```

$O(\min(n, W))$ . The maximal key of the map.

```
deleteMin :: IntMap a -> IntMap a
```

$O(\min(n, W))$ . Delete the minimal key. An error is thrown if the `IntMap` is already empty. Note, this is not the same behavior `Map`.

```
deleteMax :: IntMap a -> IntMap a
```

$O(\min(n, W))$ . Delete the maximal key. An error is thrown if the `IntMap` is already empty. Note, this is not the same behavior `Map`.

```
deleteFindMin :: IntMap a -> ((Key, a), IntMap a)
```

$O(\min(n, W))$ . Delete and find the minimal element.

```
deleteFindMax :: IntMap a -> ((Key, a), IntMap a)
```

$O(\min(n, W))$ . Delete and find the maximal element.

```
updateMin :: (a -> Maybe a) -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . 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) -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . Update the value at the maximal key.

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

```
updateMinWithKey :: (Key -> a -> Maybe a) -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . Update the value at the minimal key.

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

```
updateMaxWithKey :: (Key -> a -> Maybe a) -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . Update the value at the maximal key.

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

```
minView :: IntMap a -> Maybe (a, IntMap a)
```

$O(\min(n, W))$ . Retrieves the minimal key of the map, and the map stripped of that element, or `Nothing` if passed an empty map.

```
maxView :: IntMap a -> Maybe (a, IntMap a)
```

$O(\min(n, W))$ . Retrieves the maximal key of the map, and the map stripped of that element, or `Nothing` if passed an empty map.

```
minViewWithKey :: IntMap a -> Maybe ((Key, a), IntMap a)
```

$O(\min(n, W))$ . 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 :: IntMap a -> Maybe ((Key, a), IntMap a)
```

$O(\min(n, W))$ . 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
```

## 2.12 Debugging

`showTree :: Show a => IntMap a -> String`

$O(n)$ . Show the tree that implements the map. The tree is shown in a compressed, hanging format.

`showTreeWith :: Show a => Bool -> Bool -> IntMap a -> String`

$O(n)$ . The expression `(showTreeWith hang wide map)` shows the tree that implements the map. If `hang` is `True`, a *hanging* tree is shown otherwise a rotated tree is shown. If `wide` is `True`, an extra wide version is shown.

## 2.13 Internal types

`type Mask = Int`

`type Prefix = Int`

`type Nat = Word`

## 2.14 Utility

`natFromInt :: Key -> Nat`

`intFromNat :: Nat -> Key`

`shiftRL :: Nat -> Key -> Nat`

`shiftLL :: Nat -> Key -> Nat`

`join :: Prefix -> IntMap a -> Prefix -> IntMap a -> IntMap a`

`bin :: Prefix -> Mask -> IntMap a -> IntMap a -> IntMap a`

`zero :: Key -> Mask -> Bool`

`nomatch :: Key -> Prefix -> Mask -> Bool`

`match :: Key -> Prefix -> Mask -> Bool`

`mask :: Key -> Mask -> Prefix`

`maskW :: Nat -> Nat -> Prefix`

`shorter :: Mask -> Mask -> Bool`

```
branchMask :: Prefix -> Prefix -> Mask
```

```
highestBitMask :: Nat -> Nat
```

```
foldlStrict :: (a -> b -> a) -> a -> [b] -> a
```

## Chapter 3

# Data.IntMap.Strict

---

```
module Data.IntMap.Strict (
  IntMap, Key, (!), (\\), null, size, member, notMember, lookup,
  findWithDefault, lookupLT, lookupGT, lookupLE, lookupGE, empty,
  singleton, insert, insertWith, insertWithKey, insertLookupWithKey,
  delete, adjust, adjustWithKey, update, updateWithKey,
  updateLookupWithKey, alter, union, unionWith, unionWithKey, unions,
  unionsWith, difference, differenceWith, differenceWithKey,
  intersection, intersectionWith, intersectionWithKey, mergeWithKey, map,
  mapWithKey, traverseWithKey, mapAccum, mapAccumWithKey,
  mapAccumRWithKey, mapKeys, mapKeysWith, mapKeysMonotonic, foldr,
  foldl, foldrWithKey, foldlWithKey, foldr', foldl', foldrWithKey',
  foldlWithKey', elems, keys, assocs, keysSet, fromSet, toList,
  fromList, fromListWith, fromListWithKey, toAscList, toDescList,
  fromAscList, fromAscListWith, fromAscListWithKey, fromDistinctAscList,
  filter, filterWithKey, partition, partitionWithKey, mapMaybe,
  mapMaybeWithKey, mapEither, mapEitherWithKey, split, splitLookup,
  isSubmapOf, isSubmapOfBy, isProperSubmapOf, isProperSubmapOfBy,
  findMin, findMax, deleteMin, deleteMax, deleteFindMin, deleteFindMax,
  updateMin, updateMax, updateMinWithKey, updateMaxWithKey, minView,
  maxView, minViewWithKey, maxViewWithKey, showTree, showTreeWith
) where
```

---

An efficient implementation of maps from integer keys to values (dictionaries).

API of this module is strict in both the keys and the values. If you need value-lazy maps, use `Lazy` instead. The `IntMap` type itself is shared between the

lazy and strict modules, meaning that the same `IntMap` value can be passed to functions in both modules (although that is rarely needed).

These modules are intended to be imported qualified, to avoid name clashes with Prelude functions, e.g.

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

The implementation is based on *big-endian patricia trees*. This data structure performs especially well on binary operations like `union` and `intersection`. However, my benchmarks show that it is also (much) faster on insertions and deletions when compared to a generic size-balanced map implementation (see `Data.Map`).

- Chris Okasaki and Andy Gill, "Fast Mergeable Integer Maps", Workshop on ML, September 1998, pages 77-86, <http://citeseer.ist.psu.edu/okasaki98fast.html>
- D.R. Morrison, "PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric", Journal of the ACM, 15(4), October 1968, pages 514-534.

Operation comments contain the operation time complexity in the Big-O notation [http://en.wikipedia.org/wiki/Big\\_O\\_notation](http://en.wikipedia.org/wiki/Big_O_notation). Many operations have a worst-case complexity of  $O(\min(n, W))$ . This means that the operation can become linear in the number of elements with a maximum of  $W$  – the number of bits in an `Int` (32 or 64).

Be aware that the `Functor`, `Traversable` and `Data` instances are the same as for the `Lazy` module, so if they are used on strict maps, the resulting maps will be lazy.

### 3.1 Strictness properties

This module satisfies the following strictness properties:

1. Key and value arguments are evaluated to WHNF;
2. Keys and values are evaluated to WHNF before they are stored in the map.

Here are some examples that illustrate the first property:

```
insertWith (\ new old -> old) k undefined m == undefined
delete undefined m == undefined
```

Here are some examples that illustrate the second property:

```
map (\ v -> undefined) m == undefined    -- m is not empty
mapKeys (\ k -> undefined) m == undefined -- m is not empty
```

## 3.2 Map type

`data IntMap a`

A map of integers to values `a`.

```
instance Functor IntMap
instance Typeable1 IntMap
instance Foldable IntMap
instance Traversable IntMap
instance Eq a => Eq (IntMap a)
instance Data a => Data (IntMap a)
instance Ord a => Ord (IntMap a)
instance Read e => Read (IntMap e)
instance Show a => Show (IntMap a)
instance NFData a => NFData (IntMap a)
instance Monoid (IntMap a)
```

`type Key = Int`

## 3.3 Operators

`(!) :: IntMap a -> Key -> a`

$O(\min(n, W))$ . 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'
```

`(\\) :: IntMap a -> IntMap b -> IntMap a`

Same as `difference`.

### 3.4 Query

`null :: IntMap a -> Bool`

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

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

`size :: IntMap a -> Int`

$O(n)$ . Number of elements in the map.

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

`member :: Key -> IntMap a -> Bool`

$O(\min(n, W))$ . Is the key a member of the map?

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

`notMember :: Key -> IntMap a -> Bool`

$O(\min(n, W))$ . Is the key not a member of the map?

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

`lookup :: Key -> IntMap a -> Maybe a`

$O(\min(n, W))$ . Lookup the value at a key in the map. See also `lookup`.

`findWithDefault :: a -> Key -> IntMap a -> a`

$O(\min(n, W))$ . The expression `(findWithDefault def k map)` returns the value at key `k` or returns `def` when the key is not an element of the map.

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

`lookupLT :: Key -> IntMap a -> Maybe (Key, a)`

$O(\log n)$ . 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 :: Key -> IntMap a -> Maybe (Key, a)
```

$O(\log n)$ . 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 :: Key -> IntMap a -> Maybe (Key, a)
```

$O(\log n)$ . 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 :: Key -> IntMap a -> Maybe (Key, a)
```

$O(\log n)$ . 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
```

## 3.5 Construction

```
empty :: IntMap a
```

$O(1)$ . The empty map.

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

```
singleton :: Key -> a -> IntMap a
```

$O(1)$ . A map of one element.

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

### 3.5.1 Insertion

```
insert :: Key -> a -> IntMap a -> IntMap a
```

*O(min(n, W))*. Insert a new key/value pair in the map. If the key is already present in the map, the associated value is replaced with the supplied value, i.e. `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 :: (a -> a -> a) -> Key -> a -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . Insert with a combining function. `insertWith f key value` 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 `f new_value old_value`.

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

```
insertWithKey :: (Key -> a -> a -> a)
               -> Key -> a -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . Insert with a combining function. `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 `f key new_value old value`.

```
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")]
insertWithKey f 5 "xxx" empty == singleton 5 "xxx"
```

If the key exists in the map, this function is lazy in `x` but strict in the result of `f`.

```
insertLookupWithKey :: (Key -> a -> a -> a)
                    -> Key -> a -> IntMap a -> (Maybe a, IntMap a)
```

$O(\min(n, W))$ . 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 [(5,"xxx"), (3,"b")])
insertLookupWithKey f 7 "xxx" (fromList [(5,"a"), (3,"b")]) == (Nothing, fromList [(5,"xxx"), (3,"b")])
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,
```

### 3.5.2 Delete/Update

`delete :: Key -> IntMap a -> IntMap a`

$O(\min(n, W))$ . 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 :: (a -> a) -> Key -> IntMap a -> IntMap a`

$O(\min(n, W))$ . Adjust a value at a specific key. 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 :: (Key -> a -> a) -> Key -> IntMap a -> IntMap a`

$O(\min(n, W))$ . 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 :: (a -> Maybe a) -> Key -> IntMap a -> IntMap a`

$O(\min(n, W))$ . 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 :: (Key -> a -> Maybe a)
               -> Key -> IntMap a -> IntMap a
```

$O(\min(n, W))$ . The expression `(update 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 :: (Key -> a -> Maybe a)
                    -> Key -> IntMap a -> (Maybe a, IntMap a)
```

$O(\min(n, W))$ . Lookup and update. The function returns original value, if it is updated. This is different behavior than `updateLookupWithKey`. 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 "a", fromList [(3, "b")])
updateLookupWithKey f 7 (fromList [(5,"a"), (3,"b")]) == (Nothing, fromList [(3, "b")])
updateLookupWithKey f 3 (fromList [(5,"a"), (3,"b")]) == (Just "b", singleton 5 "a")
```

```
alter :: (Maybe a -> Maybe a) -> Key -> IntMap a -> IntMap a
```

$O(\log n)$ . 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 an `IntMap`. In short: `lookup k (alter f k m) = f (lookup k m)`.

## 3.6 Combine

### 3.6.1 Union

```
union :: IntMap a -> IntMap a -> IntMap a
```

$O(n+m)$ . The (left-biased) union of two maps. It prefers the first map 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 :: (a -> a -> a) -> IntMap a -> IntMap a -> IntMap a
```

$O(n+m)$ . The union with a combining function.

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

```
unionWithKey :: (Key -> a -> a -> a)
              -> IntMap a -> IntMap a -> IntMap a
```

$O(n+m)$ . The 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|A"), (5, "a|C"), (7, "C")]
```

```
unions :: [IntMap a] -> IntMap a
```

The union of a list of maps.

```
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 :: (a -> a -> a) -> [IntMap a] -> IntMap a
```

The union of a list of maps, with a combining operation.

```
unionsWith (++) [(fromList [(5, "a"), (3, "b")]), (fromList [(5, "A"), (7, "C")]), (fromList [(5, "A3"), (3, "b3")])]
      == fromList [(3, "bB3"), (5, "aAA3"), (7, "C")]
```

### 3.6.2 Difference

```
difference :: IntMap a -> IntMap b -> IntMap a
```

$O(n+m)$ . Difference between two maps (based on keys).

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

```
differenceWith :: (a -> b -> Maybe a)
               -> IntMap a -> IntMap b -> IntMap a
```

$O(n+m)$ . Difference with a combining function.

```
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 :: (Key -> a -> b -> Maybe a)
                  -> IntMap a -> IntMap b -> IntMap a
```

$O(n+m)$ . 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"
```

### 3.6.3 Intersection

`intersection :: IntMap a -> IntMap b -> IntMap a`

$O(n+m)$ . The (left-biased) intersection of two maps (based on keys).

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

`intersectionWith :: (a -> b -> c)`  
`-> IntMap a -> IntMap b -> IntMap c`

$O(n+m)$ . The intersection with a combining function.

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

`intersectionWithKey :: (Key -> a -> b -> c)`  
`-> IntMap a -> IntMap b -> IntMap c`

$O(n+m)$ . The 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")])
```

### 3.6.4 Universal combining function

`mergeWithKey :: (Key -> a -> b -> Maybe c)`  
`-> (IntMap a -> IntMap c)`  
`-> (IntMap b -> IntMap c) -> IntMap a -> IntMap b -> IntMap c`

$O(n+m)$ . A high-performance universal combining function. Using `mergeWithKey`, all combining functions can be defined without any loss of efficiency (with exception of `union`, `difference` and `intersection`, where sharing of some nodes is lost with `mergeWithKey`).

Please make sure you know what is going on when using `mergeWithKey`, otherwise you can be surprised by unexpected code growth or even corruption of the data structure.

When `mergeWithKey` is given three arguments, it is inlined to the call site. You should therefore use `mergeWithKey` only to define your 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 em
```

When calling `mergeWithKey combine only1 only2`, a function combining two `IntMaps` 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`.

## 3.7 Traversal

### 3.7.1 Map

```
map :: (a -> b) -> IntMap a -> IntMap b
```

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

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

```
mapWithKey :: (Key -> a -> b) -> IntMap a -> IntMap b
```

$O(n)$ . 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 => (Key -> a -> t b)
-> IntMap a -> t (IntMap b)
```

$O(n)$ . `traverseWithKey f s == fromList $ traverse ((k, v) -> (,) k $ f k v) (toList m)` That is, behaves exactly like a regular `traverse` except that the traversing function also has access to the key associated with a value.

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

```
mapAccum :: (a -> b -> (a, c)) -> a -> IntMap b -> (a, IntMap c)
```

$O(n)$ . 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
```

```
mapAccumWithKey :: (a -> Key -> b -> (a, c))
                  -> a -> IntMap b -> (a, IntMap c)
```

$O(n)$ . 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-
```

```
mapAccumRWithKey :: (a -> Key -> b -> (a, c))
                  -> a -> IntMap b -> (a, IntMap c)
```

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

```
mapKeys :: (Key -> Key) -> IntMap a -> IntMap a
```

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

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, "a"), (2, "b")]
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 :: (a -> a -> a)
              -> (Key -> Key) -> IntMap a -> IntMap a
```

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

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`.

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

```
mapKeysMonotonic :: (Key -> Key) -> IntMap a -> IntMap a
```

$O(n * \min(n, W))$ . `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`. *The precondition is not checked.* 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 slightly better performance than `mapKeys`.

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

## 3.8 Folds

```
foldr :: (a -> b -> b) -> b -> IntMap a -> b
```

$O(n)$ . 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 -> IntMap b -> a
```

$O(n)$ . 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 :: (Int -> a -> b -> b) -> b -> IntMap a -> b
```

$O(n)$ . 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 -> Int -> b -> a) -> a -> IntMap b -> a`

$O(n)$ . 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)"
```

### 3.8.1 Strict folds

`foldr' :: (a -> b -> b) -> b -> IntMap a -> b`

$O(n)$ . 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 -> IntMap b -> a`

$O(n)$ . 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' :: (Int -> a -> b -> b) -> b -> IntMap a -> b`

$O(n)$ . 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 -> Int -> b -> a) -> a -> IntMap b -> a`

$O(n)$ . 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.

## 3.9 Conversion

`elems :: IntMap a -> [a]`

$O(n)$ . 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 :: IntMap a -> [Key]
```

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

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

```
assocs :: IntMap a -> [(Key, a)]
```

$O(n)$ . An alias for `toAscList`. Returns 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 == []
```

```
keysSet :: IntMap a -> IntSet
```

$O(n * \min(n, W))$ . The set of all keys of the map.

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

```
fromSet :: (Key -> a) -> IntSet -> IntMap a
```

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

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

### 3.9.1 Lists

```
toList :: IntMap a -> [(Key, a)]
```

$O(n)$ . 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 == []
```

```
fromList :: [(Key, a)] -> IntMap a
```

$O(n * \min(n, W))$ . Create a map from a list of key/value pairs.

```

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 :: (a -> a -> a) -> [(Key, a)] -> IntMap a

```

$O(n * \min(n, W))$ . Create a map from a list of key/value pairs with a combining function. See also `fromAscListWith`.

```

fromListWith (++) [(5,"a"), (5,"b"), (3,"b"), (3,"a"), (5,"a")] == fromList [(3, "ab")
fromListWith (++) [] == empty

```

```

fromListWithKey :: (Key -> a -> a -> a) -> [(Key, a)] -> IntMap a

```

$O(n * \min(n, W))$ . Build a map from a list of key/value pairs with a combining function. See also `fromAscListWithKey`.

```

fromListWith (++) [(5,"a"), (5,"b"), (3,"b"), (3,"a"), (5,"a")] == fromList [(3, "ab")
fromListWith (++) [] == empty

```

### 3.9.2 Ordered lists

```

toAscList :: IntMap a -> [(Key, a)]

```

$O(n)$ . 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 :: IntMap a -> [(Key, a)]

```

$O(n)$ . 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")]

```

```

fromAscList :: [(Key, a)] -> IntMap a

```

$O(n)$ . Build a map from a list of key/value pairs where the keys are in ascending order.

```

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

```

```

fromAscListWith :: (a -> a -> a) -> [(Key, a)] -> IntMap a

```

$O(n)$ . Build a map from a list of key/value pairs where the keys are in ascending order, with a combining function on equal keys. *The precondition (input list is ascending) is not checked.*

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

```
fromAscListWithKey :: (Key -> a -> a -> a)
                  -> [(Key, a)] -> IntMap a
```

$O(n)$ . Build a map from a list of key/value pairs where the keys are in ascending order, with a combining function on equal keys. *The precondition (input list is ascending) is not checked.*

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

```
fromDistinctAscList :: [(Key, a)] -> IntMap a
```

$O(n)$ . Build a map from a list of key/value pairs where the keys are in ascending order and all distinct. *The precondition (input list is strictly ascending) is not checked.*

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

## 3.10 Filter

```
filter :: (a -> Bool) -> IntMap a -> IntMap a
```

$O(n)$ . Filter all values that satisfy some 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
```

```
filterWithKey :: (Key -> a -> Bool) -> IntMap a -> IntMap a
```

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

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

```
partition :: (a -> Bool) -> IntMap a -> (IntMap a, IntMap a)
```

$O(n)$ . Partition the map according to some 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 :: (Key -> a -> Bool)
                  -> IntMap a -> (IntMap a, IntMap a)
```

$O(n)$ . Partition the map according to some 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",
partitionWithKey (\ k _ -> k < 7) (fromList [(5,"a"), (3,"b")]) == (fromList [(3, "b")],
partitionWithKey (\ k _ -> k > 7) (fromList [(5,"a"), (3,"b")]) == (empty, fromList [
```

```
mapMaybe :: (a -> Maybe b) -> IntMap a -> IntMap b
```

$O(n)$ . 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 :: (Key -> a -> Maybe b) -> IntMap a -> IntMap b
```

$O(n)$ . 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) -> IntMap a -> (IntMap b, IntMap c)
```

$O(n)$ . 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 :: (Key -> a -> Either b c)
                  -> IntMap a -> (IntMap b, IntMap c)
```

$O(n)$ . 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 :: Key -> IntMap a -> (IntMap a, IntMap a)`

$O(\min(n, W))$ . The expression `(split k map)` is a pair `(map1, map2)` where all keys in `map1` are lower than `k` and all 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 :: Key -> IntMap a -> (IntMap a, Maybe a, IntMap a)`

$O(\min(n, W))$ . Performs a `split` but also returns whether the pivot key was found in the original 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)
```

## 3.11 Submap

`isSubmapOf :: Eq a => IntMap a -> IntMap a -> Bool`

$O(n+m)$ . Is this a submap? Defined as `(isSubmapOf = isSubmapOfBy (==))`.

`isSubmapOfBy :: (a -> b -> Bool) -> IntMap a -> IntMap b -> Bool`

$O(n+m)$ . The expression `(isSubmapOfBy f m1 m2)` returns `True` if 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`:

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

But the following are all `False`:

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

```
isProperSubmapOf :: Eq a => IntMap a -> IntMap a -> Bool
```

$O(n+m)$ . Is this a proper submap? (ie. a submap but not equal). Defined as `(isProperSubmapOf = isProperSubmapOfBy (==))`.

```
isProperSubmapOfBy :: (a -> b -> Bool)
                  -> IntMap a -> IntMap b -> Bool
```

$O(n+m)$ . Is this a proper submap? (ie. a submap but not equal). The expression `(isProperSubmapOfBy f m1 m2)` returns `True` when `m1` and `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)])
```

## 3.12 Min/Max

```
findMin :: IntMap a -> (Key, a)
```

$O(\min(n, W))$ . The minimal key of the map.

```
findMax :: IntMap a -> (Key, a)
```

$O(\min(n, W))$ . The maximal key of the map.

```
deleteMin :: IntMap a -> IntMap a
```

$O(\min(n, W))$ . Delete the minimal key. An error is thrown if the `IntMap` is already empty. Note, this is not the same behavior `Map`.

```
deleteMax :: IntMap a -> IntMap a
```

$O(\min(n, W))$ . Delete the maximal key. An error is thrown if the `IntMap` is already empty. Note, this is not the same behavior `Map`.

```
deleteFindMin :: IntMap a -> ((Key, a), IntMap a)
```

$O(\min(n, W))$ . Delete and find the minimal element.



`deleteFindMax :: IntMap a -> ((Key, a), IntMap a)`

$O(\min(n, W))$ . Delete and find the maximal element.

`updateMin :: (a -> Maybe a) -> IntMap a -> IntMap a`

$O(\log n)$ . 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) -> IntMap a -> IntMap a`

$O(\log n)$ . 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 :: (Key -> a -> Maybe a) -> IntMap a -> IntMap a`

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

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

`updateMaxWithKey :: (Key -> a -> Maybe a) -> IntMap a -> IntMap a`

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

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

`minView :: IntMap a -> Maybe (a, IntMap a)`

$O(\min(n, W))$ . Retrieves the minimal key of the map, and the map stripped of that element, or `Nothing` if passed an empty map.

`maxView :: IntMap a -> Maybe (a, IntMap a)`

$O(\min(n, W))$ . Retrieves the maximal key of the map, and the map stripped of that element, or `Nothing` if passed an empty map.

`minViewWithKey :: IntMap a -> Maybe ((Key, a), IntMap a)`

$O(\min(n, W))$ . 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 :: IntMap a -> Maybe ((Key, a), IntMap a)`

$O(\min(n, W))$ . 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
```

### 3.13 Debugging

`showTree :: Show a => IntMap a -> String`

$O(n)$ . Show the tree that implements the map. The tree is shown in a compressed, hanging format.

`showTreeWith :: Show a => Bool -> Bool -> IntMap a -> String`

$O(n)$ . The expression `(showTreeWith hang wide map)` shows the tree that implements the map. If `hang` is `True`, a *hanging* tree is shown otherwise a rotated tree is shown. If `wide` is `True`, an extra wide version is shown.

## Chapter 4

### Data.IntSet

---

```
module Data.IntSet (  
    IntSet, (\\), null, size, member, notMember, lookupLT, lookupGT,  
    lookupLE, lookupGE, isSubsetOf, isProperSubsetOf, empty, singleton,  
    insert, delete, union, unions, difference, intersection, filter,  
    partition, split, splitMember, map, foldr, foldl, foldr', foldl',  
    fold, findMin, findMax, deleteMin, deleteMax, deleteFindMin,  
    deleteFindMax, maxView, minView, elems, toList, fromList, toAscList,  
    toDescList, fromAscList, fromDistinctAscList, showTree, showTreeWith  
    ) where
```

---

An efficient implementation of integer sets.

These modules are intended to be imported qualified, to avoid name clashes with Prelude functions, e.g.

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

The implementation is based on *big-endian patricia trees*. This data structure performs especially well on binary operations like `union` and `intersection`. However, my benchmarks show that it is also (much) faster on insertions and deletions when compared to a generic size-balanced set implementation (see `Data.Set`).

- Chris Okasaki and Andy Gill, "Fast Mergeable Integer Maps", Workshop on ML, September 1998, pages 77-86, <http://citeseer.ist.psu.edu/okasaki98fast.html>
- D.R. Morrison, "/PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric/", Journal of the ACM, 15(4), October 1968, pages 514-534.

Additionally, this implementation places bitmaps in the leaves of the tree. Their size is the natural size of a machine word (32 or 64 bits) and greatly reduce memory footprint and execution times for dense sets, e.g. sets where it is likely that many values lie close to each other. The asymptotics are not affected by this optimization.

Many operations have a worst-case complexity of  $O(\min(n, W))$ . This means that the operation can become linear in the number of elements with a maximum of  $W$  – the number of bits in an `Int` (32 or 64).

## 4.1 Strictness properties

This module satisfies the following strictness property:

- Key arguments are evaluated to WHNF

Here are some examples that illustrate the property:

```
delete undefined s == undefined
```

## 4.2 Set type

```
data IntSet
```

A set of integers.

```
instance Eq IntSet
instance Data IntSet
instance Ord IntSet
instance Read IntSet
instance Show IntSet
instance Typeable IntSet
instance NFData IntSet
instance Monoid IntSet
```

## 4.3 Operators

`(\\)` :: IntSet -> IntSet -> IntSet

$O(n+m)$ . See `difference`.

## 4.4 Query

`null` :: IntSet -> Bool

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

`size` :: IntSet -> Int

$O(n)$ . Cardinality of the set.

`member` :: Int -> IntSet -> Bool

$O(\min(n, W))$ . Is the value a member of the set?

`notMember` :: Int -> IntSet -> Bool

$O(\min(n, W))$ . Is the element not in the set?

`lookupLT` :: Int -> IntSet -> Maybe Int

$O(\log n)$ . Find largest element smaller than the given one.

```
lookupLT 3 (fromList [3, 5]) == Nothing
lookupLT 5 (fromList [3, 5]) == Just 3
```

`lookupGT` :: Int -> IntSet -> Maybe Int

$O(\log n)$ . Find smallest element greater than the given one.

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

`lookupLE` :: Int -> IntSet -> Maybe Int

$O(\log n)$ . Find largest element smaller or equal to the given one.

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

`lookupGE :: Int -> IntSet -> Maybe Int`

$O(\log n)$ . Find smallest element greater or equal to the given one.

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

`isSubsetOf :: IntSet -> IntSet -> Bool`

$O(n+m)$ . Is this a subset? (`s1 isSubsetOf s2`) tells whether `s1` is a subset of `s2`.

`isProperSubsetOf :: IntSet -> IntSet -> Bool`

$O(n+m)$ . Is this a proper subset? (ie. a subset but not equal).

## 4.5 Construction

`empty :: IntSet`

$O(1)$ . The empty set.

`singleton :: Int -> IntSet`

$O(1)$ . A set of one element.

`insert :: Int -> IntSet -> IntSet`

$O(\min(n, W))$ . Add a value to the set. There is no left- or right bias for IntSets.

`delete :: Int -> IntSet -> IntSet`

$O(\min(n, W))$ . Delete a value in the set. Returns the original set when the value was not present.

## 4.6 Combine

`union :: IntSet -> IntSet -> IntSet`

$O(n+m)$ . The union of two sets.

`unions :: [IntSet] -> IntSet`

The union of a list of sets.

`difference :: IntSet -> IntSet -> IntSet`  
 $O(n+m)$ . Difference between two sets.

`intersection :: IntSet -> IntSet -> IntSet`  
 $O(n+m)$ . The intersection of two sets.

## 4.7 Filter

`filter :: (Int -> Bool) -> IntSet -> IntSet`  
 $O(n)$ . Filter all elements that satisfy some predicate.

`partition :: (Int -> Bool) -> IntSet -> (IntSet, IntSet)`  
 $O(n)$ . partition the set according to some predicate.

`split :: Int -> IntSet -> (IntSet, IntSet)`  
 $O(\min(n, W))$ . The expression `(split x set)` is a pair `(set1, set2)` where `set1` comprises the elements of `set` less than `x` and `set2` comprises the elements of `set` greater than `x`.

```
split 3 (fromList [1..5]) == (fromList [1,2], fromList [4,5])
```

`splitMember :: Int -> IntSet -> (IntSet, Bool, IntSet)`  
 $O(\min(n, W))$ . Performs a `split` but also returns whether the pivot element was found in the original set.

## 4.8 Map

`map :: (Int -> Int) -> IntSet -> IntSet`  
 $O(n * \min(n, W))$ . `map f s` is the set obtained by applying `f` to each element of `s`.

It's worth noting that the size of the result may be smaller if, for some  $(x, y)$ ,  $x \neq y$  &&  $f\ x == f\ y$

## 4.9 Folds

`foldr :: (Int -> b -> b) -> b -> IntSet -> b`

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

For example,

```
toAscList set = foldr (:) [] set
```

`foldl :: (a -> Int -> a) -> a -> IntSet -> a`

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

For example,

```
toDescList set = foldl (flip (:)) [] set
```

### 4.9.1 Strict folds

`foldr' :: (Int -> b -> b) -> b -> IntSet -> b`

$O(n)$ . 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 -> Int -> a) -> a -> IntSet -> a`

$O(n)$ . 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.

### 4.9.2 Legacy folds

`fold :: (Int -> b -> b) -> b -> IntSet -> b`

$O(n)$ . Fold the elements in the set using the given right-associative binary operator. This function is an equivalent of `foldr` and is present for compatibility only.

*Please note that `fold` will be deprecated in the future and removed.*



## 4.10 Min/Max

`findMin :: IntSet -> Int`

$O(\min(n, W))$ . The minimal element of the set.

`findMax :: IntSet -> Int`

$O(\min(n, W))$ . The maximal element of a set.

`deleteMin :: IntSet -> IntSet`

$O(\min(n, W))$ . Delete the minimal element.

`deleteMax :: IntSet -> IntSet`

$O(\min(n, W))$ . Delete the maximal element.

`deleteFindMin :: IntSet -> (Int, IntSet)`

$O(\min(n, W))$ . Delete and find the minimal element.

`deleteFindMin set = (findMin set, deleteMin set)`

`deleteFindMax :: IntSet -> (Int, IntSet)`

$O(\min(n, W))$ . Delete and find the maximal element.

`deleteFindMax set = (findMax set, deleteMax set)`

`maxView :: IntSet -> Maybe (Int, IntSet)`

$O(\min(n, W))$ . Retrieves the maximal key of the set, and the set stripped of that element, or `Nothing` if passed an empty set.

`minView :: IntSet -> Maybe (Int, IntSet)`

$O(\min(n, W))$ . Retrieves the minimal key of the set, and the set stripped of that element, or `Nothing` if passed an empty set.

## 4.11 Conversion

### 4.11.1 List

`elems :: IntSet -> [Int]`

$O(n)$ . An alias of `toAscList`. The elements of a set in ascending order. Subject to list fusion.

`toList :: IntSet -> [Int]`

$O(n)$ . Convert the set to a list of elements. Subject to list fusion.

`fromList :: [Int] -> IntSet`

$O(n * \min(n, W))$ . Create a set from a list of integers.

### 4.11.2 Ordered list

`toAscList :: IntSet -> [Int]`

$O(n)$ . Convert the set to an ascending list of elements. Subject to list fusion.

`toDescList :: IntSet -> [Int]`

$O(n)$ . Convert the set to a descending list of elements. Subject to list fusion.

`fromAscList :: [Int] -> IntSet`

$O(n)$ . Build a set from an ascending list of elements. *The precondition (input list is ascending) is not checked.*

`fromDistinctAscList :: [Int] -> IntSet`

$O(n)$ . Build a set from an ascending list of distinct elements. *The precondition (input list is strictly ascending) is not checked.*

## 4.12 Debugging

`showTree :: IntSet -> String`

$O(n)$ . Show the tree that implements the set. The tree is shown in a compressed, hanging format.

`showTreeWith :: Bool -> Bool -> IntSet -> String`

$O(n)$ . The expression `(showTreeWith hang wide map)` shows the tree that implements the set. If `hang` is `True`, a *hanging* tree is shown otherwise a rotated tree is shown. If `wide` is `True`, an extra wide version is shown.

## Chapter 5

### Data.IntSet.Base

---

```
module Data.IntSet.Base (
  IntSet(Bin, Tip, Nil), (\\), null, size, member, notMember, lookupLT,
  lookupGT, lookupLE, lookupGE, isSubsetOf, isProperSubsetOf, empty,
  singleton, insert, delete, union, unions, difference, intersection,
  filter, partition, split, splitMember, map, foldr, foldl, foldr',
  foldl', fold, findMin, findMax, deleteMin, deleteMax, deleteFindMin,
  deleteFindMax, maxView, minView, elems, toList, fromList, toAscList,
  toDescList, fromAscList, fromDistinctAscList, showTree, showTreeWith,
  match, suffixBitMask, prefixBitMask, bitmapOf
) where
```

---

An efficient implementation of integer sets.

These modules are intended to be imported qualified, to avoid name clashes with Prelude functions, e.g.

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

The implementation is based on *big-endian patricia trees*. This data structure performs especially well on binary operations like `union` and `intersection`. However, my benchmarks show that it is also (much) faster on insertions and deletions when compared to a generic size-balanced set implementation (see `Data.Set`).

- Chris Okasaki and Andy Gill, "Fast Mergeable Integer Maps", Workshop on ML, September 1998, pages 77-86, <http://citeseer.ist.psu.edu/okasaki98fast.html>
- D.R. Morrison, "/PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric/", Journal of the ACM, 15(4), October 1968, pages 514-534.

Additionally, this implementation places bitmaps in the leaves of the tree. Their size is the natural size of a machine word (32 or 64 bits) and greatly reduce memory footprint and execution times for dense sets, e.g. sets where it is likely that many values lie close to each other. The asymptotics are not affected by this optimization.

Many operations have a worst-case complexity of  $O(\min(n, W))$ . This means that the operation can become linear in the number of elements with a maximum of  $W$  – the number of bits in an Int (32 or 64).

## 5.1 Set type

```
data IntSet
    =  Bin !Prefix !Mask !IntSet !IntSet
    |   Tip !Prefix !BitMap
    |   Nil
    A set of integers.
```

```
instance Eq IntSet
instance Data IntSet
instance Ord IntSet
instance Read IntSet
instance Show IntSet
instance Typeable IntSet
instance NFData IntSet
instance Monoid IntSet
```

## 5.2 Operators

```
(\\) :: IntSet -> IntSet -> IntSet
       $O(n+m)$ . See difference.
```

## 5.3 Query

`null :: IntSet -> Bool`

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

`size :: IntSet -> Int`

$O(n)$ . Cardinality of the set.

`member :: Int -> IntSet -> Bool`

$O(\min(n, W))$ . Is the value a member of the set?

`notMember :: Int -> IntSet -> Bool`

$O(\min(n, W))$ . Is the element not in the set?

`lookupLT :: Int -> IntSet -> Maybe Int`

$O(\log n)$ . Find largest element smaller than the given one.

```
lookupLT 3 (fromList [3, 5]) == Nothing
lookupLT 5 (fromList [3, 5]) == Just 3
```

`lookupGT :: Int -> IntSet -> Maybe Int`

$O(\log n)$ . Find smallest element greater than the given one.

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

`lookupLE :: Int -> IntSet -> Maybe Int`

$O(\log n)$ . Find largest element smaller or equal to the given one.

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

`lookupGE :: Int -> IntSet -> Maybe Int`

$O(\log n)$ . Find smallest element greater or equal to the given one.

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

`isSubsetOf :: IntSet -> IntSet -> Bool`

$O(n+m)$ . Is this a subset? (`s1 isSubsetOf s2`) tells whether `s1` is a subset of `s2`.

`isProperSubsetOf :: IntSet -> IntSet -> Bool`

$O(n+m)$ . Is this a proper subset? (ie. a subset but not equal).

## 5.4 Construction

`empty :: IntSet`

$O(1)$ . The empty set.

`singleton :: Int -> IntSet`

$O(1)$ . A set of one element.

`insert :: Int -> IntSet -> IntSet`

$O(\min(n, W))$ . Add a value to the set. There is no left- or right bias for `IntSets`.

`delete :: Int -> IntSet -> IntSet`

$O(\min(n, W))$ . Delete a value in the set. Returns the original set when the value was not present.

## 5.5 Combine

`union :: IntSet -> IntSet -> IntSet`

$O(n+m)$ . The union of two sets.

`unions :: [IntSet] -> IntSet`

The union of a list of sets.

`difference :: IntSet -> IntSet -> IntSet`

$O(n+m)$ . Difference between two sets.

`intersection :: IntSet -> IntSet -> IntSet`

$O(n+m)$ . The intersection of two sets.

## 5.6 Filter

`filter :: (Int -> Bool) -> IntSet -> IntSet`

$O(n)$ . Filter all elements that satisfy some predicate.

`partition :: (Int -> Bool) -> IntSet -> (IntSet, IntSet)`

$O(n)$ . partition the set according to some predicate.

`split :: Int -> IntSet -> (IntSet, IntSet)`

$O(\min(n, W))$ . The expression `(split x set)` is a pair `(set1, set2)` where `set1` comprises the elements of `set` less than `x` and `set2` comprises the elements of `set` greater than `x`.

```
split 3 (fromList [1..5]) == (fromList [1,2], fromList [4,5])
```

`splitMember :: Int -> IntSet -> (IntSet, Bool, IntSet)`

$O(\min(n, W))$ . Performs a `split` but also returns whether the pivot element was found in the original set.

## 5.7 Map

`map :: (Int -> Int) -> IntSet -> IntSet`

$O(n * \min(n, W))$ . `map f s` is the set obtained by applying `f` to each element of `s`.

It's worth noting that the size of the result may be smaller if, for some  $(x, y)$ ,  $x \neq y$  &&  $f x == f y$

## 5.8 Folds

`foldr :: (Int -> b -> b) -> b -> IntSet -> b`

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

For example,

```
toAscList set = foldr (:) [] set
```

`foldl :: (a -> Int -> a) -> a -> IntSet -> a`

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

For example,

```
toDescList set = foldl (flip (:)) [] set
```

### 5.8.1 Strict folds

`foldr' :: (Int -> b -> b) -> b -> IntSet -> b`

$O(n)$ . 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 -> Int -> a) -> a -> IntSet -> a`

$O(n)$ . 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.

### 5.8.2 Legacy folds

`fold :: (Int -> b -> b) -> b -> IntSet -> b`

$O(n)$ . Fold the elements in the set using the given right-associative binary operator. This function is an equivalent of `foldr` and is present for compatibility only.

*Please note that fold will be deprecated in the future and removed.*

## 5.9 Min/Max

`findMin :: IntSet -> Int`

$O(\min(n, W))$ . The minimal element of the set.

`findMax :: IntSet -> Int`

$O(\min(n, W))$ . The maximal element of a set.

`deleteMin :: IntSet -> IntSet`

$O(\min(n, W))$ . Delete the minimal element.



`deleteMax :: IntSet -> IntSet`

$O(\min(n, W))$ . Delete the maximal element.

`deleteFindMin :: IntSet -> (Int, IntSet)`

$O(\min(n, W))$ . Delete and find the minimal element.

`deleteFindMin set = (findMin set, deleteMin set)`

`deleteFindMax :: IntSet -> (Int, IntSet)`

$O(\min(n, W))$ . Delete and find the maximal element.

`deleteFindMax set = (findMax set, deleteMax set)`

`maxView :: IntSet -> Maybe (Int, IntSet)`

$O(\min(n, W))$ . Retrieves the maximal key of the set, and the set stripped of that element, or `Nothing` if passed an empty set.

`minView :: IntSet -> Maybe (Int, IntSet)`

$O(\min(n, W))$ . Retrieves the minimal key of the set, and the set stripped of that element, or `Nothing` if passed an empty set.

## 5.10 Conversion

### 5.10.1 List

`elems :: IntSet -> [Int]`

$O(n)$ . An alias of `toAscList`. The elements of a set in ascending order. Subject to list fusion.

`toList :: IntSet -> [Int]`

$O(n)$ . Convert the set to a list of elements. Subject to list fusion.

`fromList :: [Int] -> IntSet`

$O(n * \min(n, W))$ . Create a set from a list of integers.

### 5.10.2 Ordered list

`toAscList :: IntSet -> [Int]`

$O(n)$ . Convert the set to an ascending list of elements. Subject to list fusion.

`toDescList :: IntSet -> [Int]`

$O(n)$ . Convert the set to a descending list of elements. Subject to list fusion.

`fromAscList :: [Int] -> IntSet`

$O(n)$ . Build a set from an ascending list of elements. *The precondition (input list is ascending) is not checked.*

`fromDistinctAscList :: [Int] -> IntSet`

$O(n)$ . Build a set from an ascending list of distinct elements. *The precondition (input list is strictly ascending) is not checked.*

## 5.11 Debugging

`showTree :: IntSet -> String`

$O(n)$ . Show the tree that implements the set. The tree is shown in a compressed, hanging format.

`showTreeWith :: Bool -> Bool -> IntSet -> String`

$O(n)$ . The expression `(showTreeWith hang wide map)` shows the tree that implements the set. If `hang` is `True`, a *hanging* tree is shown otherwise a rotated tree is shown. If `wide` is `True`, an extra wide version is shown.

## 5.12 Internals

`match :: Int -> Prefix -> Mask -> Bool`

`suffixBitMask :: Int`

`prefixBitMask :: Int`

`bitmapOf :: Int -> BitMap`

## Chapter 6

# Data.Map.Base

---

```
module Data.Map.Base (
    Map(Bin, Tip), (!), (\\), null, size, member, notMember, lookup,
    findWithDefault, lookupLT, lookupGT, lookupLE, lookupGE, empty,
    singleton, insert, insertWith, insertWithKey, insertLookupWithKey,
    delete, adjust, adjustWithKey, update, updateWithKey,
    updateLookupWithKey, alter, union, unionWith, unionWithKey, unions,
    unionsWith, difference, differenceWith, differenceWithKey,
    intersection, intersectionWith, intersectionWithKey, mergeWithKey, map,
    mapWithKey, traverseWithKey, mapAccum, mapAccumWithKey,
    mapAccumRWithKey, mapKeys, mapKeysWith, mapKeysMonotonic, foldr,
    foldl, foldrWithKey, foldlWithKey, foldr', foldl', foldrWithKey',
    foldlWithKey', elems, keys, assocs, keysSet, fromSet, toList,
    fromList, fromListWith, fromListWithKey, toAscList, toDescList,
    fromAscList, fromAscListWith, fromAscListWithKey, fromDistinctAscList,
    filter, filterWithKey, partition, partitionWithKey, mapMaybe,
    mapMaybeWithKey, mapEither, mapEitherWithKey, split, splitLookup,
    isSubmapOf, isSubmapOfBy, isProperSubmapOf, isProperSubmapOfBy,
    lookupIndex, findIndex, elemAt, updateAt, deleteAt, findMin, findMax,
    deleteMin, deleteMax, deleteFindMin, deleteFindMax, updateMin,
    updateMax, updateMinWithKey, updateMaxWithKey, minView, maxView,
    minViewWithKey, maxViewWithKey, showTree, showTreeWith, valid, bin,
    balance, balanced, balanceL, balanceR, delta, join, merge, glue,
    trim, trimLookupLo, foldlStrict, MaybeS(NothingS, JustS), filterGt,
    filterLt
) where
```

---

An efficient implementation of maps from keys to values (dictionaries).

Since many function names (but not the type name) clash with `Prelude` names, this module is usually imported *qualified*, e.g.

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

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, <http://www.swiss.ai.mit.edu/~adams/BB/>.
- J. Nievergelt and E.M. Reingold, "*Binary search trees of bounded balance*", SIAM journal of computing 2(1), March 1973.

Note that the implementation is *left-biased* – the elements of a first argument are always preferred to the second, for example in `union` or `insert`.

Operation comments contain the operation time complexity in the Big-O notation [http://en.wikipedia.org/wiki/Big\\_O\\_notation](http://en.wikipedia.org/wiki/Big_O_notation).

## 6.1 Map type

```
data Map k a
  =  Bin !Size !k a !(Map k a) !(Map k a)
  |  Tip

  A Map from keys k to values a.

instance Typeable2 Map
instance Functor (Map k)
instance Foldable (Map k)
instance Traversable (Map k)
instance (Eq k, Eq a) => Eq (Map k a)
instance (Data k, Data a, Ord k) => Data (Map k a)
instance (Ord k, Ord v) => Ord (Map k v)
instance (Ord k, Read k, Read e) => Read (Map k e)
instance (Show k, Show a) => Show (Map k a)
instance (NFData k, NFData a) => NFData (Map k a)
instance Ord k => Monoid (Map k v)
```

## 6.2 Operators

`(!)` :: Ord k => Map k a -> k -> a

$O(\log n)$ . 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'
```

`(\\)` :: Ord k => Map k a -> Map k b -> Map k a

Same as difference.

## 6.3 Query

`null` :: Map k a -> Bool

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

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

`size` :: Map k a -> Int

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

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

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

$O(\log n)$ . 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

$O(\log n)$ . 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
```

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

$O(\log n)$ . Lookup 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
```

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

$O(\log n)$ . 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'
```

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

$O(\log n)$ . 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)`

$O(\log n)$ . 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)
```

$O(\log n)$ . 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)
```

$O(\log n)$ . 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
```

## 6.4 Construction

```
empty :: Map k a
```

$O(1)$ . The empty map.

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

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

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

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

### 6.4.1 Insertion

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

$O(\log n)$ . 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

```

$O(\log n)$ . 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, "xxx")]
insertWith (++) 7 "xxx" (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a")]
insertWith (++) 5 "xxx" empty                          == singleton 5 "xxx"

```

```

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

```

$O(\log n)$ . 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")]
insertWithKey f 5 "xxx" empty                          == singleton 5 "xxx"

```

```

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

```

$O(\log n)$ . 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")])
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")])

```



### 6.4.2 Delete/Update

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

*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`

*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`

*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`

*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`

*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)

```

$O(\log n)$ . Lookup and update. See also `updateWithKey`. The function returns 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, "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

```

$O(\log n)$ . 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")]

```

## 6.5 Combine

### 6.5.1 Union

```

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

```

$O(n+m)$ . 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)`. The implementation uses the efficient *hedge-union* algorithm. Hedge-union is more efficient on (bigset ‘union’ smallset).

```

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

```

$O(n+m)$ . Union with a combining function. The implementation uses the efficient *hedge-union* algorithm.

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

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

$O(n+m)$ . Union with a combining function. The implementation uses the efficient *hedge-union* algorithm. Hedge-union is more efficient on (bigset ‘union’ smallset).

```
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|A"), (5, "a|A"), (7, "C")]
```

```
unions :: Ord k => [Map k a] -> Map k a
```

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

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, "aAA3"), (7, "C")]
```

## 6.5.2 Difference

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

$O(n+m)$ . Difference of two maps. Return elements of the first map not existing in the second map. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

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

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

$O(n+m)$ . 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`. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

```
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
```

$O(n+m)$ . 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`. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

```
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"), (7, "C")])
== singleton 3 "3:b|B"
```

### 6.5.3 Intersection

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

$O(n+m)$ . 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:A"
```

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

$O(n+m)$ . Intersection with a combining function.

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

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

$O(n+m)$ . Intersection with a combining function. Intersection is more efficient on (bigset ‘intersection’ smallset).

```
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" ++ singleton 3 "3:b|B"
```

### 6.5.4 Universal 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
```

$O(n+m)$ . A high-performance universal combining function. This function is used to define `unionWith`, `unionWithKey`, `differenceWith`, `differenceWithKey`, `intersectionWith`, `intersectionWithKey` and can be used to define other custom combine functions.

Please make sure you know what is going on when using `mergeWithKey`, otherwise you can be surprised by unexpected code growth or even corruption of the data structure.

When `mergeWithKey` is given three arguments, it is inlined to the call site. You should therefore use `mergeWithKey` only to define your 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 `IntMaps` 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`.

## 6.6 Traversal

### 6.6.1 Map

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

$O(n)$ . 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
```

$O(n)$ . 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)
```

$O(n)$ . `traverseWithKey f s` == `fromList $ traverse ((k, v) -> (,) k $ f k v) (toList m)` That is, behaves exactly like a regular `traverse` except that the traversing function also has access to the key associated with a value.

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

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

$O(n)$ . 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", fromLis
```

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

$O(n)$ . 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-
```

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

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

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

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

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
```

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

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`.

```
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"
```

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

$O(n)$ . `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`. *The precondition is not checked.* 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`.

```
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
```

## 6.7 Folds

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

$O(n)$ . 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`

$O(n)$ . 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`

$O(n)$ . 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`

$O(n)$ . 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)"
```

### 6.7.1 Strict folds

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

$O(n)$ . 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`

$O(n)$ . 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`

$O(n)$ . 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`

$O(n)$ . 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.

## 6.8 Conversion

`elems :: Map k a -> [a]`

$O(n)$ . 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]`

$O(n)$ . 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)]`

$O(n)$ . 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 == []
```

`keysSet :: Map k a -> Set k`

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

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

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

$O(n)$ . 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"),
fromSet undefined Data.Set.empty == empty
```

### 6.8.1 Lists

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

$O(n)$ . 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 == []
```

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

$O(n \log n)$ . 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.

```
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
```

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

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

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

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

```
let f k a1 a2 = (show k) ++ a1 ++ a2
fromListWithKey f [(5,"a"), (5,"b"), (3,"b"), (3,"a"), (5,"a")] == fromList [(3, "3ab")
fromListWithKey f [] == empty
```

### 6.8.2 Ordered lists

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

$O(n)$ . 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)]
```

$O(n)$ . 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")]
```

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

$O(n)$ . Build a map from an ascending list in linear time. *The precondition (input list is ascending) is not checked.*

```
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
```

$O(n)$ . Build a map from an ascending list in linear time with a combining function for equal keys. *The precondition (input list is ascending) is not checked.*

```
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
```

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

$O(n)$ . Build a map from an ascending list in linear time with a combining function for equal keys. *The precondition (input list is ascending) is not checked.*

```
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
```

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

$O(n)$ . Build a map from an ascending list of distinct elements in linear time. *The precondition is not checked.*

```
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
```

## 6.9 Filter

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

$O(n)$ . 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
```

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

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

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

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

$O(n)$ . 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)`

$O(n)$ . 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", empty)
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")])
```

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

$O(n)$ . 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`

$O(n)$ . 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)
```

$O(n)$ . 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)
```

$O(n)$ . 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)
```

$O(\log n)$ . 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)
```

$O(\log n)$ . 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)
```

## 6.10 Submap

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

$O(n+m)$ . This function is defined as (`isSubmapOf = isSubmapOfBy (==)`).

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

$O(n+m)$ . 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)])
```

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

$O(n+m)$ . 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`

$O(n+m)$ . Is this a proper submap? (ie. a submap but not equal). The expression (`isProperSubmapOfBy f m1 m2`) returns `True` when `m1` and `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)])
```

## 6.11 Indexed

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

$O(\log n)$ . Lookup the *index* of a key. 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`

$O(\log n)$ . Return the *index* of a key. 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)`

$O(\log n)$ . Retrieve an element by *index*. Calls *error* when an invalid index is used.

```
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`

$O(\log n)$ . 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`

$O(\log n)$ . Delete the element at *index*. Defined as `(deleteAt i map = updateAt (k x -> Nothing) i map)`.

```

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

```

## 6.12 Min/Max

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

$O(\log n)$ . 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)`

$O(\log n)$ . 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`

$O(\log n)$ . 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`

$O(\log n)$ . 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)`

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

```

deleteFindMin (fromList [(5,"a"), (3,"b"), (10,"c")]) == ((3,"b"), fromList[(5,"a"),
deleteFindMin                                          Error: can not return the mi

```

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

$O(\log n)$ . 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
```

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

*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
```

*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
```

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

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

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

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

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

*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)
```

*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)
```

$O(\log n)$ . 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)
```

$O(\log n)$ . 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
```

## 6.13 Debugging

```
showTree :: (Show k, Show a) => Map k a -> String
```

$O(n)$ . Show the tree that implements the map. The tree is shown in a compressed, hanging format. See `showTreeWith`.

```
showTreeWith :: (k -> a -> String)
              -> Bool -> Bool -> Map k a -> String
```

$O(n)$ . The expression `(showTreeWith showelem hang wide map)` shows the tree that implements the map. Elements are shown using the `showElem` function. If `hang` is `True`, a *hanging* tree is shown otherwise a rotated tree is shown. If `wide` is `True`, an extra wide version is shown.

```
Map> let t = fromDistinctAscList [(x,()) | x <- [1..5]]
Map> putStrLn $ showTreeWith (\k x -> show (k,x)) True False t
(4,())
+--(2,())
|  +--(1,())
|  +--(3,())
+--(5,())

Map> putStrLn $ showTreeWith (\k x -> show (k,x)) True True t
(4,())
|
+--(2,())
| |
|  +--(1,())
|  |
|  +--(3,())
|
|
```

```

    +---(5,())

Map> putStrLn $ showTreeWith (\k x -> show (k,x)) False True t
+---(5,())
|
(4,())
|
| +---(3,())
| |
+---(2,())
|
+---(1,())

```

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

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

```

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

```

```
bin :: k -> a -> Map k a -> Map k a -> Map k a
```

```
balance :: k -> a -> Map k a -> Map k a -> Map k a
```

```
balanced :: Map k a -> Bool
```

Exported only for `Debug.QuickCheck`

```
balanceL :: k -> a -> Map k a -> Map k a -> Map k a
```

```
balanceR :: k -> a -> Map k a -> Map k a -> Map k a
```

```
delta :: Int
```

```
join :: k -> a -> Map k a -> Map k a -> Map k a
```

```
merge :: Map k a -> Map k a -> Map k a
```

```
glue :: Map k a -> Map k a -> Map k a
```

```
trim :: Ord k => MaybeS k -> MaybeS k -> Map k a -> Map k a
```

```
trimLookupLo :: Ord k => k -> MaybeS k -> Map k a -> (Maybe a, Map k a)
```

```
foldlStrict :: (a -> b -> a) -> a -> [b] -> a
```

```
data MaybeS a
```

```

=   NothingS
|   JustS !a

```

```
filterGt :: Ord k => MaybeS k -> Map k v -> Map k v
```

```
filterLt :: Ord k => MaybeS k -> Map k v -> Map k v
```

## Chapter 7

# Data.Map.Strict

---

```
module Data.Map.Strict (
    Map, (!), (\\), null, size, member, notMember, lookup,
    findWithDefault, lookupLT, lookupGT, lookupLE, lookupGE, empty,
    singleton, insert, insertWith, insertWithKey, insertLookupWithKey,
    delete, adjust, adjustWithKey, update, updateWithKey,
    updateLookupWithKey, alter, union, unionWith, unionWithKey, unions,
    unionsWith, difference, differenceWith, differenceWithKey,
    intersection, intersectionWith, intersectionWithKey, mergeWithKey, map,
    mapWithKey, traverseWithKey, mapAccum, mapAccumWithKey,
    mapAccumRWithKey, mapKeys, mapKeysWith, mapKeysMonotonic, foldr,
    foldl, foldrWithKey, foldlWithKey, foldr', foldl', foldrWithKey',
    foldlWithKey', elems, keys, assocs, keysSet, fromSet, toList,
    fromList, fromListWith, fromListWithKey, toAscList, toDescList,
    fromAscList, fromAscListWith, fromAscListWithKey, fromDistinctAscList,
    filter, filterWithKey, partition, partitionWithKey, mapMaybe,
    mapMaybeWithKey, mapEither, mapEitherWithKey, split, splitLookup,
    isSubmapOf, isSubmapOfBy, isProperSubmapOf, isProperSubmapOfBy,
    lookupIndex, findIndex, elemAt, updateAt, deleteAt, findMin, findMax,
    deleteMin, deleteMax, deleteFindMin, deleteFindMax, updateMin,
    updateMax, updateMinWithKey, updateMaxWithKey, minView, maxView,
    minViewWithKey, maxViewWithKey, showTree, showTreeWith, valid
) where
```

---

An efficient implementation of ordered maps from keys to values (dictionaries).

API of this module is strict in both the keys and the values. If you need value-lazy maps, use `Lazy` instead. 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 (although that is rarely needed).

These modules are intended to be imported qualified, to avoid name clashes with Prelude functions, e.g.

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

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, <http://www.swiss.ai.mit.edu/~adams/BB/>.
- J. Nievergelt and E.M. Reingold, "Binary search trees of bounded balance", SIAM journal of computing 2(1), March 1973.

Note that the implementation is *left-biased* – the elements of a first argument are always preferred to the second, for example in `union` or `insert`.

Operation comments contain the operation time complexity in the Big-O notation ([http://en.wikipedia.org/wiki/Big\\_O\\_notation](http://en.wikipedia.org/wiki/Big_O_notation)).

Be aware that the `Functor`, `Traversable` and `Data` instances are the same as for the `Lazy` module, so if they are used on strict maps, the resulting maps will be lazy.

## 7.1 Strictness properties

This module satisfies the following strictness properties:

1. Key and value arguments are evaluated to WHNF;
2. Keys and values are evaluated to WHNF before they are stored in the map.

Here are some examples that illustrate the first property:

```
insertWith (\ new old -> old) k undefined m == undefined
delete undefined m == undefined
```

Here are some examples that illustrate the second property:

```
map (\ v -> undefined) m == undefined    -- m is not empty
mapKeys (\ k -> undefined) m == undefined -- m is not empty
```

## 7.2 Map type

```
data Map k a
```

A Map from keys `k` to values `a`.

```
instance Typeable2 Map
instance Functor (Map k)
instance Foldable (Map k)
instance Traversable (Map k)
instance (Eq k, Eq a) => Eq (Map k a)
instance (Data k, Data a, Ord k) => Data (Map k a)
instance (Ord k, Ord v) => Ord (Map k v)
instance (Ord k, Read k, Read e) => Read (Map k e)
instance (Show k, Show a) => Show (Map k a)
instance (NFData k, NFData a) => NFData (Map k a)
instance Ord k => Monoid (Map k v)
```

## 7.3 Operators

```
(!) :: Ord k => Map k a -> k -> a
```

$O(\log n)$ . 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'
```

```
(\\) :: Ord k => Map k a -> Map k b -> Map k a
```

Same as difference.

## 7.4 Query

```
null :: Map k a -> Bool
```

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

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

```
size :: Map k a -> Int
```

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

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

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

$O(\log n)$ . 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
```

$O(\log n)$ . 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
```

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

$O(\log n)$ . Lookup 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
```

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

$O(\log n)$ . 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'
```

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

$O(\log n)$ . 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)
```

$O(\log n)$ . 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)
```

$O(\log n)$ . 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)
```

$O(\log n)$ . 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
```

## 7.5 Construction

`empty :: Map k a`

$O(1)$ . The empty map.

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

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

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

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

### 7.5.1 Insertion

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

$O(\log n)$ . 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`

$O(\log n)$ . 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, "xxx")]
insertWith (++) 7 "xxx" (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b"), (5, "a")]
insertWith (++) 5 "xxx" empty                        == singleton 5 "xxx"
```

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

$O(\log n)$ . 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"

```

```

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

```

$O(\log n)$ . 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")])

```

## 7.5.2 Delete/Update

```

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

```

$O(\log n)$ . 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

```

$O(\log n)$ . 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

```

$O(\log n)$ . 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

```

$O(\log n)$ . 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

```

$O(\log n)$ . 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)

```

$O(\log n)$ . Lookup and update. See also `updateWithKey`. The function returns 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")])
updateLookupWithKey f 7 (fromList [(5,"a"), (3,"b")]) == (Nothing, fromList [(3, "b")])
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

```

$O(\log n)$ . 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")]

```

## 7.6 Combine

### 7.6.1 Union

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

$O(n+m)$ . 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)`. The implementation uses the efficient *hedge-union* algorithm. Hedge-union is more efficient on (bigset ‘union’ smallset).

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

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

$O(n+m)$ . Union with a combining function. The implementation uses the efficient *hedge-union* algorithm.

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

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

$O(n+m)$ . Union with a combining function. The implementation uses the efficient *hedge-union* algorithm. Hedge-union is more efficient on (bigset ‘union’ smallset).

```
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, "
```

`unions :: Ord k => [Map k a] -> Map k a`

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, "B3")]),
== fromList [(3, "B3"), (5, "A3"), (7, "C")])

```

```
unionsWith :: Ord k => (a -> a -> a) -> [Map k a] -> Map k a
```

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")]), (f
== fromList [(3, "bB3"), (5, "aAA3"), (7, "C")]
```

## 7.6.2 Difference

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

$O(n+m)$ . Difference of two maps. Return elements of the first map not existing in the second map. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

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

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

$O(n+m)$ . 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`. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

```
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
```

$O(n+m)$ . 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`. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

```
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"), (7, "C")])
== singleton 3 "3:b|B"
```

### 7.6.3 Intersection

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

$O(n+m)$ . 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`

$O(n+m)$ . 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`

$O(n+m)$ . Intersection with a combining function. Intersection is more efficient on (bigset 'intersection' smallset).

```
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|b"
```

### 7.6.4 Universal 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`

$O(n+m)$ . A high-performance universal combining function. This function is used to define `unionWith`, `unionWithKey`, `differenceWith`, `differenceWithKey`, `intersectionWith`, `intersectionWithKey` and can be used to define other custom combine functions.

Please make sure you know what is going on when using `mergeWithKey`, otherwise you can be surprised by unexpected code growth or even corruption of the data structure.

When `mergeWithKey` is given three arguments, it is inlined to the call site. You should therefore use `mergeWithKey` only to define your 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)
```

When calling `mergeWithKey combine only1 only2`, a function combining two `IntMaps` 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`.

## 7.7 Traversal

### 7.7.1 Map

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

$O(n)$ . 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
```

$O(n)$ . 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)
```

$O(n)$ . `traverseWithKey f s == fromList $ traverse ((k, v) -> (,) k $ f k v) (toList m)` That is, behaves exactly like a regular `traverse` except that the traversing function also has access to the key associated with a value.

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



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

$O(n)$ . 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")])
```

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

$O(n)$ . 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")])
```

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

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

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

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

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
```

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

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`.

```
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"
```

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

$O(n)$ . `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`. The precondition is not checked. 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`.

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

## 7.8 Folds

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

$O(n)$ . 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
```

$O(n)$ . 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
```

$O(n)$ . 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
```

$O(n)$ . 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)"
```

### 7.8.1 Strict folds

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

$O(n)$ . 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
```

$O(n)$ . 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
```

$O(n)$ . 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
```

$O(n)$ . 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.

## 7.9 Conversion

`elems :: Map k a -> [a]`

$O(n)$ . 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]`

$O(n)$ . 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)]`

$O(n)$ . 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 == []
```

`keysSet :: Map k a -> Set k`

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

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

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

$O(n)$ . 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"),
fromSet undefined Data.Set.empty == empty
```

### 7.9.1 Lists

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

$O(n)$ . 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 == []
```

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

$O(n \log n)$ . 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.

```
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
```

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

```
fromListWith (++) [(5,"a"), (5,"b"), (3,"b"), (3,"a"), (5,"a")] == fromList [(3, "ab"), (5, "aba")]
fromListWith (++) [] == empty
```

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

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

```
let f k a1 a2 = (show k) ++ a1 ++ a2
fromListWithKey f [(5,"a"), (5,"b"), (3,"b"), (3,"a"), (5,"a")] == fromList [(3, "3ab"), (5, "5a5b")]
fromListWithKey f [] == empty
```

## 7.9.2 Ordered lists

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

$O(n)$ . 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)]
```

$O(n)$ . 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")]
```

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

$O(n)$ . Build a map from an ascending list in linear time. *The precondition (input list is ascending) is not checked.*

```

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

```

$O(n)$ . Build a map from an ascending list in linear time with a combining function for equal keys. *The precondition (input list is ascending) is not checked.*

```

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

```

```

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

```

$O(n)$ . Build a map from an ascending list in linear time with a combining function for equal keys. *The precondition (input list is ascending) is not checked.*

```

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

```

```

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

```

$O(n)$ . Build a map from an ascending list of distinct elements in linear time. *The precondition is not checked.*

```

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

```

## 7.10 Filter

```

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

```

$O(n)$ . 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

```

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

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

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

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

$O(n)$ . 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)
```

$O(n)$ . 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")])
```

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

$O(n)$ . 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
```

$O(n)$ . 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)
```

$O(n)$ . 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)
```

$O(n)$ . 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)
```

$O(\log n)$ . 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)
```

$O(\log n)$ . 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)
```

## 7.11 Submap

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

$O(n+m)$ . This function is defined as `(isSubmapOf = isSubmapOfBy (==))`.

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

$O(n+m)$ . 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)])

```

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

$O(n+m)$ . 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
```

$O(n+m)$ . Is this a proper submap? (ie. a submap but not equal). The expression `(isProperSubmapOfBy f m1 m2)` returns `True` when `m1` and `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)])

```

## 7.12 Indexed

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

$O(\log n)$ . Lookup the *index* of a key. 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`

$O(\log n)$ . Return the *index* of a key. 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)`

$O(\log n)$ . Retrieve an element by *index*. Calls **error** when an invalid index is used.

```
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`

$O(\log n)$ . 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")]
updateAt (\ _ _ -> Just "x") 1    (fromList [(5,"a"), (3,"b")]) == fromList [(3, "b")]
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`

$O(\log n)$ . Delete the element at *index*. Defined as `(deleteAt i map = updateAt (k x -> Nothing) i map)`.

```
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
```

## 7.13 Min/Max

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

$O(\log n)$ . 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)
```

$O(\log n)$ . 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
```

$O(\log n)$ . 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
```

$O(\log n)$ . 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)
```

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

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

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

$O(\log n)$ . 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
```

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

$O(\log n)$ . 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
```

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

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

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

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

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

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

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

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

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

*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)
```

*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

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

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

*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)
```

*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
```

## 7.14 Debugging

`showTree :: (Show k, Show a) => Map k a -> String`

$O(n)$ . Show the tree that implements the map. The tree is shown in a compressed, hanging format. See `showTreeWith`.

`showTreeWith :: (k -> a -> String)  
-> Bool -> Bool -> Map k a -> String`

$O(n)$ . The expression `(showTreeWith showelem hang wide map)` shows the tree that implements the map. Elements are shown using the `showElem` function. If `hang` is `True`, a *hanging* tree is shown otherwise a rotated tree is shown. If `wide` is `True`, an extra wide version is shown.

```
Map> let t = fromDistinctAscList [(x,()) | x <- [1..5]]
Map> putStrLn $ showTreeWith (\k x -> show (k,x)) True False t
(4,())
+--(2,())
|  +--(1,())
|  +--(3,())
+--(5,())

Map> putStrLn $ showTreeWith (\k x -> show (k,x)) True True t
(4,())
|
+--(2,())
| |
|  +--(1,())
|  |
|  +--(3,())
|
+--(5,())

Map> putStrLn $ showTreeWith (\k x -> show (k,x)) False True t
+--(5,())
|
(4,())
|
|  +--(3,())
|  |
+--(2,())
|
+--(1,())
```

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

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

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

## Chapter 8

### Data.Set.Base

---

```
module Data.Set.Base (
  Set(Bin, Tip), (\\), null, size, member, notMember, lookupLT,
  lookupGT, lookupLE, lookupGE, isSubsetOf, isProperSubsetOf, empty,
  singleton, insert, delete, union, unions, difference, intersection,
  filter, partition, split, splitMember, map, mapMonotonic, foldr,
  foldl, foldr', foldl', fold, findMin, findMax, deleteMin, deleteMax,
  deleteFindMin, deleteFindMax, maxView, minView, elems, toList,
  fromList, toAscList, toDescList, fromAscList, fromDistinctAscList,
  showTree, showTreeWith, valid, bin, balanced, join, merge
) where
```

---

An efficient implementation of sets.

These modules are intended to be imported qualified, to avoid name clashes with Prelude functions, e.g.

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

The implementation of `Set` 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, <http://www.swiss.ai.mit.edu/~adams/BB/>.

- J. Nievergelt and E.M. Reingold, "Binary search trees of bounded balance", SIAM journal of computing 2(1), March 1973.

Note that the implementation is *left-biased* – the elements of a first argument are always preferred to the second, for example in `union` or `insert`. Of course, left-biasing can only be observed when equality is an equivalence relation instead of structural equality.

## 8.1 Set type

```
data Set a
    =  Bin !Size !a !(Set a) !(Set a)
    |  Tip
    A set of values a.
```

```
instance Typeable1 Set
instance Foldable Set
instance Eq a => Eq (Set a)
instance (Data a, Ord a) => Data (Set a)
instance Ord a => Ord (Set a)
instance (Read a, Ord a) => Read (Set a)
instance Show a => Show (Set a)
instance NFData a => NFData (Set a)
instance Ord a => Monoid (Set a)
```

## 8.2 Operators

```
(\\) :: Ord a => Set a -> Set a -> Set a
       $O(n+m)$ . See difference.
```

## 8.3 Query

```
null :: Set a -> Bool
       $O(1)$ . Is this the empty set?
```

```
size :: Set a -> Int
       $O(1)$ . The number of elements in the set.
```



```
member :: Ord a => a -> Set a -> Bool
```

$O(\log n)$ . Is the element in the set?

```
notMember :: Ord a => a -> Set a -> Bool
```

$O(\log n)$ . Is the element not in the set?

```
lookupLT :: Ord a => a -> Set a -> Maybe a
```

$O(\log n)$ . Find largest element smaller than the given one.

```
lookupLT 3 (fromList [3, 5]) == Nothing
```

```
lookupLT 5 (fromList [3, 5]) == Just 3
```

```
lookupGT :: Ord a => a -> Set a -> Maybe a
```

$O(\log n)$ . Find smallest element greater than the given one.

```
lookupGT 4 (fromList [3, 5]) == Just 5
```

```
lookupGT 5 (fromList [3, 5]) == Nothing
```

```
lookupLE :: Ord a => a -> Set a -> Maybe a
```

$O(\log n)$ . Find largest element smaller or equal to the given one.

```
lookupLE 2 (fromList [3, 5]) == Nothing
```

```
lookupLE 4 (fromList [3, 5]) == Just 3
```

```
lookupLE 5 (fromList [3, 5]) == Just 5
```

```
lookupGE :: Ord a => a -> Set a -> Maybe a
```

$O(\log n)$ . Find smallest element greater or equal to the given one.

```
lookupGE 3 (fromList [3, 5]) == Just 3
```

```
lookupGE 4 (fromList [3, 5]) == Just 5
```

```
lookupGE 6 (fromList [3, 5]) == Nothing
```

```
isSubsetOf :: Ord a => Set a -> Set a -> Bool
```

$O(n+m)$ . Is this a subset? (`s1 isSubsetOf s2`) tells whether `s1` is a subset of `s2`.

```
isProperSubsetOf :: Ord a => Set a -> Set a -> Bool
```

$O(n+m)$ . Is this a proper subset? (ie. a subset but not equal).

## 8.4 Construction

`empty :: Set a`

$O(1)$ . The empty set.

`singleton :: a -> Set a`

$O(1)$ . Create a singleton set.

`insert :: Ord a => a -> Set a -> Set a`

$O(\log n)$ . Insert an element in a set. If the set already contains an element equal to the given value, it is replaced with the new value.

`delete :: Ord a => a -> Set a -> Set a`

$O(\log n)$ . Delete an element from a set.

## 8.5 Combine

`union :: Ord a => Set a -> Set a -> Set a`

$O(n+m)$ . The union of two sets, preferring the first set when equal elements are encountered. The implementation uses the efficient *hedge-union* algorithm. Hedge-union is more efficient on (bigset union smallset).

`unions :: Ord a => [Set a] -> Set a`

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

`difference :: Ord a => Set a -> Set a -> Set a`

$O(n+m)$ . Difference of two sets. The implementation uses an efficient *hedge* algorithm comparable with *hedge-union*.

`intersection :: Ord a => Set a -> Set a -> Set a`

$O(n+m)$ . The intersection of two sets. Elements of the result come from the first set, so for example

```
import qualified Data.Set as S
data AB = A | B deriving Show
instance Ord AB where compare _ _ = EQ
instance Eq AB where _ == _ = True
main = print (S.singleton A 'S.intersection' S.singleton B,
              S.singleton B 'S.intersection' S.singleton A)

prints (fromList [A],fromList [B]).
```

## 8.6 Filter

`filter :: (a -> Bool) -> Set a -> Set a`

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

`partition :: (a -> Bool) -> Set a -> (Set a, Set a)`

$O(n)$ . Partition the set into two sets, one with all elements that satisfy the predicate and one with all elements that don't satisfy the predicate. See also `split`.

`split :: Ord a => a -> Set a -> (Set a, Set a)`

$O(\log n)$ . The expression `(split x set)` is a pair `(set1,set2)` where `set1` comprises the elements of `set` less than `x` and `set2` comprises the elements of `set` greater than `x`.

`splitMember :: Ord a => a -> Set a -> (Set a, Bool, Set a)`

$O(\log n)$ . Performs a `split` but also returns whether the pivot element was found in the original set.

## 8.7 Map

`map :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b`

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

It's worth noting that the size of the result may be smaller if, for some  $(x,y)$ ,  $x \neq y$  &&  $f\ x == f\ y$

`mapMonotonic :: (a -> b) -> Set a -> Set b`

$O(n)$ . The

`mapMonotonic f s == map f s`, but works only when `f` is monotonic. *The precondition is not checked.* Semi-formally, we have:

```
and [x < y ==> f x < f y | x <- ls, y <- ls]
    ==> mapMonotonic f s == map f s
    where ls = toList s
```

## 8.8 Folds

`foldr :: (a -> b -> b) -> b -> Set a -> b`

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

For example,

```
toAscList set = foldr (:) [] set
```

`foldl :: (a -> b -> a) -> a -> Set b -> a`

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

For example,

```
toDescList set = foldl (flip (:)) [] set
```

### 8.8.1 Strict folds

`foldr' :: (a -> b -> b) -> b -> Set a -> b`

$O(n)$ . 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 -> Set b -> a`

$O(n)$ . 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.

### 8.8.2 Legacy folds

`fold :: (a -> b -> b) -> b -> Set a -> b`

$O(n)$ . Fold the elements in the set using the given right-associative binary operator. This function is an equivalent of `foldr` and is present for compatibility only.

*Please note that `fold` will be deprecated in the future and removed.*

## 8.9 Min/Max

`findMin :: Set a -> a`

$O(\log n)$ . The minimal element of a set.

`findMax :: Set a -> a`

$O(\log n)$ . The maximal element of a set.

`deleteMin :: Set a -> Set a`

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

`deleteMax :: Set a -> Set a`

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

`deleteFindMin :: Set a -> (a, Set a)`

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

`deleteFindMin set = (findMin set, deleteMin set)`

`deleteFindMax :: Set a -> (a, Set a)`

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

`deleteFindMax set = (findMax set, deleteMax set)`

`maxView :: Set a -> Maybe (a, Set a)`

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

`minView :: Set a -> Maybe (a, Set a)`

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

## 8.10 Conversion

### 8.10.1 List

`elems :: Set a -> [a]`

$O(n)$ . An alias of `toAscList`. The elements of a set in ascending order. Subject to list fusion.

```
toList :: Set a -> [a]
```

$O(n)$ . Convert the set to a list of elements. Subject to list fusion.

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

$O(n \log n)$ . Create a set from a list of elements.

### 8.10.2 Ordered list

```
toAscList :: Set a -> [a]
```

$O(n)$ . Convert the set to an ascending list of elements. Subject to list fusion.

```
toDescList :: Set a -> [a]
```

$O(n)$ . Convert the set to a descending list of elements. Subject to list fusion.

```
fromAscList :: Eq a => [a] -> Set a
```

$O(n)$ . Build a set from an ascending list in linear time. *The precondition (input list is ascending) is not checked.*

```
fromDistinctAscList :: [a] -> Set a
```

$O(n)$ . Build a set from an ascending list of distinct elements in linear time. *The precondition (input list is strictly ascending) is not checked.*

## 8.11 Debugging

```
showTree :: Show a => Set a -> String
```

$O(n)$ . Show the tree that implements the set. The tree is shown in a compressed, hanging format.

```
showTreeWith :: Show a => Bool -> Bool -> Set a -> String
```

$O(n)$ . The expression `(showTreeWith hang wide map)` shows the tree that implements the set. If `hang` is `True`, a *hanging* tree is shown otherwise a rotated tree is shown. If `wide` is `True`, an extra wide version is shown.

```
Set> putStrLn $ showTreeWith True False $ fromDistinctAscList [1..5]
4
+--2
```

```

|  +--1
|  +--3
+--5

```

```
Set> putStrLn $ showTreeWith True True $ fromDistinctAscList [1..5]
```

```

4
|
+--2
| |
|  +--1
|  |
|  +--3
|
+--5

```

```
Set> putStrLn $ showTreeWith False True $ fromDistinctAscList [1..5]
```

```

+--5
|
4
|
|  +--3
|  |
+--2
|
+--1

```

```
valid :: Ord a => Set a -> Bool
```

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

```
bin :: a -> Set a -> Set a -> Set a
```

```
balanced :: Set a -> Bool
```

```
join :: a -> Set a -> Set a -> Set a
```

```
merge :: Set a -> Set a -> Set a
```





## Chapter 9

### Data.StrictPair

---

```
module Data.StrictPair (  
    strictPair  
) where
```

---

```
strictPair :: a -> b -> (a, b)
```

Evaluate both argument to WHNF and create a pair of the result.



## Chapter 10

### Dpfs.DavisPutnamFiniteSets

---

```
module Dpfs.DavisPutnamFiniteSets (
  Atom(Xin, Calc), dpfs, dpfsAll, dpfsSat, dpfsSatAllSolutions,
  findTrueFalse, resolveTrueFalse, findSimilarAtomUnits,
  substituteSimilarUnits, findUnit', resolveUnit, findPureLiteral,
  resolvePureLiteral, findUnit', resolveUnitCalculus, findSimpRule3,
  resolveSimpRule3, findFalse_Calculus, resolveFalse_Calculus,
  findSimpUnitCalculus, resolveCalculus1, findSelection2, setSplit2,
  lookup', setToInt, dpcnf, expectJust, toList'
) where
```

---

Implementation of two extensions of the DPLL algorithm (Davis-Putnam-Logemann-Loveland)

First Extension:

Second Extension:

#### 10.0.1 Datatype

```
data Atom
  =  Xin !Int !IntSet
  |  Calc !IntSet !IntSet

instance Eq Atom
instance Show Atom
```

## 10.1 Davis-Putnam procedures

`dpfs :: FiniteSet -> [Char] -> [Char]`

`dpfs` function executes the following parts:

- parsing the formula
- cnf transformation
- dp procedure extension 1

`dpfsAll :: Int -> [Char] -> [(String, IntSet)]`

`dpfsAll` function executes the following parts:

- parsing the formula
- cnf transformation
- dp procedure extension 1 and tries to find all solutions

`dpfsSat :: CalculusSwitch`

`-> FiniteSet -> KlauselMenge -> SolutionSet`

`dpfsSat` function describes the procedure of the Davis-Putnam algorithm and returns one solutions if one exists.

`dpfsSatAllSolutions :: CalculusSwitch`

`-> FiniteSet -> KlauselMenge -> KlauselMenge`

`dpfsAllSolutions` function describes the procedure of the Davis-Putnam algorithm and returns all solutions if any exists.

## 10.2 Rules: 1st Extension

`findTrueFalse :: Int -> KlauselMenge -> Maybe Atom`

`findTrueFalse` runs through the clause set and returns an atom which can be simplify.

`resolveTrueFalse :: Int -> Atom -> KlauselMenge -> KlauselMenge`

`resolveTrueFalse` simplifies the clause set for the given atom.

`findSimilarAtomUnits :: Int -> KlauselMenge -> Maybe Atom`

The `findSimilarAtomUnits` function iterates through the clause set and returns the first unit a.

```
substituteSimilarUnits :: Atom -> KlauselMenge -> KlauselMenge
```

Add documentation here

```
findUnit' :: KlauselMenge -> Maybe AtomTupel
```

The `findUnit'` function return the first unit (a elem M') from the clause set if one exist.

```
resolveUnit :: AtomTupel -> KlauselMenge -> KlauselMenge
```

Add documentation here

```
findPureLiteral :: KlauselMenge -> Maybe AtomTupel
```

The `findPureLiteral` function returns the first pure literal.

```
resolvePureLiteral :: AtomTupel -> KlauselMenge -> KlauselMenge
```

The `resolvePureLiteral` builds up a new clause set while skipping clauses which contains the pure unit.

## 10.3 Rules: 2nd Extension

```
findUnit' :: KlauselMenge -> Maybe AtomTupel
```

The `findUnit'` function return the first unit (a elem M') from the clause set if one exist.

```
resolveUnitCalculus :: AtomTupel -> KlauselMenge -> KlauselMenge
```

The `resolveUnitCalculus` function removes the given atom from the clause set.

```
findSimpRule3 :: KlauselMenge
               -> SolutionSet -> Maybe ((Int, IntSet), Atom)
```

The `findSimpRule3` function runs through the clause set and returns an Atom if any simplifications are possible for rule 3.

```
resolveSimpRule3 :: ((Int, IntSet), Atom)
                 -> KlauselMenge -> SolutionSet -> SolutionSet
```

The `resolveSimpRule3` function updates the solutionSet for the given calc atom

```
findFalse_Calculus :: KlauselMenge -> Maybe Atom
```

The `findFalse_Calculus` find the first calc atom with one or both sets are empty.

```
resolveFalse_Calculus :: KlauselMenge -> KlauselMenge
```

The `resolveFalse_Calculus` function runs through the clause set and removes all atoms with one or both sets are empty.

```
findSimpUnitCalculus :: KlauselMenge
                    -> SolutionSet -> Maybe (Atom, SolutionSet)
```

The `findSimpUnitCalculus` function runs through the clause set and returns an Atom if any simplifications are possible. (Rule 1 + 2)

```
resolveCalculus1 :: Atom
                -> KlauselMenge -> SolutionSet -> KlauselMenge
```

The `resolveCalculus` function simplifies the clause set and returns the updated clause set (Rule 1 + 2)

```
findSelection2 :: KlauselMenge -> SolutionSet -> Maybe Atom
```

The `findSelection` function returns an calc atom, if and only if this calc atom is splitable into two pieces.

```
setSplit2 :: Atom -> KlauselMenge -> SolutionSet -> KlauselMenge
```

The `setSplit` function splits the passed calc atom into two or more pieces.

## 10.4 Utilities

```
lookup' :: Int -> [Atom] -> Maybe IntSet
```

The helper `lookup'` function looks up a set of one literal a

```
setToInt :: Int -> SolutionSet -> Int
```

```
dpcnf :: FiniteSet -> Pexpr (Int, [Int]) -> KlauselMenge
```

```
expectJust :: Maybe a -> a
```

```
toList' :: IntSet -> [Int]
```

# Chapter 11

## Dpfs.Parser

---

```
module Dpfs.Parser (
  Pexpr(Ptrue,
        Pfalse,
        Pvar,
        Pselem,
        PSelem,
        PvarElem,
        Pnot,
        Pand,
        Por,
        Pimpl,
        Pequiv),
  parseProp
) where
```

---

```
data Pexpr name
```

```
= Ptrue
| Pfalse
| Pvar name
| Pselem name
| PSelem [Pexpr name]
| PvarElem (Pexpr name, Pexpr name)
| Pnot (Pexpr name)
| Pand [Pexpr name]
| Por [Pexpr name]
| Pimpl (Pexpr name) (Pexpr name)
| Pequiv (Pexpr name) (Pexpr name)

instance Eq name => Eq (Pexpr name)
instance Ord name => Ord (Pexpr name)
instance Show name => Show (Pexpr name)

parseProp :: [Char] -> Pexpr String
```



# Chapter 12

## Dpfs.Simp

---

```
module Dpfs.Simp (
  cnf, cnffast, pelimequiv, pelimimpl, pelimnot, pelimTF, pelimTFR,
  deleteAll, pelimflat, pelimflatr, porsublst, pandsublst, portop,
  pandtop, cnfRemoveTaut, cnfRemoveTaut', cnfRemoveTaut'', numPropSubs,
  pVarNumber, pVarNumberList, pSelemsToList, pSelemsToList', showVar
) where
```

---

Substitute all occurrences of propositional variables to numbers. Transforms an propositional logic formula into an conjunctive normal form.

```
cnf :: Int -> Pexpr (Int, [Int]) -> Pexpr (Int, [Int])
    cnf transformiert eine aussagenlogische Formel in eine Konjunktive Normalform
```

```
cnffast :: Int -> Pexpr (Int, [Int]) -> Pexpr (Int, [Int])
    cnffast transformiert eine aussagenlogische Formel in eine Konjunktive Normalform (lineare CNF)
```

```
pelimequiv :: Pexpr (Int, [Int]) -> Pexpr (Int, [Int])
    pelimequiv ersetzt quivalenzen innerhalb der Formel.  $(a = b) \rightarrow ((a = b) \wedge (b = a))$ 
```

`pelimimpl :: Pexpr (Int, [Int]) -> Pexpr (Int, [Int])`

`pelimimpl` ersetzt Implikationen innerhalb der Formel.  $(a \Rightarrow b) \rightarrow (\neg a) \vee b$

`pelimnot :: Int -> Pexpr (Int, [Int]) -> Pexpr (Int, [Int])`

`pelimnot` ersetzt Vorkommen von  $(\neg \text{Ptrue})$  bzw.  $(\neg \text{Pfalse})$  zu  $\text{Pfalse}$  bzw.  $\text{Ptrue}$

`pelimTF :: Pexpr (Int, [Int]) -> Pexpr (Int, [Int])`

`pelimTF` lscht alle Vorkommen von  $\text{Ptrue}$  und  $\text{Pfalse}$  innerhalb der Formel.

`pelimTFR :: Eq name => Pexpr name -> Pexpr name`

`deleteAll :: Eq a => a -> [a] -> [a]`

`pelimflat :: Pexpr (Int, [Int]) -> Pexpr (Int, [Int])`

`pelimflat` fasst verschachtelte Und- und Oder-Verknüpfungen zusammen.

`pelimflatr :: Pexpr t -> Pexpr t`

`porsublst :: Pexpr t -> [Pexpr t]`

`pandsublst :: Pexpr t -> [Pexpr t]`

`portop :: Pexpr t -> Bool`

`pandtop :: Pexpr t -> Bool`

`cnfRemoveTaut :: Pexpr (Int, [Int]) -> Pexpr (Int, [Int])`

`cnfRemoveTaut` lscht Klauseln: – indem Atome als Menge genau die festgelegte Menge besitzen, und – indem ein Atom  $x$  mehrfach vorkommt und die Mengen dieser Atome disjunkt sind.

`cnfRemoveTaut' :: (Eq a1, Eq a) => [Pexpr (a, [a1])] -> [Pexpr (a, [a1])]`

`-> [Pexpr (a, [a1])]`

`cnfRemoveTaut'' :: (Eq a, Eq a1) => [Pexpr (a, [a1])] -> Bool`

`numPropSubs :: Int`

`-> Pexpr String -> [(String, Int)], Pexpr (Int, [Int])`

`numPropSubs` substituiert alle vorkommenden Variablen durch Int-Zahlen und zusätzlich wird der Typ `PvarElem` ersetzt durch `Pvar`. Das Ergebnis ist ein Tupel aus dem Variablen-Bindungen zu den substituierten Zahlen und der substituierte Liste selbst .

```

pVarNumber :: Int
    -> (Int, [(String, Int)])
    -> Pexpr String -> ((Int, [(String, Int)]), Pexpr (Int, [Int]))

```

pVarNumber substituiert alle vorkommenden Variablen durch Int-Zahlen und ersetzt den Typ PvarElem durch Pvar (Num,Liste der Elemente).

```

pVarNumberList :: Int                                -> (Int, [(String, Int)])                                -> ([Pexpr (Int, [Int],

```

```

pSelemsToList :: Pexpr String -> [Int]

```

```

pSelemsToList' :: [Pexpr String] -> [Int]

```

```

showVar :: Pexpr t -> t

```



## Chapter 13

### Dpfs.SimpCnf

---

```
module Dpfs.SimpCnf (  
    Fresh, fastcnfInt, pcnf  
    ) where
```

---

```
class Ord a => Fresh a
```

```
instance Fresh Char  
instance Fresh Int  
instance (Show a, Fresh a) => Fresh [a]  
instance Fresh (Int, [Int])
```

```
fastcnfInt :: (Ord a, Fresh a, Show a) => Pexpr a -> Pexpr a  
    Add documentation here
```

```
pcnf :: (Ord a, Show a) => Pexpr a -> Pexpr a
```