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

Стр.

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*, реализующая 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, 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

Описываемая структура 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) по-

смотреть 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 части (x) будет построена описанная структура данных. В части (x) будет проведена деамортизация и получены оценки в худшем случае. В части (x) 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: $x < y \overset{def}{\iff} repr(x) < repr(y)$, где

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

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 $\log n$. 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: $x \in \mathcal{X}$
если *размер буфера* $> b$ **то**
    создать двоичную кучу T из элементов буфера;
    очистить буфер;
    добавить кучу T в MH;
    если $n > 2^{b+1}$ **то**
        $b = b + 1$;
добавить x в конец буфера;

Расписать подробнее, как изменяется размер буфера; как зависит от удалённых элементов?

1.2.2 Операция нахождения минимума

Максимальный элемент может находиться в любой из трёх частей. Для его нахождения необходимо просмотреть все элементы B, а также MH.top() и HH.top(), если соответствующая куча непуста.

1.2.3 Операция извлечения минимума

Опишем 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.

алгоритмы надо бы взять в рамочку

Function Yield(x)

```
если x ∈ 𝒳 то
    return x;
иначе
    если x.hasLeftChild то HH.insert(x.leftChild);
    если x.hasRightChild то 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 MH
        return Yield (MH);
    case HH
        T = HH.top();
        HH.extractMin();
        return Yield (T);
```

Для удаления минимального элемента его нужно найти, как описано в пункте 1.2.3. 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*— двоичная куча, поддерживающая следующие операции:

1. Извлечь 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). Кроме того, в куче будет отдельно храниться актуальный номер версии V. Таким образом, 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 и в дальнейшем пользоваться только ссылками версии V.

Насколько формальноо нужно?

Расписать, почему версий в каждой вершине будет немного, и как их удалять. Сказать, как проталкивать версию вглубь при расклеивании дерева.

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.

Сначала мы докажем, что достигается асимптотика в $O(1)$ на добавление в худшем случае. После будет показано, как поступать с запросами на удаление, пришедшими в середине балансировки, и что они не ухудшают асипмтотику.

***Теорема* 1.3.1.** *Пусть размер буфера равен b. Тогда в любой момент времени* $|MH| < 2^{b+1}$.

Доказательство. Пусть n— количество вызовов операции **insert**. Тогда согласно алгоритму 1.2.1 верно $n < 2^{b+1}$. Но $|MH| \leq n$, откуда следует требуемое неравенство. □

***Теорема* 1.3.2.** *Пусть размер буфера равен b. Тогда существует некоторая константа C такая, что балансировку можно выполнить за не более чем $b \cdot C$ сравнений в худшем случае.*

Доказательство. Покажем, что каждая часть требует $O(b)$ сравнений.

I. Построить двоичную кучу можно за $O(b)$ сравнений, согласно something by somewhere

II. Из теоремы 1.3.1 получаем, что высота MH не превосходит b. Значит, добавление возможно за $O(b)$, согласно something by somewhere.

III. Переключение версии требует 0 сравнений.

Каждый шаг выполняется за $O(b)$, значит, балансировка в целом выполняется за $O(b)$. □

Для ребалансировки 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.

Кроме этого, хотелось бы не добавлять слагаемых $O(\log n)$ в асимптотику удаления, оставив из обязательных операций с такой сложностью только просмотр буфера, поскольку это будет играть роль в дальнейших оптимизациях.

Между очисткой буфера и переключением версии MH старые элементы буфера находятся «в подвешенном состоянии». Назовём их множество T. В течение операции I T— строящаяся двоичная куча, в течение операции II— двоичная куча, после операции III T перестаёт существовать. Таким образом, во случае поступления запроса на удаление во время операций I и II необходимо также просмотреть множество T на предмет наличия в нём максимального элемента.

Рассмотрим, как производить удаление минимума в зависимости от стадии балансировки и от того, где находится минимальный элемент.

I. Построение кучи

- Если минимальный элемент найден в буфере, удалить его и ничего больше не делать.
- Если минимальный элемент найден в множестве T, достроить бинарную кучу на T, извлечь минимум из T, добавить T в НН, отменить балансировку и вернуть извлечённый минимум.
- Если минимальный элемент найден в МН или в НН, удалить его согласно алгоритму 1.

II. Добавление T в МН В этом случае необходимо добавить T в MH, отменить балансировку и найти и удалить минимальный элемент согласно алгоритму 1.

III. Переключение версии / После балансировки Операция III атомарна, поэтому в процессе её выполнения запрос на удаление прийти не может. Если запрос пришёл после выполнения балансировки, нужно найти и удалить минимальный элемент согласно алгоритму 1.

Таким образом, мы готовы сформулировать итоговый алгоритм добавления в кучу и удаления из неё минимального элемента.

1.4 Алгоритм

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

1.4.1 Доказательство корректности

Тут ещё ничего нет (А что вообще должно быть?)

Algorithm 3: Инициализация деамортизированной кучи

begin

```
BalancingState ← NoAction;
BufferCapacity ← 1;
BufferSize ← BufferCapacity;
InsertionsCount ← 1;
C ← константа из теор. 1.3.2 ;
```

1.4.2 Доказательство времени работы

Тут ещё ничего нет (А будет ссылка на две теоремы и доказательство маленькой высоты НН)

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

Data: $x \in \mathcal{X}$
begin
  если *BalancingState* $\in$ *{StateI, StateII }* **то**
    выполнить C операций по балансировке;
    если *закончилась стадия I* **то**
      BalancingState $\leftarrow$ StateII ;
    если *закончилась стадия II* **то**
      выполнить переключение версии MH;
      BalancingState $\leftarrow$ NoAction ;
  добавить x в конец буфера;
  BufferSize $\leftarrow$ BufferSize + 1;
  InsertionsCount $\leftarrow$ InsertionsCount + 1;
  если *BufferSize* = *BufferCapacity* **то**
    BufferSize $\leftarrow$ 0;
    BalancingState $\leftarrow$ StateI ;
    начать балансировку на элементах буфера и очистить буфер;
    while *InsertionsCount* $> 2^{BufferCapacity}$ **do**
      BufferCapacity $\leftarrow$ BufferCapacity + 1;

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

```
begin
    если BalancingState = StateI то
        закончить построение кучи на T;
        добавить T в HH;
    если BalancingState = StateII то
        // экстренно завершаем балансировку
        BalancingState ← NoAction;
        отменить отложенное добавление в MH;
        добавить T в HH;
    BalancingState ← 0;
    просмотреть буфер B;
    просмотреть вершины MH и HH, если соответствующие кучи непусты;
    среди просмотренных элементов определить положение минимального
    элемента Location : B, MH, HH;
    switch Location do
        case B
            удалить минимум из буфера;
        case MH
            return Yield (MH);
        case HH
            T = HH.top();
            HH.extractMin();
            return Yield (T);
```

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 $(O(\log^* n), O(\log^* n \cdot \log k))$.

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 МН: одна во внутренней структуре, «куча куч», вторая— во внешней, «кучас 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 бу-

фер 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.

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.