Persistent data structures, episode 2: black magic

Михаил Беляев

29 ноября 2019 г.

Биноминальная (двоичная) куча

- Не путать с бинарной кучей
- Состоит из *биноминальных деревьев*
- Деревья отвечают heap property

Биноминальная (двоичная) куча

- Не путать с бинарной кучей
- Состоит из *биноминальных деревьев*
- Деревья отвечают heap property

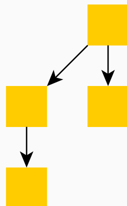
- Простейшее дерево состоит из одного элемента, он же минимум, с рангом 0
- Дерево ранга N состоит из двух деревьев ранга $(N-1)$, причем левое подвешено к вершине правого
- В дереве ранга N находится ровно 2^N элементов

Биноминальное дерево

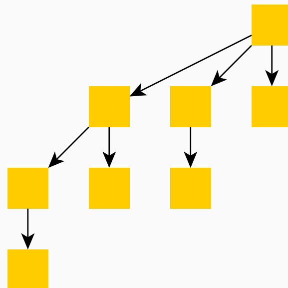
Rank 0

Rank 1

Rank 2



Rank 3



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, где на позиции, на которых есть деревья, стоит 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

Биноминальная 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)
```

Биноминальная 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
```

Биноминальная куча: реализация

```
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'
         else insTree (link t1 t2) (merge ts1' ts2')
```

Биноминальная 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

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

- После этого операция становится константной

Биноминальная куча: характеристики

- Вставка элемента работает 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

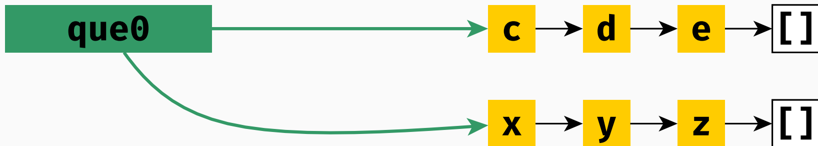
Также называется методом расписаний

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

- Но как можно делать reverse по кусочкам???
- На самом деле — несложно, мы просто в конец выходного списка добавляем *обещание* перевернуть входной список
- Но чтобы это работало, нужно делать это только тогда, когда их длины равны
    - Как за этим уследить?
- Ещё нужно обеспечить, что элементы активизируются в нужный нам момент, один раз
    - Но они могут быть в середине списка!

RealTime Queues

- Но как можно делать reverse по кусочкам???
- На самом деле — несложно, мы просто в конец выходного списка добавляем *обещание* перевернуть входной список
- Но чтобы это работало, нужно делать это только тогда, когда их длины равны
    - Как за этим уследить?
- Ещё нужно обеспечить, что элементы активизируются в нужный нам момент, один раз
    - Но они могут быть в середине списка!
- Решение: специальный список, содержащий «ещё ленивые» элементы и одновременно служащий счётчиком

RealTime Queues

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

Real-time Queues

```
-- Добавить в конец выходного списка обещание входного
rotate [] (y:_) a = y:a
rotate (x:xs) (y:ys) a =
    x : rotate (Queue xs ys (y:a))

-- Зафорсить ровно 1 элемент из расписания
exec (Queue f r (x:s)) = (Queue f r s)
exec (Queue f r [])    = let f' = rotate f r []
                           in Queue f' [] f'
```

Real-time Queues

```
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)
```

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 лекции)

```
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
```

Catenable List: реализация

```
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 q')
          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 «Purely functional data structures»

Какие ещё бывают персистентные структуры данных

Что ещё можно использовать в структурах данных, что нельзя в Haskell?

Какие ещё бывают персистентные структуры данных

Что ещё можно использовать в структурах данных, что нельзя в Haskell?

- Побочные эффекты
  - Стоп, а разве у нас не чистота по определению?
- Массивы
  - Стоп, а как это – персистентность и массивы?..

Going random!

- Существует целый класс структур данных, основанных на рандоме
- Мы уже много раз говорили, что `random()` — это нечистая функция
- Но нам-то нужен рандом только ради вероятностей!

Treap

- Одна из любимых структур данных автора лекции
- Treap = Tree + Heap
    - Он же декартово дерево
    - Он же дерамида
- Популяризирован среди олимпиадников неким Николаем Дуровым

Treap

Идея:

- В каждой ячейке два значения
  - По первому мы — бинарное дерево поиска
  - По второму мы — куча
- Куча очень хорошо балансируется сама => дерево поиска не нужно балансировать
- Но у нас же только одно значение?..

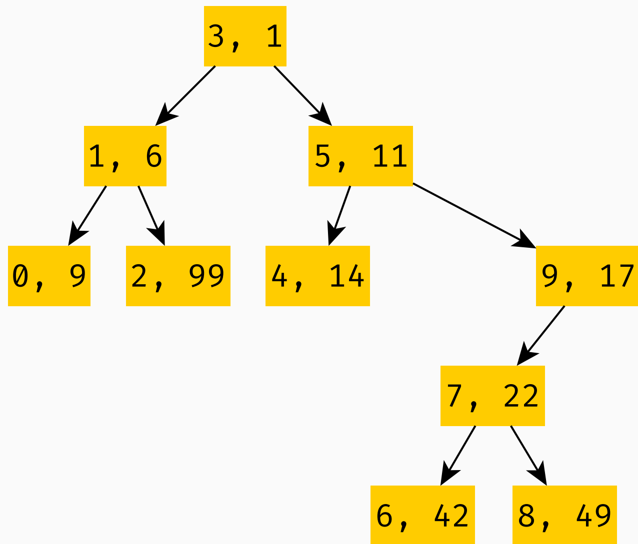
Treap

Идея:

- В каждой ячейке два значения
  - По первому мы — бинарное дерево поиска
  - По второму мы — куча
- Куча очень хорошо балансируется сама => дерево поиска не нужно балансировать
- Но у нас же только одно значение?..

- Вот тут и нужен рандом, второе берётся из генератора случайных чисел
- При идеальном распределении вероятность выйти по высоте за $4 * log(N)$ **крайне мала**

Treap



Treap: реализация

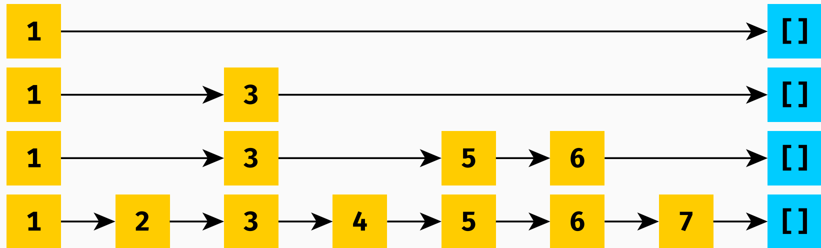
- Две операции: split и merge
- Первая «распиливает» дерево по элементу и получает два дерева
    - Игнорирует свойство хипа, но следит за БДП
- Вторая склеивает два дерева
    - Игнорирует сортированность, но следит за свойством хипа

Подробности?
Есть отличная статья на (о ужас) хабре

Skip list

- Все знают, что вместо дерева поиска можно использовать отсортированный массив
- Массивов нет
- Обычный связный список – это медленно
- Почему бы не вставить в него ссылки больше, чем на 1 элемент вперёд?
- А на сколько? А сколько ссылок?

Skip list



Skip list: поиск элемента

- Идём по первому списку, сравниваем элементы с текущим
- Если «перепрыгнули» — идём назад и спускаемся на уровень
- Повторить до готовности
- $O(log(N))$, если количество уровней тоже $O(log(N))$ и ссылки **оптимальны**

Skip list: как собственно организовать ссылки?

- Берём первый список
- **Кидаем монетку**
    - Если да – вставляем в него и все нижележащие
    - Если нет – спускаемся вниз и повторяем цикл
- При хорошем генераторе оптимальность ссылок гарантируется

Массивы

- Уже говорилось, что массивы копировать дорого
- А что, если это *мааааленькие* массивы?

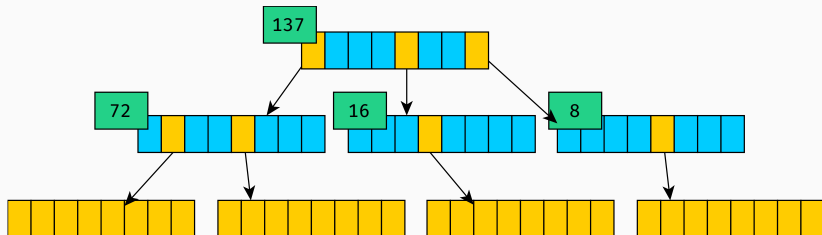
Индексное дерево

- aka Array-Mapped Trie (AMT)
- Берём бинарное дерево и делаем больше детей у каждой ноды
- Сколько? Ну, например, 32 или 64
- Массивы небольшого размера мы можем копировать
- Индексироваться будем по **отдельным битам** внутри значений

Индексное дерево

- Хранить все ссылки в массиве всё равно дороговато
- Те, которых нет, хранить не будем
- Рядом положим **маску** – число, в котором битами отмечено, где в массиве есть элементы

Индексное дерево



Индексное дерево: в чём профит?

- Копировать маленькие массивы дёшево
- Всё остальное отрабатывает персистентно, как и с обычным деревом
- Варьируя размер массива, можно регулировать персистентность vs высоту
- И размер массива, и максимальная высота – это **константы**

HAMT

- Очень хочется хеш-таблицы сделать персистентными
- Но хеш-таблица — это массив списков

HAMT

- Очень хочется хеш-таблицы сделать персистентными
- Но хеш-таблица — это массив списков
- Берём индексное дерево и в качестве индекса берём хеш

HAMT

- Hash array-mapped trie
- Строго говоря, везде константы => доступ константный, как и у **настоящей** хеш-таблицы
- На практике всё, конечно, хуже

Итог 2

- Рассмотрели несколько не очень функционально чистых, но важных персистентных СД
- Можно сказать, что на сегодня это — один из главных вкладов ФП в «общее дело» алгоритмов общего пользования

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