

Persistent data structures

Михаил Беляев

22 ноября 2019 г.

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 структур 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?

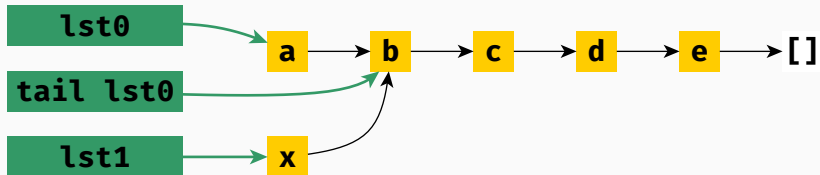
Далее я (следом за всеми ненаучными источниками)
буду использовать термин
«персистентная структура данных»
в значении
«структура данных, хорошо подходящая для
персистентного использования по своим
характеристикам»

Это не вполне корректно, но так всем будет проще

Пример персистентной структуры данных — список

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

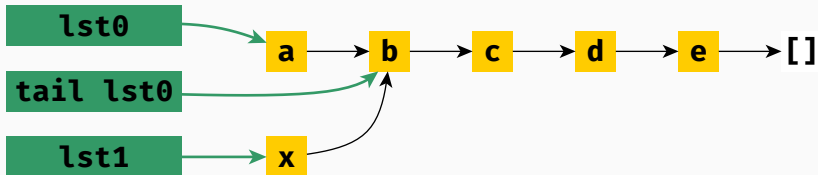
```
lst1 = x:(tail lst0)
```



Пример персистентной структуры данных — список

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

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



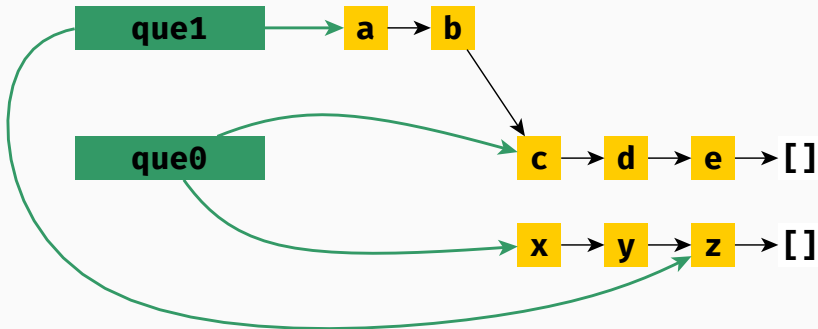
Можете ли вы привести пример операции, при которой всё будет не так радужно?

Очередь из лекции 7

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

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

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

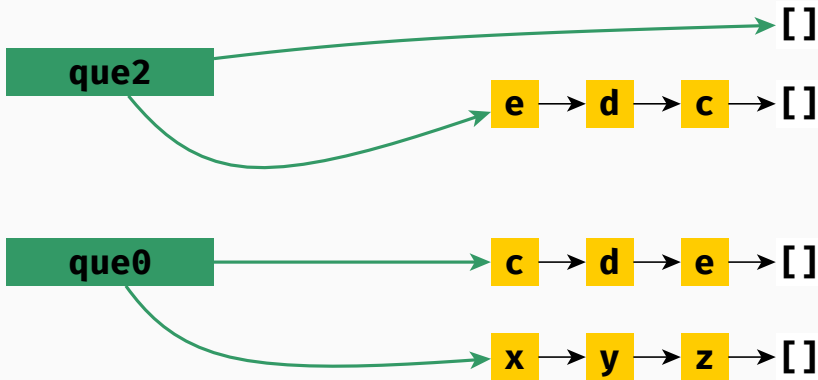


Очередь из лекции 7

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

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

```
que2 = discard $ discard $ discard $ que0
```



- Типичное использование связного 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

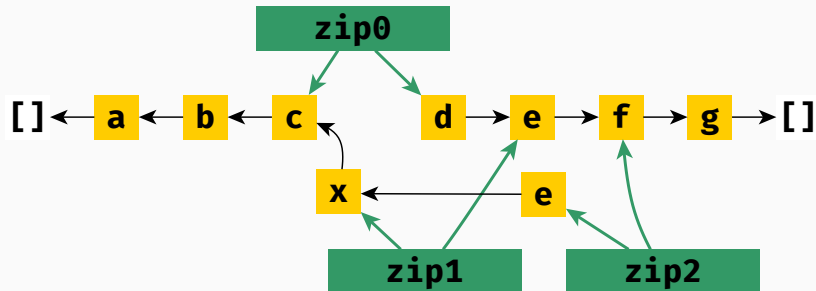
- Можно показать, что все операции, которые имели сложность $O(1)$ у обычного списка, имеют амортизированную сложность $O(1)$ у zipper
- При этом он позволяет модифицировать данные «на ходу»
- Zipper — это персистентная структура данных
- Zipper можно расширить на другие структуры данных

Персистентный zipper

```
zip0 = ...
```

```
zip1 = removeRight $ putLeft x zip0
```

```
zip2 = goRight zip1
```



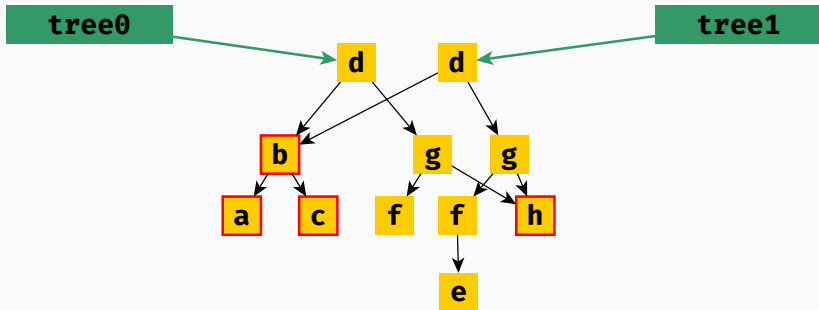
Бинарные деревья поиска

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: красный и чёрный
- 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
```

Бинарные деревья поиска

- Самобалансирующихся деревьев поиска много
 - AVL trees
 - Scapegoat trees
 - Splay trees
 - etc
- Принципы там примерно те же самые

Бинарные деревья поиска при персистентном использовании

- БДП позволяют себя менять с копированием очень маленькой ($O(\log(N))$ в идеальном случае) части памяти, остальная же память делится между версиями
- Но:
 - Ровно по высоте получается менять только небалансированные деревья
 - Высота логарифмическая только у балансированных
 - Нужно делать выбор
- На практике это не так важно, потому что в любом случае $\log(N)$ **намного меньше** N

Хип, куча или пирамида

- 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

Левосторонняя куча (leftist heap)

- Бинарное дерево
(но не бинарное дерево поиска!)
- 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

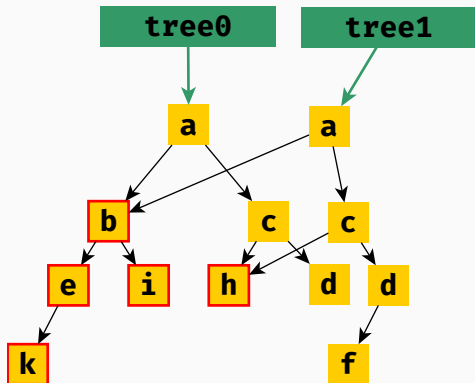
`insert x h = merge (Heap 1 x E E) h`

`findMin (Heap _ x _ _) = x`

`deleteMin (Heap _ _ a b) = merge a b`

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

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



Промежуточный итог

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

Вопрос сложности

- Вспомним лекцию про методы банкира и физика
- Амортизированная сложность работает за счёт «накопления» резервов в структуре
- Если структуру можно «бесплатно» копировать, что с накопленными резервами?

Вопрос сложности

- Вспомним лекцию про методы банкира и физика
- 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 это можно обойти, а иногда применить чёрную магию ленивости
 - Об этом — в следующей лекции

Зачем нужно персистентное использование вне ФП

Зачем нужно персистентное использование вне ФП

- Хранение множества версий одного и того же состояния
 - Стек undo/redo
- Менеджмент состояния в UI
 - Например, персистентный DOM в веб-программировании
- Параллельные алгоритмы
 - Проще подменить одну версию другой, чем жонглировать мьютексами
- etc.

- Зачастую, под персистентным использованием подразумевают одну-две версии СД
- Это совсем не то же самое, что тысячи версий
- Есть структуры данных, которые хорошо работают в первом случае и плохо — во втором

Какие ещё есть сложности

- Не работают без GC
 - Напрямую проистекает из плюсов
- Неудобство в использовании
 - «Глубокие» апдейты
- Нужно чётко понимать требуемый набор операций
 - Виновата сложно амортизированная сложность



<http://kspt.icc.spbstu.ru/course/lang>
belyaev@kspt.icc.spbstu.ru



КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ И
ПРОГРАММНЫХ ТЕХНОЛОГИЙ СПбПУ



ПОЛИТЕХ
Санкт-Петербургский
политехнический университет
Петра Великого