МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)

На правах рукописи
УДК xxx.xxx

Смирнов Иван Федорович

НАЗВАНИЕ ДИССЕРТАЦИОННОЙ РАБОТЫ

Специальность XX.XX.XX —
«Название специальности»

Выпускная квалификационная работа
кандидата физико-математических наук

Научный руководитель:
уч. степень, уч. звание
Бабенко Максим Александрович

Долгопрудный — 2016

Оглавление

Введение

Нужен ли автореферат и как-то особо отформатированное введение?

Очередь с приоритетом — это абстрактный тип данных, позволяющая поддерживать множество элементов частично упорядоченным [1]. Будем считать, что элемент — это пара $(key, item)$, где key — это приоритет. В данной работе рассматриваются очереди с приоритетом, поддерживающие следующие базовые операции: **insert**$(key, item)$, добавляющая элемент $(key, item)$ в множество; и **extractMin()**, удаляющая из множества элемент с минимальным приоритетом и возвращающая его.

Данная структура данных хорошо изучена. В бородатом году Дональдом Кнутом или типа того было предложено решение с помощью двоичной кучи, позволяющее выполнять операции **insert** и **extractMin** за $O(\log n)$ сравнений, где n — количество элементов в множестве. В дальнейшем были предложены решения с лучшей асимптотикой (в большинстве своём, однако, неприменимые на практике из-за высокой скрытой константы); оптимальной с некоторой точки зрения является *Brodal queue*, реализаующая не помню асимптотику.

В данной работе будет изучена задача приоритетной очереди, оптимальной при условии ограниченного числа удалений. Пусть операция **insert** была вызвана n раз, **extractMin** k раз, при этом на каждую операцию **insert** было затрачено $O(1)$ сравнений. Тогда известно (и будет доказано), что на операции **extractMin** необходимо в худшем случае потратить $O(k \log k)$ сравнений. Будет построена структура данных, приближающаяся к данной нижней оценке с точностью до множителя $O(\log^* n)$.

Глава 1. Основная схема

В этой главе будет построена структура данных `SimpleCascadeHeap`, реализующая операции **insert** за $O(1)$ и **extractMin** за $O(\log k + \log n)$ сравнений, где за n обозначено количество вставок, за k — извлечений к моменту выполнения очередной операции.

1.1 Обзор

`SimpleCascadeHeap` построена на базе двоичной кучи. Для улучшения асимптотики используется буферизированная вставка и дополнительный буфер ограниченного размера для извлечения элементов. Структура состоит из трёх частей: буфер вставки размера $\log n$, в который добавляются элементы; промежуточной «кучи куч» (двоичной кучи, элементами которой являются другие двоичные кучи); и конечной кучи (*Head Heap*), буфера для извлечения. Отношение порядка на множестве непустых двоичных куч порождается отношением порядка на элементах, лежащих в корне.

При добавлении элемента он попадает в буфер вставки. Затем, если буфер переполняется, на его элементах строится двоичная куча и добавляется в промежуточную кучу. Для нахождения и удаления минимального элемента необходимо пробежаться по всем элементам буфера, а также посмотреть в вершину промежуточной кучи и Head Heap. После этого элемент нужно удалить из всех куч, в которых он лежит. При этом удаление минимального элемента из всех куч, кроме Head Heap, происходит следующим образом: корень удаляется, а оба его поддерева добавляются в Head Heap, если они непусты.

На добавление одного элемента требуется $O(1)$ времени амортизированно: накопив буфер размера $\log n$, нужно потратить $O(\log n)$ действий на то, чтобы добавить элемент в промежуточную кучу, что даёт требуемую оценку в среднем.

Можно видеть, что размер Head Heap в любой момент времени не превосходит $O(k)$, где k — количество удалений, поскольку глубина любой «кучи куч» константна. Для удаления элемента необходимо: a) просмотреть буфер, b) посмотреть на корень промежуточной кучи и Head Heap, c) выполнить $O(1)$ вставок в Head Heap. Части (b) и (c) занимают $O(\log k)$ времени, часть (a) — $O(\log n)$. Узким местом является добавление в буфер. Оказывается, что на буфере можно рекурсивно построить аналогичную структуру, существенно уменьшив слагаемое $O(\log n)$ в асимптотике удаления и не ухудшив при этом операцию вставки.

В главе 1 будет построена описанная структура данных, проведена деамортизация и получены оценки в худшем случае. В части 2 будет построена улучшенная версия структуры под названием `CascadeHeap`, оценка на добавление будет улучшена до $O(\log k \cdot \log^* n)$, а также будут описаны несколько возможных трейд-оффов между временем добавления и извлечения минимума.

1.2 Описание

Определение 1.2.1. Пусть $\mathcal{X}$ — некоторое линейно упорядоченное множество. Назовём $\mathcal{H}(\mathcal{X})$ множество $\{\mathcal{X} \cup$все непустые двоичные кучи над элементами $\mathcal{X}\}$. Введём на $\mathcal{H}(\mathcal{X})$ линейный порядок следующим образом: $x < y \overset{def}{\Longleftrightarrow} repr(x) < repr(y)$, где

$$repr(x) = \begin{cases} x, & x \in \mathcal{X} \\ x.top(), & x \text{ — двоичная куча над элементами } \mathcal{X} \end{cases}$$

Обозначение. Назовём $\mathcal{H}_k(\mathcal{X})$ множество $\underbrace{\mathcal{H}(\mathcal{H}(\ldots(}_{k}\mathcal{X})\ldots))$.

Обозначим за $\mathcal{X}$ множество элементов (то есть пар $(key, item)$), над которыми оперирует структура данных. На нём определён линейный порядок.

Определение 1.2.2. Level x— *уровень* элемента x, т.е. минимальное $k : x \in \mathcal{H}_k(\mathcal{X}), x \notin \mathcal{H}_{k-1}(\mathcal{X})$. Иными словами, это максимальная глубина вложенности двоичных куч внутри элемента.

Структура данных состоит из трёх частей:

1. *буфер B* — массив элементов $\mathcal{X}$ размера b. b будет изменяться в процессе работы алгоритма и поддерживаться примерно равным $\log n$. В начале работы $b = 1$. Элементы изначально попадают в буфер.

2. *промежуточная куча* `MH` *(middle heap)* — двоичная куча над $\mathcal{H}(\mathcal{X}) \setminus \mathcal{X}$. При заполнении буфера на нём строится двоичная куча и добавляется в `MH`.
3. *конечная куча* `HH` *(Head Heap)* — двоичная куча над $\mathcal{H}_2(\mathcal{X})$. Её размер всегда равен $O(k)$.

1.2.1 Операция вставки

Algorithm 1: Операция **insert**

Data: $x \in \mathcal{X}$
добавить x в конец буфера;
if *размер буфера* $> \log_2 n$ **then**
  создать двоичную кучу T из элементов буфера;
  очистить буфер;
  добавить кучу T в `MH`;

1.2.2 Операция извлечения минимума

Добавим `MH` в `HH`. После этого максимальный элемент может находиться либо в буфере, либо в вершине `HH`. Просмотрим все элементы буфера за $O(|B|)$ и вершину `HH`. Если минимум найден в буфере, удалим его и вернём. Иначе нужно удалить корень из `HH`.

Сложность в том, что вершины `HH` — не элементы, а кучи. Опишем, как корректно производить удаление в таком случае. Для этого введём вспомогательную функцию `Yield`. На вход она принимает произвольный аргумент из $\mathcal{H}_2(\mathcal{X})$ и возвращает минимальный элемент из $\mathcal{X}$, содержащийся в множестве (по сути, корень корня корня...). Каждая просмотренная вершина удаляется из своей кучи, её оба ребёнка добавляются в `HH`.

Теперь мы готовы описать алгоритм извлечения минимума (алгоритм 2). Для удаления минимального элемента его нужно найти, просмотрев буфер и ко-

Function Yield(x)

```
if x ∈ 𝒳 then
    return x;
else
    if x.hasLeftChild then HH.insert(x.leftChild);
    if x.hasRightChild then HH.insert(x.rightChild);
    return Yield (x.top());
```

Algorithm 2: Операция **extractMin**

```
Data: x — минимальный элемент или куча, содержащая его
switch положение x do
    case буфер
        удалить x из буфера и вернуть его;
    case HH
        T = HH.top();
        HH.extractMin();
        return Yield (T);
```

рень HH. После этого нужно выполнить операцию **extractMin** из всех куч, содержащих элемент. Причём «честно» эта операция выполняется только для верхнего уровня Head Heap; для того, чтобы обработать остальные, вызывается процедура Yield, добавляющая левое и правое поддерево элемента в HH вместо непосредственно его удаления. Это сделано для того, чтобы асимптотика операции не зависела от высоты никакой кучи, кроме HH.

1.3 Деамортизация

В данной части будет проведена деамортизация операции добавления, т.е. получена оценка в $O(1)$ сравнений на добавление в худшем случае, и проведён анализ всех операций.

1.3.1 Куча с версиями

Для деамортизации нам понадобится частично персистентная куча с поддержкой отложенных операцией, которую мы назовём *кучей с версиями*. Неформально говоря, требуется делать следующее: добавлять элемент «по чуть-чуть» так, чтобы он не был виден раньше времени, а потом атомарно «переключиться», чтобы добавленный элемент появился в куче. Кроме того, необходима возможность прервать добавление в любой момент.

Определение 1.3.1. *Куча с версиями* — надстройка над двоичной кучей, поддерживающая следующие операции:

1. Удалить корень и вернуть два его поддерева. Если в данный момент происходит вставка, отменить её и также вернуть вставляемый элемент. После этой операции добавлений больше не будет.
2. Если в данный момент не происходит вставка, сделать элемент x текущим вставляемым элементом.
3. Если в данный момент происходит вставка, проделать t операций по вставке.
4. Если вставка происходила и уже закончена, атомарно добавить вставленный элемент.

В каждой вершине двоичной кучи хранится два указателя на потомков — правого и левого (возможно, пустые). Будем вместо каждого указателя хранить кортеж пар (указатель, номер версии), где версия — некоторое натуральное число. Кроме того, в корне делева будет отдельно храниться актуальный номер версии V, изначально равный 0. Для того, чтобы получить явное дерево, нужно для каждой вершины взять указатель с максимальной версией, не превосходящей V.

Для выполнения отложенной вставки нужно вставлять элемент как обычно, но вместо изменения указателей создавать новые, версии $V + 1$. После завершения вставки можно атомарно перейти на новую версию, увеличив V. Если необходимо отменить вставку и удалить корень, нужно просто вернуть оба поддерева корня (согласно версии V) и удалённый элемент, а обоим детям корня установить версию равной V.

При добавлении нового указателя в вершину необходимо удалить из неё все указатели, кроме актуального на данный момент. Несложно видеть, что при этом из каждой вершины всегда будет исходить не более двух указателей, причём среди них всегда будет актуальный.

1.3.2 Деамортизация операции вставки

Напомним, как проводится операция вставки: элемент добавляется в буфер, а при заполнении буфера на нём строится двоичная куча и добавляется в `MH`. Теперь `MH` будет кучей с версиями, и работу после переполнения буфера можно разделить на три итерации:

I. Очистить буфер и построить двоичную кучу T на элементах, которые в нём были
II. Отложенно добавить T в `MH`
III. Переключить версию в `MH`, чтобы применить добавление

Назовём эту процедуру *балансировкой*.

Все эти операции будут равномерно выполнены, пока буфер заполняется в следующий раз, требуя $O(1)$ дополнительной работы на каждое добавление элемента. Это будет доказано в разделе 1.5.

Для балансировки буфера размера b нужно после каждого из последующих b добавлений проделывать C операций процедур I, II, III после добавления. Таким образом, к моменту следующего переполнения буфера балансировка будет закончена.

1.3.3 Удаления в процессе балансировки

Во время балансировки структура находится в нестабильном состоянии. Если в это время поступает запрос **extractMin**, необходимо отменить балансировку, при этом не потеряв никакой информации. В этой главе будет описано, как это делать в зависимости от того, во время какой стадии балансировки пришёл запрос.

Если балансировка находится в I стадии, необходимо достроить двоичную кучу на множестве T, тем самым переведя балансировку в стадию II. Если балан-

Algorithm 3: Инициализация деамортизированной кучи

```
begin
    BalancingState ← NoAction;
    BufferSize ← 0;
    InsertionsCount ← 1;
    C ← константа из леммы 1.5.2 ;
```

сировка находится в II стадии (в том числе после выполнения только что описанной операции), необходимо вставить T в `HH`. После этого нужно в любом случае вставить `MH` в `HH`.

После подготовки к удалению `MH` оказывается пуста. Таким образом, для удаления минимума достаточно просмотреть буфер и корень `HH` так же, как было описано в параграфе 1.2.2.

Теперь мы готовы целиком описать алгоритмы вставки в `SimpleCascadeHeap` и удаления минимума.

1.4 Алгоритм

Алгоритмы 3, 4, 5 описывают соответственно инициализацию структуры данных, добавление элемента и извлечение минимального элемента.

1.5 Доказательство корректности и асимптотики

***Теорема* 1.5.1** (о корректности и асимптотике)**.** *Для любой последовательности из n операций **insert** и k операций **extractMin**, в которой запрос **extractMin** может поступать только к непустой куче, верно следующее:*

1. *(корректность) каждая операция **extractMin** удаляет минимальный элемент из находящихся в куче к тому моменту;*
2. *(асимптотика) каждая операция **insert** требует $O(1)$ сравнений, каждая операция **extractMin** требует $O(\log n + \log k)$ сравнений, где k — количество вызовов **extractMin** к тому моменту.*

Algorithm 4: Операция **insert** в деамортизированной куче

Data: $x \in \mathcal{X}$
begin
  if $\mathtt{BalancingState} \in \{\mathtt{StateI}, \mathtt{StateII}\}$ **then**
    выполнить `C` операций по балансировке;
    if *закончилась стадия I* **then**
      `BalancingState` $\leftarrow$ `StateII`;
    if *закончилась стадия II* **then**
      выполнить переключение версии `MH`;
      `BalancingState` $\leftarrow$ `NoAction`;
  добавить x в конец буфера;
  `BufferSize` $\leftarrow$ `BufferSize` $+ 1$;
  `InsertionsCount` $\leftarrow$ `InsertionsCount` $+ 1$;
  if $\mathtt{BufferSize} > \log_2 \mathtt{InsertionsCount}$ **then**
    `BufferSize` $\leftarrow 0$;
    `BalancingState` $\leftarrow$ `StateI`;
    начать балансировку на элементах буфера и очистить буфер;

Algorithm 5: Операция **extractMin** в деамортизированной куче

begin
  if $\mathtt{BalancingState} = \mathtt{StateI}$ **then**
    закончить построение кучи на T;
    добавить T в `HH`;
  if $\mathtt{BalancingState} = \mathtt{StateII}$ **then**
    отменить отложенное добавление в `MH`;
    добавить T в `HH`;
  if *МН непуста* **then** вставить `MH` в `HH`;
  `BalancingState` $\leftarrow$ `NoAction`;
  просмотреть буфер B и вершину `HH`, если куча непуста, найти среди них минимальный элемент;
  if *минимум в B* **then**
    удалить минимум из буфера;
  else
    T = `HH`.top();
    `HH`.extractMin();
    return `Yield` (T);

В дальнейшем в этом разделе символы n и k имеют указанное выше значение, если явно не сказано иное.

***Лемма* 1.5.1.** *Пусть размер буфера при последнем переполнении был равен b. Тогда $|MH| \leq 2^b$.*

Доказательство. Из условия на переполнение буфера из алгоритма 4 имеем $n - 1 \leq 2^{b-1}$, откуда получаем $n \leq 2^b$. Но $|MH| \leq n$, откуда и следует требуемое неравенство. □

***Лемма* 1.5.2.** *Пусть размер буфера при последнем переполнении был равен b. Тогда существует некоторая константа C такая, что балансировку можно выполнить за не более чем $b \cdot C$ сравнений в худшем случае.*

Доказательство. Балансировка состоит из двух частей: построение двоичной кучи на b элементах и добавление элемента в $\mathtt{MH}$. Первая часть реализуется за $O(b)$ сравнений ([1]). Вторая часть реализуется за $O(\log |\mathtt{MH}|)$ сравнений ([1]). Но из теоремы 1.5.1 мы имеем $\log_2 |\mathtt{MH}| \leq b$. Значит, балансировка реализуется за $O(b)$ сравнений, и существование искомой константы C следует из определения «O большого». □

***Лемма* 1.5.3.** *Две балансировки не могут идти одновременно, т.е. к моменту начала балансировки предыдущая балансировка уже завершилась.*

Доказательство. Заметим, что если переполнение буфера возникло при размере b, то следующее переполнение возникнет при бóльшем размере буфера, поскольку логарифм — монотонная функция. Значит, в течение следующих b запросов **insert** балансировка не произойдёт. Во время каждого из этих запросов будет проведено C операций по балансировке, где C — константа из леммы 1.5.2. Но из этой же леммы известно, что $b \cdot C$ операций достаточно для завершения балансировки. Значит, к моменту следующего переполнения балансировка будет завершена. □

***Лемма* 1.5.4.** *Пусть $H \in \mathcal{H}_k$ для некоторого k — «куча куч … куч» уровня k, построенная на множестве элементов X. Тогда в корне H находится минимальный элемент, т.е. $\min\limits_{x \in X} x = H.\underbrace{\mathbf{top}().\ldots.\mathbf{top}()}_{k \text{times}}$.*

Доказательство. Докажем индукцией по Level H.

Если Level $H = 0$, то H — элемент X, и утверждение учевидно.

Пусть Level $H = k > 0$. Тогда все элементы H — кучи с уровнем $< k$, и по индукции в их корнях лежит минимальное значение. Но по свойству бинарной кучи в корне H лежит минимальный из корней всех элементов H, что и требовалось доказать. □

***Теорема* 1.5.2** (о сохранении инвариантов). *В любой момент соблюдаются следующие инварианты:*

1. *`MH` и все её элементы — корректные бинарные кучи;*
2. *`HH` и все её элементы — корректные бинарные кучи;*
3. *Если балансировка находится в стадии II, то множество T, которое вставляется в `MH` — корректная бинарная куча;*
4. *$|\mathtt{HH}| \leq 6k$.*

Доказательство. При инициализации, когда структура пуста, все инварианты выполнены. Докажем, что они выполняются после всех операций.

При выполнении операции **insert** происходит две вещи: непосредственно добавление элемента в буфер и, возможно, несколько операций по балансировке. Добавление в буфер не затрагивает ни одно из интересующих нас множеств. Посмотрим на балансировку.

При выполнении первой стадии балансировки, опять же, ни одно из интересующих нас множеств не изменяется. К моменту завершения первой стадии множетство T представляет собой бинарную кучу согласно алгоритму 4 и в дальнейнем не изменяется до окончания балансировки. Значит, инвариант (3) выполнен.

При выполнении второй стадии балансировки изменяется только «будущая» версия `MH`. К моменту завершения второй стадии «будущая» версия является корректной кучей согласно алгоритму 4, кроме того, вставляемый элемент является корректной кучей по инварианту (3). Значит, инвариант (1) сохранится во время второй стадии балансировки и её завершения.

Операцию **extractMin** можно разбить на две части: завершение балансировки и непосредственно извлечение минимального элемента. Проведение первой стадии балансировки не нарушает инварианты, как мы видели ранее. Для завершения второй стадии необходимо добавить множества T и `MH` в `HH`; оба этих множества являются корректными кучами по инвариантам (1) и (3). Таким образом, мы не нарушим инварант (2) и добавим в $\texttt{HH}$ не более двух элементов.

Если извлечение минимума произошло из буфера, инварианты не нарушаются. Если извлечение произошло из HH, то, исходя из операции $\texttt{Yield}$, в $\texttt{HH}$ добавилось не более $2 \cdot \operatorname{Level}(\texttt{HH}.\mathbf{top}())$ элементов. Кроме того, все добавленные элементы — корректные бинарные кучи. Значит, инвариант (2) выполнен.

$\operatorname{Level}(\texttt{HH}.\mathbf{top}()) < \operatorname{Level}\texttt{HH} \leq 3$. Значит, суммарно за одну операцию **extractMin** в $\texttt{HH}$ добавляется не более $2 + 2 \cdot 2 = 6$ элементов, что доказывает сохранение инварианта (4).

$\square$

Доказательство теоремы 1.5.1 (о корректности и асимптотике). Напрямую следует из теоремы 1.5.2 о сохрании инвариантов и леммы 1.5.4. $\square$

Глава 2. Улучшение асимптотики

В предыдущей главе была описана структура очереди с приоритетом, позволяющая добавлять элемент за $O(1)$ сравнений и удалять минимум за $O(\log k + \log n)$ сравнений, где n — количество добавлений, k — количество удалений к моменту вызова операции удаления. Обе оценки выполняются в худшем случае.

В этой главе вторая оценка будет улучшена. Будет описана структура данных `CascadeHeap`, позволяющая улучшить асимптотику удаления до $(O \log k + \log\log\ldots\log n)$ для любого наперёд заданного константного числа итераций логарифма, сохранив при этом константную сложность добавления. Кроме того, (напрямую из предыдущей) будут получены очереди с приоритетом со сложностью вставки и удаления, соответственно, $(O(t), O(t \cdot \log k + \log^{(t)} n))$ и $(O(\log^* n), O(\log^* n \cdot \log k))$.

Поменять t на r и добавить хвостик к заявленной асимптотике

Анонс

Посмотрим на операцию **extractMin** в структуре данных, описанной в предыдущей главе. Видно, что узкое место в асимптотике — просмотр буфера и достраивание кучи T, если приходится экстренно завершать балансировку на первой стадии: эта часть работает за $O(\log n)$, в то время как остальные за $O(\log k)$. Значит, нужно избавиться от полного просмотра буфера при удалении.

Вообще говоря, просматривать весь буфер не нужно, требуется только уметь быстро находить минимум. Для этого можно рекурсивно построить такую же структуру данных размера $\log n$ на элементах буфера и вместо просмотра буфера делать запрос к ней. Во внутренней структуре, в свою очередь, будет свой буфер (размера $\log\log n$). Появятся две промежуточных кучи MH: одна во внутренней структуре, «куча куч», вторая — во внешней, «куча куч куч». Head Heap сохранится в единственном экземпляре и будет использоваться как при внутренней балансировке, так и при внешней.

Основная идея `CascadeHeap` заключается в том, чтобы рекурсивно строить аналогичную структуру на буфере предыдущего уровня до тех пор, пока буфер не станет достаточно маленького размера. В зависимости от глубины вложенности может достигаться разная комбинация асимптотик времени добавления и извлечения, как было сказано в начале этой главы.

В первой части будет описана `CascadeHeap` глубины вложенности 2 с амортизированными оценками временной сложности. В следующих частях этой главы будет более детально рассмотрена произвольная глубина вложенности, произведена деамортизация и доказана асимптотика и корректность.

2.1 `CascadeHeap` вложенности 2

Описываемая в этом разделе структура — надстройка над `SimpleCascadeHeap`, в которой для уменьшения размера буфера добавляется ещё один уровень буферизации. Это позволяет уменьшить время добавления до $O(\log k + \log\log n)$.

`CascadeHeap` вложенности 2 (или `CascadeHeap[2]`) состоит из следующих частей:

1. Буфер, куда изначально попадают элементы. Его размер растёт при добавлении элементов и поддерживается примерно равным $\log\log n$.
2. Промежуточная куча $\mathtt{MH}_2$. Сюда попадают кучи, построенные на элементах буфера. Размер $\mathtt{MH}_2$ не превосходит $O(\log n)$.
3. Промежуточная куча $\mathtt{MH}_1$, «куча куч куч». Когда $\mathtt{MH}_2$ становится слишком большого размера, её нужно добавить в $\mathtt{MH}_1$.
4. Конечная куча `HH`, используемая так же, как и в `SimpleCascadeHeap`.

Алгоритм 6 более подробно иллюстрирует вставку элемента. В данном случае оценка $O(1)$ на вставку амортизированная, деамортизация будет проведена позже.

Переполнение буфера возникает примерно каждые $\log\log n$ вставок и требует $\log\log n$ дополнительных действий на вставку элемента в $\mathtt{MH}_2$. Переполнение $\mathtt{MH}_2$ возникает примерно каждые $\log n$ вставок и требует $\log n$ действий на встав-

Algorithm 6: Операция **insert** в `CascadeHeap[2]`

Data: $x \in \mathcal{X}$
добавить x в конец буфера;
if *размер буфера* $> \log_2 \log_2 n$ **then**
  создать двоичную кучу T из элементов буфера;
  очистить буфер;
  добавить кучу T в $\mathtt{MH}_2$;
if *размер* $MH_2 > \log_2 \log_2 n$ **then**
  добавить $\mathtt{MH}_2$ в $\mathtt{MH}_1$;
  очистить $\mathtt{MH}_2$;

ку в $\mathtt{MH}_1$. Отсюда следует амортизированная оценка в $O(1)$ сравнений на вставку одного элемента.

Для удаления необходимо, как и в `SimpleCascadeHeap`, вставить $\mathtt{MH}_1$ и $\mathtt{MH}_2$ в `HH`, затем просмотреть буфер и вершину `HH` и поступить согласно алгоритму 2.

вставить сюда какую-нибудь картинку про то, что происходит

2.2 `CascadeHeap` произвольной вложенности

По аналогии с `CascadeHeap[2]` можно определить `CascadeHeap[r]` для произвольного натурального r. `CascadeHeap[r]` состоит из следующих частей:

1. Буфер, куда изначально попадают элементы. Его размер растёт при добавлении элементов и поддерживается примерно равным $\log^{(r)} n$.
2. Промежуточные кучи $\mathtt{MH}_1$, …, $\mathtt{MH}_r$. Размер кучи $\mathtt{MH}_t$ не превосходит $\log^{(t-1)} n$, $1 \leq t \leq r$.
3. Конечная куча `HH`, используемая так же, как и в `SimpleCascadeHeap`.

Вставка происходит по аналогии с `SimpleCascadeHeap` и `CascadeHeap[2]`: элемент добавляется в буфер, при переполнении буфера на нём строится двоичная куча и вставляется в $\mathtt{MH}_r$, при переполнении кучи $\mathtt{MH}_r$ она вставляется в $\mathtt{MH}_{r-1}$ и т. д. На каждом уровне при переполнении суммарно выполняется $O(n)$ действий, поэтому суммарное амортизированное время вставки — $O(r)$.

Удаление минимума происходит по аналогии с предыдущими описанными структурами. $\mathtt{MH}_1$, …, $\mathtt{MH}_r$ добавляются в `HH`, затем минимум извлекается из `HH` или из буфера. Можно показать, что размер `HH` не превосходит $O(k \cdot r)$, поэтому на одно удаление требуется $O(\log k + \log r + \log^{(r)} n)$ сравнений. Обе заявленные асимптотики будут доказаны далее.

2.3 `CascadeHeap` неограниченной вложенности

Параметр r в `CascadeHeap*` можно устремить к бесконечности, фактически сделав r изменяемым и равным $\log^* n$. Тогда количество промежуточных куч будет расти с ростом n. Когда самая большая `MH` экспоненциально...

Подумать, как одинаково описать `MH` в `CascadeHeap[r]` и `CascadeHeap*`. Пока что хочется занумеровать `MH` в обратном порядке.

Список литературы

1. Алгоритмы: построение и анализ = Introduction to Algorithms / Т. Кормен [и др.] ; под ред. И. В. Красикова. — 2-е изд. — М. : Вильямс, 2005. — 1296 с.