Фибоначчиева куча

Фибоначчиева куча (англ. *Fibonacci heap*) — структура данных, отвечающая интерфейсу приоритетная очередь. Эта структура данных имеет меньшую амортизированную сложность, чем такие приоритетные очереди как биномиальная куча и двоичная куча. Изначально эта структура данных была разработана Майклом Фридманом[1] и Робертом Тарьяном[2] при работе по улучшению асимптотической сложности алгоритма Дейкстры. Свое название Фибоначчиева куча получила из-за использования некоторых свойств чисел Фибоначчи[3] в потенциальном анализе этой реализации.

Содержание

- 1 Структура
- 2 Реализация
  - 2.1 Создание кучи
  - 2.2 Вставка элемента
  - 2.3 Получение минимального элемента
  - 2.4 Соедининение двух куч
  - 2.5 Удаление минимального элемента
    - 2.5.1 Прорежение деревьев
  - 2.6 Уменьшение значения элемента
    - 2.6.1 Вырезание
    - 2.6.2 Каскадное вырезание
  - 2.7 Удаление элемента
- 3 Время работы
  - 3.1 Потенциал
  - 3.2 Создание кучи
  - 3.3 Вставка элемента
  - 3.4 Получение минимального элемента
  - 3.5 Соедининение двух куч
  - 3.6 Удаление минимального элемента
  - 3.7 Уменьшение значения элемента
  - 3.8 Удаление элемента
  - 3.9 Итоговая таблица
- 4 Недостатки и достоинства
- 5 См. также
- 6 Примечания
- 7 Источники информации

Структура

Фибоначчиева куча — набор из подвешенных деревьев удовлетворяющих свойству: каждый предок не больше своих детей(если дерево на минимум). Это означает, что минимум всей кучи это один из корней этих деревьев. Одним из главных преимуществ Фибоначчиевой кучи — гибкость её структуры из-за того, что на деревья не наложены никакие ограничения по форме. Например, Фибоначчиева куча может состоять хоть из деревьев в каждом из которых по одному элементу. Такая гибкость позволяет выполнять некоторые операции лениво, оставляя работу более поздним операциям. Далее будут даны некоторые определения, которые понадобятся в дальнейшем.

Определение:

Степень вершины (англ. *degree*) — количество детей данной вершины. Далее будем обозначать как $degree(x)$, где x это вершина.

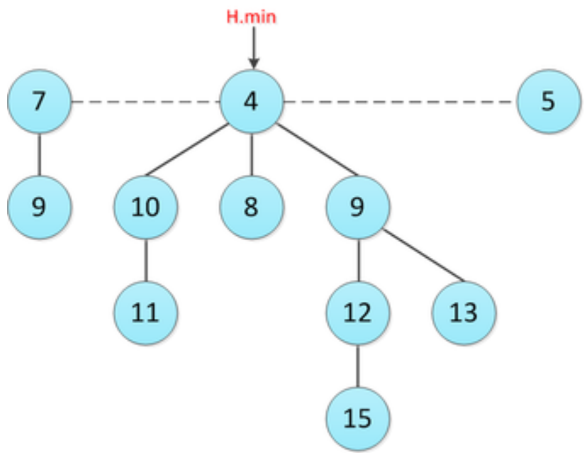
Определение:

Степень кучи (англ. *degree*) — наибольшая степень вершины этой кучи. Далее будем обозначать как $degree(H)$, где H

это куча.

Реализация

Для возможности быстрого удаления элемента из произвольного места и объединением с другим списком будем хранить их в циклическом двусвязном списке. Также будем хранить и все уровни поддерева. Исходя из этого структура каждого узла будет выглядеть вот так.



Пример фибоначчиевой кучи

```
struct Node
   int key      // ключ
   Node parent  // указатель на родительский узел
   Node child   // указатель на один из дочерних узлов
   Node left    // указатель на левый узел того же предка
   Node right   // указатель на правый узел того же предка
   int degree   // степень вершины
   boolean mark // был ли удален в процессе изменения ключа ребенок этой вершины)
```

Также стоит упомянуть, что нам нужен указатель только на одного ребенка, поскольку остальные хранятся в двусвязном списке с ним. Для доступа ко всей куче нам тоже нужен всего один элемент, поэтому разумно хранить именно указатель на минимум кучи (он обязательно один из корней), а для получения размера за константное время будем хранить размер кучи отдельно.

```
struct fibonacciHeap
   int size // текущее количество узлов
   Node min // указатель на корень дерева с минимальным ключом
```

Создание кучи

Инициализация кучи.

```
function buildHeap:
   min
   size = 0
```

Вставка элемента

Данная операция вставляет новый элемент в список корней правее минимума и при необходимости меняет указатель на минимум кучи.

```
function insert(x: int):
   Node newNode                // создаем новый узел
   newNode.key = x             // инициализируем ключ нового узла
   if size = 0                // если куче нет элементов, то только что добавленный минимальный
       min = newNode
       min.left = newNode
       min.right = newNode
   else                        // иначе аккуратно меняем указатели в списке, чтобы не перепутать указатели
       Node prevRight = min.right
       min.right = newNode
       newNode.left = min
       newNode.right = prevRight
       prevRight.left = newNode
   if newNode.key < min.key
       min = newNode           // меняем указатель на минимум, если надо
   newNode.parent
   size++                      // не забываем увеличить переменную size
```

Получение минимального элемента

Получение минимума всей кучи.

```
int getMin:
   return min.key
```

Соедининение двух куч

Для сливание двух Фибоначчиевых куч необходимо просто объединить их корневые списки, а также обновить минимум новой кучи, если понадобится. Вынесем в вспомогательную функцию $unionLists$ логику, объединяющую два списка вершины, которых подаются ей в качестве аргументов.

```
function unionLists(first: Node, second: Node):
   Node L = first.left               // аккуратно меняем указатели местами указатели
   Node R = second.right
   second.right = first
   first.left = second
   L.right = R
   R.left = L
```

Сливаем два корневых списка в один и обновляем минимум, если нужно.

```
function merge(that: fibonacciHeap):
   if that.size = 0              // если вторая куча пуста, нечего добавлять
       return
   if size = 0                   // если наша куча пуста, то результатом будет вторая куча
       min = that.min
       size = that.size
   else
       unionLists(min, that.min)   // объединяем два корневых списка
       size += that.size
   if min  or (that.min  and that.min < min) // если минимум кучи изменился, то надо обновить указатель
       min = that.min
```

Удаление минимального элемента

Первая рассматриваемая операция, в ходе которой значительно меняется структура кучи. Здесь используется вспомогательная процедура $consolidate$, благодаря которой собственно и достигается желанная амортизированная оценка. В данном случае $min = \varnothing$ не рассматривается и считается нарушением предусловий $deleteMin$

```
int deleteMin:
   Node prevMin = min
   unionLists(min, min.child)    // список детей min объединяем с корневым
   Node L = min.left             // аккуратно удаляем min из списка
   Node R = min.right
   L.right = R
   R.left = L
   if prevMin.right = prevMin   // отдельно рассмотрим случай с одним элементом
       min
       return
   min = min.right               // пока что перекинем указаиель min на правого сына, а далее consolidate() скорректирует min в проце
сce выполнения
   consolidate()
   size--
   return prevMin.key
```

Прорежение деревьев

Данная процедура принимает кучу и преобразует ее таким образом, что в корневом списке остается не более $degree(H) + 1$ вершин.

Для этого возьмем массив списков указателей на корни деревьев $A[0 \ldots D[H]]$, где $degree(H)$ — максимальная степень вершины в текущем корневом списке.

Затем происходит процесс, аналогичный слиянию биномиальных куч: добавляем поочередно каждый корень, смотря на его степень. Пусть она равна d. Если в соответствующей ячейке A еще нету вершины, записываем текущую вершину туда. Иначе подвешиваем одно дерево к другому, и пытаемся также добавить дерево, степень корня которого уже равна $d + 1$. Продолжаем, пока не найдем свободную ячейку. Подвешиваем мы его следующим образом: в корневой список добавляем корень минимальный из тех двух, а корень другого добавляем в список детей корневой вершины. Чтобы лучше понять этот процесс лучше воспользоваться визуализатором (https://www.cs.usfca.edu/~galles/visualization/FibonacciHeap.html)

```
function consolidate:
   A = Node[]
   A[min.degree] = min                    // создаем массив и инициализируем его min
   Node current = min.right
   while A[current.degree]  current     // пока элементы массива меняются
       if A[current.degree]          // если ячейка пустая, то положим в нее текущий элемент
           A[current.degree] = current
           current = current.right
       else                               // иначе подвесим к меньшему из текущего корня и того, который лежит в ячейке другой
           Node conflict = A[current.degree]
           Node addTo, adding
           if conflict.key < current.key
               addTo = conflict
               adding = current
           else
               addTo = current
               adding = conflict
           unionLists(addTo.child, adding)
           adding.parent = addTo
           addTo.degree++
           current = addTo
       if min.key > current.key          // обновляем минимум, если нужно
           min = current
```

Пример

Изначально добавляем в нашу кучу 7 элементов $56, 22, 84, 32, 85, 15, 16$. После этого выполним операцию извлечения минимума:

Начальное состояние кучи

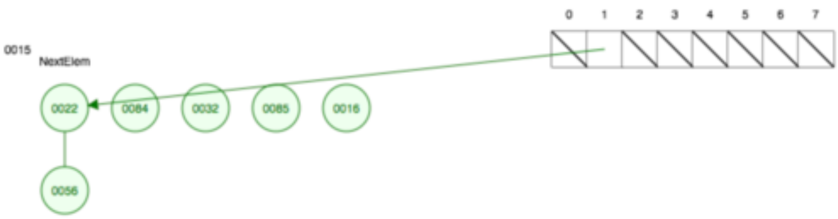
- Удалим минимальный элемент из циклического корневого списка и заведем массив A для дальнейшего прорежения.

Удаление мимимума и создание массива

- Начнем процесс протяжения с первого элемента — 56. Его степень равна 0 поэтому запишем его адрес в нулевую ячейку массива.

Состояние массива после первой итерации

- Следующий элемент 22 тоже имеет степень 0. Возникает конфликт, который решается подвешиванием к меньшему корню большего. То есть к 22 подвешиваем 56 и увеличиваем степень 22 на 1. В итоге степень 22 равна 1. Записываем адрес 22 по индексу 1 в массив.

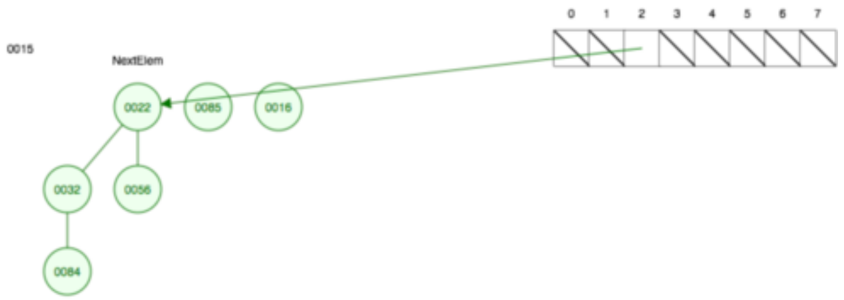


Состояние после второй итерации

- Делаем тоже самое, что и на предыдущих итерациях, но теперь объединяем 32 и 84

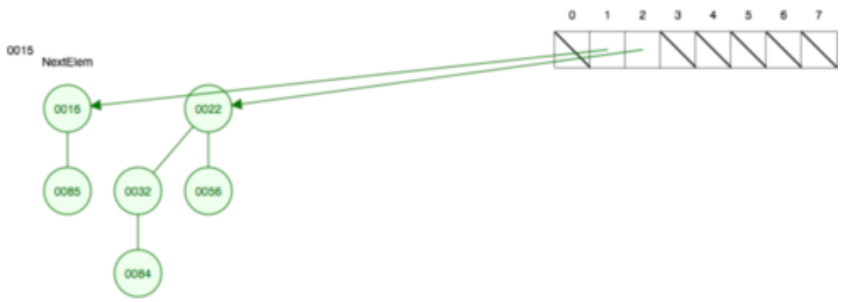
Состояние после четвертой итерации

- Теперь у нас два элемента со степенью 1 в корневом списке. Объединим их подвесив к меньшему корню — 22, больший — 32. Теперь степень 22 равна 2, запишем на 2 позицию массива обновленное значение.



Состояние после пятой итерации

- Ну и наконец аналогично объедений последние два элемента.



Финальное состояние кучи

Уменьшение значения элемента

Основная идея: хотим, чтобы учетная стоимость данной операции была $O(1)$. Было бы хорошо, чтобы вершина не всплывала до корня, и тогда дерево не придется сильно перестраивать. Для этого при удобном случае будем вырезать поддерево полностью и перемещать его в корневой список. Итак, сам алгоритм:

1. Проверяем, если новое значение ключа все же не меньше значения ключа родителя, то все хорошо, и мы выходим.
2. Иначе, вырезаем дерево с текущей вершиной в корневой список, и производим каскадное вырезание родителя.

```
function decreaseKey(x: Node, newValue: int):
   if newValue > x.parent.key  // если после изменения структура дерева сохранится, то меняем и выходим
       x.key = newValue
       return
   Node parent = x.parent     // иначе вызываем cut и cascadingCut
   cut(x)
   cascadingCut(x.parent)
```

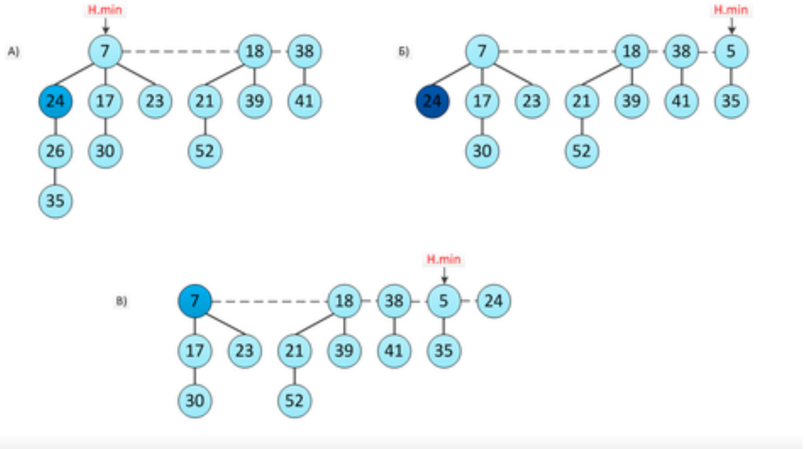
Вырезание

При вырезании вершины мы удаляем ее из списка детей своего родителя, уменьшаем степень ее родителя ($x.p.degree$) и снимаем пометку с текущей вершины ($x.mark = false$).

```
function cut(x: Node)
   Node L = x.left
   Node R = x.right
   R.left = L                 // аккуратно удаляем текущую вершину
   L.right = R
   x.right = x
   x.left = x
   x.parent.degree--
   if x.parent.child = x    // чтобы родитель не потерял ссылку на сыновей проверяем:
       if x.right = x.      // если узел который мы вырезаем содержится в родителе, то меняем его на соседний
           x.parent.child   // иначе у родителя больше нет детей
       else
           x.parent.child = x.right
   x.parent
   unionLists(min, x)        // вставляем наше поддерево в корневой список
```

Каскадное вырезание

Перед вызовом каскадного вырезания нам известно, удаляли ли ребенка у этой вершины. Если у вершины до этого не удаляли дочерний узел ($x.mark = false$), то мы помечаем эту вершину ($x.mark = true$) и прекращаем выполнение операции. В противном случае применяем операцию cut для текущей вершины и запускаем каскадное вырезание от родителя.



Пример каскадного вырезания

```
function cascadingCut(x: Node)
   while x.mark = true  // пока у нас помеченые вершины вырезаем их
       cut(x)
       x = x.parent
   x.mark = true         // последнюю вершину нужно пометить — у нее удаляли ребенка
```

Пример

Рисунок иллюстрирует пример каскадного вырезания:

- Изначально, куча состояла из 3 фибоначчиевых деревьев. У вершины с ключом 24 отсутствует 1 ребенок.
- Уменьшаем ключ 26 до 5 и делаем операцию cut этого дерева. Получаем кучу с 4 деревьями и новым минимумом. Но у вершины с ключом 24 был удален второй ребенок, поэтому запускам операцию $\mathrm{cascadingCut}$ для этой вершины: вырезаем ее, помещаем в корневой список и помечаем ее родителя.
- У вершины с ключом 7 удален лишь один ребенок, поэтому операция $\mathrm{cascadingCut}$ от нее не запускается. В итоге, получаем кучу, состоящую из 5 фибоначчиевых деревьев.

Удаление элемента

Удаление вершины реализуется через уменьшение ее ключа до $-\infty$ и последующим извлечением минимума.

```
function delete(x: Node)
   decreaseKey(x, )
   deleteMin()
```

Время работы

Потенциал

Для анализа производительности операций введем потенциал для фибоначчиевой кучи как $\Phi = trees + 2 * marked$, где $trees$ — количество элементов в корневом списке кучи, а $marked$ — количество вершин, у которых удален один ребенок (то есть вершин с пометкой $x.mark = true$). Договоримся, что единицы потенциала достаточно для оплаты константного количества работы.

Создание кучи

Очевидно, что реальное время работы — $O(1)$.

Вставка элемента

Для оценки амортизированной стоимости операции рассмотрим исходную кучу H и получившуюся в результате вставки нового элемента кучу H'. $trees(H') = trees(H) + 1$ и $marked(H') = marked(H)$. Следовательно, увеличение потенциала составляет $(trees(H) + 1 + 2 * marked(H)) - (trees(H) + 2 * marked(H)) = 1$. Так как реальное время работы составляет $O(1)$, то амортизированная стоимость данной операции также равна $O(1)$.

Получение минимального элемента

Истинное время работы — $O(1)$.

Соедининение двух куч

Реальное время работы — $O(1)$. Амортизированное время работы также $O(1)$, поскольку, при объединении двух куч в одну, потенциалы обеих куч суммируются, итоговая сумма потенциалов не изменяется, $\Phi_{n+1} - \Phi_n = 0$.

Удаление минимального элемента

Для доказательства времени работы этого алгоритма нам понадобится доказать несколько вспомогательных утверждений.

Лемма:

Для всех целых $n \geqslant 2$

$F_n = 1 + \sum\limits_{i=0}^{n-2} F_i$, где F_n — n-ое число Фибоначчи, определяемое формулой:

$$F_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F_{n-1} + F_{n-2}, & n \geqslant 2 \end{cases}$$

Доказательство:

$\triangleright$

Докажем лемму по индукции:

при $n = 2$

$F_2 = 1 + \sum\limits_{i=0}^{0} F_i = 1 + 0 = 1$, что действительно верно.

По индукции предполагаем, что $F_{n-1} = 1 + \sum\limits_{i=0}^{n-3} F_i$. Тогда

$$F_n = F_{n-1} + F_{n-2} = 1 + \sum_{i=0}^{n-3} F_i + F_{n-2} = 1 + \sum_{i=0}^{n-2} F_i$$

$\triangleleft$

Лемма:

Фибоначчиево дерево порядка n содержит не менее F_n вершин.

Доказательство:

$\triangleright$

Докажем это утверждение по индукции. Пусть s_n — минимальный размер фибоначчиева дерева порядка n.

При $n = 0$

$s_0 = 1 > F_0$.

При $n = 1$

$s_1 = 1 = F_1$.

Предположим по индукции, что для всех $i < n$ $s_i \geqslant F_i$. Пусть в нашем дереве удалено поддерево порядка $n - 1$. Тогда

$$s_n = 1 + \sum_{i=0}^{n-2} s_i \geqslant 1 + \sum_{i=0}^{n-2} F_i$$

Но по предыдущей лемме :

$1 + \sum\limits_{i=0}^{n-2} F_i = F_n$. Следовательно, $s_n \geqslant F_n$

$\triangleleft$

Лемма:

$F_n = O(\varphi^n)$, где $\varphi = \frac{1+\sqrt{5}}{2}$

Доказательство:

▷

Для начала докажем, что $F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$

Используем для этого математическую индукцию.

При $n = 0$

$F_0 = \frac{\varphi^0 - (-\varphi)^0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = 0$, что верно.

При $n = 1$

$F_1 = \frac{\varphi^1 - (-\varphi)^{-1}}{\sqrt{5}} = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}) = \frac{2\sqrt{5}}{2\sqrt{5}} = 1$, что также верно.

По индукции предполагаем, что $F_{n-1} = \frac{\varphi^{n-1} - (-\varphi)^{1-n}}{\sqrt{5}}$ и $F_{n-2} = \frac{\varphi^{n-2} - (-\varphi)^{2-n}}{\sqrt{5}}$. Тогда

$$F_n = F_{n-1} + F_{n-2} = \frac{\varphi^{n-1} - (-\varphi)^{1-n}}{\sqrt{5}} + \frac{\varphi^{n-2} - (-\varphi)^{2-n}}{\sqrt{5}} =$$

$$= \frac{1}{\sqrt{5}} (\varphi^{n-1} - (-\varphi)^{1-n} + \varphi^{n-2} - (-\varphi)^{2-n}) = \frac{1}{\sqrt{5}} (\varphi^n(\varphi^{-1} + \varphi^{-2}) - (-\varphi)^{-n}(-\varphi + \varphi^2))$$

Подставив вместо φ его значение, нетрудно убедится, что $\varphi^{-1} + \varphi^{-2} = -\varphi + \varphi^2 = 1$

Поскольку $\left|(-\varphi)^{-1}\right| < 1$, то выполняются неравенства $\frac{(-\varphi)^{-n}}{\sqrt{5}} < \frac{1}{\sqrt{5}} < \frac{1}{2}$. Таким образом, n-ое число Фибоначчи равно $\frac{\varphi^n}{\sqrt{5}}$, округленному до ближайшего целого числа. Следовательно, $F_n = O(\varphi^n)$.

◁

Лемма:

Максимальная степень $degree$ произвольной вершины в фибоначчиевой куче с n вершинами равна $O(\log n)$

Доказательство:

▷

Пусть x — произвольная вершина в фибоначчиевой куче с n вершинами, и пусть k — степень вершины x. Тогда по доказанному выше в дереве, корень которого x, содержится не менее F_k вершин, что в свою очередь по лемме равно $O(\varphi^k)$. То есть

$$n \geqslant \varphi^k$$

Логарифмируя по основанию φ, получаем

$$\log_\varphi n \geqslant k$$

Таким образом, максимальная степень $degree$ произвольной вершины равна $O(\log n)$.

◁

Итоговая асимптотика операции $\mathrm{extraxtMin}$, учитывая и вспомогательную функцию $\mathrm{consolidate}$, время работы которой доказывается ниже, равно: $O(1) + O(degree) + O(degree) = O(degree)$. По доказанной выше лемме $O(degree) = O(\log(n))$.

Учетная стоимость $\mathrm{consolidate}$ равна $O(degree)$. Докажем это:

Изначально в корневом списке было не более $degree + trees - 1$ вершин, поскольку он состоит из исходного списка корней с $trees$ узлами, минус извлеченный узел и плюс дочерние узлы, количество которых не превышает $degree$. В ходе операции $\mathrm{consolidate}$ мы сделали $O(degree + trees)$ слияний деревьев. Потенциал перед извлечением минимума равен $trees + 2 * marked$, а после не превышает $degree + 1 + 2 * marked$, поскольку в корневом списке остается не более $degree + 1$ узлов, а количество помеченных узлов не изменяется. Таким образом, амортизированная стоимость не превосходит

$$O(degree + trees) + (degree + 1 + 2 * marked) - (trees + 2 * marked) = O(degree) + O(trees) - trees$$

Поскольку мы договорились, что можем масштабировать единицу потенциала таким образом, чтобы покрывать константное количество работы, то итоговая амортизационная оценка — $O(degree)$

Уменьшение значения элемента

Докажем, что амортизированное время работы операции $\mathrm{decreaseKey}$ есть $O(1)$. Поскольку в процедуре нет циклов, ее время работы определяется лишь количеством рекурсивных вызовов каскадного вырезания.

Пусть мы вызвали процедуру каскадного вырезания подверглось k раз. Так как реальное время работы каждой итерации $\mathrm{cascadingCut}$ составляет $O(1)$, то реальное время работы операции $\mathrm{decreaseKey}$ — $O(k)$.

Рассмотрим, как изменится потенциал в результате выполнения данной операции. Пусть H — фибоначчиева куча до вызова $\mathrm{decreaseKey}$. Тогда после k итераций операции $\mathrm{cascadingCut}$ вершин с пометкой $x.mark = true$ стало как минимум на $k - 2$ меньше, потому что каждый вызов каскадного вырезания, за исключением последнего, уменьшает количество помеченных вершин на одну, и в результате последнего вызова одну вершину мы можем пометить. В корневом списке прибавилось k новых деревьев ($k - 1$ дерево за счет каскадного вырезания и еще одно из-за самого первого вызова операции cut).

В итоге, изменение потенциала составляет:
$\Phi_i - \Phi_{i-1} = ((trees + k) + 2 * (marked + k - 2)) - (trees + 2 * marked) = 4 - k$. Следовательно, амортизированная стоимость не превышает $O(k) + 4 - k$. Но поскольку мы можем соответствующим образом масштабировать единицы потенциала, то амортизированная стоимость операции $\mathrm{decreaseKey}$ равна $O(1)$.

Удаление элемента

Амортизированное время работы: $O(1) + O(degree) = O(degree)$.

Поскольку ранее мы показали, что $degree = O(\log n)$, то соответствующие оценки доказаны.

Итоговая таблица

Операция	Амортизированная сложность
$\mathrm{makeHeap}$	$O(1)$
insert	$O(1)$
getMin	$O(1)$
merge	$O(1)$
$\mathrm{extractMin}$	$O(\log n)$
$\mathrm{decreaseKey}$	$O(1)$
delete	$O(\log n)$

Недостатки и достоинства

Недостатки:

- Большое потребление памяти на узел(минимум 21 байт)
- Большая константа времени работы, что делает ее малоприменимой для реальных задач
- Некоторые операции в худшем случае могут работать за $O(n)$ времени

Достоинства:

- Одно из лучших асимптотических времен работы для всех операций

См. также

- Приоритетные очереди
- Двоичная куча
- Биномиальная куча
- Левосторонняя куча
- Тонкая куча
- Толстая куча на избыточном счетчике
- Куча Бродала-Окасаки

Примечания

1. Майкл Фридман — Википедия
2. Роберт Тарьян — Википедия
3. Числа Фибоначчи — Википедия

Источники информации

- Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн — Алгоритмы: построение и анализ. — М.: Издательский дом «Вильямс», 2005. — С. 1296. — ISBN 5-8459-0857-4
- Числа Фибоначчи — Википедия
- Фибоначчиева куча — Википедия
- Fibonacci heap visualization (https://www.cs.usfca.edu/~galles/visualization/FibonacciHeap.html)
- Фибоначчиевы кучи — INTUIT.ru (http://www.intuit.ru/department/algorithms/dscm/7/2.html)
- Fibonacci Heaps — Duke University (http://www.cs.duke.edu/courses/fall05/cps230/L-11.pdf)
- Fibonacci Heaps — Princeton University (https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf)