Чисто функциональные структуры данных

С примерами кода на Haskell

Косарев Дмитрий

матмех СПбГУ

27 сентября 2020 г.

Дата сборки: 27 сентября 2020 г.

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

Чисто функциональные структуры данных

- Следуя книге Okasaki "Pure Functional Data Structures"
- Чисто функциональные – это
  - используя только чистые функции
  - проектируя с помощью индуктивных (алгебраических) типов
- С примерами кода на языке Haskell
- В принципе, можно бы и на Python, но оно будет выглядеть отвратительно
- Поэтому сначала gentle introduction to Haskell ©

- Будет непонятно – кричите!

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

Чистые функции

Определение

Чистая функция – это

- Детерминированная
- В процессе работы не совершающая “побочных эффектов”

Т.е. запрещены: ввод-вывод, случайные значения, присваивания

N.B. Это свойство *функции*, а не языка программирования

Индуктивные типы данных

По сути: объединение enum'ов и record'ов в единый способ описания типов.

```c
// C or C++
enum typ {
  A, B
}
struct a_contents {
  Int    first;
  String second;
};
struct b_contents {
};

size_t foo(typ tag, (void*)cntnts );
```

```haskell
-- Haskell
data Typ = A Int String
         | B


  -- A and B are called
  -- ‘constructors‘

foo :: Typ → Size
```

Индуктивные типы данных (2/2). Использование

```c
/* C */
size_t foo(typ tag, (void*)cntnts ) {
  switch (tag) {
  case A:
    int x    = ((a_contents*)cntnts).first;
    String s = ((a_contents*)cntnts).second;
    ...
    break;
  case B:
    ...
    break;
  }
  assert(0); /* unreachable */
  return 0;
}
```

```haskell
-- Haskell
foo (A x s) = ...
foo B       = ...


-- A and B are called
-- ‘constructors‘


-- compiler warns if some
-- cases are not taken
-- care of
```

Индуктивные типы данных. Важные примеры.

```haskell
data Nat =
   Zero                                  -- Zero :: Nat
 | Succ Nat                              -- Succ :: Nat → Nat

data List a =                                -- Nil :: List a
    Nil                                      -- Cons :: a → List a → List
 | Cons a (List a)                           -- Cons 5 Nil :: List Int

data Tree  a =
    Leaf                            -- Leaf :: Tree a
  | Node (Tree a) a (Tree a)        -- Node :: Tree a → a → Tree a → Tree
```

Индуктивные типы данных. Встроенный список Haskell

```haskell
-- user-defined linked list in Haskell
data List a =                          -- Nil :: List a
    Nil                                -- Cons :: a → List a → List
  | Cons a (List a)                    -- Cons 5 Nil :: List Int

-- built-in linked list in Haskell
data [] a = [] | a : [a]               -- [] :: [a]
                                       -- (:) :: a → [a] → [a]
                                       -- 1:2:3:[] :: [Int]
                                       -- [1,2,3,4] :: [Int]
```

Индуктивные типы данных и множества

```haskell
data Nat =
    Zero                                -- Zero :: Nat
  | Succ Nat                            -- Succ :: Nat → Nat
```

Тип **Nat** описывает индуктивное множество значений, населяющих тип **Nat**, такое, что оно:

1. содержит значение **Zero**
2. замкнуто относительно операции **Succ :: Nat → Nat**
3. минимально

Для остальных индуктивных типов данных (связные списки, деревья) рассуждения аналогичны.

Классы типов $\sim$ интерфейсы

```haskell
-- a la interface implementation
instance STACK [] where
  empty      = []
  isEmpty [] = True
  isEmpty _  = False
  cons   x xs = x:xs
```

Устойчивость (persistence)

Отличительной особенностью функциональных структур данных является то, что они всегда *устойчивы* (persistent) — обновление функциональной структуры не уничтожает старую версию, а создает новую, которая с ней сосуществует.

Устойчивость достигается путем *копирования* затронутых узлов структуры данных, и все изменения проводятся на копии, а не на оригинале.

Поскольку узлы никогда напрямую не модифицируются, все незатронутые узлы могут *совместно использоваться* (be shared) между старой и новой версией структуры данных без опасения, что изменения одной версии непроизвольно окажутся видны другой.

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

Сигнатура Stack. Реализация через встроенные списки

```haskell
class STACK s where
  empty   :: s a
  isEmpty :: s a → Bool
  cons    :: a → s a → s a
  head    :: s a → a
  tail    :: s a → s a
```

```haskell
instance STACK [] where
  empty       = []
  isEmpty []  = True
  isEmpty _   = False
  cons   x xs = x:xs

  head []     = error "empty list"
  head (x:_)  = x
  tail []     = error "empty list"
  tail (_:xs) = xs
```

Сигнатура Stack. Реализация через новый тип данных

```haskell
class STACK s where
  empty   :: s a
  isEmpty :: s a → Bool
  cons    :: a → s a → s a
  head    :: s a → a
  tail    :: s a → s a
```

```haskell
data Stack a = Nil | Cons a (Stack a)

instance STACK Stack where
  empty     = Nil
  isEmpty Nil = True
  isEmpty _   = False
  cons   x xs = Cons x xs

  head Nil = error "empty list"
  head (Cons x _) = x
  tail Nil = error "empty list"
  tail (Cons _ xs) = xs
```

Конкатенация списков

```
(++) :: STACK l ⇒ l a → l a → l a
```

В императивной среде легко сделать за O(1), если хранить указатель на конец.

Конкатенация в императивной среде

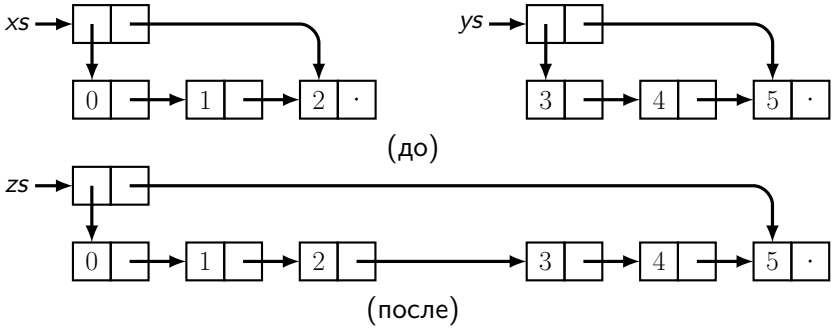


Рис.: Выполнение xs ++ ys в императивной среде. Эта операция уничтожает списки-аргументы xs и ys (их использовать больше нельзя)

Конкатенация в функциональной среде

В функциональной среде мы не можем деструктивно модифицировать. Поэтому

- добавляем последний элемент первого списка ко второму
- добавляем *пред*последний элемент первого списка к результату
- и т.д.

```
(++) :: STACK l ⇒ l a → l a → l a
(++) xs ys =
  if isEmpty xs
  then ys
  else cons (head xs) (tail xs ++ ys)
```

Если нам доступно внутреннее представление, то можно написать более короткий идиоматичный код

```
(++) []     ys = ys
(++) (x:xs) ys = x:(xs ++ ys)
```

Конкатенация

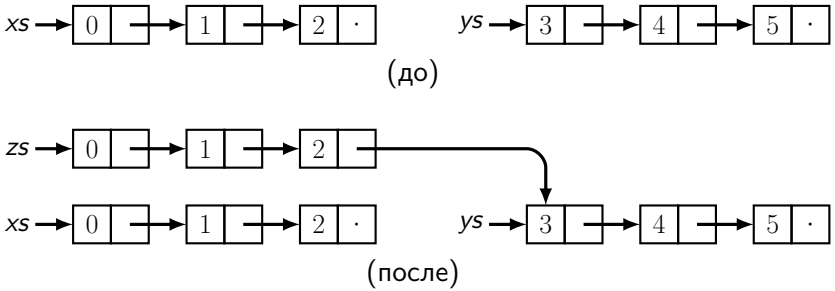


Рис.: Выполнение `zs = xs ++ ys` в функциональной среде. Заметим, что списки-аргументы `xs` и `ys` не затронуты операцией.

Несмотря на большой объем копирования, заметим, что второй список копировать не пришлось

Update

```
update :: [a] → Int → a → [a]
update []     _ _ = error "subscript"
update (x:xs) 0 y = y : xs
update (x:xs) n y = x : (update xs (n-1) y)
```

Здесь мы не копируем весь список-аргумент.

Копировать приходится только сам узел, подлежащий модификации (узел i) и узлы, содержащие прямые или косвенные указатели на i.

Другими словами, чтобы изменить один узел, мы копируем все узлы на пути от корня к изменяемому. Все узлы, не находящиеся на этом пути, используются как исходной, так и обновленной версиями.

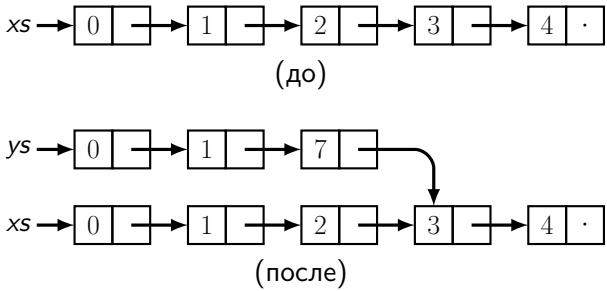


Рис.: Выполнение `ys = update xs 2 7`. Обратите внимание на совместное использование структуры списками `xs` и `ys`.

Замечание

Такой стиль программирования очень сильно упрощается при наличии автоматической сборки мусора. Очень важно освободить память от тех копий, которые больше не нужны, но многочисленные совместно используемые узлы делают ручную сборку мусора нетривиальной задачей.

Упражнение

Напишите функцию `suffixes` типа `[a]` $\rightarrow$ `[a]`, которая принимает как аргумент список `xs` и возвращает список всех его суффиксов в убывающем порядке длины. Например,

```
suffixes [1,2,3,4] = [[1,2,3,4],[2,3,4],[3,4],[4],[]]
```

Покажите, что список суффиксов можно породить за время $O(n)$ и занять при этом $O(n)$ памяти.

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

Двоичные деревья поиска

Если узел структуры содержит более одного указателя, оказываются возможны более сложные сценарии совместного использования памяти. Хорошим примером совместного использования такого вида служат *двоичные деревья поиска*.

```
data Tree a = E | T (Tree a) a (Tree a)
```

Двоичные деревья поиска — это двоичные деревья, в которых элементы хранятся во внутренних узлах в *симметричном* (symmetric) порядке, то есть, элемент в каждом узле больше любого элемента в левом поддереве этого узла и меньше любого элемента в правом поддереве.

Сигнатура для множеств упорядоченных элементов

```
class SET s where
  empty :: s a
  insert :: (Ord a) ⇒ a → s a → s a
  member :: (Ord a) ⇒ a → s a → Bool
```

Сигнатура для множеств значение «пустое множество», а также функции добавления нового элемента и проверки на членство.

В более практической реализации, вероятно, будут присутствовать и многие другие функции, например, для удаления элемента или перечисления всех элементов.

Функция member

Ищет в дереве, сравнивая запрошенный элемент с находящимся в корне дерева.

```
member :: (Ord a) ⇒ a → Tree a → Bool
member _ E                = False
member x (T l y _) | x<y = member x l
member x (T _ y r) | x>y = member x r
member _ _                = True
```

Если мы когда-либо натыкаемся на пустое дерево, значит, запрашиваемый элемент не является членом множества, и мы возвращаем значение `False`.

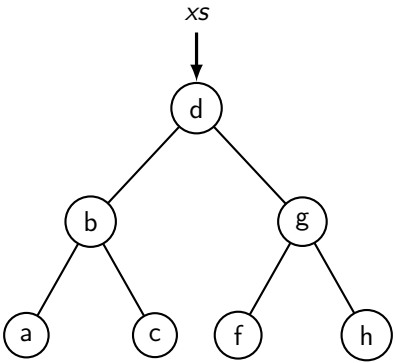
Если запрошенный элемент **меньше** корневого, мы рекурсивно ищем в левом поддереве.
Если он **больше**, рекурсивно ищем в правом поддереве.
Наконец, в оставшемся случае запрошенный элемент **равен** корневому, и мы возвращаем значение `True`.

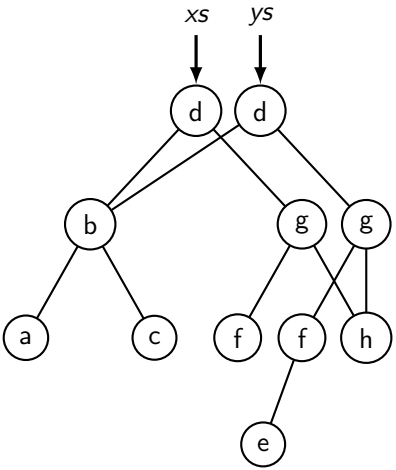
Функция insert

```
insert :: (Ord a) ⇒  a → Tree a → Tree a
insert x E                = T E x E
insert x (T l y r) | x<y = T (insert x l) y r
insert x (T l y r) | x>y = T l y (insert x l)
insert _ t                = t
```

Функция `insert` проводит поиск в дереве по той же стратегии, что и `member`, но только по пути она копирует каждый элемент.

Когда, наконец, оказывается достигнут пустой узел, он заменяется на узел, содержащий новый элемент.





Выполнение `ys = insert "e" xs`.

Для большинства деревьев путь поиска содержит лишь небольшую долю узлов в дереве.

Громадное большинство узлов находятся в совместно используемых поддеревьях.

Упражнение

Андерсон В худшем случае member производит $2d$ сравнений, где d — глубина дерева. Перепишите ее так, чтобы она делала не более $d + 1$ сравнений, сохраняя элемент, который *может* оказаться равным запрашиваемому (например, последний элемент, для которого операция $<$ вернула значение «истина» или $\leq$ — «ложь», и производя проверку на равенство только по достижении дна дерева.

Упражнение

Вставка уже существующего элемента в двоичное дерево поиска копирует весь путь поиска, хотя скопированные узлы неотличимы от исходных. Перепишите insert так, чтобы она избегала копирования с помощью исключений. Установите только один обработчик исключений для всей операции поиска, а не по обработчику на итерацию.

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

Как правило, множества и конечные отображения поддерживают эффективный доступ к произвольным элементам. Однако иногда требуется эффективный доступ только к *минимальному* элементу. Структура данных, поддерживающая такой режим доступа, называется *очередь с приоритетами* (priority queue) или *куча* (heap).

```
class Heap h where
  empty     :: Ord a ⇒ h a
  isEmpty   :: Ord a ⇒ h a → Bool

  insert    :: Ord a ⇒ a → h a → h a
  merge     :: Ord a ⇒ h a → h a → h a

  findMin   :: Ord a ⇒ h a → a
  deleteMin :: Ord a ⇒ h a → h a
```

Определение (*Порядок кучи* (heap-ordered))

Элемент при каждой вершине не больше элементов в поддеревьях.

При таком упорядочении минимальный элемент дерева всегда находится в корне.

Определение (*Правая периферия* (right spine) узла)

Самого правого пути от данного узла до пустого

Ранг узла определяется как длина его правой периферии.

Определение (Свойство *левоориентированности* (leftist property))

Ранг любого левого поддерева не меньше ранга его сестринской правой вершины.

Простым следствием свойства левоориентированности является то, что правая периферия любого узла — кратчайший путь от него к пустому узлу.

Левоориентированные кучи представляют собой двоичные деревья с порядком кучи, обладающие свойством левоориентированности.

Левоориентированные кучи

Если у нас есть некоторый тип упорядоченных элементов `e`, мы можем представить левоориентированные кучи как двоичные деревья, снабженные информацией о ранге.

```
data LeftistHeap a = E | T Int a (LeftistHeap a) (LeftistHeap a)
```

Заметим, что элементы правой периферии левоориентированной кучи (да и любого дерева с порядком кучи) расположены в порядке возрастания.

Главная идея левоориентированной кучи заключается в том, что для слияния двух куч достаточно слить их правые периферии как упорядоченные списки, а затем вдоль полученного пути обменивать местами поддеревья при вершинах, чтобы восстановить свойство левоориентированности.

```
merge h E = h
merge E h = h
merge h1@(T _ x a1 b1) h2@(T _ y a2 b2) =
  if x ⩽ y
  then makeT x a1 (merge b1 h2)
  else makeT y a2 (merge h1 b2)
```

где makeT — вспомогательная функция, вычисляющая ранг вершины **T** и, если необходимо, меняющая местами ее поддеревья.

```
rank E = 0
rank (T r _ _ _) = r

makeT x a b = if rank a ⩾ rank b
              then T (rank b + 1) x a b
              else T (rank a + 1) x b a
```

Поскольку длина правой периферии любой вершины в худшем случае логарифмическая, merge выполняется за время $O(\log n)$.

```
insert x h = merge (T 1 x E E) h
findMin E = error "empty heap"
findMin (T _ x a b) = x

deleteMin E = error "empty heap"
deleteMin (T _ x a b) = merge a b
```

Поскольку `merge` выполняется за время $O(\log n)$, столько же занимают и `insert` с `deleteMin`.

Очевидно, что `findMin` выполняется за $O(1)$.

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

Биномиальные кучи

Биномиальные очереди , которые мы, чтобы избежать путаницы с очередями FIFO, будем называть *биномиальными кучами* (binomial heaps) — ещё одна распространенная реализация куч.

Биномиальные кучи устроены сложнее, чем левоориентированные, и, на первый взгляд, не возмещают эту сложность никакими преимуществами.

Однако, с помощью дополнительных хитростей (амортизация), можно заставить `insert` и `merge` выполняться за время $O(1)$.

Биномиальные деревья. Пример

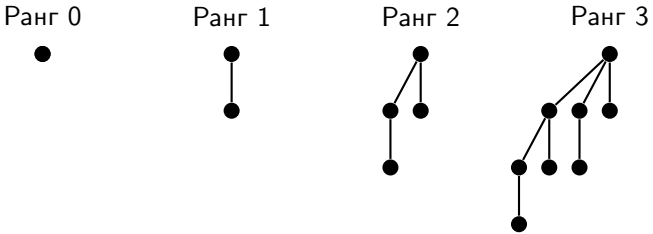


Рис.: Биномиальные деревья рангов 0–3.

Биномиальные деревья. Определение

Биномиальные кучи строятся из более простых объектов, называемых биномиальными деревьями. Биномиальные деревья индуктивно определяются так:

- Биномиальное дерево ранга 0 представляет собой одиночный узел.
- Биномиальное дерево ранга $r+1$ получается путем *связывания* (linking) двух биномиальных деревьев ранга r, так что одно из них становится самым левым потомком второго.

Из этого определения видно, что биномиальное дерево ранга r содержит ровно 2^r элементов.

Существует второе, эквивалентное первому, определение биномиальных деревьев, которым иногда удобнее пользоваться: биномиальное дерево ранга r представляет собой узел с r потомками $t_1 \ldots t_r$, где каждое t_i является биномиальным деревом ранга $(r-i)$.

Каждый список потомков хранится в *убывающем* (sic!) порядке рангов, а элементы хранятся вместе с рангом кучи. Чтобы сохранять этот порядок рангов, мы всегда привязываем дерево с большим корнем к дереву с меньшим.

```
link t1@(Node r x1 c1) t2@(Node _ x2 c2) =
  if x1 ⩽ x2
  then Node (r+1) x1 (t2 : c1)
  else Node (r+1) x2 (t1 : c2)
```

Будем привязывать деревья только с одинаковым рангом

Определяем биномиальную кучу как

- коллекцию биномиальных деревьев
- каждое из которых имеет порядок кучи
- никакие два дерева не совпадают по рангу

Например, список деревьев в порядке возрастания ранга.

```
data Tree a = Node Int a [Tree a]
```

Биномиальные кучи и числа

Поскольку каждое биномиальное дерево содержит 2^r элементов, и никакие два дерева по рангу не совпадают, деревья размера n в точности соответствуют единицам в двоичном представлении n.

Например, число $21_{10} = 10101_2$, и поэтому биномиальная куча размера 21 содержит одно дерево ранга 0, одно ранга 2, и одно ранга 4 (размерами, соответственно, 1, 4 и 16).

Заметим, что так же, как двоичное представление n содержит не более $\lfloor log(n+1) \rfloor$ единиц, биномиальная куча размера n содержит не более $\lfloor log(n+1) \rfloor$ деревьев.

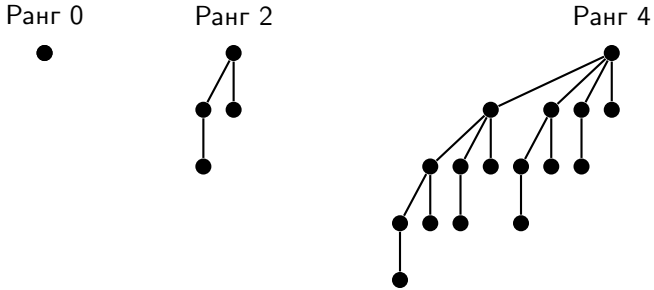


Рис.: Число $21_{10} = 10101_2$, и поэтому биномиальная куча размера 21 содержит одно дерево ранга 0, одно ранга 2, и одно ранга 4 (размерами, соответственно, 1, 4 и 16).

insert – аналогично сложению

Чтобы внести элемент в кучу, мы сначала создаем одноэлементное дерево (т. е., биномиальное дерево ранга 0), затем поднимаемся по списку существующих деревьев в порядке возрастания рангов, связывая при этом одноранговые деревья. Каждое связывание соответствует переносу в двоичной арифметике.

```
insTree t [] = [t]
insTree t ts@(t' : ts') =
  if rank t < rank t' then t:ts else insTree (link t t') ts'
insert x (BH ts) = BH (insTree (Node 0 x []) ts)
```

В худшем случае, при вставке в кучу размера $n = 2^k - 1$, требуется k связываний и $O(k) = O(\log n)$ времени.

merge – аналогично сложению

При слиянии двух куч мы проходим через оба списка деревьев в порядке возрастания ранга и связываем по пути деревья равного ранга. Как и прежде, каждое связывание соответствует переносу в двоичной арифметике.

```
mrg ts1 [] = ts1
mrg [] ts2 = ts2
mrg ts1@(t1:ts'1) ts2@(t2:ts'2)
  | rank t1 < rank t2 = t1 : mrg ts'1 ts2
  | rank t2 < rank t1 = t2 : mrg ts1 ts'2
  | otherwise = insTree (link t1 t2) (mrg ts'1 ts'2)
merge (BH ts1) (BH ts2) = BH (mrg ts1 ts2)
```

Поиск минимального элемента

Функции `findMin` и `deleteMin` вызывают вспомогательную функцию `removeMinTree`, которая находит дерево с минимальным корнем, исключает его из списка и возвращает как это дерево, так и список оставшихся деревьев.

```
removeMinTree [] = error "empty heap"
removeMinTree [t] = (t, [])
removeMinTree (t:ts) =
  if root t < root t' then (t, ts) else (t', t:ts')
  where (t', ts') = removeMinTree ts
```

Функция `findMin` просто возвращает корень найденного дерева

```
findMin (BH ts) = root t
  where (t, _) = removeMinTree ts
rank (Node r x c) = r
```

Удаление минимального элемента

Функция `deleteMin` устроена немного похитрее.

Отбросив корень найденного дерева, мы ещё должны вернуть его потомков в список остальных деревьев. Заметим, что список потомков *почти* уже соответствует определению биномиальной кучи. Это коллекция биномиальных деревьев с неповторяющимися рангами, но только отсортирована она не по возрастанию, а по убыванию ранга. Таким образом, обратив список потомков, мы преобразуем его в биномиальную кучу, а затем сливаем с оставшимися деревьями.

```
deleteMin (BH ts) = BH (mrg (reverse ts1) ts2)
  where (Node _ x ts1, ts2) = removeMinTree ts
```

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

Красно-чёрные деревья

Двоичные деревья поиска хорошо ведут себя на случайных или неупорядоченных данных, однако на упорядоченных данных их производительность резко падает, и каждая операция может занимать до $O(n)$ времени.

Решение этой проблемы состоит в том, чтобы каждое дерево поддерживать в приблизительно сбалансированном состоянии. Тогда каждая операция выполняется не хуже, чем за время $O(\log n)$.

Одним из наиболее популярных семейств сбалансированных двоичных деревьев поиска являются красно-чёрные .

Красно-чёрное дерево представляет собой двоичное дерево поиска, в котором каждый узел окрашен либо красным, либо чёрным. Мы добавляем поле цвета в тип двоичных деревьев поиска

```
data Color = R | B
```

Все пустые узлы считаются чёрными, поэтому пустой конструктор **E** в поле цвета не нуждается.

Красно-чёрные деревья. Инварианты

Мы требуем, чтобы всякое красно-чёрное дерево соблюдало два инварианта:

- **Инвариант 1.** У красного узла не может быть красного ребёнка.
- **Инвариант 2.** Каждый путь от корня дерева до пустого узла содержит одинаковое количество чёрных узлов.

Вместе эти два инварианта гарантируют, что самый длинный возможный путь по красно-чёрному дереву, где красные и чёрные узлы чередуются, не более чем вдвое длиннее самого короткого, состоящего только из чёрных узлов.

Функция `member` для красно-чёрных деревьев не обращает внимания на цвета. За исключением wildcard в варианте для конструктора **T**, она не отличается от функции `member` для несбалансированных деревьев.

```
member x E = False
member x (T _ a y b) = if x < y then member x a
                       else if x > y then member x b
```

Функция `insert` интереснее: она должна поддерживать два инварианта балансировки.

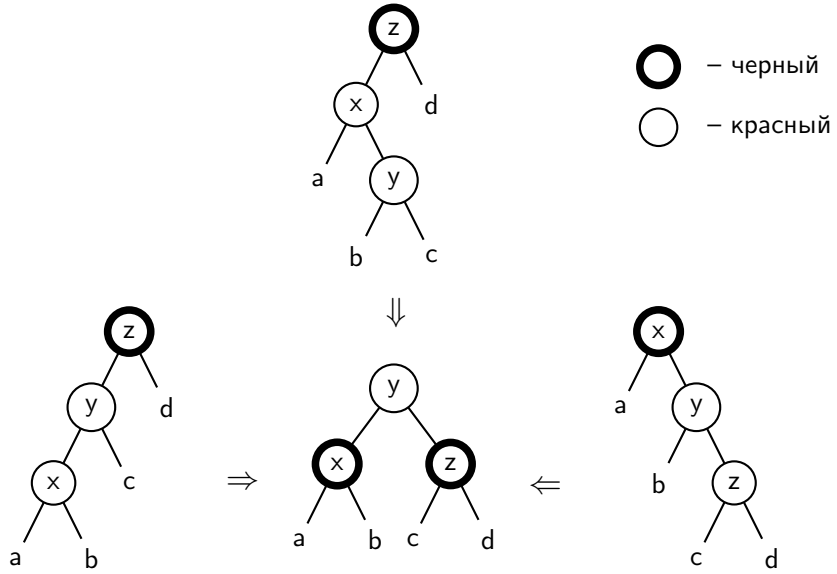
```
insert x s = T B a y b
  where ins E = T R E x E
        ins s@(T color a y b) =
          if x < y then balance color (ins a) y b
          else if x > y then balance color a y (ins b)
               else s
```

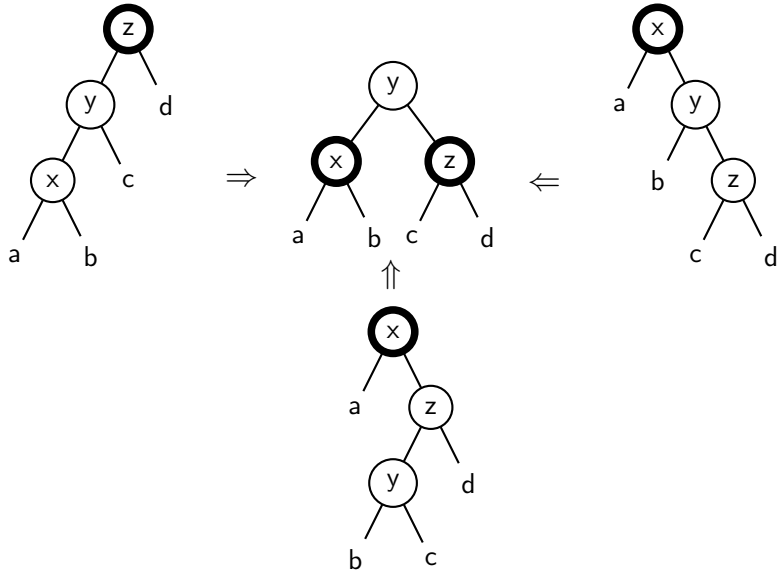
Эта функция содержит три существенных изменения по сравнению с `insert` для несбалансированных деревьев поиска. Во-первых, когда мы создаем новый узел в ветке `ins E`, мы сначала окрашиваем его в красный цвет. Во-вторых, независимо от цвета, возвращаемого `ins`, в окончательном результате мы корень окрашиваем чёрным. Наконец, в ветках `x < y` и `x > y` мы вызовы конструктора `T` заменяем на обращения к функции `balance`. Функция `balance` действует подобно конструктору `T`, но только она переупорядочивает свои аргументы, чтобы обеспечить выполнение инвариантов баланса.

```
balance B (T R (T R a x b) y c) z d = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d) = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d)) = T R (T B a x b) y (T B c z d)
```

Если новый узел окрашен красным, мы сохраняем Инвариант 2, но в случае, если отец нового узла тоже красный, нарушается Инвариант 1. Мы временно позволяем существовать одному такому нарушению, и переносим его снизу вверх по мере перебалансирования. Функция `balance` обнаруживает и исправляет красно-красные нарушения, когда обрабатывает чёрного родителя красного узла с красным ребёнком. Такая чёрно-красно-красная цепочка может возникнуть в четырёх различных конфигурациях, в зависимости от того, левым или правым ребёнком является каждая из красных вершин. Однако в каждом из этих случаев решение одно и то же: нужно преобразовать чёрно-красно-красный путь в красную вершину с двумя чёрными детьми, как показано на рисунке ниже.

После балансировки некоторого поддерева красный корень этого поддерева может оказаться ребёнком ещё одного красного узла. Таким образом, балансировка продолжается до самого корня дерева. На самом верху дерева мы можем получить красную вершину с красным ребёнком, но без чёрного родителя. С этим вариантом мы справляемся, всегда перекрашивая корень в чёрное.


z
x
d
a
y
b
c
– черный
– красный
⇓
z
y
d
x
c
a
b
⇒
y
x
z
a
b
c
d
⇐
x
a
y
b
z
c
d


z
y
d
x
c
a
b
⇒
y
x
z
a
b
c
d
⇐
x
a
y
b
z
c
d
⇑
x
a
z
y
d
b
c

Указание разработчикам

Даже без дополнительных оптимизаций наша реализация сбалансированных двоичных деревьев поиска — одна из самых быстрых среди имеющихся. С оптимизациями вроде описанных в Упражнениях 3.1 и 6.2 она просто летает!

Почему это выглядит короче императивной реализации?

Замечание

Одна из причин, почему наша реализация выглядит настолько проще, чем типичное описание красно-чёрных деревьев , состоит в том, что мы используем несколько другие преобразования перебалансировки.
В императивных реализациях обычно наши четыре проблематичных случая разбиваются на восемь, в зависимости от цвета узла, соседствующего с красной вершиной с красным ребёнком. Знание цвета этого узла в некоторых случаях позволяет совершить меньше присваиваний, а в некоторых других завершить балансировку раньше. Однако в функциональной среде мы в любом случае копируем все эти вершины, и таким образом, не можем ни сократить число присваиваний, ни прекратить копирование раньше времени, так что для использования более сложных преобразований нет причины.

Упражнение

Напишите функцию `fromOrdList` типа `[a]` $\rightarrow$ **`Tree`** `a`, преобразующую отсортированный список без повторений в красно-чёрное дерево. Функция должна выполняться за время $O(n)$.

Упражнение

Приведенная нами функция `balance` производит несколько ненужных проверок. Например, когда функция `ins` рекурсивно вызывается для левого ребёнка, не требуется проверять красно-красные нарушения на правом ребёнке.

1. Разбейте `balance` на две функции `lbalance` и `rbalance`, которые проверяют, соответственно, нарушения инварианта в левом и правом ребёнке. Замените обращения к `balance` внутри `ins` на вызовы `lbalance` либо `rbalance`.
2. Ту же самую логику можно распространить ещё на шаг и убрать одну из проверок для внуков. Перепишите `ins` так, чтобы она никогда не проверяла цвет узлов, не находящихся на пути поиска.

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

Методы амортизированного анализа

Реализации с амортизированными характеристиками производительности часто оказываются проще и быстрее, чем реализации со сравнимыми жёсткими характеристиками.

К сожалению, простой подход к амортизации, рассматриваемый в этой главе, конфликтует с идеей устойчивости — эти структуры, будучи используемы как устойчивые, могут быть весьма неэффективны. Однако на практике многие приложения устойчивости не требуют, и часто для таких приложений реализации, представленные в этой главе, могут быть замечательным выбором.

Чтобы совместить амортизацию и устойчивость стоит применить *ленивые вычисления*.

Понятие амортизации возникает из следующего наблюдения. Имея последовательность операций, мы можем интересоваться временем, которое отнимает вся эта последовательность, однако при этом нам может быть безразлично время каждой отдельной операции.

Например, имея n операций, мы можем желать, чтобы время всей последовательности было ограничено показателем $O(n)$, не настаивая, чтобы каждая из этих операций происходила за время $O(1)$. Нас может устраивать, чтобы некоторые из операций занимали $O(\log n)$ или даже $O(n)$, при условии, что общая стоимость всей последовательности будет $O(n)$.

Такая дополнительная степень свободы открывает широкое пространство возможностей при проектировании, и часто позволяет найти более простые и быстрые решения, чем варианты с аналогичными жёсткими ограничениями.

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$$

где a_i — амортизированная стоимость i-й операции, t_i — ее реальная стоимость, а m — общее число операций.

Обычно доказывается несколько более сильный результат: что на любой промежуточной стадии в последовательности операций общая текущая амортизированная стоимость является верхней границей для общей текущей реальной стоимости, т. е. для любого j

$$\sum_{i=1}^{j} a_i \geq \sum_{i=1}^{j} t_i$$

Определение

Разница между общей текущей амортизированной стоимостью и общей текущей реальной стоимостью называется *текущие накопления* (accumulated savings).

Таким образом, общая текущая амортизированная стоимость является верхней границей для общей текущей реальной стоимости тогда и только тогда, когда текущие накопления неотрицательны.

Амортизация позволяет некоторым операциям быть дороже, чем их амортизированная стоимость. Такие операции называются *дорогими* (expensive). Операции, для которых амортизированная стоимость превышает реальную, называются *дешевыми* (cheap). Дорогие операции уменьшают текущие накопления, а дешевые их увеличивают.

Главное при доказательстве амортизированных характеристик стоимости — показать, что дорогие операции случаются только тогда, когда текущих накоплений хватает, чтобы покрыть их дополнительную стоимость.

- *Метод банкира* (banker's method)
  - *кредит* (credits)
- *Метод физика* (physicist's method)
  - *потенциал* (potential)

Кредит и потенциал являются лишь средствами анализа; ни то, ни другое не присутствует в тексте программы (разве что, возможно, в комментариях).

В методе банкира текущие накопления представляются как *кредит* (credits), привязанный к определенным ячейкам структуры данных. Этот кредит используется, чтобы расплатиться за будущие операции доступа к этим ячейкам. Амортизированная стоимость операции определяется как ее реальная стоимость плюс размер кредита, выделяемого этой операцией, минус размер кредита, который она расходует, т. е.,

$$a_i = t_i + c_i - \bar{c}_i$$

где c_i — размер кредита, выделяемого операцией i, а $\bar{c}_i$ — размер кредита, расходуемого операцией i.

$$a_i = t_i + c_i - \bar{c}_i$$

где c_i — размер кредита, выделяемого операцией i, а $\bar{c}_i$ — размер кредита, расходуемого операцией i.

Каждая единица кредита должна быть выделена, прежде чем израсходована, и нельзя расходовать кредит дважды. Таким образом, $\sum c_i \geq \sum \bar{c}_i$, а следовательно, как и требуется, $\sum a_i \geq \sum t_i$.

Как правило, доказательства с использованием метода банкира определяют *инвариант кредита* (credit invariant), регулирующий распределение кредита так, чтобы при всякой дорогой операции достаточное его количество было выделено в нужных ячейках структуры для покрытия стоимости операции.

Метод физика

Определяется функция Φ, отображающая всякий объект d на действительное число, называемое его *потенциалом* (potential). Потенциал обычно выбирается так, чтобы изначально равняться нулю и оставаться неотрицательным. В таком случае потенциал представляет нижнюю границу текущих накоплений.

Пусть объект d_i будет результатом операции i и аргументом операции $i+1$. Тогда амортизированная стоимость операции i определяется как сумма реальной стоимости и изменения потенциалов между d_{i-1} и d_i, т. е.,

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

текущих накоплений.

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

Текущая реальная стоимость последовательности операций равна

$$\begin{aligned}
\sum_{i=1}^{j} t_i &= \sum_{i=0}^{j} (a_i + \Phi(d_{i-1}) - \Phi(d_i)) \\
&= \sum_{i=1}^{j} a_i + \sum_{i=1}^{j} (\Phi(d_{i-1}) - \Phi(d_i)) \\
&= \sum_{i=1}^{j} a_i + \Phi(d_0) - \Phi(d_j)
\end{aligned}$$

Если Φ выбран таким образом, что $\Phi(d_0)$ равен нулю, а $\Phi(d_j)$ неотрицателен, мы имеем $\Phi(d_j) \geq \Phi(d_0)$, так что, как и требуется, текущая общая амортизированная стоимость является верхней границей для текущей общей реальной стоимости.

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

Чисто функциональные очереди

```
class Queue q where
  empty   :: q a
  isEmpty :: q a → Bool

  snoc    :: q a → a → q a
  head    :: q a → a
```

Самая распространенная чисто функциональная реализация очередей представляет собой пару списков, `f` и `r`, где `f` содержит головные элементы очереди в правильном порядке, а `r` состоит из хвостовых элементов в обратном порядке.

Например, очередь, содержащая целые числа 1...6, может быть представлена списками `f=[1,2,3]` и `r=[6,5,4]`. Это представление можно описать следующим типом:

```
data Queue a = Queue [a] [a]
```

В этом представлении голова очереди — первый элемент `f`, так что функции `head` и `tail` возвращают и отбрасывают этот элемент, соответственно.

```
head (x : f, r) = x
tail (x : f, r) = f
```

Подобным образом, хвостом очереди является первый элемент `r`, так что `snoc` добавляет к `r` новый.

```
snoc (f,r) x = (f, x : r)
```

Инвариант очереди

Элементы добавляются к `r` и убираются из `f`, так что они должны как-то переезжать из одного списка в другой. Этот переезд осуществляется путем обращения `r` и установки его на место `f` всякий раз, когда в противном случае `f` оказался бы пустым.

Одновременно `r` устанавливается в `[]`. Наша цель — поддерживать инвариант, что список `f` может быть пустым только в том случае, когда список `r` также пуст (т. е., пуста вся очередь).

Заметим, что если бы `f` был пустым при непустом `r`, то первый элемент очереди находился бы в конце `r`, и доступ к нему занимал бы $O(n)$ времени. Поддерживая инвариант, мы гарантируем, что функция `head` всегда может найти голову очереди за $O(1)$ времени.

Добавление и удаление из очереди

```
snoc ( [], _) x = ([x], [])
snoc (  f, r) x = (f,  x :: r)
tail ([x], r)   = (rev r, [])
tail (x:f, r)   = (f, r)
```

Заметим, что в первой ветке `snoc` используется wildcard. В этом случае поле `r` проверять не нужно, поскольку из инварианта мы знаем, что если список `f` равен `[]`, то `r` также пуст.

Чуть более изящный способ записать эти функции — вынести те части `snoc` и `tail`, которые поддерживают инвариант, в отдельную функцию `checkf`. Она заменяет `f` на `rev r`, если `f` пуст, а в противном случае ничего не делает.

```
checkf ([], r) = (reverse r, [])
checkf q       = q

snoc (f,  r) x = checkf (f, x:r)
tail (x:f,r)   = checkf (f, r)
```

Функции `snoc` и `head` всегда завершаются за время $O(1)$, но `tail` в худшем случае отнимает $O(n)$ времени.

Однако, используя либо метод банкира, либо метод физика, мы можем показать, что как `snoc`, так и `tail` занимают амортизированное время $O(1)$.

Чисто функциональная очередь и метод банкира

Инвариант: каждый элемент в хвостовом списке связан с одной единицей кредита.

Каждый вызов `snoc` для непустой очереди занимает один реальный шаг и выделяет одну единицу кредита для элемента хвостового списка; таким образом, общая амортизированная стоимость равна двум.

Вызов `tail`, не обращающий хвостовой список, занимает один шаг, не выделяет и не тратит никакого кредита, и, таким образом, имеет амортизированную стоимость 1.

Наконец, вызов `tail`, обращающий хвостовой список, занимает $(m + 1)$ реальных шагов, где m — длина хвостового списка, и тратит m единиц кредита, содержащиеся в этом списке, так что амортизированная стоимость получается $m + 1 - m = 1$.

Чисто функциональная очередь и метод физика

В методе физика мы определяем функцию потенциала Φ как длину хвостового списка.

Тогда всякий `snoc` к непустой очереди занимает один реальный шаг и увеличивает потенциал на единицу, так что амортизированная стоимость равна двум.

Вызов `tail` без обращения хвостовой очереди занимает один реальный шаг и не изменяет потенциал, так что амортизированная стоимость равна одному.

Наконец, вызов `tail` с обращением очереди занимает $(m + 1)$ реальных шагов, но при этом устанавливает хвостовой список равным `[]`, уменьшая таким образом потенциал на m, так что амортизированная стоимость равна $m + 1 - m = 1$.

Вывод

У чисто функциональной очереди функция `tail` за $O(n)$ в худшем случае и за $O(1)$ амортизированного.

Указание разработчикам

Эта реализация очередей идеальна в приложениях, где не требуется устойчивости и где приемлемы амортизированные показатели производительности.

Если совместить ленивые вычисления и амортизированные методы, то можно получить устойчивые очереди с хорошими амортизированными характеристиками.

Ленивые вычисления (очень кратко)

Можно представлять число списком цифр. Тогда сложение будет работать за $O(n)$ из-за переносов.

А ещё можно представить с помощью ленивого варианта списка (так называемый *поток* (stream)). Как он будет проводить сложение?

- Вычислит младший разряд за $O(1)$
- Вычисления остальных разрядов проведет потом, если они понадобятся

N.B. Оценивать сложность алгоритмов в присутствие ленивых вычислений очень сложно.

N.B. В языке Haskell *все* вычисления по умолчанию такие.

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

В Разделе 5 мы показали, что вставка в биномиальную кучу проходит в худшем случае за время $O(\log n)$. Здесь мы доказываем, что на самом деле амортизированное ограничение на время вставки составляет $O(1)$.

Метод физика. Потенциал биномиальной кучи – число деревьев в ней.
Заметим, что это число равно количеству единиц в двоичном представлении n, числа элементов в куче. Вызов `insert` занимает $k+1$ шаг, где k — число обращений к `link`. Если изначально в куче было t деревьев, то после вставки окажется $t-k+1$ деревьев. Таким образом, изменение потенциала составляет $(t-k+1)-t=1-k$, а амортизированная стоимость вставки $(k+1)+(1-k)=2$.

Упражнение

Повторите доказательство с использованием метода банкира.

Можно доказать, что `merge` и `deleteMin` работают за $O(\log n)$

Для полноты картины нам нужно показать, что амортизированная стоимость операций `merge` и `deleteMin` по-прежнему составляет $O(\log n)$.

`deleteMin` не доставляет здесь никаких трудностей, но в случае `merge` требуется небольшое расширение метода физика (нужно учесть, что операции могут возвращать больше одного объекта).

Оглавление

1. Введение в Haskell
2. Списки и их конкатенация
3. Двоичные деревья поиска
4. Левоориентированные кучи
5. Биномиальные кучи
6. Красно-чёрные деревья
7. Методы амортизированного анализа
8. Чисто функциональные очереди и их амортизация
9. Биномиальные кучи и амортизация
10. Расширяющиеся (splay) кучи

Расширяющиеся деревья (splay trees)

Расширяющиеся деревья (splay trees) — возможно, самая известная и успешно применяемая амортизированная структура данных.

Расширяющиеся деревья являются ближайшими родственниками двоичных сбалансированных деревьев поиска, но они не хранят никакую информацию о балансе явно.

Вместо этого каждая операция перестраивает дерево при помощи некоторых простых преобразований, которые имеют тенденцию увеличивать сбалансированность. Несмотря на то, что каждая конкретная операция может занимать до $O(n)$ времени, можно показать, что амортизированная стоимость ее не превышает $O(\log n)$.

Расширяющиеся vs. деревья поиска

Важное различие между расширяющимися и сбалансированными двоичными деревьями поиска вроде красно-чёрных деревьев из Раздела 6 состоит в том, что расширяющиеся деревья перестраиваются даже во время запросов (таких, как `member`), а не только во время обновлений (таких, как `insert`).

Это свойство мешает использованию расширяющихся деревьев для реализации абстракций вроде множеств или конечных отображений в чисто функциональном окружении, поскольку приходилось бы возвращать в запросе новое дерево наряду с ответом на запрос[1].

[1] В принципе можно было бы хранить корень расширяющегося дерева в ссылочной ячейке и обновлять значение по ссылке при каждом запросе, но такое решение не является чисто функциональным.

Представление расширяющихся деревьев идентично представлению несбалансированных двоичных деревьев поиска.

Однако в отличие от несбалансированных двоичных деревьев поиска из Раздела 3, мы позволяем дереву содержать повторяющиеся элементы. Эта разница не является фундаментальным различием расширяющихся деревьев и несбалансированных двоичных деревьев поиска; она просто отражает отличие абстракции множества от абстракции кучи.

Реализация `insert`

Разобьем существующее дерево на два поддерева, чтобы одно содержало все элементы, меньше или равные новому, а второе все элементы, большие нового. Затем породим новый узел из нового элемента и двух этих поддеревьев. В отличие от вставки в обыкновенное двоичное дерево поиска, эта процедура добавляет элемент как корень дерева, а не как новый лист.

```
insert x t = T (smaller x t) x (bigger x t)
```

где `smaller` выделяет дерево из элементов, меньше или равных `x`, а `bigger` – больших `x`.

Наивная реализация bigger

По аналогии с фазой разделения быстрой сортировки, назовем новый элемент *границей* (pivot).
Можно наивно реализовать `bigger` как

```
bigger pivot E = E
bigger pivot (T a x b) =
  if x ⩽ pivot
  then bigger pivot b
  else T (bigger pivot a) x b
```

однако при таком решении не делается никакой попытки перестроить дерево, добиваясь лучшего баланса.

Правильная реализация bigger

Вместо этого мы применяем простую эвристику для перестройки: каждый раз, пройдя по двум левым ветвям подряд, мы проворачиваем два пройденных узла.

```
bigger pivot E = E
bigger pivot (T a x b) =
  if x ⩽ pivot
  then bigger pivot b
  else case a of
    E          → T E x b
    T a1 y a2 →
        if y ⩽ pivot
        then T (bigger pivot a2) x b)
        else T (bigger pivot a1) y (T a2 x b)
```

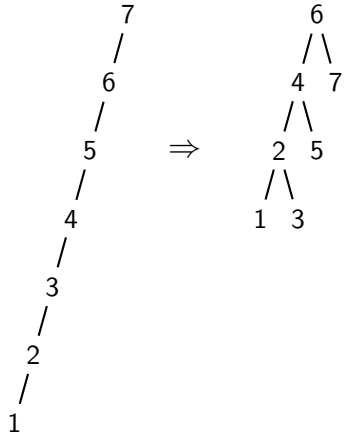


Рис.: Вызов функции `bigger` с граничным элементом `pivot` $= 0$ на сильно несбалансированном дереве.

На Рис. 6 показано, как bigger действует на сильно несбалансированное дерево.

Несмотря на то, что результат по-прежнему не является сбалансированным в обычном смысле, новое дерево намного сбалансированнее исходного; глубина каждого узла уменьшилась примерно наполовину, от d до $\lfloor d/2 \rfloor$ или $\lfloor d/2 \rfloor + 1$.

Разумеется, мы не всегда можем уполовинить глубину каждого узла в дереве, но мы можем уполовинить глубину каждого узла, лежащего на пути поиска.

В сущности, в этом и состоит принцип расширяющихся деревьев: нужно перестраивать путь поиска так, чтобы глубина каждого лежащего на пути узла уменьшалась примерно вполовину.

Рассмотрим теперь `findMin` и `deleteMin`. Минимальный элемент расширяющегося дерева хранится в самой левой его вершине типа **T**. Найти эту вершину несложно.

```
findMin E = error "empty heap"
findMin (T E x b) = x
```

Функция `deleteMin` должна уничтожить самый левый узел и одновременно перестроить дерево таким же образом, как это делает `bigger`. Поскольку мы всегда рассматриваем только левую ветвь, сравнения не нужны.

```
deleteMin E = error "empty heap"
deleteMin (T E x b) = b
deleteMin (T (T E x b) y c) = T b y c
```

N.B. Функция слияния `merge` довольно неэффективна и для многих входов занимает $O(n)$ времени.

Можно показать методом физика, что `insert` выполняется за время $O(\log n)$.

Конец.

Подробнее в книге Окасаки "Чисто функциональные структуры данных".