Чисто функциональные структуры данных

Без примеров кода (ну почти)

Косарев Дмитрий

матмех СПбГУ

22 ноября 2020 г.

Дата сборки: 22 ноября 2020 г.

Оглавление

Оглавление

Введение

Введение

Введение

Введение

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

Определение (Неизменяемые структуры данных (immutable data structures))

Которые с течением времени не изменяются ☺

Определение (Устойчивые структуры данных (persistent data structures))

Имеют доступ (не уничтожают) предыдущее своё состояние

Почти то же самое, только акцент смещён

Замечание

Так как старые узлы есть, то можно их использовать (share) в новой версии структуры данных

Определение (Неустойчивые структуры данных называются *эфемерными (ephemeral)*)

Оглавление

Самая простая структура данных: связный список

Определение (Связный список)

... вы же знаете, да?

Определение (Список)

Самая простая структура данных: связный список

Определение (Связный список)

... вы же знаете, да?

Определение (Список)

Структура данных, у которой для некоторой выбранной стороны (например, голова списка) добавление и удаление элементов работает из $O(1)$

Конкатенация связных списков в императивной среде

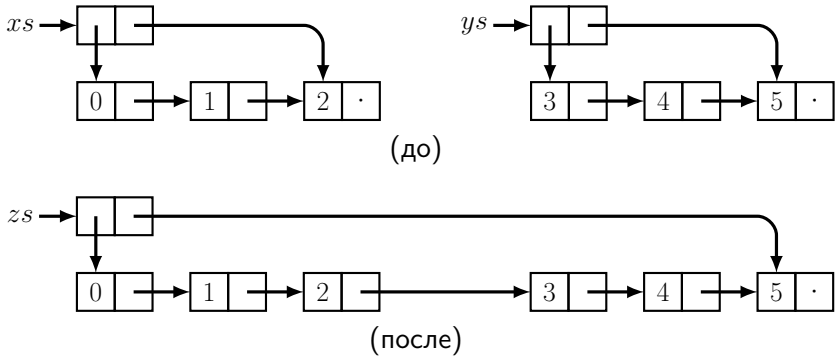


Рис.: Выполнение конкатенации списков xs и ys в императивной среде. Эта операция уничтожает списки-аргументы xs и ys (их использовать больше нельзя)

Конкатенация чисто функциональных списков

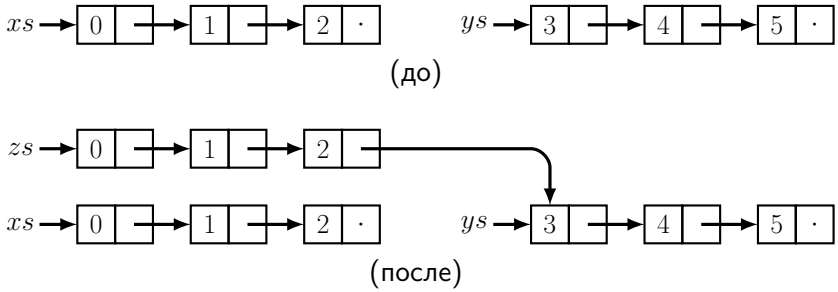


Рис.: Выполнение `zs = xs ++ ys` в функциональной среде. Заметим, что списки-аргументы `xs` и `ys` не затронуты операцией.

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

Как реализовать конкатенацию ++ списков xs и ys?

- Если `xs` пустой, то `ys` – ответ
- Иначе `xs` состоит из головы `h` и хвоста `tl`, а ответ – это прицепление головы `h` к хвосту `tl++ys`

Сложность: O(length(xs))

Как сложно обращаться к n-му элементу?
Ответ: O(n), что несколько печалит

Ассоциативность конкатенации

В теории конкатенация ассоциативна

$$(((a_1 +\!\!+ a_2) +\!\!+ a_3) +\!\!+ \ldots +\!\!+ a_n) \equiv (a_1 +\!\!+ (a_2 +\!\!+ (a_3 +\!\!+ (\ldots +\!\!+ a_n))))$$

На практике то, что слева будет работать сильно медленнее того, что справа.

Указание разработчикам

Иногда, для эффективной реализации надо переписывать алгоритмы, чтобы короткие списки конкатенировались с длинными. В идеале: всегда конкатенировать один элемент с длинным списком.

Оглавление

Методы амортизированного анализа

Стандартная нотация для сложности $O(\cdot)$ – оценка в худшем случае

Но мы можем себе позволить большую свободу:

- Будем делать $n + 1$ действий
- Большинство действий будет дешёвыми: $O(1)$
- Одно будет дорогим: например, $(O(n))$
- Стандартная асимптотическая сложность будет $(O(n))$
- Сложность *в среднем* при исполнении n операций (*амортизированная сложность*) вполне может быть $O(1)$ за одну операцию

Такая дополнительная степень свободы иногда позволяет спроектировать более простую и эффективную реализацию

Как оценивать амортизированную сложность? Метод банкира

Определение (*Текущие накопления* (accumulated savings))

Разница между общей текущей амортизированной стоимостью и общей текущей реальной стоимостью.

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

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

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.

- Каждая единица кредита должна быть выделена, прежде чем израсходована
- Нельзя расходовать кредит дважды

Таким образом, $\sum c_i \geq \sum \bar{c}_i$, а следовательно, как и требуется, $\sum a_i \geq \sum t_i$.

Оглавление

Чисто функциональные очереди

Интерфейс:

- `empty: queue → bool`
- `enqueue*int → queue` добавление в очередь
- `head: queue → int` посмотреть на головной элемент
- `tail: queue → queue` изъять головной элемент

Самая распространенная чисто функциональная реализация очередей представляет собой пару списков, `f` и `r`,

- `f` (front) содержит головные элементы очереди в правильном порядке,
- `r` (reversed) состоит из хвостовых элементов в обратном порядке

Например, очередь, содержащая целые числа `f=[1,2,3,4,5,6]`, может быть представлена списками `f=[1,2,3]` и `r=[6,5,4]`.

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.

Определение (Инвариант очереди)

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

Функция удаления из очереди `tail: queue` $\to$ `queue` принимает очередь как пару списков `f` и `r`

- Если `f` пуст – ошибка
- Если `f` состоит из одного элемента `x`, то возвращаем пару из `reverse(r)` и пустого списка
- Если `f` состоит из головного элемента `x` и хвоста `tl`, то возвращаем пару `reverse(r)` и списка `tl`

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

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

Чисто функциональная очередь и метод банкира

Определение (Инвариант)

Каждый элемент в хвостовом списке связан с одной единицей кредита.

Каждый вызов `enqueue` 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 $m + 1 - m = 1$.

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.

Оглавление

Ленивые вычисления

Идея (Ленивые вычисления)

Если надо что-то сделать, то выполняем, когда понадобится результат этого действия, т.е. откладываем вычисление на потом

Идея (Мемоизация ленивых вычислений)

Если вычисление понадобилось, то мы вычисляем и запоминаем результат. Когда оно понадобится кому-то ещё, вернем уже посчитанный результат

Ленивые списки (потоки)

Определение (Поток (stream))

Это список, где вычисления подсписков в нём отложены на потом, а вычисление элементов не отложено на потом (или не обязательно отложено)

С потоками легко описать, например, "все возможные натуральные числа"

Нотация

Добавление элемента x к хвосту xs: $\texttt{Cons}(x, xs)$
Пустой поток: $\texttt{Nil}$
Откладывание на потом f: f

Замечание

Потоки могут быть конечными, а могут быть бесконечными. Пока до конца не посчитаешь – не поймешь

Пример: фибоначчи

Пусть будет функция `zip` : $stream \times stream \to stream$, которая складывает потоки поэлементно

Поток чисел фибоначчи описывается так:

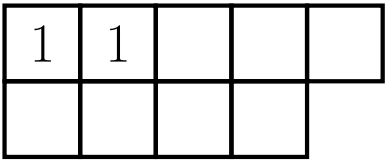
$$fibs \equiv \$\texttt{Cons}(1, zip(fibs, tail(fibs)))$$

Пример: фибоначчи

Пусть будет функция `zip` : $stream \times stream \to stream$, которая складывает потоки поэлементно

Поток чисел фибоначчи описывается так:

$$fibs \equiv \$\mathtt{Cons}(1, zip(fibs, tail(fibs)))$$

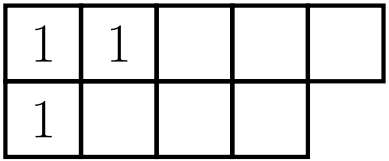


Пример: фибоначчи

Пусть будет функция `zip` : $stream \times stream \to stream$, которая складывает потоки поэлементно

Поток чисел фибоначчи описывается так:

$$fibs \equiv \$\mathtt{Cons}(1, zip(fibs, tail(fibs)))$$

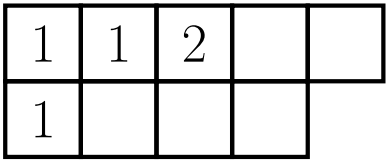


Пример: фибоначчи

Пусть будет функция `zip` : $stream \times stream \to stream$, которая складывает потоки поэлементно

Поток чисел фибоначчи описывается так:

$$fibs \equiv \$\mathtt{Cons}(1, zip(fibs, tail(fibs)))$$

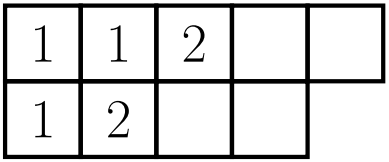


Пример: фибоначчи

Пусть будет функция `zip` : $stream \times stream \to stream$, которая складывает потоки поэлементно

Поток чисел фибоначчи описывается так:

$$fibs \equiv \$\mathtt{Cons}(1, zip(fibs, tail(fibs)))$$



Пример: фибоначчи

Пусть будет функция `zip` : $stream \times stream \to stream$, которая складывает потоки поэлементно

Поток чисел фибоначчи описывается так:

$$fibs \equiv \$\texttt{Cons}(1, zip(fibs, tail(fibs)))$$

1	1	2	3	
1	2			

и т.д.

Оглавление

Улучшаем: banker's queue

Замечание

Эта реализация будет работать за амортизированную $O(1)$ и быть устойчивой

1. Вместо списков используем потоки
2. Явно храним длины
3. Инвариант: $|f| > |r|$

В момент, когда потоки сравняются под длине конструируем новый f как $f +\!\!+ reverse(r)$.

Обращение списка

- не будет считаться слишком рано из-за ленивости
- не будет считаться дважды из-за мемоизации

Оглавление

Планирование (scheduling)

Проблема:

- Мы делаем n дешёвых вычислений
- Затем одно дорогое за $O(n)$
- И из-за этого обязаны заявлять только амортизированную сложность

Идея планирования (scheduling)

Разобьём дорогое вычисление на n составляющих константной стоимости. Каждый раз, выполняя дешёвое вычисление, будем по чуть-чуть выполнять дорогое.

Real-time queue

Вспоминаем очередь банкира: там мы полагались на вычисление `f ++ reverse (r)`
Теперь будет использовать для этого специальную функцию `rotate`

$$rotate(f, r, a) = f +\!\!+ reverse(r) +\!\!+ a$$

Третий параметр – аккумулятор, будет хранить частичные результаты `reverse(r)`.
Очевидно, что

$$rotate(f, r, \$Nil) = f +\!\!+ reverse(r)$$

Когда перестраиваем очередь?

Будем перестраивать очередь, когда $|R| = |F| + 1$. Это будет сохраняться на всём протяжении перестроения

База индукции

$$\begin{aligned} \mathit{rotate}(\$\mathtt{Nil}, \$\mathtt{Cons}(y, \$\mathtt{Nil}), a) &\equiv \$\mathtt{Nil} +\!\!+ \mathit{reverse}(\$\mathtt{Cons}(y, \$\mathtt{Nil})) +\!\!+ a \\ &\equiv \$\mathtt{Cons}(y, a) \end{aligned}$$

Переход

$$\begin{aligned} \mathit{rotate}(\$\mathtt{Cons}(x, f), \$\mathtt{Cons}(y, r), a) &\equiv \$\mathtt{Cons}(x, f) +\!\!+ \mathit{reverse}(\$\mathtt{Cons}(y, r)) +\!\!+ a \\ &\equiv \$\mathtt{Cons}(x, f +\!\!+ \mathit{reverse}(\$\mathtt{Cons}(y, r)) +\!\!+ a) \\ &\equiv \$\mathtt{Cons}(x, f +\!\!+ \mathit{reverse}(r) +\!\!+ \$\mathtt{Cons}(y, a)) \\ &\equiv \$\mathtt{Cons}(x, \mathit{rotate}(f, r, \$\mathtt{Cons}(y, a))) \end{aligned}$$

Как реализовать *rotate*?

Напоминаю, мы хотим заменить в очереди банкира $f +\!\!+ \mathit{reverse}(r)$ на $\mathit{rotate}(f, r, \texttt{\$Nil})$

Реализация $\mathit{rotate}(f, r, a)$

- Если $f \equiv \texttt{\$Nil}$ и $r \equiv \texttt{\$Cons}(y, tl)$, то возвращаем $\texttt{\$Cons}(y, a)$
- Если $f \equiv \texttt{\$Cons}(x, f)$ и $r \equiv \texttt{\$Cons}(y, r')$, то возвращаем $\texttt{\$Cons}(x, \mathit{rotate}(f', r', \texttt{\$Cons}(y, a)))$
- Другие случаи не возможны из-за инварианта $|R| = |F| + 1$

Замечание

rotate выполняет константное количество вычислений, при этом откладывая вычисление ещё одно вызова rotate от аргументов меньшей длины

Тип данных для очередей реального времени

- Новое поле S с типом "поток элементов", хранит расписание форсирования вычислений в F
  - S будет суффиксом F, таким что **все элементы впереди посчитаны до конца**[1]
  - Форсирование вычислений в F достигается форсирование головного элемента S
  - Инвариант $|S| \equiv |F| - |R|$
- R конструируется как есть, поэтому это просто список
- Не храним длины

[1] Это будет важно при оценке сложности

Нотация

Добавление x в обычный список xs записываем как $x :: xs$

Добавление в очередь $enqueue(f, r, s, x) \equiv queue(f, x :: r, S)$
Удаление из очереди: $tail(f, r, s) \equiv queue(f', r, s)$ при $f \equiv \$\mathtt{Cons}(x, f')$

Дополнительная функция псевдо-конструктор *queue* поддерживает инвариант $|S| \equiv |F| - |R|$, но в момент вызова аргументы удовлетворяют $|S| \equiv |F| - |R|+1$

Реализация *queue*:

- Если $s = \$\mathtt{Cons}(x, s)$, выдаем новую очередь из f, r и s (инвариант тривиально сохраняется)
- Если s пустой, то надо посчитать $f' \equiv rotate(f, r, \$\mathtt{Nil})$ и вернуть f' вместо f, $\mathtt{\$Nil}$ и f' вместо s

Про оценку сложности

Чтобы стоимость была константой необходимо

- Тратить константу на работу
- Форсировать вычислений только на константную стоимость

Оцениваем:

- Все конструирования, такие как `$Nil`, `$Cons`$(\cdot, \cdot)$, и тело *rotate* выполняют константу работы
- Вызов *rotate* форсирует голову фронта, но мы помним, что перед планированием *rotate* фронт уже был посчитан, так что это тоже константа работы

Итоги по очередям

Очередь\Операция	enqueue	head	tail
Банкира	$O(1)^*$	$O(1)^*$	$O(1)^*$
Real-time	$O(1)$	$O(1)$	$O(1)$

Амортизированные оценки обозначаются с*.

Оглавление

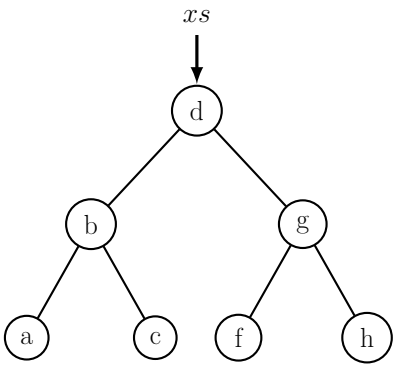
Деревья

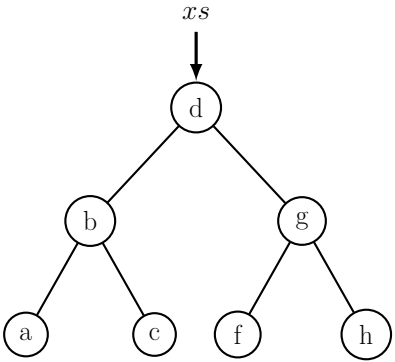
Хранят элементы по-разному

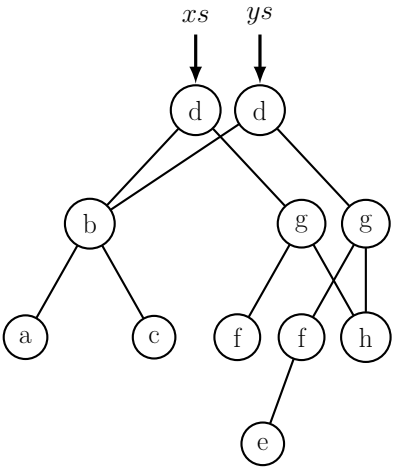
- Только в узлах
- Только в листьях
- И там, и там

По форме бывают разные

- Бинарные
- n-арные
- другие







Выполнение `ys` $\equiv$ `add("e",xs)`.

Для большинства деревьев путь, который надо изменить, содержит лишь небольшую долю узлов в дереве. Громадное большинство узлов будет находиться в совместно используемых поддеревьях.

Оглавление

Двоичные деревья поиска

Определение (Двоичные деревья поиска)

Двоичные деревья, в которых элементы симметричном (symmetric) порядке, то есть, элемент в каждом узле больше любого элемента в левом поддереве этого узла и меньше любого элемента в правом поддереве.

Двоичные деревья поиска: вставка

Например, пусть двоичное дерево поиска каких-то значений – это

- Либо лист без значений
- Либо узел, который хранит значение и двое других двоичных деревьев поиска

Функция `insert: tree*int → tree` вставки значения x в дерево:

- Вставка в пустое дерево тривиальна
- Иначе наше дерево – это узел из значения y и двух других поддеревьев `l` и `r`
  - Если $x < y$, то ответ – это дерево из y, `insert x l` и `r`
  - Если $x > y$, то ответ – это дерево из y, `l` и `insert x r`
  - Иначе не нужно добавлять, дерево из y, `l` и `r` – это ответ

Функция `member: tree*int → bool` пишется аналогично

Оглавление

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.

Определение (Красно-чёрные деревья)

Это двоичные деревья поиска особой структуры

- либо узел, состоящий из цвета, значения и двух поддеревьев
  - где цвет бывает либо красный, либо черный
- либо лист без значений (считается черным)

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.

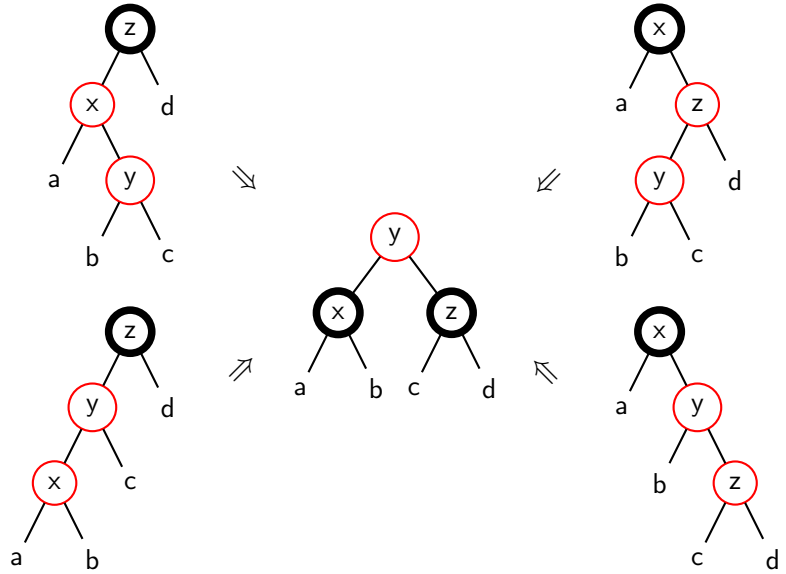
Вместе эти два инварианта гарантируют, что самый длинный возможный путь по красно-чёрному дереву, где красные и чёрные узлы чередуются, не более чем вдвое длиннее самого короткого, состоящего только из чёрных узлов.

Вставка делается нетривиально

Функция `insert: tree*int → tree` вставки значения x в дерево реализуется с помощью функции `balance`:

- Вставка в пустое дерево тривиальна
- Иначе вставляем в дерево, которое состоит из: значения y, цвета c и двух других поддеревьев `l` и `r`
  - Если $x = y$, то возвращаем дерево как есть
  - Если $x < y$, нужно вызвать `balance(c, insert x l, y, r)`
  - Если $x > y$, нужно вызвать `balance(c, l, y, insert x r)`

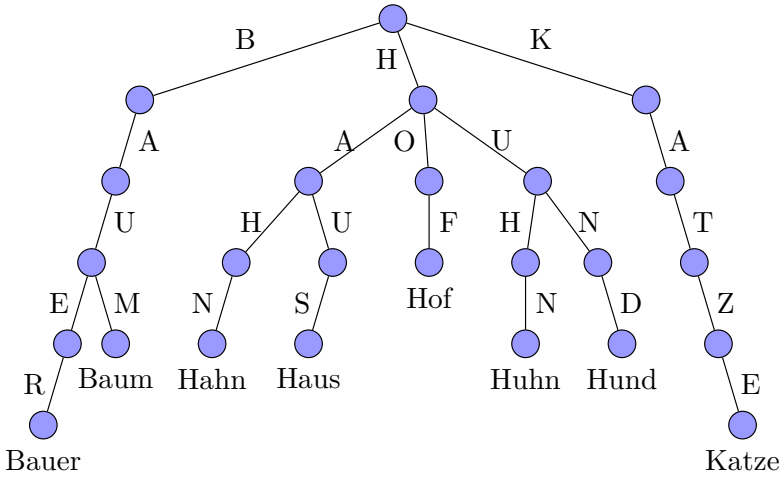
Функция `balance: color*tree*int*tree → tree` конструирует узел дерева, переупорядочивая, если нужно


z
x
d
a
y
b
c
x
a
z
y
d
b
c
y
x
z
a
b
c
d
z
y
d
x
c
a
b
x
a
y
b
z
c
d

Оглавление

Префиксные деревья (trie)

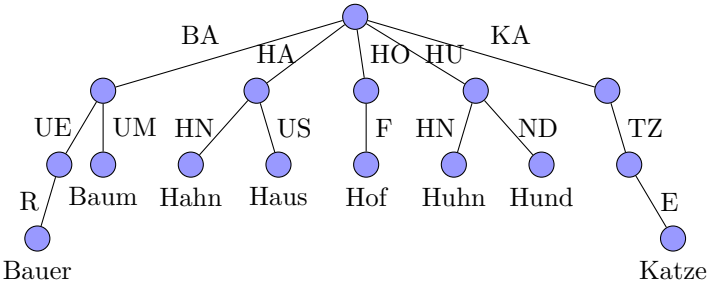
Желаем хранить последовательности так, чтобы начинающиеся с одного и того же были рядом



Префиксные деревья (trie)

Можно сжимать ребра, ускоряя доступ к листу, но увеличивая количество ветвей

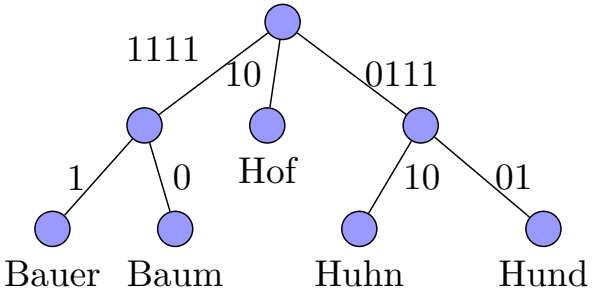
Если сжать их до максимума, то структура начнет напоминать массив



Префиксные деревья (trie), где ключи – числа

Конечное отображение (map)

- $63 = 11111_2 \mapsto$ Bauer
- $31 = 01111_2 \mapsto$ Baum
- $2 = 10_2 \mapsto$ Hof
- $71 = 100111_2 \mapsto$ Huhn
- $39 = 010111_2 \mapsto$ Hund



Важный апгрейд: HAMT (Hash Array Mapped Trie)

Оглавление

Оглавление

Левоориентированные (leftist) 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).

Операции:

- Вставка: `int`$\star$heap $\to$ heap
- Слияние: heap$\star$heap $\to$ heap
- Минимум: heap $\to$ `int` (если пустая – исключение)
- Удаление минимума: heap $\to$ heap (если пустая – исключение)

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

Элемент при каждой вершине не больше элементов в поддеревьях.

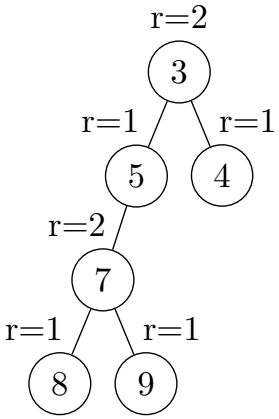
При таком упорядочении минимальный элемент дерева всегда находится в корне, *но это не дерево поиска*

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

Самого правый путь от данного узла до пустого

Определение (Ранг)

Ранг узла — длина его правой периферии.



Левоориентированные кучи

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.

Представление левоорентированных куч

Двоичные деревья, снабженные информацией о ранге, т.е.

- В листьях ничего нет (и ранг всегда 0)
- В узлах: хранимый элемент, два поддерева и ранг (int)

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

Слияние

Идея

Достаточно слить их правые периферии как упорядоченные списки, а затем вдоль полученного пути обменивать местами поддеревья при вершинах, чтобы восстановить свойство левоориентированности.

Функция `merge: heap*heap → heap`

- Если одна из куч пустая – возвращаем другую
- Если имеем два узла: h1, состоящий из `(x,l1,r1)` и h2 — `(x,l2,r2)`
  - При $x \leqslant y$ возвращаем `makeT(x,l1, merge(r1, h2))`
  - Иначе `makeT(y,l2, merge(h1, r2))`

Функция `makeT: int*heap*heap → heap` принимает `(x,l,r)`:

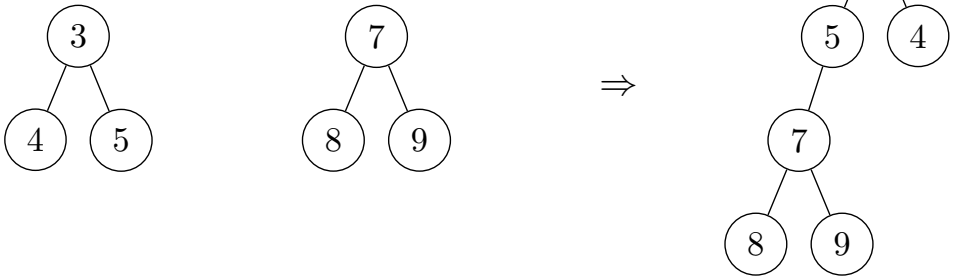
- Если `rank(l)` $\geqslant$ `rank(r)` то строим дерево из `(1+rank(b), x, a, b)`
- Иначе как `(1+rank(a), x, b, a)`

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

Итого: сложность левоориентированных куч

- Слияние ($O(log\ n)$)
- Минимум – заглядывание в корень ($O(1)$)
- Вставка – это слияние с одноэлементным деревом ($O(log\ n)$)
- Удаление – слияние левого поддерева с правым ($O(log\ n)$)

Пример: слияние двух левоориентированных куч



Оглавление

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 ухищрений (избавление от амортизации), можно заставить `insert` и `merge` выполняться за время $O(1)$.

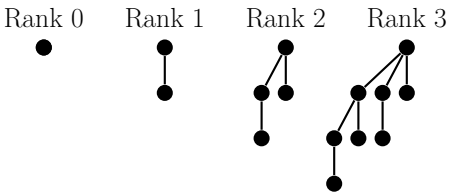
Биномиальные кучи строятся из более простых объектов, называемых биномиальными деревьями.

Биномиальные деревья. Пример

Определение (Биномиальные деревья (опр. 1, индуктивное))

- 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.

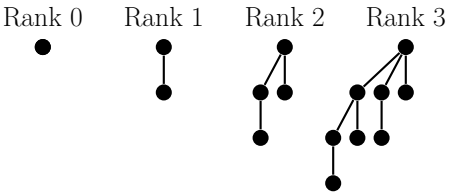


Биномиальные деревья. Пример

Существует второе, эквивалентное первому, определение биномиальных деревьев, которым иногда удобнее пользоваться.

Определение (Биномиальные деревья (опр. 2))

Биномиальное дерево ранга r представляет собой узел с r потомками $t_1 \dots t_r$, где каждое t_i является биномиальным деревом ранга $(r - i)$.



Реализация биномиальных деревьев:

- Храним узлы с рангом, значением, список деревьев-потомков
- Потомки хранятся в порядке *убывания*[2] ранга
- Элементы подвешиваются согласно "порядку кучи" (деревья с большими корнями подвешиваются к узлам с меньшими)

Элементы хранятся в порядке кучи. Чтобы сохранять этот порядок кучи, мы всегда подцепляем дерево с большим корнем к узлу с меньшим.

Функция `link: heap×heap → heap` принимает дерево `t1` из `(r,x1,c1)` и дерево `t2` из `(r2,x2,c2)`

- Если `x1<x2` строим дерево из `(r+1, x1, t2::c1)`
- Иначе `(r+1, x2, t1::c1)`
- Инвариант: связываем деревья только с одинаковым рангом: **`assert`**`(r1≡r2)`

[2] Это будет важно при удалении

Реализация биномиальной кучи:

- Список биномиальных деревьев с "порядком кучи"
- Отсортированный в порядке *возрастания*[3] рангов

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.

[3] Это будет важно при удалении

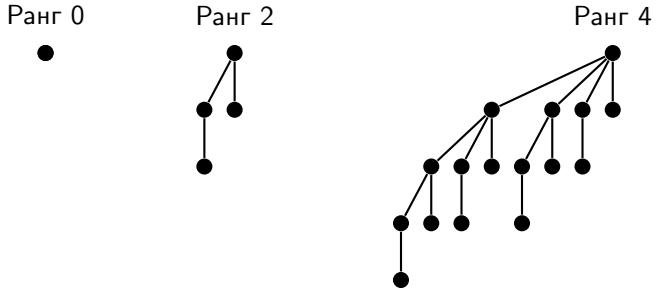


Рис.: Число $21_{10} = 10101_2$, и поэтому биномиальная куча размера 21 содержит одно дерево ранга 0, одно ранга 2, и одно ранга 4 (размерами, соответственно, 1, 4 и 16).

Вставка аналогично инкременту

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.

Функция `insTree: tree * tree list → tree`

- Вставка в пустой список возвращает одноэлементный
- Нужно посмотреть на вставляемое дерево t и на головное дерево `t2` из списка `ts`
  - Если `rank(t)<rank(t2)`, то вернуть `t :: ts`
  - Иначе вернуть `insTree(link(t,t2), tail(ts))`

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

Слияние – 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.

Функция `merge: heap * heap → heap`

- Если одна из куч пустая, то возвращаем другую
- Иначе у нас есть `ts1`$\equiv$`t1 :: ts1'` и `ts2`$\equiv$`t2 :: ts2'`
- если `rank(t1)<rank(t2)`, выдаем `t1 :: merge (ts1', ts2)`
- если `rank(t2)<rank(t1)`, выдаем `t2 :: merge (ts1, ts2')`
- если равны, то `insTree(link(t1,t2), merge (ts1', ts2')`

Операции работы с минимумом

Дополнительная функция `removeMinTree: tree list → tree * (tree list)` удаляет из списка дерево с минимальным значением в корне, и выдает это дерево и остаток списка

Функция `findMin` – вызвать `removeMinTree` и заглянуть в корень полученного дерева

Функция `delete` – вызвать `removeMinTree`, взять список потомков полученного дерева, перевернуть и слить с учетом рангов со вторым списком

Куча\Операция	findMin	deleteMin	insert	merge
Leftist	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Биномиальная	$O(1)$[4]	$O(\log n)$	$O(\log n)$	$O(\log n)$
Бин. + амортизация			$O(1)^*$	
Бин. + расписания			$O(1)$	
bootstrapped	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$

Пропуск означает, что точную оценку забыли подсмотреть в литературе
Амортизированные оценки обозначаются с*.

[4] Изложено $O(\log n)$, но можно сделать $O(1)$

Оглавление

Оглавление

Общие замечания по упражнениям

Если явно не оговорено иное, то...

Предлагается реализовать устойчивую структуру данных на вашем любимом языке обывательском (C#, C, etc), и на каком-нибудь функциональном языке (OCaml,Haskell,F#, Scala 3), а затем сравнить размер/сложность двух реализаций. Ожидаются реализации в виде чистых функций (ну может понадобится присваивание для эмуляции мемоизации, в остальном – едва ли)

Обозначение A/B/C говорит, что за решение на обывательском языке будет начислено A баллов, на классическом функциональном B; C баллов начисляется дополнительно, если обучающийся может сравнить реализации, указать на преимущества и недостатки одной и второй, и т.д.

Упражнения на ленивость I

Упражнение (?/?/?)

По аналогии с вычислением последовательности фибоначчи, сделайте вычисление простых чисел решетом Эратосфена

Упражнение (?/?/?)

Дан поток потоков чисел *xss*. Функция `merge` должно объединить это всё в один поток чисел.

Ограничение: $\forall(i < \infty)\forall(j < \infty) \quad \big((xss[i][j] \equiv n) \quad \Longrightarrow \quad \exists(k < \infty) \quad (merge(xss)[k] \equiv n)\big)$

Упражнения на ленивость II

Упражнение (?/?/?)

Дано дерево с числами только в листьях. Построить новое дерево, где все числа предыдущего дерева заменены на минимум от этих чисел. Ограничение: за один проход.

Замечание

Крайне рекомендуется использовать язык, где все вычисления ленивы по умолчанию (например, Haskell)

Упражнения на очереди I

Упражнение (?/?/?)

Реализуйте чисто функциональную очередь

Упражнение (?/?/?)

Реализуйте очередь банкира

Упражнение (?/?/?)

Реализуйте очередь реального времени

Упражнения на очереди I

Упражнение (?/?/?)

Реализуйте чисто функциональную очередь

Упражнение (?/?/?)

Реализуйте очередь банкира

Упражнение (?/?/?)

Реализуйте очередь реального времени

Упражнения на кучи I

Упражнение (?/?/?)

Реализуйте левоориентированную кучу

Упражнение (?/?/?)

Реализуйте weight-biased левоориентированную кучу (упражнение в книге 3.4)

Упражнение (?/?/?)

Реализуйте биномиальную кучу, храня аннотации ранга реже (упражнение в книге 3.6)

Упражнение (?/?/?)

Реализуйте биномиальную кучу с явным минимумом (упражнение в книге 3.7)

Упражнения на деревья I

Упражнение (?/?/?)

Реализуйте красно-черное дерево, где балансировка делает меньше проверок (упражнение в книге 3.10)

Упражнение (?/?/?)

Реализуйте префиксное дерево

Упражнение (?/?/?)

Реализуйте HAMT

Конец.

Подробнее в книге К.Окасаки "Чисто функциональные структуры данных".

Ссылки I


Книга "Чисто функциональные структуры данных"
Chris Okasaki


Heap - Leftist Tree


«Immutability changes everything»
Pat Helland


Immutable data structures for functional JS
Anjana Vakil