Libraries (1A)

Young Won Lim
6/5/17

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Based on

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

Using Libraries

```
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"))
```

https://downloads.haskell.org/~ghc/latest/docs/html/libraries/containers-0.5.7.1/Data-Map-Lazy.html

fromList (1)

fromList :: Eq key => (key -> Int32) -> [(key, val)] -> IO (HashTable key val)
base Data.HashTable
Convert a list of key/value pairs into a hash table. Equality on keys is taken from the Eq instance for the key type.

fromList :: [(Key, a)] -> IntMap a
containers Data.IntMap.Strict, containers Data.IntMap.Lazy
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")]

fromList :: [Key] -> IntSet
containers Data.IntSet
O(n*min(n,W)). Create a set from a list of integers.

fromList :: [a] -> Seq a
containers Data.Sequence
O(n). Create a sequence from a finite list of elements. There is a function toList in the opposite direction for all instances of the Foldable class, including Seq.

https://www.haskell.org/hoogle/?hoogle=fromList

fromList (2)

fromList :: Ord a => [a] -> Set a

containers Data.Set

O(n*log n). Create a set from a list of elements. If the elemens are ordered, linear-time implementation is used, with the performance equal to fromDistinctAscList.

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

containers Data.Map.Lazy, containers Data.Map.Strict

O(n*log n). Build a map from a list of key/value pairs. See also fromAscList. If the list contains more than one value for the same key, the last value for the key is retained. If the keys of the list are ordered, linear-time implementation is used, with the performance equal to fromDistinctAscList.

> 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")]

https://www.haskell.org/hoogle/?hoogle=fromList

lookup (1)

lookup :: Eq a => a -> [(a, b)] -> Maybe b
base Prelude, base Data.List
lookup key assocs looks up a key in an association list.

lookup :: HashTable key val -> key -> IO (Maybe val)
base Data.HashTable
Looks up the value of a key in the hash table.

lookup :: Key -> IntMap a -> Maybe a
containers Data.IntMap.Strict, containers Data.IntMap.Lazy
O(min(n,W)). Lookup the value at a key in the map. See also lookup.

lookup :: Ord k => k -> Map k a -> Maybe a
containers Data.Map.Lazy, containers Data.Map.Strict
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:

https://www.haskell.org/hoogle/?hoogle=fromList

lookup (2)

```
> 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
```

https://www.haskell.org/hoogle/?hoogle=fromList

elem

elem :: Eq a => a -> [a] -> Bool
base Prelude, base Data.List

elem is the list membership predicate,
usually written in infix form, e.g., x `elem` xs.
For the result to be False, the list must be finite;
True, however, results from an element equal to x found
at a finite index of a finite or infinite list.

```
1 `elem` [1, 2, 4] -- True
2 `elem` [1, 2, 4] -- True
3 `elem` [1, 2, 4] -- False
```

https://www.haskell.org/hoogle/?hoogle=fromList

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf