МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)»

ФАКУЛЬТЕТ ИННОВАЦИЙ И ВЫСОКИХ ТЕХНОЛОГИЙ
КАФЕДРА АНАЛИЗА ДАННЫХ

Выпускная квалификационная работа по направлению
01.03.02 «Прикладные математика и информатика»
НА ТЕМУ:

ОПТИМАЛЬНАЯ ОЧЕРЕДЬ С ПРИОРИТЕТОМ ПРИ ОГРАНИЧЕНИИ НА КОЛИЧЕСТВО УДАЛЕНИЙ

Студент Смирнов И. Ф.

Научный руководитель Колесниченко И. И.

Научный руководитель к.ф-м.н. Бабенко М. А.

Зав. зам. кафедрой д.ф-м-н., профессор Бунина Е. И.

МОСКВА — 2016

Оглавление

Стр.

Введение

Очередь с приоритетом — абстрактный тип данных, поддерживающий две обязательные операции — добавить элемент и извлечь минимум. Будем считать, что элемент — это пара $(key, item)$, где key — это приоритет. В данной работе рассматриваются очереди с приоритетом, поддерживающие базовые операции: **insert**$(key, item)$, добавляющая элемент $(key, item)$ в множество; и **extractMin()**, удаляющая и возвращающая элемент с минимальным приоритетом.

Эта структура данных хорошо изучена. Ещё в прошлом веке было предложено решение с помощью двоичной кучи[1, с. 179], позволяющее выполнять обе операции за $O(\log n)$ сравнений, где n — количество элементов в множестве. Позднее были предложены решения с лучшей асимптотикой, обычно, однако, неприменимые на практике из-за высокой скрытой константы. В 1996 году Gerth Brodal опубликовал структуру *Brodal queue*[2], реализующую вставку, нахождение минимума и слияние двух куч за $O(1)$ и извлечение за $O(\log n)$ в худшем случае. Эта оценка в некотором смысле теоретически оптимальна.

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. Мы построим структуру данных, приближающуюся к данной оценке с точностью до множителя $O(\log^* n)$.

В первой главе будет построена структура `SimpleCascadeHeap`, иллюстрирующая общую идею алгоритма. Для достижения показанного времени работы мы использовали техники бутстрэппинга («кучи из куч») и буферизованных вставок. Во второй главе на базе `SimpleCascadeHeap` будут построены структуры `CascadeHeap[r]` и `CascadeHeap*`. Оценки на количество сравнений указаны в таблице 1.

Таблица 1 — Сравнение асимптотик разных версий `CascadeHeap`

Название	Сложность **insert**	Сложность **extractMin**
`SimpleCascadeHeap`	$O(1)$	$O(\log n)$
`CascadeHeap[r]`	$O(r)$	$O(\log^{(r)} n + r(\log k + \log r))$
`CascadeHeap*`	$O(\log^* n)$	$O(\log^* n(\log k + \log\log^* n))$

Глава 1. Основная схема

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

1.1 Обзор

`SimpleCascadeHeap` будет построена на базе двоичной кучи. Для улучшения асимптотики будет использоваться буферизированная вставка и дополнительный буфер ограниченного размера для извлечения элементов. Структура будет состоять из трёх частей: буфер вставки размера $\log n$, в который добавляются элементы; промежуточной «кучи куч» (двоичной кучи, элементами которой являются другие двоичные кучи) *Middle Heap* (`MH`); и конечной кучи (*Head Heap*, `HH`), буфера для извлечения.

Отношение порядка на множестве непустых двоичных куч порождается отношением порядка на элементах, лежащих в корне. То есть для сравнения двух куч нужно сравнить минимальные элементы каждой из них.

При добавлении элемента он попадает в буфер вставки. Затем, если буфер переполняется, на его элементах строится двоичная куча и добавляется в Middle Heap.

Для нахождения и удаления минимального элемента необходимо пробежаться по всем элементам буфера, а также посмотреть в вершину Middle Heap и Head Heap. После этого элемент нужно удалить из той кучи, в которой он найден. 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. Таким образом размер Heap Heap поддерживается равным $O(k)$, и все операции, кроме просмотра буфера, выполняются за $O(\log |\texttt{HH}|)$ и не зависят от высоты других куч.

[1] Вообще говоря, $k \leq n$, поэтому здесь можно написать $O(\log n)$. Однако для общности с полученными в следующей главе асимптотиками здесь будет использована именно такая запись.

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.

Далее в этой главе будет построена описанная структура данных, проведена деамортизация и получены оценки в худшем случае. В главе 2 будет построена улучшенная версия структуры под названием `CascadeHeap`, оценка на извлечение минимума будет улучшена до $O(\log^* n(\log k + \log\log^* n))$, а также будут описаны возможные трейд-оффы 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: $x < y \overset{def}{\iff} repr(x) < repr(y)$, где

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

Иными словами, $repr(x)$ — минимальный представитель объекта x.

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

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 $\log n$ (формально ограничения на размер буфера описаны в алгоритме 1). 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

Алгоритм 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`.

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());
```

Алгоритм 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);
```

Теперь мы готовы описать алгоритм извлечения минимума (алгоритм 2). Для удаления минимального элемента его нужно найти, просмотрев буфер и корень `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 добавления запрещены.
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.
5. Отменить вставку и вернуть вставляемый элемент. После этой операции добавления запрещены.

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.

Для более подробного описания операции отложенной вставки введём несколько *состояний* для кучи с версиями и опишем переходы между ними. Возможные состояния перечислены в списке ниже.

1. Обычное состояние, в котором куча с версиями выглядит как обычная куча.
2. Происходит вставка, и её результаты пока не видны.
3. Куча заблокирована, новые вставки запрещены.

Когда начинается новая вставка, происходит переход из состояния (1) в (2); в состоянии (2) новые вставки запрещены. Когда вставка завершается, происходит переключение версии и переход из состояния (2) в (1). Когда отменяется вставка или происходит удаление корня, происходит переход в состояние (3) из (1) или (2) в зависимости от того, происходила ли в тот момент отложенная вставка.

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 балансировка 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.

Алгоритм 3: Инициализация деамортизированной кучи

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

1.4 Алгоритм

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

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

```
Data: x ∈ 𝒳
begin
    if BalancingState ∈ {State1, State2} then
        выполнить C операций по балансировке;
        if закончилась стадия I then
            BalancingState ← State2;
        if закончилась стадия II then
            выполнить переключение версии MH;
            BalancingState ← NoAction;
    добавить x в конец буфера;
    BufferSize ← BufferSize + 1;
    InsertionsCount ← InsertionsCount + 1;
    if BufferSize > log_2 InsertionsCount then
        BufferSize ← 0;
        BalancingState ← State1;
        начать балансировку на элементах буфера и очистить буфер;
```

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

```
begin
    if BalancingState = State1 then
        закончить построение кучи на T;
        добавить T в HH;
    if BalancingState = State2 then
        отменить отложенное добавление в MH;
        добавить T в HH;
    if MH непуста then вставить MH в HH;
    BalancingState ← 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);
```

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, причём обе оценки верны в худшем случае.*

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 $|\texttt{MH}| \leq 2^b$.*

Доказательство. Из условия на переполнение буфера из алгоритма 4 имеем $n - 1 \leq 2^{b-1}$, откуда получаем $n \leq 2^b$. Но $|\texttt{MH}| \leq n$, 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 MH. Первая часть реализуется за $O(b)$ сравнений ([1, с. 181]). Вторая часть реализуется за $O(\log |\texttt{MH}|)$ сравнений ([1, с. 181]). Но из теоремы 1.5.1 мы имеем $\log_2 |\texttt{MH}| \leq b$. 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.* $\min\limits_{x \in X} x = H.\underbrace{\textbf{top}().\ldots.\textbf{top}()}_{k\text{times}}$.

Доказательство. Докажем индукцией по $\text{level}(H)$.

Если $\text{level}(H) = 0$, то H — элемент X, и утверждение учевидно.

Пусть $\text{level}(H) = k > 0$. 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. $|\texttt{HH}| \leq 6k$.

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 `HH` не более двух элементов.

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

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

□

Доказательство теоремы 1.5.1 (о корректности и асимптотике). Напрямую следует из теоремы 1.5.2 о сохранении инвариантов и леммы 1.5.4. □

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, $(O(r), O(\log^{(r)} n + r(\log k + \log r)))$ для любого $r > 0$ и $(O(\log^* n), O(\log^* n(\log k + \log\log^* n)))$.

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 $\texttt{MH}_1$. Сюда попадают кучи, построенные на элементах буфера. Размер $\texttt{MH}_1$ не превосходит $O(\log n)$.
3. Промежуточная куча $\texttt{MH}_2$, «куча куч куч». Когда $\texttt{MH}_1$ становится слишком большого размера, её нужно добавить в $\texttt{MH}_2$.
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.

Алгоритм 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 $\texttt{MH}_1$;
if *размер* $\texttt{MH}_1 > \log_2\log_2 n$ **then**
  добавить $\texttt{MH}_1$ в $\texttt{MH}_2$;
  очистить $\texttt{MH}_1$;

Переполнение буфера возникает примерно каждые $\log\log n$ вставок и требует $\log\log n$ дополнительных действий на вставку элемента в $\texttt{MH}_1$. Переполнение $\texttt{MH}_1$ возникает примерно каждые $\log n$ вставок и требует $\log n$ действий на вставку в $\texttt{MH}_2$. Отсюда следует амортизированная оценка в $O(1)$ сравнений на вставку одного элемента.

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

Описание `CascadeHeap[2]` приведено только для упрощения понимания того, как устроена многоуровневая структура. В дальнейшем все описания и доказательства будут приведены только для общего случая, т. е. `CascadeHeap[r]`.

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 $\texttt{MH}_1$, …, $\texttt{MH}_r$. Размер кучи $\texttt{MH}_t$ не превосходит $\log^{(r-t)} n$, $1 \le t < r$. Размер кучи $\texttt{MH}_r$ не превосходит n.
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 $\texttt{MH}_1$, при переполнении кучи $\texttt{MH}_1$ она вставляется в $\texttt{MH}_2$ и т. д., то есть когда очередная промежуточная куча (кроме максимальной) достигает своего предельного размера, она вставляется в промежуточную кучу следующего уровня. 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. $\texttt{MH}_1$, …, $\texttt{MH}_r$ 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

В предыдущей секции количество промежуточных куч (параметр r) было фиксированным. Размер каждой следующей кучи поддерживался равным экспоненте от размера предыдущей. Можно сохранить это свойство, при этом не ограничивая количество куч. Таким образом, $\texttt{MH}_1$ переполняется при размере 2, $\texttt{MH}_2$ — при размере $2^2 = 4$, $\texttt{MH}_3$ — при размере $2^{2^2} = 16$, …, $\texttt{MH}_t$ — при размере ${}^t2$, где ${}^an$ — операция *тетрации*.

Определение 2.3.1. Для любого положительного вещественного $a > 0$ и неотрицательного целого $n \geq 0$, тетрацию ${}^na$ можно определить реккурентно:

1. ${}^0a = 1$,
2. ${}^na = a^{({}^{n-1}a)}$, $n > 0$.

Иными словами, тетрация — это результат вычисления «степенной башни» высоты n из чисел a ([3]).

Одновременно можно избавиться от буфера и добавлять элементы сразу в $\texttt{MH}_1$, поскольку добавление в кучу размера ≤ 2 требует $O(1)$ сравнений. В остальном вставка элемента и извлечение минимума абсолютно аналогичны рассмотренным ранее структурам.

2.4 Деамортизация

Как и в случае с SimpleCascadeHeap, деамортизация нужна только для процесса балансировки. Здесь способ, рассмотренный в разделе 1.3, будет применён для каждого уровня в отдельности. Вначале мы рассмотрим деамортизацию CascadeHeap[r], затем описанный способ будет адаптирован для CascadeHeap*.

2.4.1 Деамортизация `CascadeHeap[r]`

Сделаем все $\texttt{MH}_t$ кучами с версией (см. параграф 1.3.1). При вставке возможны два вида переполнений:

1. Переполнение буфера. При таком переполнении нужно построить кучу на элементах буфера, а затем вставить её в $\texttt{MH}_1$. Назовём это *балансировкой нулевого уровня*.
2. Переполнение одной из куч $\texttt{MH}_1$, …, $\texttt{MH}_{r-1}$. При таком переполнении нужно только вставить переполнившуюся куча в промежуточную кучу следующего уровня. Назовём процесс вставки $\texttt{MH}_t$ в $\texttt{MH}_{t+1}$ *балансировкой t-го уровня*.

В любой момент времени на каждом уровне может происходить балансировка. При выполнении операции **insert** надо провести несколько операций по балансировке на каждом из них. Эти балансировки независимы: если какая-то куча $\texttt{MH}_t$ переполнилась и вставляется в $\texttt{MH}_{t+1}$, её состояние «замораживается», и в неё саму больше не будет ничего вставлено. Если в момент переполнения $\texttt{MH}_t$ в неё производится отложенная вставка, то вставку нужно завершить и сделать вставляемый элемент единственным элементом новой $\texttt{MH}_t$.

Далее будет показано, что балансировка t-го уровня может выполняться в течение $\log^{(r-t)} n$ вставок и требует $O(\log^{(r-t)} n)$ времени, значит, деамортизацию на каждом уровне можно выполнять за $O(1)$ сравнений при каждом вызове **insert**. Так можно достичь оценки в $O(r)$ сравнений на операцию добавления в худшем случае.

Для того, чтобы выполнить операцию **extractMin**, нужно экстренно завершить балансировку на каждом уровне, затем, как обычно, добавить $\texttt{MH}_1$, …, $\texttt{MH}_r$ в $\texttt{HH}$ и извлечь минимум из $\texttt{HH}$ или из буфера. Завершение балансировки на первом уровне требует достраивания кучи на элементах буфера и отмены вставки в $\texttt{MH}_1$, для этого необходимо $O(\log^{(r)} n)$ сравнений. Завершение балансировки на уровнях после первого включает в себя только отмену вставки и производится за $O(1)$ (если быть точным, за 0 сравнений).

2.4.2 Деамортизация `CascadeHeap*`

Деамортизация `CascadeHeap*` происходит абсолютно аналогично. Различие лишь в отсутствии нулевого уровня балансировки: в `CascadeHeap*`, в отличие от `CascadeHeap[r]`, нет буфера, поэтому завершение балансировки происходит за $O(1)$ на каждом уровне.

2.5 Алгоритм

За T_0, …, T_r, …обозначены отложенно вставляемые кучи. T_t вставляется в $\texttt{MH}_{t+1}$. Во время первой стадии балансировки нулевого уровня T_0 также обозначает кучу в процессе построения.

Инициализация и операции для `CascadeHeap[r]` показаны в алгоритмах 7–9, для `CascadeHeap*` — в алгоритмах 10–12.

Алгоритм 7: Инициализация `CascadeHeap[r]`

```
begin
    BalancingState ← (NoAction, …, NoAction) (r times);
    BufferSize ← 0;
    InsertionsCount ← 0;
    C ← константа из леммы 2.6.3;
```

Алгоритм 8: Операция **insert** в `CascadeHeap[r]`

```
Data: x ∈ 𝒳
begin
    if BalancingState[0] ∈ {State1, State2} then
        выполнить C операций по балансировке на нулевом уровне;
        if закончилась стадия I then
            BalancingState[0] ← State2;
    for t = 0 to r − 1 do
        if BalancingState[t] = State2 then
            выполнить C операций по балансировке на t-м уровне;
            if закончилась стадия II then
                выполнить переключение версии MH_{t+1};
                BalancingState[t] ← NoAction;
    добавить x в конец буфера;
    BufferSize ← BufferSize + 1;
    InsertionsCount ← InsertionsCount + 1;
    if BufferSize > log_2 InsertionsCount then
        BufferSize ← 0;
        BalancingState[0] ← State1;
        T[0] ← буфер;
        начать балансировку на элементах буфера и очистить буфер;
    for t = 1 to r − 1 do
        if MH_t.size() > log_2^{(r−t)} InsertionsCount then
            BalancingState[t] ← State2;
            начать отложенно вставлять MH_t в MH_{t+1};
            T[t] ← MH_t;
            // MH_t теперь пуста
            if BalancingState[t] = State2 then
                положить T[t-1] единственным элементом MH_t;
```

Алгоритм 9: Операция **extractMin** в `CascadeHeap[r]`

```
begin
    if BalancingState[0] = State1 then
        закончить построение кучи на T[0];
        добавить T[0] в HH;
        BalancingState[0] ← NoAction;
    for t = 0 to r − 1 do
        if BalancingState[t] = State2 then
            отменить отложенное добавление в MH_{t+1};
            добавить T[t] в HH;
            BalancingState[t] ← NoAction;
    for t = 1 to r do
        добавить MH_t в HH;
    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);
```

Алгоритм 10: Инициализация `CascadeHeap*`

```
begin
    BalancingState ← (NoAction, NoAction, ...) ;   // infinite array
    MaximumLevel ← 1;
    C ← константа из леммы 2.6.3;
```

Алгоритм 11: Операция **insert** в `CascadeHeap*`

```
Data: x ∈ 𝒳
begin
    for t = 1 to MaximumLevel do
        if BalancingState[t] = State2 then
            выполнить C операций по балансировке на t-м уровне;
            if закончилась стадия II then
                выполнить переключение версии MH_{t+1};
                BalancingState[t] ← NoAction;
    добавить x в MH_1;
    for t = 1 to MaximumLevel do
        if MH_t.size() ≥ 2^t then
            BalancingState[t] ← State2;
            начать отложенно вставлять MH_t в MH_{t+1};
            T[t] ← MH_t;
            // MH_t теперь пуста
            if BalancingState[t] = State2 then
                положить T[t-1] единственным элементом MH_t;
            MaximumLevel ← max(MaximumLevel, t+1);
```

Алгоритм 12: Операция **extractMin** в `CascadeHeap*`

```
begin
    if BalancingState[0] = State1 then
        закончить построение кучи на T[0];
        добавить T[0] в HH;
        BalancingState[0] ← NoAction;
    for t = 0 to r − 1 do
        if BalancingState[t] = State2 then
            отменить отложенное добавление в MH_{t+1};
            добавить T[t] в HH;
            BalancingState[t] ← NoAction;
    for t = 1 to r do
        добавить MH_t в HH;
    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);
```

2.6 Доказательства

***Теорема* 2.6.1** (о корректности и асимптотике `CascadeHeap[r]`)**.** *Для любого натурального $r > 1$, 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 $O(r)$ сравнений, каждая операция **extractMin** требует $O(\log^{(r)} n + r(\log k + \log r))$ сравнений, где k — количество вызовов **extractMin** к тому моменту, причём обе оценки верны в худшем случае.*

***Теорема* 2.6.2** (о корректности и асимптотике `CascadeHeap*`)**.** *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 $O(\log^* n)$ сравнений, каждая операция **extractMin** требует $O(\log^* n(\log k + \log\log^* n))$ сравнений, где k — количество вызовов **extractMin** к тому моменту, причём обе оценки верны в худшем случае.*

Как и в секции 1.5, для доказательства этих двух теорем будет сформулировано и доказано несколько лемм. Если явно не сказано иное, каждая лемма относится как и к `CascadeHeap[r]`, так и к `CascadeHeap*`. Все леммы и теоремы, в которых фигурирует буфер, относятся только к `CascadeHeap[r]`. Во всех доказательствах параметр r считается произвольным положительным натуральным числом.

Замечание. Если явно не сказано иное, под записью $\log x$ подразумевается двоичный логарифм ($\log_2 x$). То же самое относится к повторному логарифму ($\log^{(r)} x$) и к итеративному ($\log^* x$).

***Лемма* 2.6.1.** *Пусть в MH_t для некоторого t переполнилась в некоторый момент времени. Тогда для любого $b \geq 0$ после b вставок $|MH_t| \leq b + 1$.*

Доказательство. Сначала заметим, что операция **extractMin** может только уменьшить количество элементов в $\texttt{MH}_t$.

Элементы могут попасть в $\texttt{MH}_t$ двумя способами:

1. в результате переполнения $\texttt{MH}_t$ в неё попадает куча, отложенно вставляемая в $\texttt{MH}_t$;
2. в $\texttt{MH}_t$ добавляется новый элемент в результате завершившейся балансировки на уровне $t-1$.

После события первого типа $|\texttt{MH}_t| = 1$. Событие второго типа может произойти не чаще, чем один раз на вставку, потому что две балансировки не могут идти одновременно. Значит, при увеличении b на 1 в $\texttt{MH}_t$ попадает не более одного элемента, что и доказывает лемму. □

***Лемма* 2.6.2.** *Пусть на t-м уровне ($t \geq 0$, в случае `CascadeHeap*` $t > 0$) произошло переполнение кучи $\texttt{MH}_t$ или буфера после n вставок, и следующее переполнение на этом уровне произойдёт после $n' = n + b$ вставок для некоторого b, если за это время не произойдёт вызова **extractMin**. Тогда $|\texttt{MH}_{t+1}| \leq 2^b$.*

Доказательство для `CascadeHeap[r]`. Обозначим $s = \log^{(r-t+1)} n$. Из условий на переполнение имеем:

$$b > \log^{(r-t)}(n+b) \quad \text{по лемме 2.6.1}$$
$$b > \log^{(r-t)} n = \log s$$
$$2^b > s$$
$$|\texttt{MH}_{t+1}| \leq \log^{(r+t-1)} n = s \quad \text{из условий переполнения}$$
$$|\texttt{MH}_{t+1}| < 2^b$$

□

Доказательство для `CascadeHeap*`.

$$b \geq {}^t2 \quad \text{по лемме 2.6.1}$$
$$2^b \geq {}^{t+1}2$$
$$|\texttt{MH}_{t+1}| \leq {}^{t+1}2 \quad \text{из условий переполнения}$$
$$|\texttt{MH}_{t+1}| \leq 2^b$$

□

***Лемма* 2.6.3.** *Пусть на t-м уровне ($t \geq 0$, в случае `CascadeHeap*` $t > 0$) произошло переполнение кучи $\texttt{MH}_t$ или буфера после n вставок, и следующее переполнение на этом уровне произойдёт после $n' = n + b$ вставок для некоторого b,*

если за это время не произойдёт вызова ***extractMin****. Тогда найдётся константа* C *такая, что балансировку можно выполнить за не более чем* $C \cdot b$ *сравнений в худшем случае.*

Доказательство. Балансировка может состоять из не более чем двух стадий: построение кучи на элементах буфера и вставка в промежуточную кучу следующего уровня.

Рассмотрим первую стадию. Пусть размер буфера при переполнении равен s. Тогда $s \leq b$ в силу монотонности повторного логарифма. На s элементах можно построить бинарную кучу за $O(s) \in O(b)$ сравнений согласно [1, с. 181].

Рассмотрим вторую стадию. Из леммы 2.6.2 известно, что $|\texttt{MH}_t| \leq 2^b$. Вставку реализуется за $O(|\texttt{MH}_t|)$ сравнений, что есть $O(b)$.

Обе части балансировки реализуются за $O(b)$ сравнений. Значит, вся балансировка на отдельно взятом уровне реализуется за $O(b)$ сравнений, и существование искомой константы следует из определения «O большого». □

***Лемма* 2.6.4.** *На каждом уровне одновременно не может идти более чем одна балансировка.*

Доказательство. Во время каждой операции **insert** на каждом уровне проделывается C шагов балансировки, где C — константа из леммы 2.6.3. Но эта константа определена так, чтобы балансировка успела завершиться за b шагов, где b — минимально возможное количество вставок до начала следующей балансировки. □

***Лемма* 2.6.5.** *Количество уровней в* `CascadeHeap*` *(т. е.* MaximumLevel*) не превосходит* $\log^* n + 1$.

Доказательство. Из условий на переполнение следует, что если $\mathsf{MaximumLevel} = t$, то $n \geq {}^{t-1}2$. Значит, $\log^* n \geq t - 1$, то есть $t \leq \log^* n + 1$. □

***Лемма* 2.6.6.** *Для любого валидного* $t > 0$ *верно* $\text{level}(\texttt{MH}_t) \leq t + 1$.

Доказательство. Докажем индукцией по t.

Если $t = 1$, то все элементы $\texttt{MH}_t$ — кучи (уровня 1), построенные на элементах буфера (в случае `CascadeHeap[r]`) или единичные элементы (в случае `CascadeHeap*`). В обоих случаях $\text{level}\,\texttt{MH}_1 \leq 2$.

Если $t > 1$, то все элементы $\texttt{MH}_t$ имеют уровень, не превосходящий максимальный уровень $\texttt{MH}_{t-1}$. Тогда из предположения индукции получаем, что $\text{level}(\texttt{MH}_t) \le \text{level}(\texttt{MH}_{t-1}) + 1 \le t + 1$. □

***Лемма* 2.6.7.** *Положим*

$$R = \begin{cases} r \text{ в случае } \texttt{CascadeHeap[r]} \\ \text{MaximumLevel в случае } \texttt{CascadeHeap*} \end{cases}.$$

Тогда $\text{level}(\texttt{HH}) \le R + 2$.

Доказательство. Все элементы, попадающие в HH, попадают туда из какой-то $\texttt{MH}_t$ или из кучи, которая когда-то была одной из $\texttt{MH}_t$. По построению $t \le R$. Из теоремы 2.6.6 уровень добавляемого элемента не превосходит $R + 1$. Но тогда $\text{level}(\texttt{HH}) \le R + 2$. □

***Теорема* 2.6.3** (о сохранении инвариантов)**.** *В любой момент сохраняются следующие инварианты:*

1. $\texttt{MH}_t$ *и все её элементы — корректные бинарные кучи для любого валидного* t*;*
2. HH *и все её элементы — корректные бинарные кучи;*
3. *Если балансировка на* t*-м уровне находится в стадии II, то множество* **T[t]***, которое вставляется в* $\texttt{MH}_{t+1}$ *— корректная бинарная куча;*

4r. (версия `CascadeHeap[r]`*)* $|\texttt{HH}| \le (4r + 4)k$*;*

4. (версия* `CascadeHeap*`*)* $|\texttt{HH}| \le k \cdot (4 \log^* n + 8)$.

Доказательство. Доказательство этой теоремы техническое и построено так же, как и доказательство теоремы 1.5.2.

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 или $\texttt{MH}_1$ и, возможно, несколько операций по балансировке. Добавление в буфер не затрагивает ни одно из интересующих нас множеств. Добавление в $\texttt{MH}_1$ происходит только в `CascadeHeap*`. Поскольку в ней нет нулевого уровня балансировки, на инвариант (3) вставка не влияет. Инвариант (1) сохраняется по свойству вставки в кучу.

Посмотрим на балансировку.

При выполнении первой стадии балансировки, опять же, ни одно из интересующих нас множеств не изменяется. К моменту завершения первой стадии множетство $\mathsf{T}[0]$ представляет собой бинарную кучу согласно алгоритму 8 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 алгоритмам 8 и 11, 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 все множества $\mathsf{T}[t]$ и $\texttt{MH}_t$ в `HH`; все они являются корректными кучами по инвариантам (1) и (3). Таким образом, мы не нарушим инварант (2).

Посчитаем, сколько элементов было добавлено в `HH`. Для каждого уровня t могло быть добавлено не более двух новых элементов ($\texttt{MH}_t$ и $\mathsf{T}[t]$). Значит, в `CascadeHeap[r]` было добавлено не более $2r$ элементов, в `CascadeHeap*` — не более $2(\log^* n + 1)$ (теор. 2.6.5).

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

В случае `CascadeHeap[r]` за одну операцию удаления в `HH` было добавлено не более $2r + 2\,\text{level}(\texttt{HH}) \leq 2r + 2(r+2) = 4r + 4$ элементов.

В случае `CascadeHeap*` в `HH` было добавлено не более $2(\log^* n + 1) + 2\,\text{level}(\texttt{HH}) \leq 2(\log^* n + 1) + 2(\log^* n + 3) = 4\log^* n + 8$ элементов. Кроме того, $\log^* n$ монотонно неубывает по n. Это доказывает соблюдение инвариантов (4r) и (4*). В этих утверждениях мы использовали теорему 2.6.7. □

Доказательство теорем 2.6.1 и 2.6.2. Доказательство этих теорем напрямую следует из теоремы 2.6.3 о сохранении инвариантов и леммы 2.6.4. □

Наконец, докажем теоретическую нижнюю оценку.

Теорема* 2.6.4.** *Пусть некоторая структура данных реализует операции* ***insert *и* ***extractMin****, причём операция* ***insert*** *требует* $O(1)$ *сравнений в худшем случае. Тогда для выполнения* k *операций* ***extractMin*** *требуется* $\Omega(k \log k)$ *сравнений в худшем случае.*

Доказательство. Заметим, что выполнить k извлечений минимума может оказаться не проще, чем отсортировать k элементов. Из теоретико-информационных соображений известно, что на это требуется $\Theta(k \log k)$ сравнений[1, с. 222].

На все операции вставки было потрачено $O(n)$ сравнений. Тогда при $k = \omega(\frac{n}{\log n})$ имеем $\Theta(k \log k) \notin O(n)$, то есть сравнений, получаемых от добавления элементов, недостаточно для проведения сортировки. Значит, в худшем случае требуется $\Omega(k \log k)$ дополнительных сравнений. □

Глава 3. Заключение

Нами была построена структура данных `CascadeHeap`, улучшающая известные на данный момент верхние оценки на реализацию очереди с приоритетом при ограниченном числе удалений. Построенная нами оценка очень близка к теоретически оптимальной.

Прикладной ценности наша стуктура данных не имеет из-за очень высокой скрытой в асимптотике константы. Однако полученный результат может использоваться для улучшения верхних оценок других алгоритмов.

Ниже в этой главе мы рассмотрим возможное применение `CascadeHeap` и опишем возможные дальнейшие направления работы по теме.

3.1 Применение к задачам частичной сортировки

Задача `PartialSorting` — это упрощённая форма задачи сортировки.[4] В ней требуется по множеству из n сравнимых элементов выдать k наименьших в отсортированном порядке. При $k = n$ задача `PartialSorting` эквивалентна сортировке.

***Теорема* 3.1.1.** *Любое решение задачи* `PartialSorting` *при* $k > 0$ *требует* $\Omega(n + k \log k)$ *сравнений в худшем случае.*

Доказательство. Задача `PartialSorting` сводится к задачам выбора наименьших k элементов и их сортировки. Отсортировать k элементов невозможно быстрее, чем за $\Omega(k \log k)$ из теоретико-информационных соображений[1, с. 222]. Для выбора наименьших k элементов нужно как минимум выбрать минимальный, что требует $n - 1 = \Omega(n)$ сравнений. Сумма этих нижних оценок даёт требуемую оценку в $\Omega(n + k \log k)$. □

Объявленная оценка является строгой: существуют алгоритмы, решающие `PartialSorting` за $O(n + k \log k)$; самый простой из них построен на базе QuickSort и QuickSelect.

Задача `IncrementalSorting` — усложнённая версия задачи `PartialSorting`. В этой формулировке, как и прежде, все элементы заданы заранее, однако значение k заранее неизвестно и требуется уметь быстро переходить от k отсортированных минимальных элементов к $k+1$ (иначе говоря, каждый раз возвращать минимальный элемент среди оставшихся). Ясно, что `IncrementalSorting` не проще `PartialSorting`, поэтому нижняя граница из теор. 3.1.1 верна и здесь. Алгоритм *IncrementalQuickSort*[5], достигающий нижнюю оценку амортизированно в худшем случае, был построен Paredes и Navarro на базе QuickSelect. Однако алгоритма, достигающего нижней оценки в худшем случае и не амортизированно (то есть возвращающего k-й минимум за $O(\log k)$ с препроцессингом за $O(n)$) на момент написания статьи нам не известно.

Задачу `PriorityQueue` можно назвать усложнённой версией `PartialSorting`. Фактически, это описание операций очереди с приоритетом: требуется вставлять элемент в структуру и извлекать из структуры минимальный к данному моменту элемент. Нижняя оценка из теор. 3.1.1 верна и для этой задачи. Paredes и Navarro представили структуру данных *QuickHeap*[6], решающую эту задачу за $O(n + k \log k)$ амортизированно и в среднем (представленный ими алгоритм является вероятностным). Не вероятностное решение этой задачи с такой асимптотикой авторам не известно, даже если разрешить амортизированные оценки сложности.

Все упомянутые здесь задачи используются как подзадачи в более сложных алгоритмах. Например, в работе [6] показано, как использовать `IncrementalSorting` и `PriorityQueue` для оптимизации алгоритмов Прима и Краскала построения остовного дерева.

Представленная в статье структура данных `CascadeHeap` позволяет решать три упомянутые здесь задачи в худшем случае и без амортизаци за лучшую асимптотику, чем было представлено ранее. Теоретическая оценка достигается с точностью до множителя $\log^* n \cdot \log \log^* n$. Это можно использовать для улучшения известных верхних оценок решений некоторых задач.

Стоит заметить, что `CascadeHeap` достаточно сложна и громоздка для реализации на практике, поэтому, хоть данная оценка и очень близка к теоретически оптимальной, практической ценности наша структура данных не имеет.

3.2 Дальнейшая работа

Все асимптотические оценки, приведённые в статье, рассматриваются в модели сравнений. Однако можно немного модифицировать построенную структуру данных так, чтобы оценки остались верны и в более сложных моделях, таких, как Pointer Machine и RAM-модель. Для этого требуется более аккуратный анализ, особенно в части деамортизации. Например, сейчас операция копирования буфера размера $O(\log^{(r)})$ считается выполнимой за $O(1)$, поскольку не требует сравнений. Копирование активно используется в части деамортизации, однако при желании его можно избежать.

Стоит обратить внимание, что при операции **extractMin** все промежуточные кучи очищаются. За счёт этого операция вставки в `CascadeHeap*` на самом деле занимает не $O(\log^* n)$, а $O(\log^* q)$, где q — количество вставок с момента последнего извлечения. Авторам не удалось получить из этого никаких асимптотических улучшений, однако можно попробовать вывести из этого *queueish*-свойство[1] или аналогичные ему.

[1] Свойство queueish означает, что время доступа к элементу x лежит в $O(\log q(x))$, где $q(x)$ — количество элементов, просмотренных с момента последего доступа к x.

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.
2. *Brodal G. S.*, *Okasaki C.* Optimal Purely Functional Priority Queues // Journal of Functional Programming. — 1996. — Т. 6. — С. 839—857.
3. Тетрация — Википедия. — URL: https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%82%D1%80%D0%B0%D1%86%D0%B8%D1%8F (дата обр. 23.06.2016).
4. Partial Sorting — Wikipedia. — URL: https://en.wikipedia.org/wiki/Partial_sorting (дата обр. 26.06.2016).
5. *Navarro G.*, *Paredes R.* Optimal Incremental Sorting // Proceedings of the Meeting on Algorithm Engineering & Expermiments. — Miami, Florida : Society for Industrial, Applied Mathematics, 2006. — С. 171—182. — URL: http://dl.acm.org/citation.cfm?id=2791171.2791187.
6. *Navarro G.*, *Paredes R.* On Sorting, Heaps, and Minimum Spanning Trees // Algorithmica. — 2010. — Т. 57, № 4. — С. 585—620. — DOI: 10.1007/s00453-010-9400-6. — URL: http://dx.doi.org/10.1007/s00453-010-9400-6.