Конспект о $SoftHeap$.

Артём Юсупов

6 мая 2019 г.

1 Аннотация.

Что вообще такое Soft heap? $SoftHeap$ это всего лишь вариация на тему приоритетной очереди (здесь и далее везде имеется в виду очередь для минимума). Эту структуру придумал Бернард Чазхель для быстрого решения задачи MST. На самом деле это целое семейство структур данных, зависящих от параметра ε. Давайте посмотрим на операции нашей кучи и поймём зачем она нужна:
- $Create()$ $O(1)$
- $Insert(Heap, key)$ $O^*(\frac{1}{\varepsilon})$
- $Meld(Heap_1, Heap_2)$ $O^*(1)$
- $Delete(Heap, element)$ $O^*(1)$
- $FindMin(Heap)$ $O^*(1)$
Думаю, пояснений требует только операция $Meld$. Она берёт две кучи и объединяет их в одну. При этом исходные кучи становятся невалидными. Зачем же нужна наша куча? В дальнейшем мы покажем, что с её помощью можно быстро решать задачу MST, кроме того она хороша в приближённых алгоритмах.

Интересный вопрос. Внимательный читатель задастся вопросом: неужели мы преодолели теоретический барьер на время сортировки и можем теперь сортировать за $O(nlog(1/\varepsilon))$? На самом деле не совсем. Дело в том, что наша куча будет иногда увеличивать ключи в элементах. Такие элементы мы будем называть испорченными. В чём идея? Ну, таким образом будет уменьшаться энтропия в наших данных (можно считать, что энтропия это "вариативность"данных). Это одна из идей, лежащих в основе нашей структуры данных. Второй идеей будет то, что мы будем объединять элементы в группы и оперировать с этими группами, тем самым экономя время.

2 Введение.

Формальные требования к структуре данных. В самом начале мы фиксирум число $\varepsilon \in (0; 1/2]$. Теперь мы будем требовать следующее: в каждый момент времени, если было произведено n операций $Insert$, то испорченных элементов в куче не более, чем $n\varepsilon$.

Важное замечание. Давайте поймём, сколько на самом деле мы требуем от нашей структуры. Заметим, что не стоит думать, что в нашей куче за всё время будет только $n\varepsilon$ испорченных элементов. Мы можем испортить несколько элементов, затем удалить их, затем снова испортить. Таким образом мы можем добиться того, чтобы большинство элементов были испорчены.
Несмотря на предыдущее замечание $SoftHeap$ весьма полезна.

Полезные фичи.

- Если мы положим $\varepsilon = 1/2n$, где n - финальное количество вставок, то наша куча преваращается в обычную кучу с логарифимическим временем вставки.
- Более интересным является тот факт, что наша куча по сути содержит в себе неявный поиск приближённой медианы. Давайте положим ε равным маленькой константе. Тогда мы можем сделать n вставок и $\lfloor n/2 \rfloor$ удалений минимума(поиск + удаление). Это займёт линейное время. Среди удалённых ключей выберем наибольший. Этот ключ отличается от медианы не более чем на $n\varepsilon$ позиций.
- Предыдущее замечание не должно вводить читателя в заблужение, что $SoftHeap$ это просто структура для поиска медианы. Она может намного больше. Например, давайте покажем, что иногда $SoftHeap$ хорошо работает почти как обычная куча. Положим $\varepsilon = 1/2$. Если нам дана последовательность операций $Insert$ и $FindMin$ и вставляемые ключи идут в невозрастающем порядке, то, несмотря на высокое значение ε, как минимум в половине случаев мы будем давать правильный ответ на запрос минимума.

3 Приложения.

Здесь приведены несколько примеров применений посерьёзнее.

- $SoftHeap$ была создана в основном для решения задачи MST за $O(m\alpha(m, n))$, где α - обратная функция Аккермана.
- Другое, более простое применение, это динамическое поддеражние процентилей. Например, если мы хотим узнать, с какого значения начинаются лучшие 1/10 студентов, то с помощью $SoftHeap$ мы можем амортизировано быстро это сделать.
- $SoftHeaps$ дают интересный метод нахождения медианы в линейное время. Более того, даже k - й элемент. Как это сделать? Положим $\varepsilon = 1/3$. Вставим n наших значений, удалим 1/3. Далее вызовем операции $FindMin$ и $Delete$ $n/3$ раз. Выберем наибольший номер, среди найденных. Тогда он делит исходный массив (в худшем случае) на части, относящиеся как 1 : 2. Тогда суммируя ряд $2n/3 + 4n/9 + ...$ получаем $O(n)$.
- Четвёртым применением является приблиизительная сортировка. Это такая сортировка, что в ней максимум $n\varepsilon^2$ пар инверитрованы. Это оставляется в качестве упражнения для читателя.
- Пятым применением является более сильный вариант приближённой сортировки. Это такая сортировка, в которой для каждого элемента, его позиция отличается от позиции в честной сортировке не более чем на $n\varepsilon$. Здесь не будет приведено всего алгоритма, а только его набросок без доказательства. Сначала мы вставляем все n элементов в нашу кучу. Затем делаем операции $FindMin$ и $Delete$. Затем разбиваем эти операции на l групп, в каждой по $\lceil 2\varepsilon n \rceil$ операций (можем считать, что ровно столько). Для каждого элемента будем хранить "время когда он испортился и его настоящее значение. Затем в каждой группе будем выбирать максимальный настоящий ключ среди тех, которые до начала операции с это группой не были испорчены. Утверждается, что эти ключи будут идти в возрастающем порядке и поделят наши элементы на группы из непересекающихся полуинтервалов. Кроме того, в каждом таком полуинтервале лежат от εn до $6\varepsilon n$ настоящих ключей. Ну тогда заменив теперь ε на $\varepsilon/6$ и вставив каждый элемент в нужный интервал за $O(1)$ получим требуемое.

4 Сама структура.

Здесь будет приведена не оригинальная реализация Чazзеля из его статьи (она базировалась на модификации биномиальной кучи), а более упрощённая, которая была предложена

Капланом и Цвиком[ISBN 978-0-89871-680-1].

Наша куча будет представлять из себя список бинарных деревьев, деревья состоят из узлов, в каждом узле хранится $list[x]$ список ключей (возможно пустой) и $ckey$ - верхняя граница. Для всех элементов в $list[x]$ будем считать, что их ключ это $ckey[x]$(Здесь нам и нужно будет увеличивать ключи). Будем обозначать как $right[x]$ и $left[x]$ левые и парвые деревья узла x соответственно. Если x не имеет, например, левого ребёнка, то будем писать $left[x] = \perp$. Кроме того, у каждого узла будет свой ранг. Ранг сыновей (если они есть) на 1 меньше, чем ранг родителя. После того, как узел был создан, его ранг никогда не меняется. Будет обозначать его $rank[x]$. Также каждому x поставим в соответствие $size[x]$. Пусть $r = \lceil \log_2 \frac{1}{\varepsilon} \rceil + 5$. Тогда $size[x] = s_k$, если $rank[x] = k$.

$$s_k = \begin{cases} 1, & if\ k \leq r \\ \lceil \frac{3s_{k-1}}{2} \rceil, & otherwise \end{cases}$$

$s_0 = s_1 = ... = s_r = 1,\ s_{r+1} = 2, s_{r+2} = 3, ...$
Кроме того, будем поддерживать инвариант кучи на наших $ckey$.
Также хотим, чтобы деревья в кучу были упорядочены по рангу их корней.
В основном списке, в каждом узле будет указатель на соответствующее дерево, а также указатели $next$ и $prev$, смысл которых понятен. Дополнительно будем хранить в каждом $suffix - min$, указатель на узел с минимальным $ckey$ на суффиксе. Рангом кучи будем называть максимальный ранг её деревьев.

5 Основные операции.

Sift. Это основная операция, на которой все остальные будут базироваться. Что она будет делать? Если количество элементов в $list[x]$ становится меньше, чем $size[x]/2$, и при это x не является листом, то мы можем использовать $sift(x)$, чтобы добавить элементов из сыновей. Без ограничения общности можем счиать, что $ckey[left[x]] < ckey[right[x]]$. Тогда мы конкатенируем списки $list[x]$ и $list[left[x]]$, записываем результат в $list[x]$, а $left[x]$ очищаем. Затем, если $left[x]$ был листом, то $left[x] = \perp$, иначе рекурсивно вызваемся от $left[x]$.

Meld. Сначала научимся объединять два дерева одного ранга. Пусть на вход даются деревья x и y, тогда создадим пустой узел z и положим $left[z] \leftarrow x, right[z] \leftarrow y$. Затем вызовем $Sift(z)$.
Теперь, чтобы объединить 2 кучи будем действовать как в биномиальной куче. В конце обновим все $suffix_min$.

Insert. Реализуется через $Meld$ и создание кучи с 1 элементом.

ExtractMin. Смотрим на $suffix - min$ у головы списка, возвращаем произвольный элемент из $list[suffix - min[head]]$. Если размер $list[x]$ становится слишком маленьким, то вызываем $Sift(x)$.

Delete. Просто реализуется лениво через пометку $Deleted[item] \leftarrow True$, вся работа выполняется в $ExtractMin$.

6 Корректность.

6.1 Корректность операций.

6.1.1 Лемма

Формулировка. $ExtractMin$ всегда возвращает элемент с минимальным ключом.

Докакзательство. Действительно, этот элемент - это один из корней. Кроме того, т.к. его ключ минимален, то $suffix - min$ укажет на его узел.

6.2 Оценки на число испорченных вершин.

6.2.1 Лемма

Формулировка. Если x - узел ранга не больше r, тогда $|list[x]| = 1$, иначе если x не лист и имеет ранг $k \geq r$, то $\frac{1}{2}size[x] \leq |list[x]| \leq 3size[x]$.

Доказательство. Первая часть очевидна.
Нижняя граница во второй части тоже (из операции $Sift(x)$).
Дальше давайте доказывать по индукции наше утверждение. Для $k \leq r$ оно выполняется. Далее рассмотрим $k \geq r$. Если x имеет такой ранг, то пока мы не вызвали $Sift(x)$, то всё очевидно. Когда же мы вызываем $Sift(x)$, то $|list[x]| < size[x]$ и $rank[left[x]] = k - 1$, $size[left[x]] < 3size[left[x]] \leq 3\frac{2}{3}size[x] = 2size[x]$.

6.2.2 Лемма

Формулировка. Если было вставлено n элементов, то количество узлов ранга k не превосходит $n/2^k$.

Докакзательство. Просто индукция и соображения того, узел ранга k образуется из 2 узлов ранга $k - 1$.

6.2.3 Лемма

Формулировка. Количество испорченных вершин не превосходит $n\varepsilon$ при n вставках.

Доказательство. Каждый узел ранга не больше r содержит лишь 1 элемент. Поэтому испорченные элементы только в узла ранга $> k$. Просуммируем:

$$\sum_{k>r} \frac{n}{2^k} 3s_k \leq \frac{n}{2^r} \sum_{k>r} 6(\frac{1}{2})^{k-r}(\frac{3}{2})^{k-r} = \frac{6n}{2^r} \sum_{i \geq 1} (\frac{3}{4})^i = \frac{18n}{2^r} < \varepsilon n$$

.

7 Анализ времени работы.

Мы введём потенциал для куч, деревьев и узлов. Потенциал кучи ранга k будет $k+1$. Потенциал деерва с корнем x это $(r+2)del(x)$, где $del(x)$ - количество удалённых элементов из дерева с момента создания или с момента последнего $Sift$. Также введём потенцилы узлов. Потенциал корневого узла ранга k будет $k+7$. Если x не корневой, то 1. Кроме того, у нас будет заряд(монетки) на элементах. В дальнейшем мы будем часто говорить о потенциале как о монетках, надеюсь вас это не смутит.

Sift. Начнём с анализа нашей основной операции. Пусть k - ранг x, $y \leftarrow left[x]$. Если $|list[y]| < \frac{1}{2}size[y]$, то y - лист. Тогда y исчезнет в результате операции. Тогда общее время работы будет 0.
С другой стороны, если $|list[y]| \geq \frac{1}{2}size[y] \geq \lceil\frac{1}{2}size[y]\rceil$. Тогда мы можем разделить время этой работы на все элементы из $list[y]$. Тогда заряд каждого элемента изменится не более, чем не $\lceil\frac{size[y]}{2}^{-1} = \lceil\frac{s_{k-1}}{2}\rceil^{-1}$. Элемент может быть заряжен только один раз на каждом ранге, тогда суммарный заряд будет

$$\sum_{k\geq 0}\lceil\frac{s_k}{2}\rceil^{-1} \leq r+2\sum_{i\geq 0}(\frac{2}{3})^i = r+6 = O(\log(\frac{1}{\varepsilon}))$$

. Далее рассмотрим операцию $combine(x,y)$, которая объединяет 2 дерева одного ранга. Потенциал узлов x и y уменьшается с $k+7$ до 1. Тогда общий потенциал уменьшется на $2k+12$. Далее, так как образуется новое дерево, то потенциал увеличивается на $k+8$. Остаётся $k+4$. Затем 1 потенциал тратится, чтобы оплатить время операции и ещё один тратится, если текущее дерево имеет наибольший ранг. Остаётся $k+2$, которые тратятся на операцию $update-suffix-min$. Тогда понятно $Meld$ работает за константу.

ExtractMin. Пусть x - корень дерева, в $list[x]$ содержится минимальный элемент e. Если после удаления e x выполнено, что $|list[x]| \geq \frac{1}{2}size[x]$, или $|list[x]| > 0$ и x - лист, тогда больше действий предпринимать не надо. Тогда суммараная стоимость будет $1+(r+2) = r+3$. Это прибавим к заряду e, больше e не будет заряжаться.
В ином случае может быть выполнена операция $Sift(x)$. Тогда $del(x) \geq \lceil\frac{size[x]}{2}\rceil$, и потенциал дерева был как минимум $(r+2)\lceil\frac{size[x]}{2}\rceil = (r+2)\lceil\frac{s_k}{2}\rceil$, где $k = rank[x]$. Можно показать, что

$$(r+2)\lceil\frac{s_k}{2}\rceil \geq (k+1).$$

Кроме того, для $0 \leq k \leq r+1$, мы имеем $(r+2)\lceil\frac{s_k}{2}\rceil = r+2 \geq k+1$. Для $k = r+2$, мы имеем $s_{r+2} = 3$, $r+2\lceil\frac{s_{r+2}}{2}\rceil = 2(r+2) > r+3$. Для $k \geq r+3$, мы имеем $(r+2)\lceil\frac{s_k}{2}\rceil \geq \frac{r+2}{2}(\frac{3}{2})^{k-r} \geq k+1$, т.к. $\frac{1}{2}(\frac{3}{2})^{k-r} \geq \frac{k+1}{r+2}$, для $k \geq r+3$. Тогда этого хватает, чтобы заплатить за $update-suffix-min$.
Наконец, проанализируем случай, когда x лист, который надо удалить. Ну тогда у нас освобождается $k+7$ единиц потенциала, чего хватает, чтобы заплатить за $update-suffix-min$.

Insert. Можно понять, что для каждого узла в будущем потребуется не более, чем $8+(r+6)+(r+3) = 2r+17 = O(\log(\frac{1}{\varepsilon}))$ единиц потенциала. Этим мы завершаем наше доказательство.