

# Persistent data structures

---

Михаил Беляев

November 23, 2016

## Mutable vs Immutable

- Изменяемые структуры данных: скрытое изменяемое состояние
  - Позволяют очень быстрые алгоритмы
  - Просты (?)
- Неизменяемые структуры данных: никаких внутренних изменений структуры, каждое изменение создаёт новый экземпляр
  - Меньше проблем в нахождении ошибок
  - Лучшая работа в параллельном окружении
  - Сложность реализации

В чисто функциональной программе все структуры данных неизменяемые.

# Использование структуры данных

- Эфемерное
  - Стандартное использование — изменение на месте либо передача в функции
- Персистентное
  - Персистентность — способность состояния «пережить своего создателя»
  - В применении к структурам данных — возможность структуры данных существовать в нескольких состояниях одновременно

Зачем персистентное использование может быть нужно?

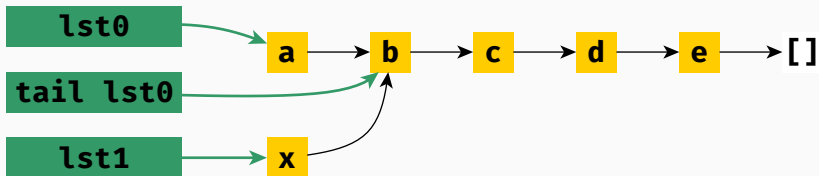
# Реализация персистентного использования на практике

- Как использовать персистентно изменяемую структуру данных?
- Как использовать персистентно неизменяемую структуру данных?

## Пример персистентной структуры данных — список

```
lst0 = [a,b,c,d,e]
```

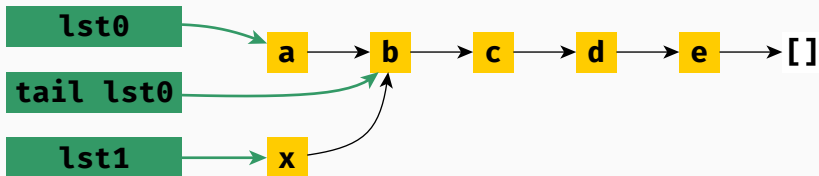
```
lst1 = x:(tail lst0)
```



## Пример персистентной структуры данных — список

```
lst0 = [a,b,c,d,e]
```

```
lst1 = x:(tail lst0)
```



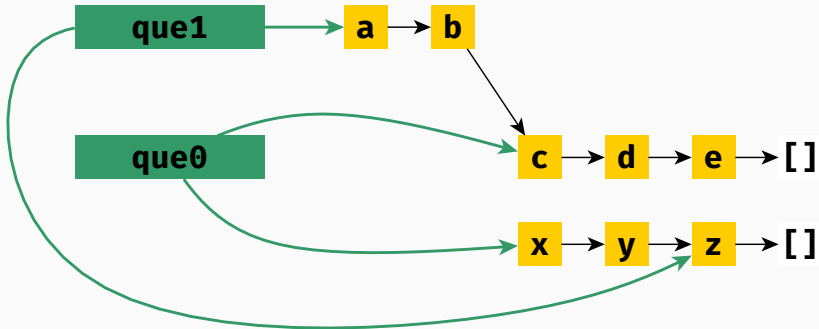
Можете ли вы привести пример операции, при которой всё будет не так радужно?

## Очередь с предыдущей лекции

`discard q = q' where (q', _) = dequeue q`

`que0 = Queue ...`

`que1 = discard $ discard $ que0 'enqueue' b 'enqueue' a`

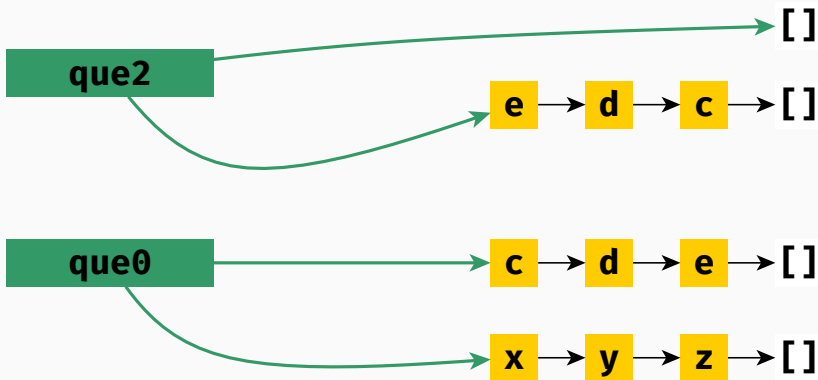


## Очередь с предыдущей лекции

```
discard q = q' where (q', _) = dequeue q
```

```
que0 = Queue ...
```

```
que1 = discard $ discard $ discard $ que0
```





- Типичное использование связанного списка в императивном программировании — вставка/удаление множества элементов в какую-то точку в середине
- В ФП у вас такой возможности нет
- Сделать это эффективно, не меняя структуру данных (список) невозможно

# Zipper: идея



# Zipper: реализация

```
data Zipper a = Zipper [a] [a]
```

```
fromList lst = Zipper [] lst
```

```
goRight z@(Zipper _ []) = z
```

```
goRight (Zipper l (rh:rt)) = Zipper (rh:l) rt
```

```
goLeft z@(Zipper [] _) = z
```

```
goLeft (Zipper (lh:lt) r) = Zipper lt (lh:r)
```

```
putRight x (Zipper l r) = Zipper l (x:r)
```

```
putLeft x (Zipper l r) = Zipper (x:l) r
```

```
removeRight (Zipper l (_:rt)) = Zipper l rt
```

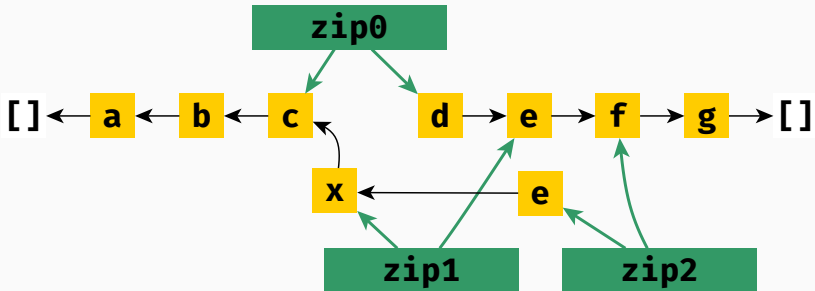
```
removeLeft (Zipper (_:lt) r) = Zipper lt r
```

## Zipper: характеристики

- Можно показать, что все операции, которые имели сложность  $O(1)$  у обычного списка, имеют амортизированную сложность  $O(1)$  у zipperа
- Zipper это персистентная структура данных
- Zipper можно определить для очень многих структур данных

# Персистентный zipper

```
zip0 = ...  
zip1 = removeRight $ putLeft x zip0  
zip2 = goRight zip1
```

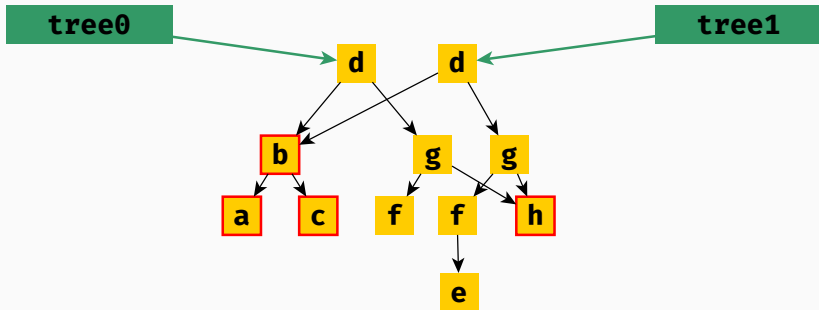


Что такое бинарное дерево поиска?

Как они работают на практике?

# Бинарные деревья поиска

- Пересоздаются только вершины по пути к изменяемому элементу (если иного не требует протокол балансировки)



# Красно-чёрное дерево

- Одно из самых известных самобалансирующихся деревьев поиска
- Вершины имеют два цвета, листья всегда чёрные
- Инварианты:
  - Красная вершина не может иметь красных детей
  - Все пути от корня до листа содержат одинаковое количество чёрных вершин



# Красно-чёрное дерево: реализация

```
data Color = R | B
data RBTree a = E | RBTree Color (RBTree a) a (RBTree a)
member x E = False
member x (RBTree _ l v r) | x == v      = True
member x (RBTree _ l v r) | x < v      = member x l
member x (RBTree _ l v r) | otherwise = member x r
```

# Красно-чёрное дерево: реализация

```
insert x E =
  RBTree B a y b
    where (RBTree _ a y b) = ins s
          ins E = RBTree R E x E
          ins s@(RBTree c a y b) | x == y = s
          ins (RBTree c a y b)   | x < y =
              balance c (ins a) y b
          ins (RBTree c a y b)   | x > y =
              balance c a y (ins b)
```

## Красно-чёрное дерево: реализация

```
balance B (RBTREE R (RBTREE R a x b) y c) z d =  
    RBTREE R (RBTREE B a x b) y (RBTREE B c z d)  
balance B (RBTREE R a x (RBTREE R b y c)) z d =  
    RBTREE R (RBTREE B a x b) y (RBTREE B c z d)  
balance B a x (RBTREE R (RBTREE R b y c) z d) =  
    RBTREE R (RBTREE B a x b) y (RBTREE B c z d)  
balance B a x (RBTREE R b y (RBTREE R c z d)) =  
    RBTREE R (RBTREE B a x b) y (RBTREE B c z d)  
balance color left value right = RBTREE color left value right
```

- Структура данных с константным доступом к **минимальному** элементу
- Обычно реализуется как дерево (которое, в свою очередь, может быть упаковано в массив)
- Дерево обладает heap property — дети всегда больше своих родителей

- Бинарное дерево
- Ранг левого ребёнка всегда больше или равен рангу правого ребёнка
- Ранг это длина *правой хорды* (самого правого пути до листа)
- Проще всего хранить ранг прямо в дереве

## Левосторонняя куча

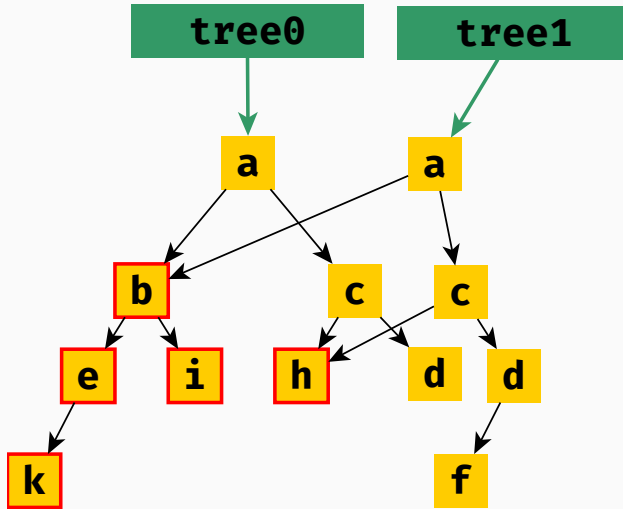
```
data Heap a = E | Heap Integer a (Heap a) (Heap a)
rank (Heap r _ _ _) = r
rank E = 0
makeHeap x a b | rank a > rank b = Heap (rank b + 1) x a b
               | rank b > rank a = Heap (rank a + 1) x b a
merge h E = h
merge E h = h
merge lh@(Heap _ lv ll lr) rh@(Heap _ rv rl rr) =
  if (rv > lv)
  then makeHeap lv ll (merge lr rh)
  else makeHeap rv rl (merge lh rr)
```

## Левосторонняя куча

```
insert x h = merge (Heap 1 x E E) h  
findMin (Heap _ x _ _) = x  
deleteMin (Heap _ _ a b) = merge a b
```

```
tree0 = Heap ...
```

```
tree1 = f 'insert' tree0
```

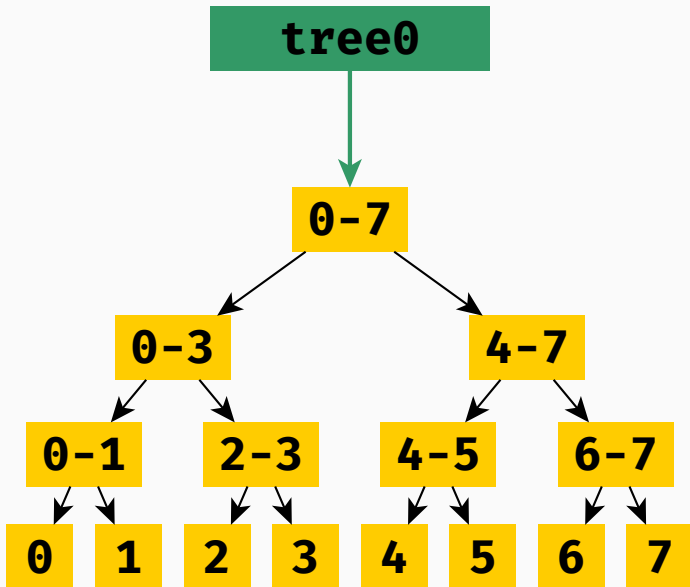




## Дерево отрезков (сегментное дерево)

- Заранее сбалансированное бинарное дерево
- Структура данных, описывающая данные на заданном промежутке
- Позволяет делать запросы вида «минимум на отрезке» за логарифм
  - Минимум нужно хранить в каждом узле
  - Вместо минимума можно считать любую *ассоциативную* операцию

# Дерево отрезков



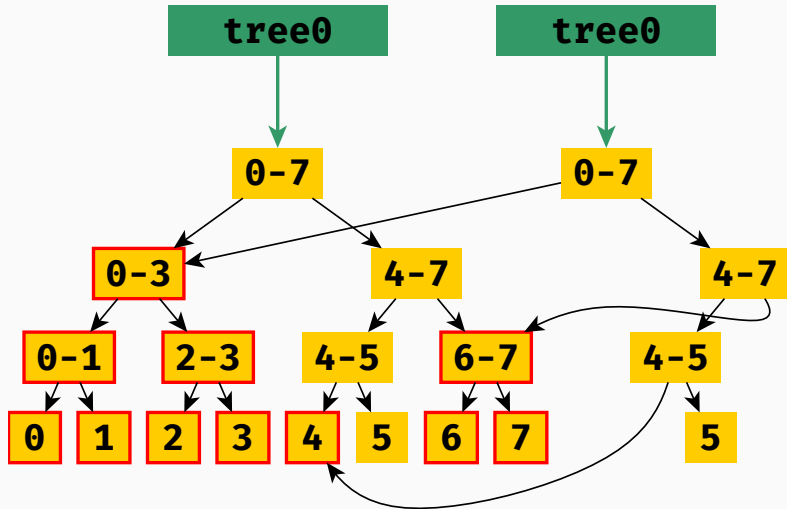
## Дерево отрезков

```
data SegTree a = Leaf a | SegTree a (SegTree a) (SegTree a)
data SegRange a = SegRange Integer Integer (SegTree a)
value (Leaf a) = a
value (SegTree a _ _) = a
set i x r = let SegRange min max tree = r
             avg' = avg min max
             in
             SegRange min max $ set' i x min avg' max tree
```

## Дерево отрезков

```
set' i x min avg max (Leaf _) = Leaf x
set' i x min avg max (SegTree a l r) | i < avg =
  let newAvg = (avg - min)/2
      l' = set' i x min newAvg avg l
      v' = f (value l') (value r)
  in SegTree v' l' r
set' i x min avg max (SegTree a l r) | i >= avg =
  let newAvg = (max - avg - 1)/2
      r' = set' i x (avg + 1) newAvg max r
      v' = f (value l) (value r')
  in SegTree v' l r'
```

# Дерево отрезков



- Рассмотрено большое количество персистентных структур данных
- Персистентность позволяет:
  - Уменьшить сложность по памяти
  - Использовать данные в персистентном режиме

- Амортизированная сложность работает за счёт «накопления» резервов в структуре
- Если структуру можно «бесплатно» копировать, что с накопленными резервами?

- Амортизированная сложность работает за счёт «накопления» резервов в структуре
- Если структуру можно «бесплатно» копировать, что с накопленными резервами?
- К сожалению, персистентное использование ломает амортизированную сложность
- Иногда ленивые вычисления позволяют это обойти



