Лекция по алгоритмам #5
Тема: Кучи

(k-heap, leftist, skew, add in O(1), merge in O(1), build in O(n))

2 сентября

Собрано 19 января 2015 г. в 20:08

Содержание

1 Heap

Куча – структура данных реализующая интерфейс очереди с приоритетами (Priority Queue).
Basic interface:

- Add (Insert)
- GetMin
- DelMin

Extended interface:

- Delete
- (Decrease/Increase)Key
  Delete + Add
- Mergable $\rightarrow$ Add, Delete
  Add: если реализован Merge, то реализован Add через Merge с новым элементом
  Delete: удаляем элемет, Merge'им его детей, и подвешиваем результат к родителю удалённого элемента
- Find $\mathcal{O}(1)$
  храним обратные ссылки, поддерживаем ответ по элементу, храним в массиве hash-table

2 K-Heap

Binary Heap – частный случай K-Heap, поэтому все операции знакомые по Binary Heap переносятся и на K-Heap, однао некоторые оценки времени отличаются.
Рассмотрим операции $SiftUp$ и $SiftDown$:

Время работы	**SiftUp**	**SiftDown**
Binary Heap	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(\log_2 n)$
K-Heap	$\mathcal{O}(\log_k n)$	$\mathcal{O}(k \log_k n)$

Обратите вниение на время работы операции $SiftDown$ у K-Heap – $\mathcal{O}(\frac{k}{log_k n})$. Почему так происходит? При просеивании вниз необходимо просмотреть всех детей число которых равно k.

Построение кучи на массиве без использования дополнительной памяти (Inplace)

Удобнее всего k-ичную кучу хранить в виде массива $A[0..N-1]$, у которого нулевой элемент $A[0]$ — элемент в корне, а потомками элемента $A[i]$ являются $A[2i+1] \dots A[2i+K]$. Соответственно, отец элемента $A[i]$ – это элемент $A[\frac{i-1}{k}]$.

Построение K-Heap за $\mathcal{O}(n)$ и доказательство времени построения

Дан массив $A[0..N-1]$ Требуется построить К-кучу с минимумом в корне. Наиболее очевидный способ построить такую кучу из неупорядоченного массива — сделать нулевой элемент массива корнем, а дальше по очереди добавить все его элементы в конец кучи и запускать от каждого добавленного элемента $SiftUp$. Временная оценка такого алгоритма $\mathcal{O}(n \log n)$. Однако можно построить кучу еще быстрее — за $\mathcal{O}(n)$.

Представим, что в массиве хранится дерево ($A[0]$ – корень, а потомками элемента $A[i]$ являются $A[ki+1] \dots A[ki+k]$). Сделаем $SiftDown$ для вершин, имеющих хотя бы одного потомка: от $\frac{N}{K}$ до 0,— так как поддеревья, состоящие из одной вершины без потомков, уже упорядочены.

Почему на выходе мы получим кучу:

При вызове $SiftDown$ для вершины, ее поддеревья являются кучами. После выполнения $SiftDown$ эта вершина с ее поддеревьями будут также являться кучей. Значит, после выполнения всех $SiftDown$ получится куча.

Доказательство времени работы за $\mathcal{O}(n)$:

Число вершин на высоте h в куче из N элементов не превосходит $\lceil \frac{N}{K^h} \rceil$. Высота кучи не превосходит $\log_k N$. Обозначим за H высоту дерева, тогда время построения не превосходит

$\sum_{h=1}^{H} \frac{N}{K^h} \cdot K \cdot h = N \cdot K \sum_{h=1}^{H} \frac{h}{K^h}$

Докажем вспомогательную лемму:

Лемма:

$\sum_{h=1}^{\infty} \frac{h}{K^h} = \frac{K}{(K-1)^2}$

Доказательство леммы:

Обозначим $\sum_{h=1}^{\infty} \frac{h}{K^h} = S$. Заметим, что $\frac{n}{K^n} = \frac{1}{K} \cdot \frac{n-1}{K^{n-1}} + \frac{1}{K^n}$. $\sum_{h=1}^{\infty} \frac{1}{K^h}$ – это сумма бесконечной убывающей геометрической прогрессии, и она равна $\frac{\frac{1}{K}}{1-\frac{1}{K}} = \frac{1}{K-1}$. Получаем $S = \frac{1}{K} \cdot S + \frac{1}{K-1}$, откуда $S = \frac{K}{(K-1)^2}$. Подствляя в нашу формулу результат леммы, получаем $N \cdot (\frac{K}{K-1})^2 \leq 4 \cdot N = \mathcal{O}(n)$.

3 Leftist heap

Левосторонняя, левацкая куча (Leftist heap) — двоичное левосторонее дерево (не обязательно сбалансированное), но с соблюдением порядка кучи (heap order).
Свободной позицией назовем место в дереве, куда может быть вставлена новая вершина. Само дерево будет являться свободной позицией, если оно не содержит вершин. Если же у какой-то внутренней вершины нет сына, то на его месте — свободная позиция.

Обозначим за $d(v)$ расстояние вниз от вершины v до ближайшей свободной позиции в поддереве, а за $size(v)$ число вершин в поддереве с вершиной v. Тогда для для левацкой кучи выполняется следующее:

- $d(l[v]) \geq d(r[v])$ – по определению левацкой кучи
- $\log_2 size(v) \geq d(v)$ – если бы все свободные позиции были на глубине более логарифма, то мы получили бы полное дерево с количеством вершин более n

Если для какой-то вершины это 1 условие не выполняется, то это легко устраняется: можно за $\mathcal{O}(1)$ поменять местами левого и правого ребенка, что не повлияет на порядок кучи.

Merge(x, y): // x, y — корни двух деревьев

1. if $(x = \emptyset)$: return y
2. if $(y = \emptyset)$: return x
3. if $(y.key < x.key)$: $swap(x, y)$
   // Воспользуемся тем, что куча левосторонняя. Правая ветка — самая короткая
   //и не длиннее логарифма. Пойдем направо и сольем правое поддерево с вершиной .
4. $x.right = merge(x.right, y)$
   // Могло возникнуть нарушение левостороннести кучи
5. if $(d(x.right) > d(x.left))$: $swap(x.left, x.right)$
6. $d(x) = min(d(x.left), d(x.right)) + 1$
   // пересчитаем расстояние до ближайшей свободной позиции
7. return x
   // Каждый раз идем из уже существующей вершины только
   // в правое поддерево — не более логарифма вызовов (по лемме)

$Time \leq d(Heap_2) + d(Heap_1) + \mathcal{O}(\log n)$

4 Skew heap

Вспомним левацкую кучу. Теперь давайте откажемся от требования, что $d(v.left) > d(v.right)$

Merge

1. if $(x_1 > x_2)$ $swap(x_1, x_2)$
2. $R = Merge(R, heap_2)$
3. $swap(L, R)$

Мы получили кривую кучу или Skew heap.

Утверждение: $Merge$ работает амортизированно за $\mathcal{O}(\log n)$

Доказательство:

Пусть φ – количество правых тяжёлых детей (тяжёлым назовём ребёнка, если размер его поддерева больше половины всей кучи).

Рассмотрим два случая:

1) Пусть $Merge$ спускался вправо только в лёгкого сына, тогда было совершено менее $\log n$ шагов, т.к. скаждым шагом размер поддерева уменьшался хотябы в 2 раза. За 1 вызов $Merge$ в этом случае увеличилось не более чем на $\log n$ (каждый шаг весит не более, чем 1 по результатамswap – а).

2)Мы можем увеличить количество правых тяжёлых детей.

Утверждение: Когда мы переходим в правого сына φ уменьшается на 1

Доказательство:

Так происходит из-за того, что мы делаем $swap$. Мы меняем левого и правого детей местами, тогда тяжёлым становится левый сын, а количество правых тяжёлых уменьшается на 1.

Изначально $\varphi_0 = 0$, мы предполагаем, что у нас каждая куча состоит из 1 элемента. $\varphi = \varphi^+ + \varphi^- \geq 0$, где φ^+ – суммарное количество увеличений φ , а φ^- – суммарное количество уменьшений φ. Отсюда следует, что $\varphi^+ \geq \varphi^-$, $\varphi^+ \leq M \log n$, здесь и далее M – количество вызовов функции $Merge$.

$Merge$ = Т + Л (сумма для тяжелых и легких детей) = Т + $\log n$.

$\sum Merge = \sum T + M\log n = \varphi^- + M\log n$.

Среднее арифметическое:

$\frac{\sum Merge}{M} = \frac{\sum T + M\log n}{M} = \frac{\varphi^- + M\log n}{M} \le \frac{2M\log n}{M} = 2\log n = \mathcal{O}(\log n)$, здесь мы воспользовались тем, что $\varphi^- \le \varphi^+ \le M\log n$.

5 Add and Merge in $\mathcal{O}(1)$

Теперь наша цель – получить кучу, в которой операции Add и $Merge$ выполняются за амортизированное $\mathcal{O}(1)$, а операция $DelMin$ – за $\mathcal{O}(\log n)$.
Нам интересно, сколько будут вместе, в среднем, работать эти операции.
Пусть Add выполнилась A раз, $Merge$ – M раз, $DelMin$ – D раз, тогда:

$Time \leq A + M + D(\log n)$.

1. Пусть у нас есть структура, которая может делать:

   - Add, $Merge$, $DelMin$ за $\mathcal{O}(\log n)$
   - $Build$ за $\mathcal{O}(n)$ (строит кучу от n элементов)

   Мы можем получить из этой структуры такую, что Add в ней будет происходить за $\mathcal{O}(1)$:

   Произошло k раз Add. Перед $Merge$ и $DelMin$ вызываем $Build$ ($\mathcal{O}(k)$) от добавленных элементов и $Merge$ – им получившиеся структуры (новую и старую).

   В среднем на каждый Add мы тратим единицу $Build \rightarrow \mathcal{O}(1)$.

   $Merge$ и $DelMin$ на данном этапе работают за $\mathcal{O}(\log n)$.

   $NewDelMin = Build + Merge + Del$

   $NewMerge = 2(Build + Merge) + Merge$ – приводим каждую из двух структур к новому виду

2. Add $\mathcal{O}(1)$, $Merge$ $\mathcal{O}(\log n)$, $DelMin$ $\mathcal{O}(\log n)$ – пусть структура Q удовлетворяет таким условиям.

   Новая цель: $Merge$ за $\mathcal{O}(1)$

   Строим новую структуру B, она содержит ключ и структуру Q типа B.

   $B = \langle X, Q\langle B \rangle \rangle$ либо $B = null$, X – ключ, глобальный минимум (типа int).

   Q – это очередь элементов типа B, где каждый элемент – это ключ и очередь элементов типа B. (Некоторая абстрактная структура). Q хранит пары и сравнивает их по ключу.

   Add (B, x)

   1. return $Merge(B, newB(x, null))$

   Merge(B_1, B_2)

   1. if $(B_1.x > B_2.x)$: $swap(B_1, B_2)$
   2. $B_1.Q = B_1.Q.Add(B_2)$

3. return B

DelMin (B) // $\langle x, B\rangle$, – минимум, B – все остальные

1. $\langle B_2, Q_2\rangle = B.Q.DelMin()$
2. return $\langle x, \langle B.x, B_2.Q.Merge(Q_2)\rangle\rangle$ // Присоединили Q_2

Как выглядит структура:
$\langle x, B\rangle$
$B = \langle x, Q\rangle$
$Q = \langle B2, Q2\rangle$

Что получили в итоге:

- Add $\mathcal{O}(1)$
- $Merge$ $\mathcal{O}(1)$
- $DelMin$ $\mathcal{O}(\log n)$

Замечание по поводу времени работы:
Если начальная структура работала за амортизированное время, то данная тоже работает за амортизированное $\mathcal{O}(1)$, а если просто, то за чистое $\mathcal{O}(1)$.