Persistent data structures, episode 2

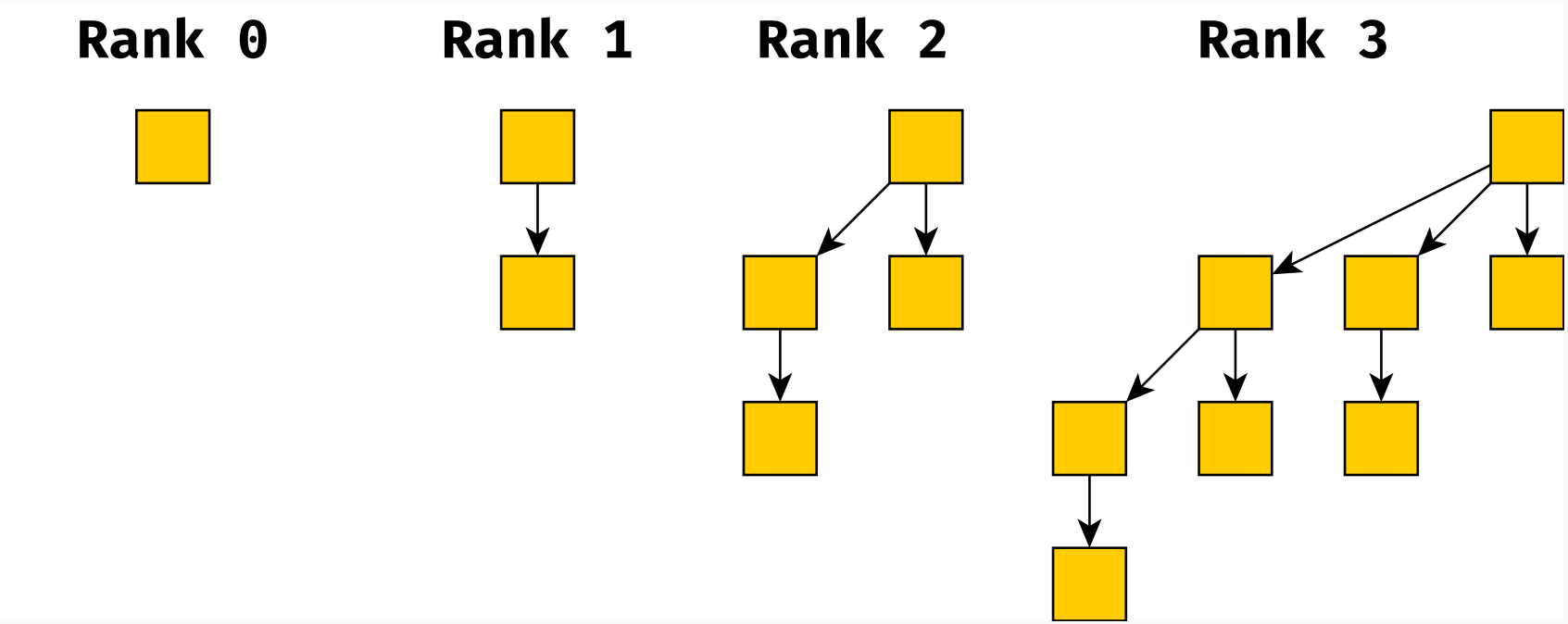
Михаил Беляев

10 декабря 2015

Биноминальная (двоичная) куча

- Состоит из *биноминальных деревьев*
- Деревья отвечают heap property
- Простейшее дерево состоит из одного элемента, он же минимум, с рангом 0
- Дерево ранга N состоит из двух деревьев ранга $(N - 1)$, причем левое подвешено к вершине правого
- В дереве ранга N находится ровно 2^N элементов

Биноминальное дерево



Биноминальная куча

- Биноминальная куча — это список биноминальных деревьев с «пробелами»
- На позиции с номером N всегда стоит дерево ранга N

Количество элементов в биноминальной куче

$$K = \Sigma 2^i$$

для всех i, которые заполнены деревьями

Знакомая формула?

Количество элементов в биноминальной куче

- Количество элементов равно числу, двоичным представлением которого является куча, если на позиции, на которых есть деревья, поставить 1, а где нет — 0
- Аналогия с двоичными числами на этом не заканчивается

Объединение двух биноминальных куч

- Объединение аналогично сложению двоичных чисел
  - На позиции, в которых в одной куче есть дерево, а в другой нет, помещается это дерево
  - Если в обоих кучах есть дерево, деревья объединяются в дерево размера $N + 1$ и переносятся в следующую позицию
- Таким образом, мы всегда сливаем деревья только с одинаковым рангом

Биномиальная куча: реализация

```haskell
data Tree a = Node Integer a ([Tree a])
type Heap a = [Tree a]

rank (Node r _ _) = r
value (Node _ v _) = v

link n1@(Node r1 x1 c1) n2@(Node r2 x2 c2) =
    if(x1 <= x2) then Node (r1+1) x1 (n2:c1)
                 else Node (r1+1) x2 (n1:c2)
```

Биномиальная куча: реализация

```
insTree t [] = [t]
insTree t ts@(t':ts') =
    if(rank t < rank t') then t:ts
                         else insTree (link t t') ts'

insert x ts = insTree (Node 0 []) ts

merge ts [] = ts
merge [] ts = ts
merge ts1@(t1:ts1') ts2@(t2:ts2') =
    if(rank t1 < rank t2)
    then t1 : merge ts1' ts2
    else if (rank t2 < rank t1)
         then t2 : merge ts1 ts2'
```

Биномиальная куча: реализация

```
removeMinTree [t] = (t, [])
removeMinTree (t:ts) =
    let (t', ts') = removeMinTree ts
    in if (value t < value t')
       then (t, ts)
       else (t', t:ts)

findMin h = value t where (t, _) = removeMinTree h
deleteMin h =
    let (Node _ x ts1, ts2) = removeMinTree h
    in merge (reverse ts1) ts2
```

Биноминальная куча: характеристики

- Поиск и удаление минимума за $O(K)$, где K это размер верхнего списка
- $K <= log(N + 1)$
- Хотелось бы доставать минимум за константу
- Можно хранить и восстанавливать минимальный элемент в куче

```haskell
data Heap a = Heap a [Tree a]
```

- После этого операция становится константной

Биноминальная куча: характеристики

- Вставка элемента работает за за $O(K)$, где K это размер верхнего списка
- $K <= log(N + 1)$
- На самом деле, вставка амортизированно константна

Воспользуемся методом физика:

- $\varphi = K$, вставка становится константной
- Можно убедиться, что все остальные операции не меняют своей сложности

Интересный факт: эта оценка остаётся корректной даже при персистентном использовании см. Chris Okasaki, «Purely functional data structures»

Проблема

- Большинство структур данных в ФП имеют амортизированно хорошую сложность
- Достаточно сложно использовать амортизированные структуры данных в персистентном режиме
    - Есть системы оценки для ленивых амортизированных структур, см. Окасаки

Обобщённое решение: - Сделать из операции с амортизированной сложностью операцию с наихудшей сложностью

Scheduling

- Применяется, когда
  1. Линейная операция имеет амортизированно константную сложность
  2. Эту операцию можно разбить на много маленьких константных операций
- Использует ленивые вычисления и техники, подобные динамическому программированию
- К сожалению, в общем случае это невозможно

Scheduling

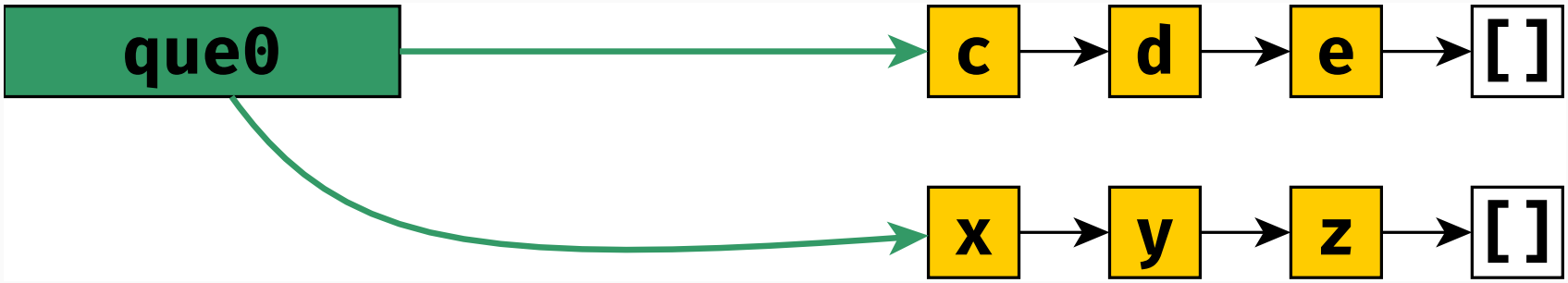
Также называется методом расписаний

Общая идея: добавляем в структуру данных ленивый список (или *поток*) *работ*, которые нужно будет выполнить в момент выплаты долга.

Пример

- Очередь из лекции про амортизацию
- Имеет линейную сложность за счёт того, что иногда требуется вызывать reverse
- reverse это монолитная операция
- Решение — лениво при каждой операции делать «кусочек» reverse и сохранять результат в промежуточном списке

Functional Queue



RealTime Queues

```
data Queue a = Queue [a] [a] [a]
emptyQ = Queue [] [] []
isEmptyQ (Queue [] _ _) = True
isEmptyQ _ = False
```

```
rotate (Queue [] (y:_) a) = y:a
rotate (Queue (x:xs) (y:ys) a) =
    x : rotate (Queue xs ys (y:a))

exec (Queue f r (x:s)) = (Queue f r s)
exec (Queue f r [])    = let f' = rotate (Queue f r [])
                         in Queue f' [] f'
```

```
push (Queue f r s) x = exec (Queue f (x:r) s)

top (Queue [] r s) = error "empty queue"
    (Queue (x:f) r s) = x
pop (Queue [] r s) = error "empty queue"
    (Queue (x:f) r s) = exec (Queue f r s)
```

Scheduling для биноминальной кучи

- Можно обеспечить worst-case $O(1)$ вставку в биноминальную кучу
- Для этого нужно хранить не только единицы бинарного представления, но и нули
  - Список в основе кучи становится ленивым
  - Расписание — это ленивый список таких ленивых списков

Проблема

- Есть структура данных-контейнер (например, обычный список)
- Нужно обеспечить быструю операцию конкатенации
- Большинство структур данных такого не позволяет

Catenable List

- Структура данных, поддерживающая все операции списка + конкатенацию
- Используется приём, в более общем виде известный как Structural Bootstrapping

Catenable List: реализация

Предположим, что у нас уже есть достаточно эффективная реализация очереди (см. предыдущие или эту лекцию)

```
data CatList q a = E | C a (q (CatList q a))
empty = E
isEmpty E = True
isEmpty _ = False
```

Catenable List: реализация

```
xs ++ E = xs
E ++ xs = xs
xs ++ ys = link xs ys

cons x xs = C x Queue.empty ++ xs
snoc xs x = xs ++ C x Queue.empty

head (C x q) = x
tail (C x q) =
    if Queue.isEmpty q then E else linkAll q
    where linkAll q = if Queue.isEmpty q' then t
                                          else link t (linkAll
          t = Queue.top q
          q' = Queue.pop q
```

Catenable List: реализация

```
link (C x q) s = C x (Queue.push q s)
```

Итог

- Рассмотрен ряд персистентных структур данных
- Пример метода расписаний над простой структурой данных
- Пример structural bootstrapping над простой структурой данных

Больше материалов можно найти в книге Криса Окасаки