Экзаменационные билеты
Алгоритмы. 5 курс. Весенний семестр.

Горбунов Егор Алексеевич

13 июня 2016 г.

1 Декартово дерево

Определение 1. Декартово дерево – это бинарное дерево, в каждой вершине которого хранится пара (k, p), причём декартово дерево является деревом поиска по ключу k и кучей по приоритету p.

Лемма 1. *Пускай нам дан набор пар $(k_1, p_1), (k_2, p_2), \ldots, (k_n, p_n)$, причём все p_i различны. Тогда существует единственное декартово дерево, построенное по этому набору пар.*

Доказательство. Будем рекурсивно (по индукции) строить декартово дерево. Все приоритеты p_i различны, а значит среди пар можно выбрать *единственную*, у которой приоритет $p_i = p_{max}$ максимальный. Относительно этого приоритета все пары мы можем однозначно разбить на две группы: в «левую» группу отправим пары у которых $p_i < p_{max}$, а в правую те, у которых $p_i \geq p_{max}$. Таким образом разделили исходную задачу размера n на более мелкие две. Построим соответственно декартово дерево на «левой» и «правой» группе и присоединим к корневой паре с приоритетом p_{max}. Дерево построено, значит существует. На каждом уровне выбор корня делается единственным способом (тут используем единственность приоритетов), а значит дерево единственно. □

Замечание 1. Алгоритм, описанный в доказательстве леммы отработает за $\mathcal{O}(n^2)$.

1.1 Построение за линейное время

Пускай нам дан *уже отсортированный* по ключам в порядке возрастания набор пар $(k_1, p_1), \ldots, (k_n, p_n)$. Построим по ним декартово дерево быстро – за $\mathcal{O}(n)$. Декартово дерево – бинарное дерево поиска по ключам k_i, а это значит, что при добавлении ключей в порядке возрастания, добавляемая вершинка должна оказаться самой правой (у нас в левой веточке ключи $< k$, а в правой $\geq k$). Заметим таким образом, что мы всегда можем добавлять новую вершину x в правую ветвь декартова дерева:

1. В самой правой ветви найти такие две вершины a и b (b – непосредственный сын a), что $a.p > x.p$ и $x.p > b.p$ (всякие граничные случаи где $a = NIL$ или $b = NIL$ не берём в голову, сейчас пофиг)

2. Перекинуть ссылки на сынишек: $x.l = b,\ a.r = x$

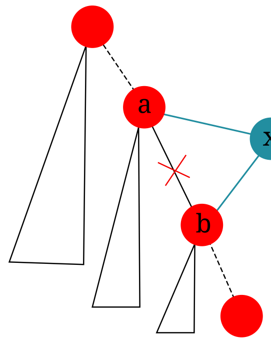

Рис. 1: Вставка вершины при построении декартова дерева

Как находить такие вершины a и b? Можно спускаться от корня вниз и вправо. Но тогда представим себе такую ситуацию: на вход подали набор пар $(1,1),(2,2),(3,3),\ldots,(n,n)$. Пары отсортированы как надо. На каждой итерации алгоритма построения дерева новая вершина будет добавляться в самый конец пути вправо, таким образом для нахождения вершины a мы каждый раз будем спускаться вниз проходя по i вершин на i-ой итерации. Это как-то квадратично, а мы хотим за $\mathcal{O}(n)$.

Будем искать место для вставки с конца правой ветки. Почему это разумно? А потому, что веточка снизу ломается влево каждый раз при добавлении новой вершины там, куда её добавили. Это наводит на разные амортизационные мысли. Имеем дерево T, которое строится.

$$\Phi_i(T) = \text{длина правой ветки дерева перед } i\text{-ой итерацией}$$

Тогда амортизационная стоимость итерации (добавления одной вершинки) равна (тут t — это то, сколько мы прошли по правой ветки снизу в поисках места вставки)

$$\tilde{c}_i = t + \Delta_i\Phi = t + ((l - t + 1) - l) = t - t + 1 = 1 = \mathcal{O}(1)$$

Таким образом амортизационная оценка на построение всего дерева: $\mathcal{O}(n)$!

1.2 Операции вставки и удаления

Для вставки и удаления введём 2 дополнительные операции: `Split` и `Merge`

- *Процедура* `Split` принимает на вход дерево T и ключик k, а возвращает 2 декартовых дерева: в первом дереве все ключики $< k$, а во втором $\geq k$.

```
def Split(T, k):
    if size(T) == 0: return (nil, nil) # nil is empty tree or something like this...
    if size(T) == 1 and T.k < k: return (T, nil)
    if size(T) == 1 and T.k >= k: return (nil, T)
    if T.k < k:
        (T.r, R) = Split(T.r, k)
        return (T, R)
    else:
        (L, T.l) = Split(T.l, k)
        return (L, T)
```

Логика проста: если ключ в корне дерева меньше k, то этот корень уже находится в искомой «левой» половине разбиения (сплита, разреза, как хотите!), а значит «правую» половину разбиения мы берём из рекурсивного разбиения правого сына, а левые остатки присоединяем к итоговой левой части.

- *Процедура* `Merge` принимает на вход два дерева L и R, причём таких, что ключи в первом дереве меньше либо равны ключам во втором и возвращает декартово дерево содержащее ключи обоих деревьев. Опять просто и рекурсивно:

```
def Merge(L, R):
        if size(L) == 0: return R
        if size(R) == 0: return L
        if L.p > R.p:
            L.r = Merge(L.r, R)
            return L
        else:
            R.l = Merge(L, R.l)
            return R
```

Тут мы просто уменьшаем задачу валидным способом...

- *Вставка в дерево*: процедура `Insert`. Процедура будет принимать дерево и новую вершину, а возвращать изменённое дерево.

```
def Insert(T, node)
        (L, R) = Split(T, node.k)
        return Merge(Merge(L, node), R)
```

- *Удаление из дерева*: процедура `Remove`. Принимаем дерево и ключ, а возвращаем дерево, их которого вершинка с ключиком была удалена.

```
def Remove(T, k)
        (L, R) = Split(T, k)
        (D, R) = Split(R, k)
        return Merge(L, R)
```

Замечание 2. Легко видеть, что время работы операций `Split` и `Merge` — $\mathcal{O}(h)$, где h — высота дерева. А значит аналогично и время работы вставки и удаления.

1.3 Дуча (Treap, Дерамида)

Определение 2. Дуча — это декартово дерево, приоритеты в котором случайные и равномерно распределённые.

Лемма 2. *Математическое ожидание высоты дучи равно $\mathcal{O}(\log n)$, где n — число вершин в ней.*

Доказательство. Обратимся к рекурсивному (тот, что за $\mathcal{O}(n^2)$) алгоритму построения декартова дерева. Заметим, что в силу случайности приоритетов мы равновероятно выбираем любой ключ при построении дерева, относительно которого все остальные ключи будут разбиты по группам $<$ или $\geq$. Это в точности та же процедура, которая происходит при применении быстрой сортировки. И дерево рекурсии для этих задач идентичны. Умеем доказывать, что дерево рекурсии `QuickSort` будет иметь высоту, в среднем (из-за случайности процесса, а не по входам), логарифмическую! □

2 Задачи `RMQ` и `LCA`