CS314. Функциональное программирование

Лекция 8. Основные структуры данных

В. Н. Брагилевский

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

7 октября 2016 г.

Содержание

Содержание

Содержание

Список (Data.List)

Определение

Список — это набор элементов (конечный или бесконечный) с быстрым доступом к началу ($O(1)$) и линейной сложностью большинства операций.

Основные приёмы использования

- Функции высшего порядка.
- Генераторы списков.

```
[ x2 + y2 | x <- xs, y <- ys,
            let x2 =  x^2, let y2 = y^2,
            x2 `mod` y2 == 0 ]
```

Data.ByteString — разновидность списков для двоичных данных.

Содержание

Последовательность (Data.Sequence)

Определение

Последовательность — это конечный проиндексированный набор элементов с быстрым доступом к началу и концу ($O(1)$) и логарифмической амортизированной сложностью многих операций.

Типы операций

- Конструирование.
- Обращение к содержимому.
- Преобразование.

Последовательности обобщают очереди и деки (двусторонние очереди). В основе реализации «2-3 finger trees», аннотированные размерами.

Конструирование и простые запросы

```
empty :: Seq a                    -- O(1)
singleton :: a -> Seq a           -- O(1)
replicate :: Int -> a -> Seq a    -- O(log n)

(<|) :: a -> Seq a -> Seq a       -- O(1)
(|>) :: Seq a -> a -> Seq a       -- O(1)

(><) :: Seq a -> Seq a -> Seq a -- O(log(min(n1, n2)))

fromList :: [a] -> Seq a          -- O(n)
```

```
null :: Seq a -> Bool  -- O(1)
length :: Seq a -> Int -- O(1)
```

Пример конструирования и обращение к концам

```
ghci> let s = (4 <| 5 <| singleton 6) >< (empty |> 7 |> 8)
ghci> s
fromList [4,5,6,7,8]
ghci> let (x :< xs) = viewl s
ghci> x
4
ghci> xs
fromList [5,6,7,8]
ghci> let (ys :> y) = viewr s
ghci> y
8
ghci> ys
fromList [4,5,6,7]
```

Обращение к концам: типы

```haskell
data ViewL a = EmptyL | a :< Seq a
data ViewR a = EmptyR | Seq a :> a

viewl :: Seq a -> ViewL a     -- O(1)
viewr :: Seq a -> ViewR a     -- O(1)
```

Пример

```haskell
endsWith :: Eq a => Seq a -> a -> Bool
endsWith s x = let (_ :> x') = viewr s
               in x == x'

sameEnds :: Eq a => Seq a -> Bool
sameEnds s = l == r
  where
    (l :< _) = viewl s
    (_ :> r) = viewr s
```

```
ghci> endsWith (1 <| 2 <| 3 <| empty) 3
True
ghci> sameEnds (1 <| 2 <| 3 <| empty)
False
```

Расширение ViewPatterns

```
endsWith :: Eq a => Seq a -> a -> Bool
endsWith s x = let (_ :> x') = viewr s
               in x == x'
```

```
{-# LANGUAGE ViewPatterns #-}

endsWith' (viewr -> _ :> x') x = x == x'
```

- Вместо сопоставления параметра функции с образцом мы вызываем для него функцию и смотрим уже на результат.

Пример для списков

```
eitherEndIsZero :: [Int] -> Bool
eitherEndIsZero (head -> 0) = True
eitherEndIsZero (last -> 0) = True
eitherEndIsZero          _  = False
```

Разные операции с последовательностями

Операции с индексами (логарифмическая сложность)

```
index :: Seq a -> Int -> a
adjust :: (a -> a) -> Int -> Seq a -> Seq a
update :: Int -> a -> Seq a -> Seq a
take :: Int -> Seq a -> Seq a
drop :: Int -> Seq a -> Seq a
splitAt :: Int -> Seq a -> (Seq a, Seq a)
```

Обращение последовательности ($O(n)$)

```
reverse :: Seq a -> Seq a
```

Имеются также операции, аналогичные операциям со списками.

Содержание

Массивы (Data.Array.IArray)

Определение

Массив — тип данных, позволяющий хранить фиксированный набор данных некоторого типа и обеспечивающий эффективный доступ к своим элементам по индексу. Индексом элемента массива может быть значение типа, принадлежащего классу типов Ix (например, Int, Bool, Char, кортежи с типами из Ix и пр.).

Основные операции

- Заполнение (монолитное и инкрементальное).
- Доступ к элементам (индексация).
- Преобразование (amap, ixmap).

Формирование массива из списка (индекс, значение)

```
array    :: (IArray a e, Ix i)
         => (i,i)          -- диапазон индексов
         -> [(i, e)]       -- список ( индекс, значение )
         -> a i e
```

```
import Data.Array.IArray

squares :: Array Int Int
squares = array (1,100) [(i, i*i) | i <- [1..100]]

element42 = squares ! 42
```

- Порядок индексов не важен, но повторения недопустимы!

Пример: массив чисел Фибоначчи

```
fibs :: Int -> Array Int Int
fibs n = a
  where a = array (0,n) ([(0, 1), (1, 1)] ++
                         [(i, a!(i-2) + a!(i-1)) | i <- [2..n]])
```

- Обращение к заполненным ранее элементам.

Формирование массива из списка значений

```
listArray :: (IArray a e, Ix i) => (i, i) -> [e] -> a i e
```

```
names :: Array Int String
names = listArray (1,16)
      ["анна", "юлия", "ольга", "мария", "александра",
       "дарья", "ирина", "наталья", "оксана", "галина",
       "виктория", "любовь", "вера", "алиса", "алла",
       "татьяна"]
```

Формирование массива с накоплением значений

```
accumArray :: (IArray a e, Ix i)
           => (e -> e' -> e)      -- Аккумулирующая функция
           -> e                   -- Элемент по умолчанию
           -> (i,i)               -- Диапазон индексов
           -> [(i, e')]           -- Список (индекс,значение)
           -> a i e
```

```
arr :: Array Int Int
arr = accumArray (+) 0 (1,3) [(1, 4),(2, 4),(1, 5),(3,5)]
```

```
ghci> arr
array (1,3) [(1,9),(2,4),(3,5)]
```

- Порядок индексов не важен, всякий раз запускается аккумулирующая функция (при первом упоминании индекса со значением по умолчанию в качестве первого аргумента).

Пример: индексирование в обратном порядке

```
import Data.Array.IArray

adds = replicate 100001 42

adds' :: Array Int Int
adds' = listArray (0, length adds-1) adds

answer = sum $ map (adds !!) [100000,99999 .. 0]

answer' = sum $ map (adds' !) [100000,99999 .. 0]
```

Испытания

```
ghci> answer
4200042
(19.95 secs, 35914784 bytes)
ghci> answer'
4200042
(0.06 secs, 51187840 bytes)
```

Пример: матрицы и сумма матриц

```
m1, m2 :: Array (Int,Int) Double
m1 = listArray ((1,1),(2,3)) [1,2,3,4,5,6]
m2 = listArray ((1,1),(2,3)) [1,2,3,4,5,6]

matSum :: Array (Int,Int) Double
          -> Array (Int,Int) Double
          -> Array (Int,Int) Double
matSum x y = array resultBounds
                   [ (i, x!i + y!i) | r <- range(lr,ur),
                     c <- range(lc,uc), let i = (r, c)]
  where
    bx@((lr,lc),(ur,uc)) = bounds x
    resultBounds
        | bx == bounds y = bx
        | otherwise = error "matSum: incompatible bounds"
```

- bounds — диапазон индексов; range — список значений из диапазона.

Преобразование массивов

```
ixmap :: (IArray a e, Ix i, Ix j) =>
                (i,i) -> (i->j) -> a j e -> a i e

amap :: (IArray a e', IArray a e, Ix i) =>
                (e' -> e) -> a i e' -> a i e
```

```
n = 100

xs :: Array Int Double
xs = listArray (1, n) $ take n $ iterate (+0.1) 0

xs_rev = ixmap (bounds xs) (\i -> n + 1 - i) xs

sines = amap (\x -> (x, sin x)) xs
```

Содержание

Содержание

Множество

Определение

Множество — тип данных, поддерживающий эффективные операции вставки, удаления и проверки принадлежности для данных некоторого типа.

- Модуль Data.Set.
- Реализация — сбалансированные бинарные деревья поиска.
- Ограничение для элементов — класс типов Ord.

Импорт модуля Data.Set и создание множества

Импорт модуля

```
import qualified Data.Set as Set
```

Создание множества

```
empty :: Set a
singleton :: a -> Set a
fromList :: Ord a => [a] -> Set a
fromAscList :: Eq a => [a] -> Set a
fromDistinctAscList :: [a] -> Set a
```

- Предусловия не проверяются!

Основные операции с множествами

```
null :: Set a -> Bool
size :: Set a -> Int
member :: Ord a => a -> Set a -> Bool
notMember :: Ord a => a -> Set a -> Bool
isSubsetOf :: Ord a => Set a -> Set a -> Bool
isProperSubsetOf :: Ord a => Set a -> Set a -> Bool
insert :: Ord a => a -> Set a -> Set a
delete :: Ord a => a -> Set a -> Set a
union :: Ord a => Set a -> Set a -> Set a
unions :: Ord a => [Set a] -> Set a
difference :: Ord a => Set a -> Set a -> Set a
intersection :: Ord a => Set a -> Set a -> Set a
```

Обработка элементов множества

```
filter :: (a -> Bool) -> Set a -> Set a
partition :: (a -> Bool) -> Set a -> (Set a, Set a)
split :: Ord a => a -> Set a -> (Set a, Set a)
splitMember :: Ord a => a -> Set a -> (Set a, Bool, Set a)
map :: Ord b => (a -> b) -> Set a -> Set b
foldr :: (a -> b -> b) -> b -> Set a -> b
foldl :: (a -> b -> a) -> a -> Set b -> a
```

Другие возможности

Множество как очередь с приоритетом

```
findMin :: Set a -> a
findMax :: Set a -> a
deleteMin :: Set a -> Set a
deleteMax :: Set a -> Set a
deleteFindMin :: Set a -> (a, Set a)
deleteFindMax :: Set a -> (a, Set a)
```

```
elems :: Set a -> [a]
toAscList :: Set a -> [a]
toDescList :: Set a -> [a]
```

Пример: проверка принадлежности множеству

Задача

Определить, сколько слов из данного набора являются женскими именами.

```
import qualified Data.Set as Set

female_names = ["анна", "юлия", "ольга", "мария", "дарья",
       "александра", "ирина", "наталья", "оксана", "галина",
       "виктория", "любовь", "вера", "алиса", "татьяна"]
some_words = take 10000000
                  $ cycle ["анна", "лето", "двор", "вера"]
answer = length $ filter (`elem` female_names) some_words
answer' = length $ filter (`Set.member` female_names')
                          some_words
  where
    female_names' = Set.fromList female_names
```

Испытания

```
ghci> length some_words
10000000
(0.28 secs, 561475440 bytes)
ghci> answer
5000000
(3.86 secs, 280926888 bytes)
ghci> answer'
5000000
(2.12 secs, 520981296 bytes)
```

Множество целых чисел

```
import qualified Data.IntSet as IntSet
```

```
ghci> let primes = IntSet.fromAscList [2,3,5,7,11,13]
ghci> 5 `IntSet.member` primes
True
ghci> 4 `IntSet.member` primes
False
```

Содержание

Отображение

Определение

Отображение (словарь) — тип данных, позволяющий хранить пары вида (ключ,значение) и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

Модуль Data.Map — отображения

```
import qualified Data.Map as Map

type Name = String
type PhoneNumber = String

phoneBook :: Map.Map Name PhoneNumber
phoneBook = Map.fromList
    [("оля","555-29-38")
    ,("женя","452-29-28")
    ,("катя","493-29-28")
    ,("маша","205-29-28")
    ,("надя","939-82-82")
    ,("юля","853-24-92")
    ]
```

Поиск записи по ключу

```
ghci> :t Map.lookup
Map.lookup :: (Ord k) => k -> Map.Map k a -> Maybe a
ghci> Map.lookup "оля" phoneBook
Just "555-29-38"
ghci> Map.lookup "надя" phoneBook
Just "939-82-82"
ghci> Map.lookup "таня" phoneBook
Nothing
```

Добавление записи и определение размера

```
ghci> :t Map.insert
Map.insert :: (Ord k) => k -> a -> Map.Map k a -> Map.Map k a
ghci> Map.lookup "таня" phoneBook
Nothing
ghci> let newBook = Map.insert "таня" "341-90-21" phoneBook
ghci> Map.lookup "таня" newBook
Just "341-90-21"
```

```
ghci> :t Map.size
Map.size :: Map.Map k a -> Int
ghci> Map.size phoneBook
6
ghci> Map.size newBook
7
```

Некоторые функции из модуля Data.Map

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

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

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

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

elems :: Map k a -> [a]

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

Некоторые функции из модуля Data.Map (2)

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

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

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

Разновидности отображений

- Data.Map.Lazy и Data.Map.Strict
- Data.IntMap.Lazy и Data.IntMap.Strict

Содержание

Типы данных

```
import qualified Data.Map as Map

data LockerState = Taken | Free deriving (Show, Eq)

type Code = String

type LockerMap = Map.Map Int (LockerState, Code)
```

Поиск шкафчика

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber lockers =
   case Map.lookup lockerNumber lockers of
     Nothing -> Left $ "Шкафчик #" ++ show lockerNumber ++
                       " не существует"
     Just (Taken, _ ) -> Left $ "Шкафчик #" ++
                 show lockerNumber ++ " уже занят"
     Just (Free, code) -> Right code
```

Испытания

```
lockers :: LockerMap
lockers = Map.fromList
  [(100,(Taken,"ZD39I"))
   ,(101,(Free,"JAH3I"))
   ,(103,(Free,"IQSA9"))
   ,(105,(Free,"QOTSA"))
   ,(109,(Taken,"893JJ"))
   ,(110,(Taken,"99292"))
   ]
```

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Шкафчик #100 уже занят"
ghci> lockerLookup 102 lockers
Left "Шкафчик #102 не существует"
ghci> lockerLookup 110 lockers
Left "Шкафчик #110 уже занят"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```

Содержание

Другие виды контейнеров

Упорядоченные контейнеры

- Data.Tree (rose tree)
- Data.Graph

Неупорядоченные контейнеры

- Data.HashSet
- Data.HashMap

Для элементов множества и ключей отображения требуется экземпляр класса типов Data.Hashable.