50001 - Algorithm Analysis and Design - Lecture 17

Oliver Killane

18/12/21

Lecture Recording

Lecture recording is available here

Mutable Nub

This can be useful for the **nub** function (removes duplicates from a list) by using a bucket based hashmap of items from the list to determine duplication.

```haskell
import Control.Monad (when)
import Data.Array.ST
    ( getElems, newListArray, readArray, writeArray, STArray )
import Control.Monad.ST ( ST, runST )


-- immutable version:
nub :: Eq a => [a] -> [a]
nub = reverse . foldl nubHelper []
  where
    nubHelper :: Eq a => [a] -> a -> [a]
    nubHelper ns c
      | c `elem` ns = ns
      | otherwise   = c:ns

-- mutable version, create a hash table of characters to track which have
-- already been seen.
nubMut :: (Hashable a, Eq a) => [a] -> [a]
nubMut xs = concat $ runST $ do
  axss <- newListArray (0, n - 1) (replicate 256 []) :: ST s (STArray s Int [a])
  sequence [do {
    let hx = hash x `mod` (n - 1)
    ys <- readArray axss hx
    unless (x `elem` ys) $ do writeArray axss hx (x : ys)}
  | x <- xs]
  getElems axss
  where
    -- number of buckets in the hash table
    n = 256
```

Quicksort

We can also implement quicksort, which can be done in place in an array (saved memory and time as accesses are constant time).

By using mutable data structures we can swap elements when reordering by reading and writing from the array.

```haskell
import Data.Array.ST ( readArray, writeArray, STArray, getElems )
import Control.Monad.ST ( ST )

-- swap elements over (classic store in temp & swap over other before writeback)
swap :: STArray s Int a -> Int -> Int -> ST s ()
swap axs i j = do
  temp <- readArray axs i
```

```
  readArray axs j >>= writeArray axs i
  writeArray axs j temp

qsort :: Ord a => [a] -> [a]
qsort xs = runST $ do
  axs <- newListArray (0,n) xs
  aqsort axs 0 n
  getElems axs
  where
    n = length xs - 1

{-
Partition around a pivot (k) (all smaller to left, larger to right
(unsorted)), then recur on these partitions
Index:      0       (k-1) k (k+1)     n
Contents: [ ..<= x .. ][x][ .. >x ..]
-}
aqsort :: Ord a => STArray s Int a -> Int -> Int -> ST s ()
aqsort axs i j
  | i >= j = return ()
  | otherwise = do
      k <- apartition axs i j
      aqsort axs i (k - 1)
      aqsort axs (k + 1) j

apartition :: Ord a => STArray s Int a -> Int -> Int -> ST s Int
apartition asx p q = do
  x <- readArray axs p
  let loop i j
    | i > j = do
      swap axs p j
      return j
    | otherwise = do
      u <- readArray axs i
      if u < x
        then do loop (i + 1) j
        else do
          swap axs i j
          loop i (j - 1)
  loop (p+1) q
```