# Лабораторная работа №6. Дисциплина «Основы функционального программирования»

Евгений Хандыго, гр. 53501/3

13 декабря 2015 г.

## 1 Структура деки

Для организации деки в Haskell используем тот же прием, что и при построении очереди. То есть будем использовать два списка, один из которых представляет переднюю часть деки (front), а другой — заднюю часть деки (back). При описании очереди для удобства и повышения эффективности операций используют инвариант, постулирующий, что перед очереди (откуда вынимаются элементы) может быть пуст тогда и только тогда, когда сама очередь пуста. При выполнении любых операций над очередью таким образом необходимо поддерживать данный инвариант путем переконструирования очереди в случае его нарушения. В случае с реализацией деки целесообразным представляется использование инварианта, который гарантировал бы примерно одинаковое распределение элементов по ее частям. В качестве такого инварианта может быть выбран следующий:

$$|F| \leq c|B| + 1 \text{ и } |B| \leq c|F| + 1,$$

где $F$ и $B$ соответственно передняя и задняя часть деки и $c \in \mathbb{R}$, $c > 1$. Как и в случае с очередью данный инвариант необходимо проверять и восстанавливать в случае необходимости после каждой операции помещения/извлечения элемента из/в очередь. Реализация деки в соответствии с представленным описанием приведена в листинге 1.1.

```haskell
data Deque a = Deque Int [a] Int [a] deriving (Show)

reverseTail :: [a] -> Int -> [a]
reverseTail xs n = reverse $ drop n xs

makeDeque :: Int -> [a] -> Int -> [a] -> Deque a
```

```haskell
makeDeque lx xs ly ys
    | ly > 3 * lx + 1   =
    let n = div (lx + ly) 2
      in let
        lx' = (lx + ly - n)
        xs' = xs ++ (reverseTail ys n)
        ys' = take n ys
        in Deque lx' xs' n ys'
    | lx > 3 * ly + 1   =
    let n = div (lx + ly) 2
      in let
        xs' = take n xs
        ly' = (lx + ly - n)
        ys' = ys ++ (reverseTail xs n)
        in Deque n xs' ly' ys'
  | otherwise      = Deque lx xs ly ys

emptyDeque :: Deque a
emptyDeque = Deque 0 [] 0 []

popFront :: Deque a -> (Maybe a, Deque a)
popFront d@(Deque 0 [] 0 [])        = (Nothing, d)
popFront (Deque 0 [] 1 [y])         = (Just y, (Deque 0 [] 0 []))
popFront d@(Deque lx (hx:tx) ly ys) = (Just hx, makeDeque (lx - 1) tx ly ys)

popBack :: Deque a -> (Maybe a, Deque a)
popBack d@(Deque 0 [] 0 [])        = (Nothing, d)
popBack (Deque 1 [x] 0 [])         = (Just x, (Deque 0 [] 0 []))
popBack d@(Deque lx xs ly (hy:ty)) = (Just hy, makeDeque lx xs (ly - 1) ty)

pushFront :: Deque a -> a -> Deque a
pushFront (Deque lx xs ly ys) x = makeDeque (lx + 1) (x:xs) ly ys

pushBack :: Deque a -> a -> Deque a
pushBack (Deque lx xs ly ys) y = makeDeque lx xs (ly + 1) (y:ys)
```

Listing 1.1: Реализация деки

## 2 Оценка амортизационной сложности методом банкира

Пусть в результате операций над декой необходимо провести ее балансировку. При этом длины списков начала и конца деки равны соответственно $l_1$ и $l_2$. Пусть также $l_1 > l_2$ и после предыдущей балансировки не были потрачены все накопленные к тому моменту монеты. Таким образом, для достижения текущего состояния было гарантировано выполнено не менее $l_1 - l_2$ операций вставки и/или удалений, каждая из которых приносит некоторое фиксированной количество монет $m$. Покажем, что при достаточно большом $m$ монет, по-

лученных в ходе преобразований деки от момента предыдущей балансировки, хватит для того, чтобы расчитытаться за необходимую на текщем этапе балансировку. Для того, чтобы восстановить инвариант необходимо произвести следующие шаги:

1. Отсчитать $\left\lfloor \frac{l_1+l_2}{2} \right\rfloor$ элементов от $F$, что потребует $O(\left\lfloor \frac{l_1+l_2}{2} \right\rfloor)$ операций.
2. Инвертировать порядок следования остатка $F$, что потребует $O(l_1 - \left\lfloor \frac{l_1+l_2}{2} \right\rfloor)$ операций.
3. Присоеденить инвертированный остаток $F$ к выходу деки, что потребует $O(l_2)$ операций.

Таким образом, за балансировку необходимо заплатить $l_1 + l_2$ монет. При этом имеется $m(l_1 - l_2)$ монет. Рассмотрим выражение

$$ml_1 - ml_2 - l_1 - l_2 = (m-1)l_1 - (m+1)l_2.$$

При этом из необходимости выполнения операции балансировки следует, что

$$l_1 > cl_2 + 1.$$

То есть для того, чтобы $(m-1)l_1 - (m+1)l_2$ было больше нуля необходимо, чтобы $\frac{m+1}{m-1} \leq c$, поскольку $l_1$ и $l_2$ неотрицательны. Таким образом, при правильном выборе $m$ можно считать, что при проведении операции балансировки всегда накопится необходимое количество монет при условии, что после предыдущей операции балансировки не было перерасхода имеющихся средств. Заметим, однако, что при инициализации деки она уже находится в сбалансированном состоянии и в распоряжении имеется 0 монет, а поскольку всякая операция балансировки не уводит нас в минус, можно считать, что монет всегда достаточно для восстановления инварианта. Таким образом, можно сделать вывод, что амортизационная сложность всех операций над декой является константной в рассмотренном случае $l_1 > l_2$. Заметим, однако, что случай $l_2 > l_2$ совершенно аналогичен рассмотенному поскольку очереди конца и начала деки симметричны, то есть в рамках доказательства могут быть заменены друг другом. В приведенной реализации параметр $c$ полагается равным трем. Для повторения приведенного доказательства в данном случае можно взять $m = 2$.

# 3 Оценка амортизационной сложности методом физика

За потенциал деки возьмем ее длину. Тогда

- Добавление элемента в начало/конец деки без балансировки увеличвает потенциал на 1 и занимет $O(1)$.
- Удаление элемента с начала/конца деки без балансировки уменьшает потенциал на 1 и занимает $O(1)$.
- Балансировка деки не изменяет потенциал и занимает $O(l)$, где $l$ — это длина деки.

Таким образом, можно сделать вывод, что амортизационная сложность всех операций над декой является константной.