The Assignment

We have to implement a data type Set via a data type Heap that in turn is stored in an Array. A Set is a list of elements where the order and multiplicity of elements do not matter. Such a type can be perfectly implemented via a Heap, since Heap itself cannot contain duplicate elements. Heap will maintain its own internal order, though, but that is irrelevant with regards to Set.

The Implementation

The Heap Module (listing 1)

The Heap module is very large and implements quite a lot of functions for pretty much anything you'd want to do with a Heap. It uses an Array for the storage. Reading from an Array happens in `O(1)`, but I don't know how fast updating a value of the Array is. Assuming it is also `O(1)`, as arrays in procedural languages are, then most of the basic Heap functions can operate in `O(lg n)` and the advanced ones in `O(n lg n)`.

The type declaration `newtype Heap a = HeapC (Int, Int, (Array Int a))` is used as a record similar to `{heapSize, heapLength, heapArray}`. It is important to note that the size of the heap is not the same as the length of the heap and that the length is always greater than or equal to the size. The length is the maximum amount of elements that can be contained in the Heap, whereas size is the number of elements currently in the Heap. Length is not absolute, since `heapInsert` will double the length of the Heap if there is not enough room. Doubling the size will ensure that there's enough space for a while, so we don't have to create a new array for every insert. This is a common way to handle dynamically expanding storages.

Since `heapInsert` is the only way elements can be added to a Heap, it is also in charge of maintaining the Heap structure. `heapInsert` will skip insertion of existing elements, in the sense that such an operation is not an error, but simply doesn't change the Heap at all. It will check for existing elements via `heapFind`, which is implemented to only search through the left and right child trees so long as the element we want to insert is smaller than the roots of those trees. Searching through the Heap in this manner has runtime of `O(lg n)`.

In case the element we want to insert does not exist in the Heap, it will call `heapIncreaseKey` to insert the element at the end of the Heap and then move it upwards in the tree. `heapIncreaseKey` has a runtime of `O(lg n)`. This means that `heapInsert` runs in `O(2 lg n)`, or if we disregard constants also `O(lg n)`.

Likewise, `heapRemoveAt` is the only way to remove elements from a Heap (`heapRemove` is just a wrapper for `heapRemoveAt` and has the extra cost of doing a `heapFind`). In the case that we want to remove the last element, `heapRemoveAt` runs in `O(1)` since that's simply decrementing the size of the Heap by 1. In any other case we need to move the last element of the Heap up to replace the element we are removing, and then call `heapSort` on that sub-tree.

`heapSort` is completely by the book (and I mean that: *Introduction to Algorithms, 2nd Edition, Cormen et al., page 130*) so it should run in the standard time of `O(lg n)`. This means that `heapRemove` runs in `O(2 lg n)` and `heapRemoveAt` runs in `O(lg n)`.

The more advanced functions of `heapMerge`, `heapIntersect`, `heapDifference`, `heapEquals`, `heapContains`, `heapMap`, `heapFilter`, and `heapFold`, all share some traits in the implementation, namely that they peel off the last elements of the victim Heap one by one and operate on that element. This property alone adds `O(n)` to the runtime, so that these functions all run in `O(n lg n)`, except for `heapFold` which stays `O(n)`.

`heapLesserOrEq` and `heapShow` work in the opposite fashion; cutting off the largest elements and operating on those. Recursing in this manner has a base cost of `O(n lg n)`, which also happens to be the runtime of both functions.

The Set Module (listing 2)

There is very little actual code in the Set module, since most of the functionality is passed directly on to Heap. The only function that actually does something is `makeSet` that calls `makeSetHelper` which recurses over a List and inserts elements into a Heap one by one. This is done in the time `O(n lg n)`.

The Testing Phase (listing 3)

The burden of testing lies heavily on Heap, since Set is more or less just a wrapper for Heap. So tests 01 through 12 are for Heap, and 13 through 18 are for Set.

Tests 01 through 05 plus 10 and 13 are basic creation and merge tests, just to make sure the heap structure is maintained, which it is.

Test 06 finds the difference between {9,5,4,3} and {7,3,2,0}, which it correctly outputs as {9,5,4}.

Test 07 find the intersection between {9,5,4,3} and {7,3,2,0}, which it correctly outputs as {3}.

Tests 08 and 09 check whether {9,5,4,3} is respectively equal to, or lesser than or equal to, {8,7,6,1}. Both are False, as expected.

Test 11 folds {9,8,7,6,5,4,3,2,1,0} over `(+)`, and returns 45 as expected.

Test 12 filters {9,8,7,6,5,4,3,2,1,0} with `(even)` and returns {8,6,4,2,0} as expected.

Test 14 is particularly interesting since it shows the lexicographically correct sorting, in this case over a Set of Sets of Integers.

Test 15 takes the List [4,6,8,2,3,5,7] and turns it into the Set {8,7,6,5,4,3,2}. Test 16 is analogous.

Test 17 outputs the union of {8,7,6,5,4,3,2} and {9,8,7,3,2} as {9,8,7,6,5,4,3,2}, correctly.

Test 18 is just to prove that it works with types other than Integer, though test 14 probably also showed that.

Output from the test program

```
Main> sTest01
"{}"
Main> sTest02
"{9,5,4,3,}"
Main> sTest03
"{8,7,6,1,}"
Main> sTest04
"{7,3,2,0,}"
Main> sTest05
"{9,8,7,6,5,4,3,1,}"
Main> sTest06
"{9,5,4,}"
Main> sTest07
"{3,}"
Main> sTest08
"False"
Main> sTest09
"False"
Main> sTest10
"{9,8,7,6,5,4,3,2,1,0,}"
Main> sTest11
"45"
Main> sTest12
"{8,6,4,2,0,}"
Main> sTest13
"{}"
Main> sTest14
"{{6,5,3,},{5,},{4,3,2,1,},{4,3,2,},{4,2,},}"
Main> sTest15
"{8,7,6,5,4,3,2,}"
Main> sTest16
"{9,8,7,3,2,}"
Main> sTest17
"{9,8,7,6,5,4,3,2,}"
Main> sTest18
"{\"Def\",\"Cde\",\"Bcd\",\"Abc\",}"
```

Result: Works as expected.

The Conclusion

Provided my assumptions about Array are correct, then I have implemented Heap as searchable in `O(lg n)` as the assignment originally prescribed. If not, then it certainly can't be worse than using List's `O(n)` runtime. Also shifted most of the code from Set directly into Heap, making Heap very reusable. Set is a new type that provides a `makeSet` function, but otherwise Set is a Heap.

Source Listing 1: The Heap Module

```
module Heap
      (Heap, heapCreate, heapSize, heapLength, heapExchange,
      heapSort, heapMaximum, heapIncreaseKey, heapInsert,
      heapFind, heapRemove, heapRemoveAt, heapMerge,
      heapLastElement, heapIntersect, heapDifference, heapEquals,
      heapShow, heapMap, heapFilter, heapContains,
      heapLesserOrEq, heapFold, heapTrimToLength, heapDoubleLength) where

import Array

newtype Heap a = HeapC (Int, Int, (Array Int a))

instance (Eq a, Ord a) => Eq (Heap a) where
      (==) = heapEquals
instance Ord a => Ord (Heap a) where
      (<=) = heapLesserOrEq

-- A Heap starts with room for 10 elements, but it will dynamically
-- expand to fit any number of elements, courtesy heapInsert calling heapDoubleLength
heapCreate :: (Heap a)
heapCreate = HeapC (0, 10, (array (1,10) []))

heapSize :: (Heap a) -> Int
heapSize (HeapC (b,q,a)) = b

heapLength :: (Heap a) -> Int
heapLength (HeapC (a,q,b)) = q

heapArray :: (Heap a) -> (Array Int a)
heapArray (HeapC (a,q,b)) = b

heapExchange :: (Heap a) -> Int -> Int -> (Heap a)
heapExchange (HeapC (d,q,a)) b c = HeapC (d, q, a // [(b,a!c),(c,a!b)])

heapSort :: Ord a => (Heap a) -> Int -> (Heap a)
heapSort (HeapC (b,q,a)) i
      = let l = i*2 ; r = i*2+1 in
            let large = heapSortHelper l b (a!l) (a!i) i in
                  let largest = heapSortHelper r b (a!r) (a!large) large in
                        if largest /= i
                              then heapSort (heapExchange (HeapC (b,q,a)) i largest) largest
                              else (HeapC (b,q,a))

heapSortHelper :: Ord a => Int -> Int -> a -> a -> Int -> Int
heapSortHelper a b c d e
      | a <= b && c > d = a
      | otherwise             = e

{-
heapFromArray :: Ord a => Int -> (Array Int a) -> (Heap a)
heapFromArray a b
      | a==0            = heapCreate
      | otherwise =
-}

heapMaximum :: (Heap a) -> a
heapMaximum (HeapC (b,q,a)) = a!1

heapLastElement :: (Heap a) -> a
heapLastElement (HeapC (b,q,a)) = a!b

heapIncreaseKey :: Ord a => (Heap a) -> Int -> a -> (Heap a)
heapIncreaseKey (HeapC (b,q,a)) i k
      | k < a!i         = error "new key is smaller than current key"
      | otherwise       = heapIncreaseKeyHelper (HeapC (b,q,(a // [(i,k)]))) i

heapIncreaseKeyHelper :: Ord a => (Heap a) -> Int -> (Heap a)
heapIncreaseKeyHelper (HeapC (b,q,a)) i
      | i > 1 && a!(i `div` 2) < a!i
            = heapIncreaseKeyHelper (heapExchange (HeapC (b,q,a)) i (i `div` 2)) (i `div` 2)
      | otherwise                               = (HeapC (b,q,a))

-- This will set the array size to anything
heapSetLength :: (Heap a) -> Int -> (Heap a)
heapSetLength (HeapC (a,q,b)) z
```

```
      | z<1       = error "heap length must be >= 1"
      | z>=a           = (HeapC (a, z, array (1,z) (assocs b)))
      | otherwise = (HeapC (z, z, array (1,z) (assocs b)))

-- This will reduce the array size to the heap size
-- Unused, but could be useful for saving a Heap to I/O
heapTrimToLength :: (Heap a) -> (Heap a)
heapTrimToLength (HeapC (a,q,b)) = (HeapC (a, a+1, array (1,a+1) (assocs b)))

-- If we run out of space, double the size of the array.
-- This ensures that we don't have to increase the size that often, since
-- creating a new Array is probably not the cheapest operation.
heapDoubleLength :: (Heap a) -> (Heap a)
heapDoubleLength (HeapC (a,q,b)) = (HeapC (a, q*2, array (1,q*2) (assocs b)))

-- It's not an error to try to insert an existing element
heapInsert :: Ord a => (Heap a) -> a -> (Heap a)
heapInsert (HeapC (b,q,a)) k
      | b+1 > q                          = heapInsert (heapDoubleLength (HeapC (b,q,a))) k
      | (heapFind (HeapC (b,q,a)) k) == 0 = heapIncreaseKey (HeapC (b+1, q, (a // [(b+1, k)]))) (b+1) k
      | otherwise                        = (HeapC (b,q,a))

heapFind :: Ord a => (Heap a) -> a -> Int
heapFind (HeapC (b,q,a)) k = heapFindHelper (HeapC (b,q,a)) k 1

-- This performs lots of checks to make sure it doesn't ask for out of bounds elements
-- It is safe to search both child trees and + their result since an element can only occur once
-- in a heap, courtesy heapInsert using heapFind.
heapFindHelper :: Ord a => (Heap a) -> a -> Int -> Int
heapFindHelper (HeapC (b,q,a)) k i
      | (i <= b) && (k == a!i)                       = i
      | (i*2 <= b) && (i*2+1 <= b) && (k <= a!(i*2)) && (k <= a!(i*2+1))
          = (heapFindHelper (HeapC (b,q,a)) k (i*2)) + (heapFindHelper (HeapC (b,q,a)) k (i*2+1))
      | (i*2 <= b) && (k <= a!(i*2))           = heapFindHelper (HeapC (b,q,a)) k (i*2)
      | (i*2+1 <= b) && (k <= a!(i*2+1))  = heapFindHelper (HeapC (b,q,a)) k (i*2+1)
      | otherwise                                    = 0 -- dirty hack, but it works

heapRemove :: Ord a => (Heap a) -> a -> (Heap a)
heapRemove (HeapC (b,q,a)) k
      | heapFind (HeapC (b,q,a)) k == 0   = (HeapC (b,q,a))
      | otherwise       = heapRemoveAt (HeapC (b,q,a)) (heapFind (HeapC (b,q,a)) k)

heapRemoveAt :: Ord a => (Heap a) -> Int -> (Heap a)
heapRemoveAt (HeapC (b,q,a)) i
      | i==0            = error "cannot remove the 0th element"
      | i>b       = error "index to remove is out of bounds"
      | i==b            = (HeapC (b-1,q,a))
      | otherwise = heapSort (HeapC (b-1,q,(a // [(i,a!b)]))) i

heapMerge :: Ord a => (Heap a) -> (Heap a) -> (Heap a)
heapMerge a (HeapC (c,q,b))
      | c==0            = a
      | otherwise = let x = heapLastElement (HeapC (c,q,b)) in
                          let y = heapRemoveAt (HeapC (c,q,b)) c in
                                heapMerge (heapInsert a x) y

heapIntersect :: Ord a => (Heap a) -> (Heap a) -> (Heap a)
heapIntersect (HeapC (a,q,b)) (HeapC (c,w,d))
      | a==0 || c==0    = heapCreate
      | otherwise       = heapIntersectHelper heapCreate (HeapC (a,q,b)) (HeapC (c,w,d))

heapIntersectHelper :: Ord a => (Heap a) -> (Heap a) -> (Heap a) -> (Heap a)
heapIntersectHelper a (HeapC (b,q,c)) (HeapC (d,w,e))
      | b==0 || d==0    = a
      | otherwise = let x = heapLastElement (HeapC (b,q,c)) in
                    let y = heapRemoveAt (HeapC (b,q,c)) b in
                     let z = heapFind (HeapC (d,w,e)) x in
                       if z == 0
                        then heapIntersectHelper a y (HeapC (d,w,e))
                        else heapIntersectHelper (heapInsert a x) y (heapRemoveAt (HeapC (d,w,e)) z)

heapDifference :: Ord a => (Heap a) -> (Heap a) -> (Heap a)
heapDifference (HeapC (a,q,b)) (HeapC (c,w,d))
      | a==0 || c==0    = (HeapC (a,q,b))
      | otherwise       = let x = heapLastElement (HeapC (c,w,d)) in
                                let y = heapRemoveAt (HeapC (c,w,d)) c in
                                      let z = heapFind (HeapC (a,q,b)) x in
                                            if z==0
```

```
                                                then heapDifference (HeapC (a,q,b)) y
                                                else heapDifference (heapRemoveAt (HeapC (a,q,b)) z)
y

heapEquals :: Ord a => (Heap a) -> (Heap a) -> Bool
heapEquals (HeapC (a,q,b)) (HeapC (c,w,d))
      | a==0 && c==0    = True
      | a/=c                  = False
      | otherwise       = let x = heapLastElement (HeapC (c,w,d)) in
                                    let y = heapRemoveAt (HeapC (c,w,d)) c in
                                          let z = heapFind (HeapC (a,q,b)) x in
                                                if z==0
                                                      then False
                                                      else heapEquals (heapRemoveAt (HeapC (a,q,b)) z) y

-- This function is lexiographically correct
heapLesserOrEq :: Ord a => (Heap a) -> (Heap a) -> Bool
heapLesserOrEq (HeapC (a,q,b)) (HeapC (c,w,d))
      | a==0 && c==0    = True
      | a==0 && c/=0    = True -- empty heaps are always lesser than non-empty heaps
      | a/=0 && c==0    = False
      | b!1 > d!1       = False
      | b!1 < d!1       = True
      | otherwise       = heapLesserOrEq (heapRemoveAt (HeapC (a,q,b)) 1) (heapRemoveAt (HeapC (c,w,d)) 1)

heapContains :: Ord a => (Heap a) -> (Heap a) -> Bool
heapContains (HeapC (a,q,b)) (HeapC (c,w,d))
      | a==0 && c==0    = True
      | a/=0 && c==0    = True -- This will be the usual end-result
      | a==0 && c/=0    = False
      | otherwise       = let x = heapLastElement (HeapC (c,w,d)) in
                                    let y = heapRemoveAt (HeapC (c,w,d)) c in
                                          let z = heapFind (HeapC (a,q,b)) x in
                                                if z==0
                                                      then False
                                                      else heapContains (HeapC (a,q,b)) y

heapMap :: (Ord a, Ord b) => (a -> b) -> (Heap a) -> (Heap b)
heapMap f (HeapC (a,q,b)) = heapMapHelper f (HeapC (a,q,b)) heapCreate

heapMapHelper :: (Ord a, Ord b) => (a -> b) -> (Heap a) -> (Heap b) -> (Heap b)
heapMapHelper f (HeapC (a,q,b)) (HeapC (c,w,d))
      | a==0            = (HeapC (c,w,d))
      | otherwise = let x = heapLastElement (HeapC (a,q,b)) in
                              let y = heapRemoveAt (HeapC (a,q,b)) a in
                                    heapMapHelper f y (heapInsert (HeapC (c,w,d)) (f x))

heapFilter :: Ord a => (a -> Bool) -> (Heap a) -> (Heap a)
heapFilter f (HeapC (a,q,b)) = heapFilterHelper f (HeapC (a,q,b)) heapCreate

heapFilterHelper :: Ord a => (a -> Bool) -> (Heap a) -> (Heap a) -> (Heap a)
heapFilterHelper f (HeapC (a,q,b)) (HeapC (c,w,d))
      | a==0            = (HeapC (c,w,d))
      | otherwise = let x = heapLastElement (HeapC (a,q,b)) in
                              let y = heapRemoveAt (HeapC (a,q,b)) a in
                                    if (f x)
                                          then heapFilterHelper f y (heapInsert (HeapC (c,w,d)) x)
                                          else heapFilterHelper f y (HeapC (c,w,d))

heapFold :: Ord b => (a -> b -> a) -> a -> (Heap b) -> a
heapFold f a (HeapC (c,q,d))
      | c==0            = a
      | otherwise = let x = heapLastElement (HeapC (c,q,d)) in
                              let y = heapRemoveAt (HeapC (c,q,d)) c in
                                    heapFold f (f a x) y

heapShow :: Ord a => (a -> String) -> (Heap a) -> String
heapShow f (HeapC (a,q,b)) = "{" ++ (heapShowHelper f (HeapC (a,q,b)) "") ++ "}"

heapShowHelper :: Ord a => (a -> String) -> (Heap a) -> String -> String
heapShowHelper f (HeapC (a,q,b)) s
      | a==0            = s
      | otherwise = let x = heapMaximum (HeapC (a,q,b)) in
                              let y = heapRemoveAt (HeapC (a,q,b)) 1 in
                                    heapShowHelper f y (s ++ (f x) ++ ",")
```

Source Listing 2: The Set Module

```
module Set
      (Set, empty, sing, memSet,
      union, inter, diff, eqSet,
      subSet, makeSet, mapSet, filterSet,
      foldSet, showSet, card) where

import Heap

newtype Set a = Set (Heap a)

instance (Eq a, Ord a) => Eq (Set a) where
      (==) = eqSet
instance Ord a => Ord (Set a) where
      (<=) = leqSet

-- Creates a new empty Set
empty :: Set a
empty = Set heapCreate

-- Makes a Set out of a
sing :: Ord a => a -> Set a
sing x = Set (heapInsert heapCreate x)

-- Checks whether a is a member of the Set
memSet :: Ord a => Set a -> a -> Bool
memSet (Set a) b
      | heapFind a b == 0     = False
      | otherwise             = True

union :: Ord a => Set a -> Set a -> Set a
union (Set a) (Set b) = Set (heapMerge a b)

inter :: Ord a => Set a -> Set a -> Set a
inter (Set a) (Set b) = Set (heapIntersect a b)

diff :: Ord a => Set a -> Set a -> Set a
diff (Set a) (Set b) = Set (heapDifference a b)

eqSet :: (Eq a, Ord a) => Set a -> Set a -> Bool
eqSet (Set a) (Set b) = heapEquals a b

leqSet :: Ord a => Set a -> Set a -> Bool
leqSet (Set a) (Set b) = heapLesserOrEq a b

-- Yes, they are reversed on purpose.
-- This asks heapContains "does b contain a?"
subSet :: Ord a => Set a -> Set a -> Bool
subSet (Set a) (Set b) = heapContains b a

makeSet :: Ord a => [a] -> Set a
makeSet []  = empty
makeSet a   = makeSetHelper a empty

makeSetHelper :: Ord a => [a] -> Set a -> Set a
makeSetHelper [] b = b
makeSetHelper (x:xs) (Set b) = makeSetHelper xs (Set (heapInsert b x))

mapSet :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
mapSet f (Set a) = Set (heapMap f a)

filterSet :: Ord a => (a -> Bool) -> Set a -> Set a
filterSet f (Set a) = Set (heapFilter f a)

-- The type differs from the prescribed (a -> a -> a) -> a -> Set a -> a
-- but that's only a good thing. This function is more general, thus more useful.
-- It will work fine if a is same type as b anyways, but now also if they differ.
foldSet :: Ord b => (a -> b -> a) -> a -> Set b -> a
foldSet f a (Set b) = heapFold f a b

showSet :: Ord a => (a -> String) -> Set a -> String
showSet f (Set a) = heapShow f a

card :: Set a -> Int
card (Set a) = heapSize a
```

Source Listing 3: The Test Program

```
import Heap
import Set

-- Testing Heap
hTest01 = heapCreate :: Heap Int
hTest02 = heapInsert (heapInsert (heapInsert (heapInsert (hTest01) 3) 9) 4) 5
hTest03 = heapInsert (heapInsert (heapInsert (heapInsert (hTest01) 1) 6) 7) 8
hTest04 = heapInsert (heapInsert (heapInsert (heapInsert (hTest01) 3) 7) 2) 0
hTest05 = heapMerge hTest02 hTest03
hTest06 = heapDifference hTest02 hTest04
hTest07 = heapIntersect hTest02 hTest04
hTest08 = heapEquals hTest02 hTest03
hTest09 = heapLesserOrEq hTest02 hTest03
hTest10 = heapMerge hTest05 hTest04
hTest11 = heapFold (+) 0 hTest10
hTest12 = heapFilter (even) hTest10

sTest01 = heapShow show hTest01
sTest02 = heapShow show hTest02
sTest03 = heapShow show hTest03
sTest04 = heapShow show hTest04
sTest05 = heapShow show hTest05
sTest06 = heapShow show hTest06
sTest07 = heapShow show hTest07
sTest08 = show hTest08
sTest09 = show hTest09
sTest10 = heapShow show hTest10
sTest11 = show hTest11
sTest12 = heapShow show hTest12

-- Testing Set
hTest13 = empty :: Set Integer
hTest14 = makeSet
            [(makeSet [4,3,2,1]), (makeSet [4,3,2]),
            (makeSet [4,2]), (makeSet [5]), (makeSet [5,6,3]), (makeSet [2,4])]
hTest15 = makeSet [4,6,8,2,3,5,7]
hTest16 = makeSet [3,7,2,9,8,9]
hTest17 = union (union hTest13 hTest15) hTest16
hTest18 = makeSet ["Abc","Bcd","Cde","Def","Bcd"]

sTest13 = showSet show hTest13
sTest14 = showSet (showSet show) hTest14
sTest15 = showSet show hTest15
sTest16 = showSet show hTest16
sTest17 = showSet show hTest17
sTest18 = showSet show hTest18
```