Конспект по алгоритмам, четвёртые полсеместра

Василий Алфёров. А лектор – Сергей Копелиович

Оглавление

0 Bugzilla **5**

1 Жадность и АВЛ-деревья **6**
1.1 $\ln N$-OPT приближение Set Cover . . . 6
1.2 Дедлайны. Hard . . . 7
1.3 Деревья поиска . . . 9
1.3.1 Обход бинарного дерева поиска . . . 10
1.3.2 NextRight, NextLeft за $\mathcal{O}(1)$ . . . 10
1.3.3 Find за $\mathcal{O}(h)$ и $\mathcal{O}(1)$ . . . 10
1.3.4 Del за $\mathcal{O}(1)$ . . . 11
1.3.5 Add $\leftarrow$ LowerBound . . . 11
1.3.6 Add за $\mathcal{O}(h)$ . . . 11
1.3.7 Персистентные деревья поиска . . . 13
1.3.8 Сбаллансированные деревья поиска . . . 13
1.4 АВЛ-деревья . . . 13
1.4.1 Add в АВЛ-дереве . . . 15

2 Центроиды и Centroid Decomposition **17**

3 Деревья поиска **20**
3.1 B-деревья . . . 20
3.1.1 Add в B-дереве . . . 21
3.1.2 Del в B-дереве . . . 21
3.1.3 Нафига козе баян? . . . 21
3.2 Красно-чёрные деревья . . . 22
3.2.1 AA-деревья . . . 23
3.2.2 Нафига козе баян? . . . 23
3.3 Декартово дерево . . . 23
3.3.1 Split в декартовом дереве . . . 24
3.3.2 Merge в декартовом дереве . . . 25
3.3.3 Нафига козе баян? . . . 25
3.4 Randomized Binary Search Tree . . . 25
3.4.1 Нафига козе баян? . . . 25
3.5 Деревья по неявному ключу . . . 25
3.5.1 Персистентные деревья по неявному ключу . . . 26
3.6 Абстрактные факты про персистентность . . . 27
3.6.1 Декартово дерево не дружит с персистентностью . . . 27

3.6.2 Garbage collection . . . 27

4 Splay-деревья и Skip-List 28

4.1 Rope . . . 28
4.2 `SkipList` . . . 28
4.3 `Splay`-деревья . . . 29
4.4 SQRT-декомпозиция . . . 31
4.4.1 Структура данных на массиве . . . 31
4.4.2 SQRT `Split`/`Rebuild` . . . 31
4.5 Notes on persistence . . . 31
4.5.1 Этот персистентный мир . . . 32
4.5.2 ~~Распилы и~~ откаты . . . 32
4.6 Персистентная очередь . . . 32

5 Деревья отрезков и персистентность 33

5.1 Дерево отрезков . . . 33
5.1.1 Дерево отрезков снизу . . . 34
5.1.2 Дерево отрезков сверху . . . 34
5.1.3 Сравнение последних двух . . . 34
5.2 Двумерные деревья . . . 35
5.2.1 ДО сортированных массивов . . . 36
5.2.2 ДО of ДД . . . 36
5.2.3 ДО of (ДО of Sorted Array) . . . 36
5.3 Scanline . . . 36
5.3.1 Прямоугольники и точки на плоскости . . . 36
5.3.2 Переход к online . . . 37
5.3.3 k-тая порядковая статистика на отрезке . . . 37

6 RMQ и LCA 38

6.1 Range Minimum Query . . . 38
6.1.1 Sparse table . . . 38
6.1.2 SparseTable++ . . . 38
6.2 LCA . . . 39
6.2.1 Двоичные подъё~~мы~~ . . . 39
6.2.2 LCA $\to$ RMQ ± 1 . . . 39
6.2.3 LCA-offline $\mathcal{O}(\alpha(n))$ (Тарьян) . . . 40
6.3 Снова RMQ . . . 40
6.3.1 RMQ$\pm$1 за $\langle n, 1\rangle$ . . . 40
6.3.2 RMQ $\to$LCA . . . 41
6.4 Level Ancestor . . . 41
6.5 Euler Tour Trees . . . 41

7 HLD и Link-Cut Trees 42

7.1 Heavy-Light Decomposition . . . 42
7.2 Link-Cut trees . . . 42
7.3 MST за $\mathcal{O}(n)$ . . . 43
7.3.1 Проверка остовного дерева на минимальность . . . 43

7.3.2 Собственно Randomized BST 44

Bugzilla

Что надо сделать и/или исправить

1. В теореме 1.1.1 понять, что оценка работает в нужную сторону.
2. Добавить центроидам строгости.
3. Поправить центроидам багу.
4. Написать `Del` в B-дереве.
5. Нарисовать B- и RB-деревьям картинок.
6. Нарисовать картинку в доказательство Splay.
7. Картинка: ДО – запрос сверху – не дерево.
8. Достижение оценки про количество вершин в ДО снизу.
9. Написать про алгоритм Вишкина.
10. Понять что-то про минимум на пути в дереве за $\mathcal{O}(V \log V)$ (надо для MST за линию).

Больше ничего сделать не надо, поэтому тут будет немного Lorem ipsum.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Жадность и АВЛ-деревья

Лекция 19 апреля 2017 года

1.1. $\ln N$-`OPT` приближение Set Cover

Напомним задачу: дано N-элементное множество и M его подмножеств. Нужно выбрать среди них минимальный набор, покрывающий всё множество.

Задача NP-полна. Но мы и не собираемся находить именно минимум. Если k – размер оптимального ответа, то мы найдём покрытие размером $\leq k \ln N$. В худшем случае. А часто – даже лучше.

Жадность простая – будем на каждом шаге выбирать множество, содержащее максимальное количество ещё не покрытых элементов. А теперь будем доказывать.

Th 1.1.1. Получим ответ размером не больше $k \ln N$.

Доказательство. Если сделаем операцию один раз над данными, в которых оптимальный ответ k, то выберем множество, в котором хотя бы $\lceil \frac{N}{k} \rceil$ невыбранных элементов. После этого оптимальный ответ на оставшихся данных мог либо уменьшится на 1, либо остаться таким же.

Если сделаем операцию один раз, получим, что невыбранных элементов осталось не больше чем

$$N\left(1 - \frac{1}{k}\right)$$

(мы убрали округление)

Если дважды, то не больше чем

$$\max\left\{N\left(1-\frac{1}{k}\right)^2, N\left(1-\frac{1}{k}\right)\left(1-\frac{1}{k-1}\right)\right\} = N\left(1-\frac{1}{k}\right)^2$$

Соответственно, если t раз, то

$$N\left(1-\frac{1}{k}\right)^t$$

При $t = k$ это равно

$$N\left(1-\frac{1}{k}\right)^k \le \frac{N}{e}$$

(тут стоит понять, что оценка именно с этой стороны (мне лень))

А при $t = k\ln n$

$$N\left(1-\frac{1}{k}\right)^{k\ln N} \le \frac{N}{e^{\ln N}} \le 1$$

Ну значит, что после $k\ln N$ операций у нас действительно не окажется непокрытых элементов, что и требовалось. □

1.2. Дедлайны. Hard

> Чтобы было удобнее, докажем гораздо более сильный факт, чем собирались
>
> С.К.

Напомним задачу. Даны n заданий, каждое задано парой $\langle t_i, d_i\rangle$ – время, которое уходит на выполнение задания и дедлайн, к которому его надо выполнить. Сейчас момент времени 0. В разных вариациях нужно понять, можно ли успеть выполнить все задания, или выполнить наибольшее возможное количество из них.

Сейчас мы знаем два решения на две версии этой задачи:

1. Жадность: выполнить всё за $\mathcal{O}(n\log n)$.

   Отсортировать задания по дедлайну и пытаться выполнить их в таком порядке.

2. Динамика: выполнить max за $\mathcal{O}(n^2)$.

   Динамика `sum[i][k]` – минимальное время, за которое можно выполнить k заданий из первых i (в порядке сортировки по дедлайну). Переходы `sum[i][k]` $\leftarrow$ `sum[i - 1][k]` и `sum[i][k]` $\leftarrow$ `sum[i - 1][k - 1]` $+ t_i$, последний только если `sum[i - 1][k - 1]` $+ t_i \le d_i$.

Сейчас будем учиться выполнять max за $\mathcal{O}(n\log n)$. Это в некотором роде комбинация двух предыдущих решений.

Решение будет жадным. Сначала отсортируем все задания по возрастанию дедлайна. Теперь идём по заданиям и пытаемся выполнить как можно больше. Пусть на текущий момент мы выполнили задания $\langle t_1, d_1\rangle, \dots, \langle t_m, d_m\rangle$, и теперь пытаемся добавить к ним задание $\langle T, D\rangle$. Будет три случая.

1. $\sum_i t_i + T \le D$.

   Тогда мы можем выполнить все предыдущие задания и наше. Ну и выполним.

2. $t_{\max} \ge T$, где $t_{\max} = \max_i t_i$.

   Тогда заменим задание с максимальным временем на $\langle T, D\rangle$.

3. Ни то, ни другое.

   Тогда не будем делать ничего.

Почему $\mathcal{O}(n\log n)$? Сортировка $\mathcal{O}(n\log n)$, потом нам надо уметь поддерживать какое-то множество элементов, пихать туда числа и доставать оттуда максимум, и всё за $\mathcal{O}(\log n)$. Это куча, это мы умеем.

Всё хорошо, осталость доказать, что работает. У нас есть динамика, которая точно работает. Будем доказывать как раз через неё.

Th 1.2.1. Рассмотрим любой момент времени i. Пусть наша жадность смогла выполнить m заданий. Упорядочим их по возрастанию времени ($t_1 \le t_2 \le ... \le t_m$). Тогда m – максимальное число заданий, которое мы можем выполнить из первых i и $\forall j$ `sum[i][j]` $= \sum_{k=1}^{j} t_k$.

Формулировка не самая очевидная: мы же вроде бы выполняем задания не по возрастанию времени выполнения, а по возрастанию дедлайна. Комментарий к этому такой: сейчас мы рассматриваем задания, которые можем выполнить к моменту времени i. Пусть мы могли из них выполнить максимум m. Тогда значение динамики `sum[i][k]` равно минимальному времени, за которое мы могли бы выполнить k из них. То есть суммарному времени k минимальных из них (помним, что задания мы выполняем без пауз).

Итак,

Доказательство. Индукция по i. База понятна – из нуля предметов больше нуля не возьмёшь.

Переход $i-1 \to i$. Пусть на шаге $i-1$ мы могли выполнить не более m задание с оптимальным ответом $\langle t_1, d_1\rangle, ..., \langle t_m, d_m\rangle$ ($t_1 \le t_2 \le ... \le t_m$). Рассмотрим любое $j \le m$. Смотрим на $t_1, ..., t_j$. Пытаемся улучшить ответ значением $\langle T, D\rangle$. Как помним из динамики, `sum[i][j] = sum[i - 1][j]`, если `sum[i - 1][j - 1]` $+ T > D$, а иначе же `sum[i][j] = min(sum[i - 1][j], sum[i - 1][j - 1]` $+T$). Исходя из предположения индукции, на шаге $i-1$ первые j в порядке сортировки по времени заданий действительно оптимальны, также как и первые $j-1$. Посмотрим, что же случится на шаге i.

1. $\sum_{k=1}^{m} t_k + T < D$.

   Тогда в массив t_k куда-то в середину (сортировка по времени выполнения же) добавится T. Это как минимум означает, что при любом требуемом j задание мы выполнить сможем, а значит смотреть надо на минимум. Когда `sum[i - 1][j]` больше, чем `sum[i - 1][j - 1]` $+ T$? Когда мы можем вставить T вместо какого-то из $t_1 ... t_j$. Ну вроде бы это как раз именно то, что случилось.

2. $T < t_{\max}$.

   Тогда t_m уберётся, а T вставится куда-то в середину. Тут нужно понять, что это абсолютно тот же самый случай.

3. Ни то, ни то.

   Тогда, очевидно, выполнить все задания вместе с новым мы не можем и на него уйдёт больше времени, чем на любое другое. Тогда непонятно, с чего бы нам вообще его делать (более строго – всегда можно сделать выбор заданий с меньшей суммой t).

Осталось лишь понять, почему оптимальное число выполненных заданий будет действительно посчитано правильно. В динамике у нас единственный переход, увеличивающий это количество – это `sum[i - 1][j - 1]` $\leftarrow$ `sum[i][j]`. Раз на предыдущем шаге мы получили именно оптимальное относительно `sum` множество заданий по предположению индукции, и при пересчёте динамики мы могли бы выполнить все те задания и новое, то и сейчас сможем. Этим и для $j = m + 1$ при увеличении m доказано утверждение теоремы. $\square$

1.3. Деревья поиска

Сейчас мы будем выращивать
много разных деревьев.

С.К.

Здесь мы вводим новую структуру данных.

Def 1.3.1. Бинарное дерево поиска (BST, **Binary** Search Tree) – бинарное дерево, в каждой вершине которого написано некоторое значение, при этом если в вершине написано значение x, то в левом поддереве все значения меньше, чем x, а в правом – больше, чем x.

Договоримся, что за n будет обозначаться количество элементов в дереве, а за h – высота дерева.

Данное определение требует уточнения – непонятно, что делать с равными элементами. Предлагается два способа.

1. Как в алгоритме Борувки – хранить пару $\langle$значение, время добавления$\rangle$, чтобы все значения были гарантированно различными.

2. Хранить слева меньшие либо равные элементы, а справа – большие либо равные. Можно ещё равные только справа, но баллансировать дерево поиска (см. далее) тогда не выйдет – если все элементы равны, дерево вытянется в бамбук направо. *Этот вопрос был задан после лекции и на него был дан именно такой ответ.*

Дерево будем хранить как структуру `Node`. В англоязычной литературе по отношению к вершинам графов применимы слова vertex и node, но когда речь идёт о графах, чаще выбирают слово vertex, а когда о деревьях – почти однозначно node. Тогда в простом варианте на текущем этапе структура выглядит так:

```
struct Node {
    int x;
    Node *L, *R;
};
```

Здесь `*L, *R` – указатели на левого и правого ребёнка, соответственно, если эти дети существуют, и `NULL` иначе.

Операции с деревом поиска нам захочется делать такие:

1. `Add`, `Del`

2. `Find(value)`, `NextRight(vertex)`, `NextLeft(vertex)`

3. `LowerBound(value) -> vertex`

4. `Min`, `Max` (за $\mathcal{O}(1)$)

Многие из этих операций хочется делать за $\mathcal{O}(1)$.
Сначала небольшое визуальное упражнение.

1.3.1. Обход бинарного дерева поиска

> Ребёнка у нас пока два: левый и правый.
>
> С.К.

Вроде бы его называют симметрическим (но это требует уточнения).

Суть: дано дерево поиска, обойти его в порядке увеличения ключей. Решение оформим как рекурсивную функцию. Нормальная мнемоника для запоминания (даже на лекции было): LxR. Более подробно так: надо обойти поддерево текущей вершины. Слева всё меньше, справа всё больше. Тогда сначало надо пойти налево, потом посетить (вывести, например) текущую вершину, а потом пойти направо. Выглядит оно так:

```
void print_tree(Node *v) {
    if (v == NULL) // условие выхода из рекурсии
        return;
    print_tree(v->L);
    cout << v->x << endl;
    print_tree(v->R);
}
```

Теперь поехали по порядку:

1.3.2. `NextRight`, `NextLeft` за $\mathcal{O}(1)$

Будем хранить в каждой вершине указатели на вершины `NextRight`, `NextLeft` и пересчитывать их. Заметим, что это простой (сортированный) двусвязный список. Мы справимся.

1.3.3. `Find` за $\mathcal{O}(h)$ и $\mathcal{O}(1)$

За $\mathcal{O}(h)$ – "спуск по дереву". Стартуем в корне, смотрим на текущее значение. Если оно совпадает с искомым – мы нашли нужную вершину. Если оно меньше искомого – идём направо, иначе – налево. В худшем случае спустимся на глубину дерева.

За $\mathcal{O}(1)$ – просто будем для каждого x хранить хеш-таблицу, которая будет нам по значению возвращать указатель на нужную вершину. Ну или говорить, что её нет. А мораль этого – ~~если вы так делаете, вам нужен доктор~~ не надо бояться добавлять нужную структуру к дереву, если надо.

1.3.4. `Del` за $\mathcal{O}(1)$

Утверждается, что `Del` = `Find` + `NextRight` + хранить отца. Сейчас поймём, почему.

Сначала сделаем `Find` и поймём, какую вершину нам нужно удалить. Обзовём её v, её родителя p, а детей – L и R. Посмотрим на неё. Будет два случая.

1. У вершины v нет правого ребёнка (см. рис. 1.1).

   Тогда нам нужно всего лишь привесить ребёнка L (даже, если он нулевой) к вершине p. Поддерево врешины L при этом никуда не денется и всё будет хорошо.

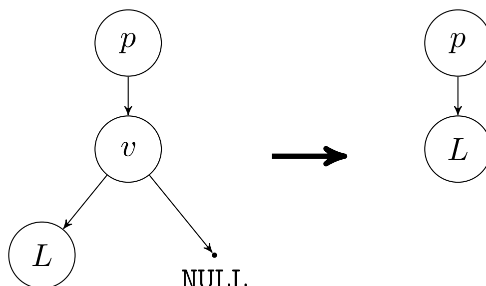


Рис. 1.1: Удаление вершины из BST: у вершины v нет правого ребёнка.

2. У вершины v есть правый ребёнок (см. рис. 1.2).

   Тогда у вершины v есть какой-то `NextRight` – вершина x, и она – минимум в поддереве вершины R (то есть будем идти налево из неё, пока не упрёмся в `NULL`, и попадём как раз в x). У неё-то при этом точно нет левого ребёнка, значит, мы её можем удалить ровно так же, как в первом случае.

   Заметим, что если свопнуть значение в ней со значением в вершине v, а потом удалим её, ничего не сломается. Ну действительно, значение, которое было в вершине x, может оказаться в вершине v (оно меньше всех значений в поддереве вершины R и больше всех значений в поддереве вершины L). Кроме того, после свопа всё поддерево вершины R будет корректно, а значит, и удаление хотя бы в поддереве вершины R оставит всё корректно. Сверху же от вершины R тем более всё будет корректно, так как новых значений туда не добавлялось.

   Ну так свопнем и удалим.

1.3.5. `Add` $\leftarrow$ `LowerBound`

То есть как делать `Add` за $Time($`LowerBound`$) + \mathcal{O}(1)$. Было оставлено как упражнение на практику, тут нужно понять, что это несложно и достаточно похоже на `Del`. Ещё один небольшой хинт: здесь подразумевается, что мы добавляем обязательно новый лист.

1.3.6. `Add` за $\mathcal{O}(h)$

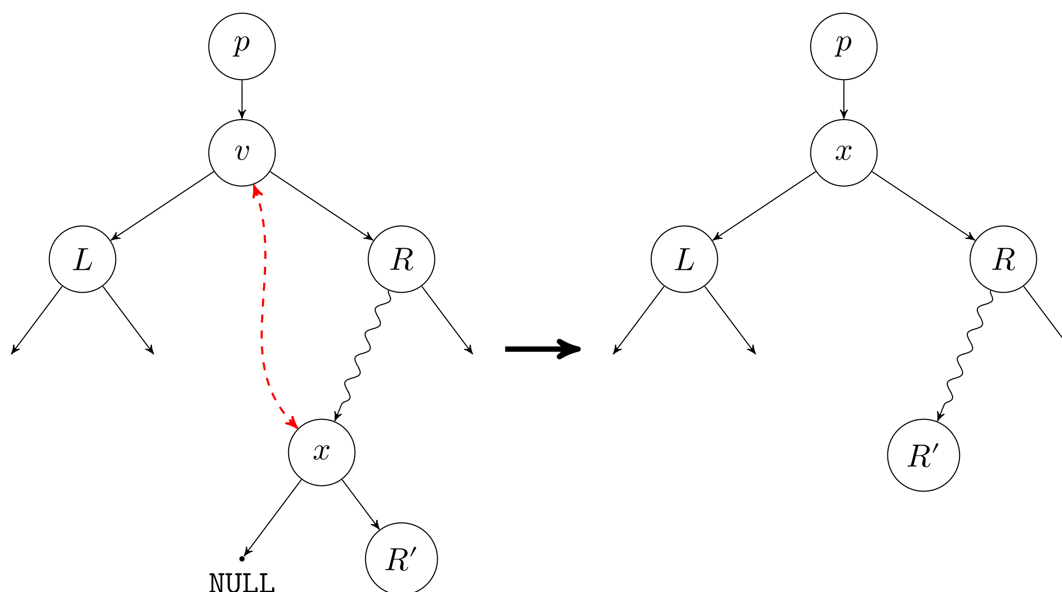


Рис. 1.2: Удаление вершины из BST: у вершины v есть правый ребёнок. Волнистой линией показано произвольное (возможно, нулевое) количество рёбер.

Тут тоже добавляем новый лист. Снова рекурсивная функция. Будем принимать какое-то дерево (ну корень) и возвращать корень нового дерева. Потом будет параграф про персистентность, там так делать будет совсем удобно. Ну и сейчас нормально.

Ну хорошо, пусть мы стоим в вершине v и хотим добавить значение x. Если v `== NULL`, мы знаем, что делать. Иначе надо добавить вершину либо налево, либо направо – зависит от значения в вершине x. Выглядит это так:

```
Node *Add(Node *root, int x) {
    if (root == NULL)
        return new Node(x); // Конструктор занулит L и R.
    if (root->x > x)
        root->L = Add(root->L, x);
    else
        root->R = Add(root->R, x);
    return root;
}
```

Из-за этих операций дерево может вытянуться в бамбук, например, если ключи будут добавляться в порядке возрастания. Ну мы скоро научимся с этим бороться.

Вообще сейчас структура здорово раздулась и выглядит как-то так:

```
struct Node {
    int x;
    Node *L, *R;
    Node *par; // предок текущей вершины
    Node *NextLeft, *NextRight; // двусвязный список
};
```

Кроме того, где-то валяется хеш-таблица. Тут, помимо прочего, надо понимать, что ни хеш-таблицу, ни список, ни родителей мы нигде не поддерживали в описании предыдущих алгоритмов и надо их поддерживать отдельно. ~~Ну, если вы это вообще пишете.~~

1.3.7. Персистентные деревья поиска

Персистентность – свойство структуры данных, при котором при любой операции на структуре сама структура не портится. То есть можно выполнять запросы в новой (изменённой) структуре, а можно в старой (то есть к структуре в том её состоянии, в котором она была до изменения). Деревья поиска прекрасны тем, что они так умеют (~~хоть и непонятно, что это даёт, они ведь пока за линию работают~~ потом мы научимся делать их работающими за $\mathcal{O}(\log n)$ и это будет совсем круто).

А идея простая – каждый раз вместо модификации вершины создавать её копию. И когда модифицируем детей вершины, создать копию этой вершины и записать туда новых детей, а старую – не трогать. Возможно, тут нужна картинка, если да, пишите, нарисую.

Полученная структура получается недеревянной, а именно ациклический граф. Но мы работаем (одно изменение) за $\mathcal{O}(h)$, что явно в свете следующих определений весьма неплохо.

1.3.8. Сбаллансированные деревья поиска

<u>Def</u> 1.3.2 (Balanced Search Tree)**.** Дерево поиска (структуру данных, а не само дерево) будем называть сбаллансированным, если $h = \mathcal{O}(\log n)$ (то есть высота дерева небольшая). Или, по другому, $\exists C > 0$, что в любой момент времени $h < C \log n$.

Пусть у нас есть сбаллансированное дерево поиска. Тогда мы умеем поддерживать динамический отсортированный массив (Tree $\to$ Sorted Array), работающий за логарифм. А ещё мы умеем делать его персистентным (Persistent Dynamic Sorted Array).

Дальше на лекции был какой-то трёп про то, что мир состоит из массивов, которые могут быть персистентными. Не уверен, что оно здесь надо, поэтому сразу к сути.

А ещё персистентным можно сделать самый обычный массив (фиксированного размера). Для этого в корень положим элемент с индексом $\frac{n}{2}$, слева построим дерево для левой половины (до $\frac{n}{2}$), а справа – после $\frac{n}{2}$. Возможно, здесь тоже нужна картинка.

После этого мы получаем сбалансированное дерево, которое содержит массив, к любой копии которого мы умеем обращаться в любой момент времени. Классно же.

1.4. АВЛ-деревья

Ну давайте собственно изобретём то самое сбаллансированное дерево поиска.

Баллансировать можно разными способами. Сегодня мы будем делать это вращениями.

Малое вращение (single, по ребру) показано на рис. 1.3. Там именно поворот направо, но есть зеркально симметричная операция – поворот налево. Большое вращение (double) – на рисунке 1.4. Там также есть симметричный вариант в другую сторону.

При вращении меняется глубина некоторых поддеревьев. Можно их обсудить отдельно.

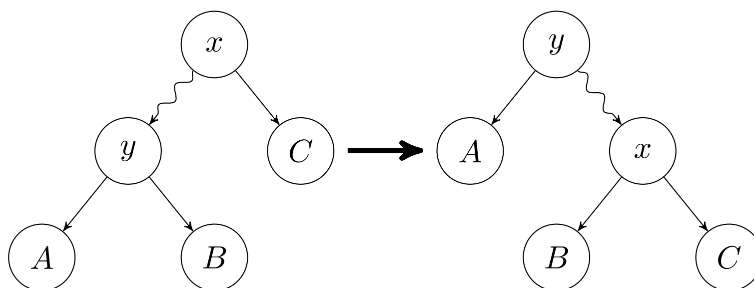


Рис. 1.3: Малое правое вращение (single rotate) ребра $x \to y$.

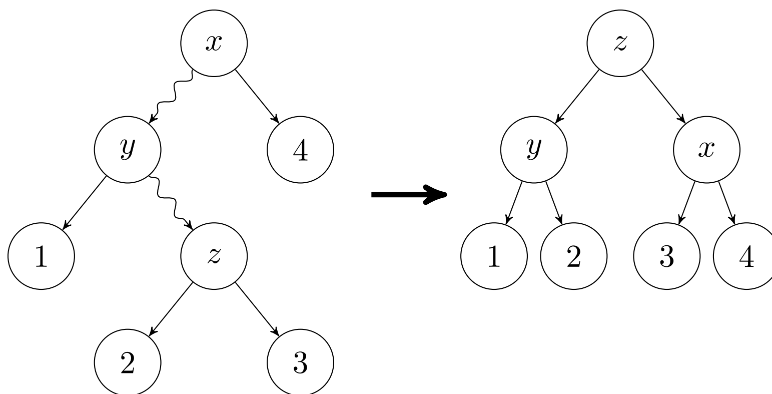


Рис. 1.4: Большое правое вращение (double rotate) пути $x \to y \to z$.

Малое вращение

Здесь очень полезно смотреть на рис. 1.3.

Заметьте, у вершин x и C длина пути от корня увеличилась, а у вершин y и A – уменьшилась. Это и логично – поворачивать направо имеет смысл, если справа у поддерева высота меньше. Но однако эта операция никак не меняет длину пути от корня до вершины B, соотвтественно, поможет не всегда.

Большое вращение

Здесь министерство душевного равновесия рекомендует к просмотру рис. 1.4.

Тут уже у вершины 1 ничего не поменялось, вершины 2 и 3 стали выше, а вершина 4 – ниже.

Теперь про то, зачем это всё надо.

Def 1.4.1 (АВЛ-инвариант)**.** Пусть для любой вершины v с детьми L и R выполняется $|h(L) - h(R)| \leq 1$, где $h(x)$ – высота поддерева вершины x. Тогда будем говорить, что дерево удовлетворяет АВЛ-инварианту.

Lm 1.4.1. Пусть дерево размера n и высоты h удовлетворяет АВЛ-инварианту. Тогда $1.6^h \leq n \leq 2^h$.

Disclaimer. Я не уверен, что это именно то доказательство, что было на лекции, ибо успешно прохлопал этот кусок ушами (хлоп-хлоп). Но факт вроде простой.

Доказательство. Правая оценка очевидна – в бинарном дереве высоты h вообще как бы не больше 2^h вершин, иначе кто-нибудь просто не поместится. Будем по этому поводу ~~пытаться их туда впихнуть~~ доказывать левую.

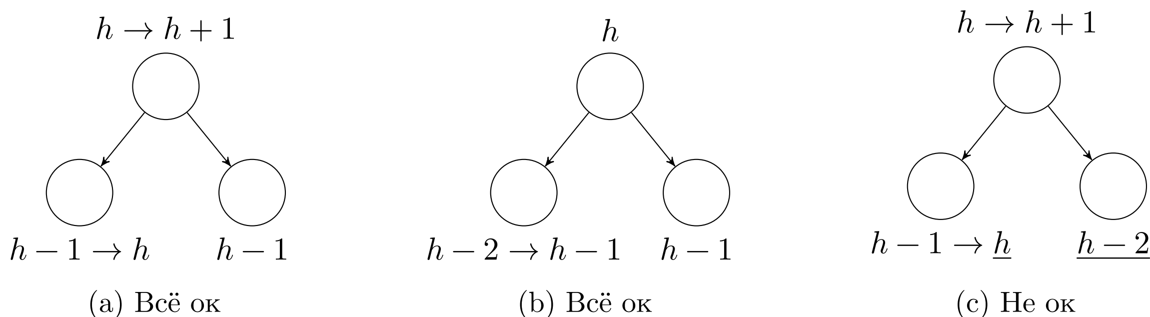


Рис. 1.5: Случаи на добавление в АВЛ-дерево: часть первая: ~~скрытая угроза~~

Пусть $T(h)$ – минимально возможное количество вершин в дереве высоты h. Понятно, что $T(0) = 1, T(1) = 2$. Кроме того, выполняется $T(h) \geq T(h-1) + T(h-2) + 1 > T(h-1) + T(h-2)$. Действительно, если у вершины высота поддерева h, то высота хотя бы одного из детей должна быть равна $h-1$. А высота второго тогда не меньше $h-2$ в силу инварианта.

Ну тут написано, что $T(h) > F_h$ (F_n – n-тое число Фибоначчи). Для доказательства леммы этого вполне достаточно. □

Corollary 1.4.1. В дереве с АВЛ-инвариантом $h = \mathcal{O}(\log n)$.

Теперь осталось научиться поддерживать АВЛ-инвариант при пересчёте вершин. Напомню, все операции мы уж точно умеем делать за высоту дерева, что нас устроит. Значит, надо научиться поддерживать инвариант при операциях, изменяющих дерево. Таких у нас две – `Add` и `Del`, но на лекции был рассказан только `Add`, а `Del` остался на практику.

1.4.1. `Add` в АВЛ-дереве

Для тупящих вроде меня (реально не очень очевидно было): мы произвольным образом добавили в дерево лист (как описано в параграфе 1.3.6 или как могло бы быть описано в параграфе 1.3.5, если бы я не был таким ленивым; кстати, в последнем случае можно не хранить список и искать `NextRight` за высоту, потому что всё равно не жалко), а теперь пытаемся сбаллансировать дерево, чтобы АВЛ-инвариант выполнялся.

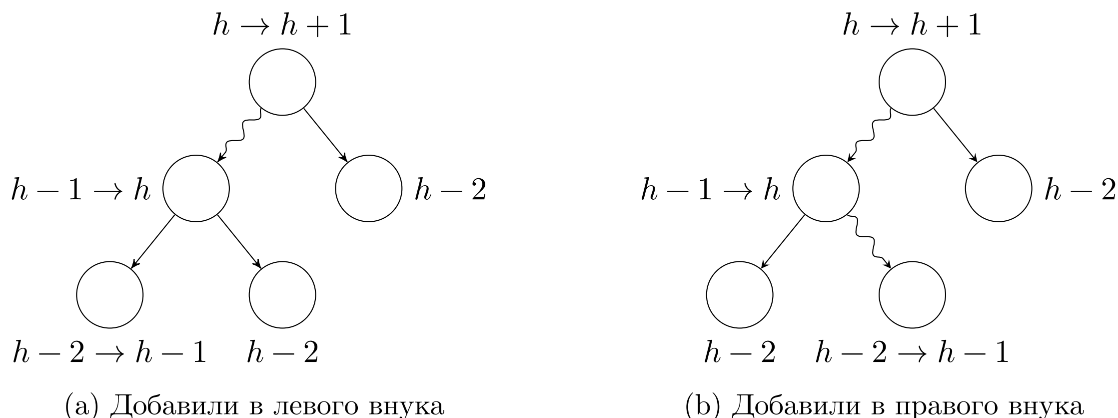
Делать будем так: прогуляемся от добавленной вершины наверх, пока не дойдём до того момента, когда станет ясно, что можно дальше не идти, и будем исправлять дисбалланс по ходу дела. Ну действительно, кроме как на этом пути дисбалланс больше нигде не мог возникнуть.

Итак, сейчас будет разбор случаев. Но их не очень много.

Случаи будут выглядеть так: мы смотрим на вершину, поддерево которой имело высоту h, и в поддерево которой добавили вершину. Смотрим на высоты детей этой вершины и что-то решаем. Они изображены на рис. 1.5.

Итак, у нас есть дерево высоты h. До добавления оно выполняло АВЛ-инвариант, значит, высоты детей были либо $h-1$ и $h-1$, либо $h-1$ и $h-2$. В первом случае всё хорошо – высота дерева просто увеличилась при добавлении на 1 (вершину добавили направо или налево, в обоих случаях теперь дети имеют высоты h и $h-1$), поэтому для этой вершины всё ок, но нужно посмотреть наверх. Второй случай имеет два подслучая.

В первом вершину добавили в дерево высотой $h-2$. Тогда для нашей вершины всё ок, а так как её высота не изменилась, то и для всех верхних всё ок и можно просто завершаться.



(a) Добавили в левого внука (b) Добавили в правого внука

Рис. 1.6: Случаи на добавление в АВЛ-дерево: часть вторая: ~~атака клонов~~

Во втором вершину добавили в дерево высотой $h-1$. Тогда возникает дисбалланс (на рисунке подчёркнут). Будем исправлять. К сожалению, снова случаи.

В новом разборе случаев рекомендуется разглядывать картинку 1.6. Стоит пояснить: АВЛ-инвариант для изменённого ребёнка нашей вершины уже выполнен, так как мы считаем, что снизу всё хорошо. Поэтому до добавления размеры внуков были равны.

Посмотрим на картинки, тем более там уже выделено то, что надо сделать – в первом случае (левый внук) надо сделать single rotate, а во втором случае – double rotate. Упражнением остаётся (по крайней мере, пока) порисовать картинки и понять, что тогда дисбалланса не будет, более того, высота дерева станет равной h, а значит, на этом снова можно заканчивать с перебаллансировкой.

Заметьте, что несмотря на то, что АВЛ-дерево – одно из самых первых сбаллансированных бинарных деревьев поиска, оно при этом одно из самых быстрых: на каждую операцию у нас всего один rotate, а переходы по ссылкам, а тем более их переприсваивание – тяжёлая операция.

Центроиды и Centroid Decomposition

Утконосьи заметки

Disclaimer. Я лично писал центроидную декомпозицию примерно два с половиной раза и достаточно давно, так что слабо отвечаю за достоверность данной главы. Но теория тут вроде ок.

Итак, вы решились научиться считать какую-нибудь стрёмную функцию на пути в дереве, но не знаете что делать. Лезете в разбор практики, а разбора-то и нет. Что делать? Грустить? Нет, давайте разбираться.

Итак, напомню, что такое центроид дерева.

Def. Центроидом дерева размера n называется такая вершина v, что при её удалении дерево распадётся на компоненты связности размером меньше $\frac{n}{2}$.

Тут надо бы доказать, что центроид всегда существует, что он почти единственный, и всё такое. Но я этого (надеюсь, пока) делать не буду. Почти единственный, если что, значит, что либо он единственный, либо их два и между ними есть ребро. Лучше я расскажу, что с этой берёзой делать.

Во-первых, её надо найти. Как? Ну подвесить дерево за какую-то вершину, посчитать `dfs`'ом размеры всех поддеревьев. Потом смотрим на вершину. Если её удалить, останутся компоненты связности – дети и то, что сверху. Размеры детей мы знаем, а размер того, что сверху – это n минус сумма размеров детей минус 1. Мы справимся понять, центроид ли вершина. Значит, справимся и найти центроид.

Тут копипаст из практики

А теперь действовать будем так. Пусть мы нашли центроид дерева. Без центроида дерево распадается на компоненты связности. Для них рекурсивно строим такую же структуру и запоминаем ссылку на структуру-отца. Центроид, найденный в компоненте связности C, будем обозначать $v(C)$. Также будем использовать обратное обозначение $C(v)$. Каждая вершина ровно один раз будет центроидом.

Заметьте, глубина рекурсии $\mathcal{O}(\log n)$. Ну каждый раз размер рассматриваемого дерева уменьшается больше, чем вдвое. Ширина рекурсии при этом $\mathcal{O}(n)$, поэтому всё вместе работает за $\mathcal{O}(n \log n)$.

Далее нам предлагается это добро хранить. Снова копипаст из практики.

Хранить будем уровень рекурсии $d(v)$, на котором вершина стала центроидом, и $p(v)$ – ссыдку на отца в дереве декомпозиции. Памяти тогда $\mathcal{O}(n)$. Заметьте, мы тогда можем

для любой вершины u найти все $d(u) = \mathcal{O}(\log n)$ вершин v, для которых $u \in C(v)$. Действительно: это просто все предки в дереве центроидной декомпозиции. Это нам поможет в следующем факте.

Итак, теперь будем считать берёзу на пути. Оказывается, для каждой пары вершин x, y на пути между ними лежит ровно одна вершина v такая, что $x \in C(v)$ и $y \in C(v)$. Это и логично: сначала деревья были в одной компоненте связности, а потом на каком-то уровне рекурсии разделились по разным. Ну тогда центроид на этом этапе обязательно лежал на пути $x \leftrightarrow y$. Ну и обратно, если бы на каком-то предыдущем уровне центроид лежал на этом пути, то они бы разделились раньше. Более того, вершину v мы можем найти за $\mathcal{O}(\log n)$ перебором уровня, на котором вершины разделились. На нём последнем у них будет совпадать предок в дереве центроидной декомпозиции. Бывает, правда, так, что $v = x$ или $v = y$. Но это абсолютно неважно.

Минимум на пути дерева

Факт, на самом деле, очень мощный. Например, мы сразу умеем искать минимум на пути в дереве (неважно, взвешены ли вершины или рёбра) за предподсчёт $\mathcal{O}(n \log n)$ и ответ $\mathcal{O}(\log n)$. Действительно, для каждого центроида v посчитаем для каждой вершины $u \in C(v)$ минимум на пути $v \rightsquigarrow u$, например, `dfs`'ом. Суммарный размер всех центроидов $\mathcal{O}(n \log n)$, например, потому что мы их за такое время построили. Теперь нам дают вершины x и y и просят посчитать минимум на пути между ними. Находим такую v на этом пути, что $x \in C(v)$ и $y \in C(v)$. Мы запомнили минимумы на пути $v \rightsquigarrow x$ и $v \rightsquigarrow y$. И вершина v лежит на пути $x \leftrightarrow y$. Ну значит, ответ – минимум этих двух величин.

На практике также надо было научиться отвечать на запрос минимума на пути в дереве за $\mathcal{O}(\log \log n)$. За $\mathcal{O}(\log n)$ у нас сейчас работает поиск центроида на пути, остальное – за $\mathcal{O}(1)$. Надо просто заметить, что центроид на пути – последний общий центроид и искать его можно бинпоиском. Линейным поиском мы перебирали $\mathcal{O}(\log n)$ вершин, значит, бинпоиском будет $\mathcal{O}(\log \log n)$. Это примечательно тем, что из известных мне способов решать эту задачу этот – один из самых эффективных по времени. Кажется, я умею отвечать на запрос за $\mathcal{O}(1)$ с предподсчётом за то же время, но писать это на практике я, конечно же, не буду.

Максимальный путь в дереве

В следующей задаче практики от нас хотели найти наибольший взвешенный путь в дереве. Сразу примечание – да, это решается динамикой за линейное время, например, так: найдём для каждого ребра наибольший *вертикальный* взвешенный путь, начинающийся в этом ребре и идущий вниз. Затем перебрать самую верхнюю вершину искомого пути и в ней выбрать два максимума по рёбрам вниз. Однако сейчас нас интересует более медленное решение центроидами.

Нам надо перебрать путь. Мы уже знаем, что делать – будем перебирать центроид на этом пути. Ну посчитаем, как в прошлый раз, два максимума, но уже с центроидами.

Вообще, центроиды дают офигительное сведение: за $\mathcal{O}(\log n)$ к асимптотике можно свести задачу про пути к задаче про пути, проходящие через корень.

Максимальный короткий путь в дереве

Далее на практике хотелось посчитать наибольший взвешенный путь длины не более l.

<u>Disclaimer</u>. Это **не** то, что рассказывалось на практике. Я не помню, что там было, но это вроде работает.

Начало такое же: перебираем центроид, смотрим на его дерево, пытаемся найти такой же путь, но обязательно проходящий через сам центроид (то есть корень). На этот раз придётся аж два раза пройтись по вершинам дерева каждого центроида.

В первый раз просто для каждой вершины запомним пару ⟨длина пути в эту вершину, вес пути в эту вершину⟩. И тут же для каждого поддерева корня и для каждой длины вертикального пути от корня в этом поддереве запомним максимальный вес вертикального пути от корня такой длины в этом поддереве. А потом ещё и посчитаем максимумы на префиксах в этом деле и получим то же, но для длины не более заданной. Заметим, что длина полученных данных суммарно для всех центроидов – $\mathcal{O}(n \log n)$, или $\mathcal{O}(|C(v)|)$ для каждого центроида, так как если в поддереве был путь от корня длины h, то в том поддереве было хотя бы h вершин уж точно.

Теперь второй проход. Перебираем один конец пути и подбираем второй. Пусть сейчас мы смотрим на вершину на глубине d. Тогда нужно взять максимум по всем поддеревьям, не совпадающим с текущим, посчитанных величин для длины $l - d$. Ну надо просто как-нибудь упорядочить поддеревья (например, в том порядке, в котором они заданы), и посчитать для каждой длины максимум на префиксе и суффиксе по деревьям. Хранить префиксные и суффиксные максимумы, однако, слишком много – их суммарно может быть квадрат. Тогда пусть конец, который мы перебираем, будет "правым" – то есть тем, который находится в более правом в нашем порядке поддереве. Будем перебирать поддеревья в нашем порядке слева направо и поддерживать префиксные максимумы (обновили, когда вышли из поддрева, причём только тот префикс, на котором в нашей текущей вершине что-то есть). Тогда вся часть суммарно работает за $\mathcal{O}(|C(v)|)$, а всё решение – за $\mathcal{O}(n \log n)$.

GCD на отрезке

Ну, и напоследок. Дан массив, хочется за $\mathcal{O}(\log n)$ уметь считать GCD на отрезке $[L; R]$.

Заметим, что массив – частный случай дерева. А именно бамбук. В терминах дерева задача выглядит как "найти GCD на пути". А это мы умеем – найдём GCD пути до центроида, GCD пути от центроида (ну предпосчитаем) и возьмём их GCD.

Я считаю, что тут уже разобрано достаточно задач для мозгового разогрева на тему центроидов. Если вы считаете иначе, вы знаете, как со мной связаться.

Деревья поиска

Лекция 26 апреля 2017 года

3.1. B-деревья

> Сейчас я вам дорисую этого осьминога.
>
> С.К.

До сих пор мы изучали бинарные деревья поиска. Казалось бы, а почему бы и не сделать их более чем бинарными? B-деревья (B-tree) развивают эту идею.

Def 3.1.1. B-деревом называется дерево поиска, в котором выполнены следующие свойства:

(a) Во всех вершинах, кроме корня, хранится количество ключей в полуинтервале $[k-1; 2k-1)$. В корне же хранится просто меньше, чем $2k$ ключей. Количество детей у вершины на 1 больше, чем количество ключей, то есть лежит в полуинтервале $[k; 2k)$, кроме корня. Ключи в вершине упорядочены, все ключи в поддереве самого левого сына меньше левого ключа, в самом правом поддереве больше самого правого ключа, а в поддереве i-того сына (не самого правого и не самого левого) лежат между $i-1$-ым ключом и i-тым ключом.

(b) Все листья находятся на одной глубине.

Число k мы выбираем заранее.

Lm 3.1.1. Глубина B-дерева $d = \mathcal{O}(\log n)$. Даже $\Theta(\log n)$.

Доказательство.

$$k^d \leq n \leq (2k-1)^d$$
$$d \leq \log_k n \ \wedge \ \log_{2k-1} n \leq d$$

□

Если $k = 2$, то это называется 2-3-дерево (2-3-Tree).

Если мы научимся добавлять и удалять в эту структуру, поддерживая свойства из опрения, то мы победили. Давайте учиться.

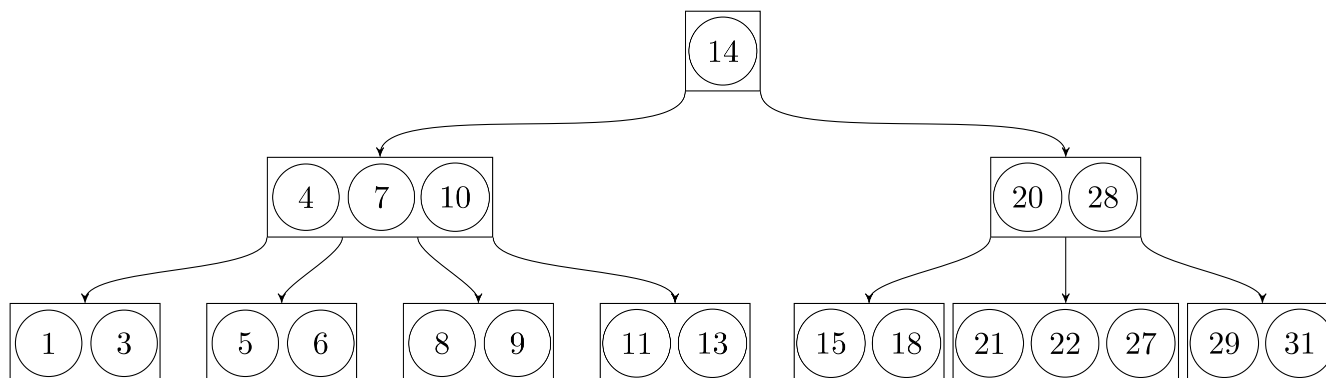


Рис. 3.1: Валидное B-дерево ($k = 3$)

3.1.1. `Add` в B-дереве

> – А куда мы дели детей?
> – А дети как висели, так и висят.
>
> Аня Швецова и С.К.

Схема такая: ищем, куда бы нам добавить новую вершину, добавляем её, после чего, возможно, ключей стало слишком много. Тогда мы спихиваем нашу проблему на нашего отца – берём средний ключ, разбиваем вершину на две вершины (слева и справа от ключа), а ключ вытаскиваем наверх к отцу, рекурсивно решаем проблему для него, и так далее, пока не дойдём до корня. Если у корня слишком много ключей, вытащим тем же макаром один из них наверх и назовём его новым корнем.

Более подробно напишу как только картинки научусь рисовать.

3.1.2. `Del` в B-дереве

Тут я пока ничего не написал.

3.1.3. Нафига козе баян?

Пусть у нас есть громадная база данных, хранящаяся на диске в удобном нам виде. Нам надо уметь быстро искать в ней значения. Напишем её как B-дерево и поставим $k = 1024$. Дисковое чтение – очень долгая операция, читает сразу по 4 килобайта. Глубина дерева тогда у нас уж точно не больше 4, поэтому поиск мы выполним за 4 дисковые операции.

При больших k значения внутри будем искать бинпоиском. Тогда всё работает за $\log k \cdot \log_k n = \log n$.

3.2. Красно-чёрные деревья

> У красных вершин все дети чёрными рождаются.
>
> С.К.

> Я называю эту глубину чёрной.
>
> С.К.

<u>Def</u> 3.2.1. Красно-чёрное (red-black, RB) дерево – бинарное дерево поиска, в котором каждая вершина покрашена в ~~один из цветов толстовки Академического Университета~~ красный либо чёрный цвет, и при этом выполняются следующие свойства:

(a) Корень чёрный.

(b) У красной вершины не может быть красных детей.

(c) У всех "пустых мест" (т.е. мест, куда можно было бы вставить вершину, или, по-другому, `NULL`-детей) одинаковая чёрная глубина (количество чёрных вершин на пути от корня).

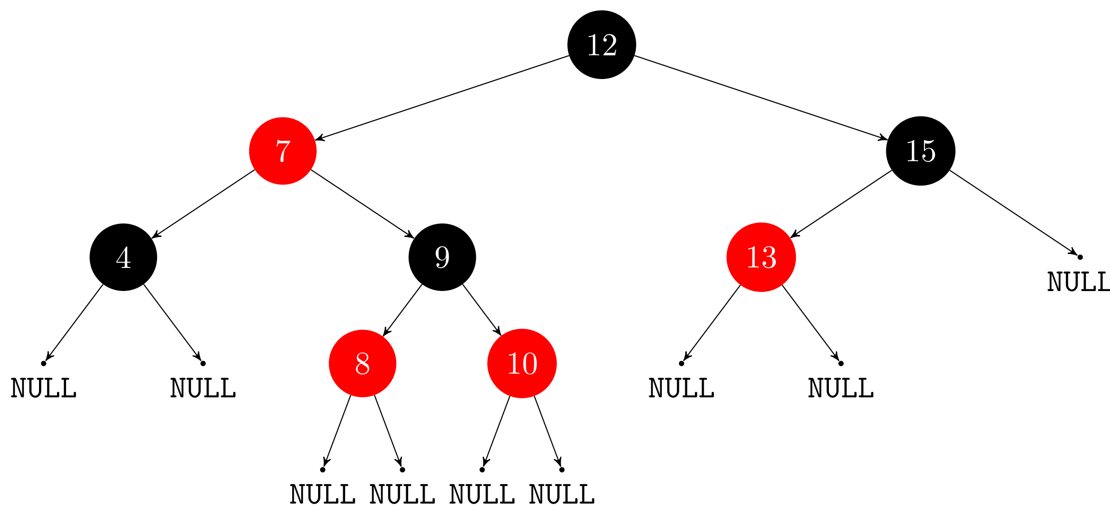


Рис. 3.2: Валидное красно-чёрное дерево

Если почитать в Кормене про RB-деревья, вы увидите это определение и разбор примерно 24 случаев добавления и удаления вершины, лол. Тут этого, конечно же, не будет, ведь нам милосердно рассказали, откуда эти деревья вылезают, дабы мы, столкнувшись с необходимостью написать красно-чёрное дерево, порисовали бы их на бумажке и вывели бы сами ровно те же случаи.

<u>Th</u> 3.2.1. Существует биекция между 2-3-4-деревьями и RB-деревьями.

На всякий случай, 2-3-4-дерево – это B-дерево, в котором разрешили от 2 до 4 детей у вершины. Разумеется, оно не очень подходит под определение, но с этим справиться вполне реально (я не проверял).

Доказательство. Доказательство будет методом пристального вглядывания в картинку **??**, которую я ещё не нарисовал. Вкратце, чёрные вершины задают один из узлов вершины, а красные – все остальные. Свойство (c) красно-чёрных деревьев гарантирует выполнение свойства (b) у B-деревьев. □

3.2.1. AA-деревья

Ну, на самом деле, так себе биекция, так как у нас не различаются случаи на картинке **??**, которую я ещё не нарисовал (впрочем, отображения достаточно). Плохо, когда красных детей двое. Так давайте запретим.

Def 3.2.2. AA-дерево (AA-tree) – красно-чёрное дерево, в котором все красные дети левые.

Теперь у нас уже не 2-3-4-дерево, а 2-3-дерево. С точки зрения кода – минус два случая, тоже неплохо.

3.2.2. Нафига козе баян?

Очень быстро работает. В частности, `std::set` реализован как красно-чёрное дерево.

3.3. Декартово дерево

Def 3.3.1. Декартово дерево (cartesian tree) – бинарное дерево, в вершинах которого хранятся пары $\langle x, y \rangle$, при этом по x оно удовлетворяет свойству бинарного дерева, а по y – свойству кучу с минимумом в корне.

Ну, почему бы, собственно, и нет.
Далее предполагаем, что y_i различны.

Lm 3.3.1. $\forall$ различных y существует единственное дерево на заданном наборе x-ов.

Доказательство. Если y различны, то корень выбирается однозначно. В оставшихся кусках действем по индукции. □

Def 3.3.2. Декартово дерево (treap) – cartesian tree, в котором y выбраны случайными.

Заметьте, что по-английски оно называется по разному, а по-русски – одинаково. Ничего не поделаешь. Если будет неоднозначность, будут использоваться английские обозначения, но в целом "декартово дерево" чаще относят к treap. Как структуру данных уж точно.

Def 3.3.3. Случайным (бинарным) деревом называется дерево, корень которого выбран равновероятно, а девая и правая структуры построены тем же макаром рекурсивно.

Заметьте, что случайное дерево не обязательно равновероятно выбрано из всех.

Lm 3.3.2. Treap – случайное дерево.

Доказательство. Корень каждого поддерева действительно выбран равновероятно – минимумом равновероятно мог быть любой элемент. □

Th 3.3.1. Математическое ожидание глубины случайного дерева $E = \mathcal{O}(\log n)$.

Перед доказательством полезно вспомнить теорему из первого семестра:

Th. Математическое ожидание времени работы `QuickSort` $\le 2n \ln n$.

Кроме того, к теореме прилагается лемма.

Lm 3.3.3. Сумма размеров всех поддеревьев равна сумме глубин всех вершин.

Доказательство. Посчитаем сумму равзмеров всех поддеревьев. Каждую вершину мы посчитаем во всех её предках, то есть [её глубину] раз. □

Доказательство теоремы 3.3.1. Формулировка теоремы требует уточнения, сейчас здесь доказано, что матожидание глубины случано выбранной вершины в случайном дереве $\mathcal{O}(\log n)$.

Построим по QuickSort'у бинарное дерево – в корень выбранный случайно элемент, слева и справа – то, что сделал `partition`, в обоих частях структуру строим рекурсивно. Заметим, что получили в точности случайное дерево.

QuickSort по сути суммарно обойдёт все поддеревья. Отсюда, матожидание суммарного размера всех поддеревьев $\mathcal{O}(n \log n)$, следовательно, по лемме 3.3.3, матожидание суммарной глубины всех вершин $\mathcal{O}(n \log n)$, а случайно выбранной из n, соответственно, $\mathcal{O}(\log n)$. □

Ну классно, осталось научиться поддерживать это дерево.

Будем учиться делать `Merge` и `Split`. Тогда `Add = Split + 2Merge` (посплитили по добавляемому значению, смёржили левую часть с новой вершиной, смёржили получившееся с правой частью), а `Del = 2Split + Merge` (высплитили удаляемую вершину слева и справа, помёржили оставшиеся части).

3.3.1. `Split` в декартовом дереве

Напомним, `Split` берёт дерево T и значение x и делит T на два дерева, в левом из которых все ключи $< x$, а справа – $> x$ (если есть равные, в зависимости от реализации они могут попасть как налево, так и направо).

Нам дали дерево (задаётся корнем v). Первым делом решим, куда пойдёт корень – в левое поддерево или в правое. Мы это знаем исходя из x и значения в корне. Если корень в левом поддереве, то всё левое поддерево корня – тем более, но непонятно, что с правым. Рекрсивно тогда поделим правое, и корень левого поддерева результата запишем как нового правого сына корня. Тогда левое дерево ответа – (получившееся) дерево корня, а правое – новое дерево результата. Если корень в правом поддереве, делаем симметрично.

Заметим, что получившиеся деревья – всё ещё treap'ы с тем же матожиданием глубины. Работает за $\mathcal{O}(h) = \mathcal{O}(\log n)$.

3.3.2. `Merge` в декартовом дереве

> В `Merge` две вершины соревнуются, кто из них круче.
>
> С.К.

Напомним, `Merge` берёт два дерева (левое и правое), все ключи в левом меньше всех ключей в правом, после чего возвращает новое дерево, в котором содержатся все ключи обоих поддеревьев.

Будем на каждом шаге решать, привешиваем ли мы левое дерево к левому сыну правого или правое дерево к правому сыну левого, после чего рекурсивно запускаться от выбранных параметров (e.g. `left->R = Merge(left->R, right)`). Кого к кому? Да того, у кого y меньше.

Мы снова на выходе получаем корректное декартово дерево. Работает за $\mathcal{O}(h) = \mathcal{O}(\log n)$.

3.3.3. Нафига козе баян?

Если надо очень быстро написать бинарное дерево поиска, вы знаете, что делать. Пишется в две функции, каждая строк в 10. И там реально сложно ошибиться.

3.4. Randomized Binary Search Tree

А почему бы и не написать бинарное дерево поиска таким, какое оно есть. Устроено оно будет как декартово. В декартовом мы интересовались y-ми ровно один раз – внутри `Merge`. Давайте там равновероятно выберем корень. Чтобы он был равновероятным, выберем корень левого поддерева (равновероятный среди всех вершин левого поддерева) с вероятностью `size(v->L) / (size(v->L) + size(v->R))`, иначе же выберем корень правого поддерева.

То, что это работает за $\mathcal{O}(\log n)$ в среднем, мы уже доказали.

3.4.1. Нафига козе баян?

Основная часть этого раздела объясняется в разделе про персистентные `Merge` и `Split`. Здесь же просто упомянем ноль дополнительной памяти. Да, конечно, нам нужно хранить размеры поддеревьев (кстати, не забывайте после изменения детей их пересчитывать), но эта информация может быть полезной. Об этом тоже чуть ниже.

3.5. Деревья по неявному ключу

> Когда вставляете в середину, все индексы едут.
>
> С.К.

Дерево по неяному ключу – это такой мощный вариант массива. Но обо всём по порядку.

K-th element

Пусть нам надо найти в бинарном дереве поиска K-тый элемент (в естественном порядке). Будем хранить размеры всех поддеревьев и пересчитывать при изменении детей. Тогда мы точно знаем, лежит ли нужный элемент в левом поддереве относительно корня, в правом или же корень и есть, а также, в первых двух случаях, какой он там по счёту. Ну в первых двух случаях вызовемся рекурсивно, а в третьем вернём корень.

`Add`(K)

Пока это достаточно неестественная операция, но потом всё встанет на свои места. Добавить в дерево элемент между K-тым и $K+1$-ым. ~~В нормальных~~ Во всех, кроме декартова, деревьях поиска это делается через сведение `Add` к `LowerBound`. Ну действительно, по сути предыдущая операция найдёт `LowerBound`. В декартовом дереве же придётся научиться отсплитить K элементов. Это делается ровно также, как и по значению, только `if` немного другой.

`Del`(K)

А это уже нормальная операция. Взять и удалить K-тый в порядке сортировки элемент. Ну надо сделать `Find` и удалить результат.

Последние три операции подводят к мысли – а зачем вообще сортировать по ключу? Да и зачем нужен ключ? Мы уже научились находить k-тый элемент по ключу, вставлять новый элемент в любое место и удалять любой элемент из середины. Естественно, что при этом все индексы едут направо или налево. Ну так и не будем хранить ключи, а дерево будет представлять собой массив (в естественном порядке обхода). Умеющий кучу всего, но всё за $\mathcal{O}(\log n)$.

Так вот, что мы ещё умеем? Например, `Split` и `Merge` (а что мешает)? А значит, мы запросто за $\mathcal{O}(\log n)$ сделаем, например, циклический сдвиг. К сожалению, константа у этого немаленькая, так что на маленьких размерностях уж точно выгоднее делать за $\mathcal{O}(n)$.

И, наконец, из разряда приколов: у нас было декартово дерево с иксами и игреками. Мы сначала сделали из него RBST и отобрали у него игреки, а потом сделали ему неявный ключ и отобрали у него иксы. Ну не милота ли?

3.5.1. Персистентные деревья по неявному ключу

На первый взгляд, ничего необычного произойти было не должно с ними. Однако же мы можем делать `Split` и `Merge`.

Нам ничто не мешает высплитить из середины кусок и смёржить его в конец. Казалось бы, что нам мешает высплитить из середины кусок и смёржить его в конец *старой* версии `split`?

Ничего не мешает. Мы получили странную ещё менее деревянную структуру, однако, тем не менее, рабочую. В ней стало резко больше элементов, но и очень жаль. Факт с практики: если смёржить 10^5 раз дерево само с собой, то можно запросто получить дерево глубиной 10^5 и размером 2^{10^5}, помещающееся, однако, в память. Но с операциями надо быть аккуратным – они будут долгими.

3.6. Абстрактные факты про персистентность

3.6.1. Декартово дерево не дружит с персистентностью

После того, как мы попробуем сделать `Merge(v, v)`, всё нафиг сломается – будут равные ключи. Поэтому в персистентном варианте надо писать RBST.

3.6.2. Garbage collection

> $\texttt{Root}_2$, ты нам больше не нужен.
>
> С.К.

Как помним, персистентность жрёт много памяти. Иногда у нас получаются версии, которые после какого-то момента не нужны. У них есть свои специфичные вершины, нафиг никому уже не нужные. Как же их удалять вовремя?

Один из способов – подсчёт ссылок. Хранить количество вершин, которым ещё нужна данная, и при обнулении удалять и уменьшать счётчики у детей. В C++ это реализовано и называется `shared_ptr`, но он долгий, лучше написать свой или юзать `make_shared` (не проверено).

Splay-деревья и Skip-List

Лекция 3 мая 2017 года

4.1. Rope

Rope – это не структура данных. Rope – это интерфейс. Интерфейс структуры данных, которая умеет много классных вещей:

`insert(i, x)` Вставить `x` перед `i`-тым элементом.

`erase(i)` Удалить `i`-тый элемент.

`get(i)` Взять `i`-тый элемент.

`split(i)` Разрезать на две структуры, в первой из которых будет `i` первых элементов.

`merge(a, b)` "Склеить" две структуры.

`rotate(i)` Циклический сдвиг на i.

И ещё много всего.

Всё это умеет Balanced BST по неявному ключу. В этой лекции научимся делать это ещё с помощью SkipList и SQRT-Decomposition.

4.2. `SkipList`

Прежде всего вспомним, что все эти операции умеет делать обычный односвязный список. Причём `merge` за $\mathcal{O}(1)$, а `insert`, `erase`, `split`, `rotate` – в некотором смысле быстро. А именно, за быстро, если мы умеем `get`, возвращающий "итератор". Эту же операцию мы не умеем быстрее, чем за линию. Будем учиться быстрее.

Идея: хранить в отдельном списке все элементы, в отдельном – каждый второй, потом каждый четвёртый, каждый восьмой, и так далее. Тогда у нас $\mathcal{O}(n)$ памяти, а `get` работает за $\mathcal{O}(\log n)$ – в каждом списке сдвинемся не более чем на один элемент. Но эти списки фиг пересчитаешь при вставке.

Use random, Luck!

Будем хранить много списков $\texttt{List}_i$, в каждом будут храниться нужные нам элементы. Будем считать, что выполняется

$$\forall x \operatorname{Pr}[x \in \texttt{List}_i] = 2^{-i}$$

Заметим, что в нулевом списке тогда лежат все элементы.

Например, это можно реализовать так: элемент переходит из i-того списка в $i+1$-ый с вероятностью $\frac{1}{2}$. Для каждого элемента каждого списка будем хранить тогда ссылку вниз – ссылку на элемент нижнего списка, из которого он перешёл в свой текущий. Заметим, что тогда мы умеем делать `get` за $\mathcal{O}(\log n)$ (матожидание количества непустых списков) – сделав линейный поиск сверху (можно для каждой ссылки дальше хранить её длину), перейдя вниз и повторив. И пересчитывать списки при этом не сложнее.

4.3. Splay-деревья

> Оно на реальных данных зажигает просто нереально!
>
> Дима Саютин

> Оно рассказано так, что его можно запомнить.
>
> С. К.

Очередной укур от Роберта Тарьяна и Даниеля Слейтора. Это BST, ни разу не сбалансированное, которое тем не менее умеет делать операции быстро, иногда за $\mathcal{O}(1)$, хранит 0 памяти в вершине и вообще всячески няшное. Единственный минус – амортизация.

Будем учиться делать `Add` и `Del` из несбаллансированного дерева.

Выразим операцию `Splay(v)` – взять вершину v и сделать её корнем. Мы это уже както делать умеем. Например, пока v – не корень, можно делать малый поворот по ребру, ведущему наверх. Но это работает долго (пример – волосатый бамбук).

Чтобы это работало быстро, нужно подниматься сразу на два ребра. Подробнее, `Splay` делает следующее: пока v – не корень, посмотри на то, что сверху, и сделай одну из трёх операций:

zig Если v – ребёнок корня (лол), то делаем один single rotate.

zig-zig Если два предыдущих ребра до v ведут в одну сторону, сделаем single rotate отца, потом single rotate v.

zig-zag Если в разные стороны, сделай double rotate.

И вот если делать так, то ВНЕЗАПНО это работает за амортизированный логарифм.

Lm 4.3.1. Пусть $x+y=1$, $x>0$, $y>0$. Тогда $(\log x + \log y) \to \max$ достигается в $x=y=\frac{1}{2}$ и равен -2.

Доказательство. Продифференцируйте и всё у вас выйдет. □

<u>Lm</u> 4.3.2. Пусть $x + y = C$, $x > 0$, $y > 0$. Тогда $\log x + \log y \leq 2\log C - 2$.

Доказательство.

$$\log x = \log \frac{x}{C} + \log C$$
$$\log y = \log \frac{y}{C} + \log C$$
$$\log x + \log y = \log \frac{x}{C} + \log \frac{y}{C} + 2\log C \underset{\text{Lm 4.3.1}}{\leq} 2\log C - 2$$

□

<u>Th</u> 4.3.1. Splay-дерево работает за амортизированный логарифм.

Доказательство. $R_v := \log(size(v))$, $\varphi := \sum_v R_v = \mathcal{O}(n \log n)$.

В точности будет доказано, что если мы подняли вершину v до уровня вершины u, то $\Delta\varphi \leq 3(R_u - R_v)$.

Разберём только zig-zig, zig-zag оставлен в кои-то веки в качестве упражнения.

Тут будет картинка, в которой слева дерево (эйлеров обход) $zyxAxBxyCyzDz$, справа – $x'Ax'y'By'z'Cz'Dz'y'x'$.

$$a_i = 2 + \Delta\varphi = 2 + \varphi' - \varphi = 2 + R_{x'} + R_{y'} + R_{z'} - R_x - R_y - R_z = 2 + R_{y'} + R_{z'} - R_x - R_y \leq$$
$$\leq 2 + R_{y'} + R_{z'} - 2R_x \leq 2 + R_{x'} + R_{z'} - 2R_x$$

Константа $t_i = 2$ на самом деле нужна, чтобы оценка в точности сошлась. Берётся из zig.

Надо показать:

$$R_{z'} \leq 2R_{x'} - R_x - 2$$
$$R_{z'} + R_x \leq 2R_{x'} - 2$$

Нужно заметить лишь, что дети z' – A и B, дети x – C и D, а большие потомки x' – A, B, C и D. Значит, осталось лишь применить лемму 4.3.2. □

Есть ещё одна теорема про Splay-деревья.

<u>Th</u> 4.3.2. Пусть k_i – количество запросов к вершине i. Splay сработает за $\sum_i k_i(3\log\frac{n}{k_i} + 1)$.

Доказательство. Потенциал вершины равен теперь не сумме размеров, а сумме сумм весов в поддеревьях, а вес вершины v равен k_v. Дальше работают те же рассуждения, что и в теореме 4.3. □

И это одна из немногих классных фич Splay-деревьев – адаптивность под запросы.

Подразумевается, что `Add` и `Del` мы при этом делаем втупую, как в BST, да. Дополнительное упражнение: проверить, что потенциалы не сломаются.

4.4. SQRT-декомпозиция

Идея корневых оптимизаций у нас уже была в первом семестре, в этом конспекте именно сейчас мы будем проходить структуру данных, на ней основанную.

4.4.1. Структура данных на массиве

Будем решать задачу RMQ (Range Minimum Query) – дан массив (в разных вариантах изменяющийся или нет, SQRT-декомпозиция умеет и так, и так), требуется уметь искать в нём минимум на отрезке.

Идея: разбить отрезок на $\sqrt{n}$ кусков длиной $\sqrt{n}$ каждый и посчитать минимум в каждом. Тогда любой отрезок выглядит либо как подотрезок какого-то куска, либо как суффикс какого-то куска + какое-то количество целых кусков + перфикс какого-то куска. В первом случае длина отрезка не превосходит $\sqrt{n}$ и мы можем посчитать ответ втупую. Во втором случае количество полных отрезков, длина префикса и длина суффикса также не превосходят $\sqrt{n}$ (суммарно $3\sqrt{n}$), минимум на внутренних отрезках мы посчитали и ответ также выдадим за $\mathcal{O}(\sqrt{n})$.

Вообще-то RMQ мы уже умеем решать быстрее, а научимся ещё быстрее. Но SQRT-декомпозиция умеет считать гораздо более сложные функции, от которых более продвинутые структуры данных могут и загнуться.

4.4.2. SQRT `Split/Rebuild`

Идея другой структуры данных, умеющей в Rope.

Будем хранить какой-нибудь массив, в котором в каком-нибудь порядке будут храниться элементы структуры Также будем хранить список из k пар чисел $[l; r)$ – ~~отрезков~~ полуинтервалов хранимого массива в том порядке, в котором они хранятся в структуре.

В самом начале у нас будет храниться массив как он есть и один отрезок $[0; n)$. Будем всё, что нужно, выражать через операцию `Split(i)` – сделать так, чтобы `i` был началом своего отрезка и вернуть указатель на этот отрезок. `Split` будет работать за $\mathcal{O}(k)$ – найти элемент, если надо, разбить отрезок на два и вставить новый в список. Если же k стало больше, чем $\sqrt{n}$ (это случается не чаще, чем каждые $\sqrt{n}$ операций), делаем `Rebuild` – то есть выписываем элементы заново, как было.

`Insert` = `Split` + вставить новый отрезок.

Также бывают `Erase`, `merge`, `split` – словом, всё, что нужно. Если на каждом отрезке хранить минимум, то можно и RMQ.

В общем, мощная штука.

4.5. Notes on persistence

`Root`, разделись, пожалуйста,
на три дерева

С. К.

4.5.1. Этот персистентный мир

Потратив лишний log памяти и времени в асимптотике, мы любую структуру данных на массивах/списках/да чём угодно можем сделать персистентной. Вот.

Ещё можно вспомнить, а то и узнать про деревья отрезков, которые тоже бывают персистентными и сказать, что в таких случаях они работают побыстрее.

4.5.2. ~~Распилы и~~ откаты

Если нам можно в offline, то можно и без лога. В таком случае мы точно знаем заранее наследников всех версий (т.е. версии, получаемые в процессе из каждой). Таким образом мы получаем дерево версий, йоторое можно обойти dfs'ом всего с одной структурой: при проходе вниз по ребру мы делаем изменения, которые это ребро предполагает, а при проходе обратно мы их *откатываем*, то есть производим в обратном порядке, чтобы вернуть предыдущую версию.

Амортизация здесь не пройдёт, как и в прошлом параграфе – если у какой-то версии амортизированной структуры много детей и после этой версии обязательно делать долгую операцию, то мы её сделаем "количество детей" раз и сломаем себе вю асимптотику.

4.6. Персистентная очередь

> Окажется, что голову я
> положил вот сюда вот.
>
> С. К.

Я не вижу смысла писать здесь что-то существенно подробное. Делайте, как было в очереди с минимумом без амортизации, и всё будет хорошо. На лекции был буквально повтор той самой очереди.

Деревья отрезков и персистентность

Лекция 10 мая 2017 года

5.1. Дерево отрезков

> Деревья отрезков все одинаковые, поэтому над ними обычно люди не думают.
>
> С.К.

Мы уже знакомы с деревьями отрезков, но не в этом конспекте. Это пора исправить.

Дерево отрезков – бинарное дерево, в листьях которого находятся элементы массива. Также хочется, чтобы оно было как-то сбаллансированным. Например, для массива размером 2^k хочется иметь полное бинарное дерево высоты k. Будем делать так (для массива длины n):

- Корень отвечает за ~~отрезок~~ полуинтервал $[0; n)$.
- Если вершина v отвечает за $[L; R)$ и $R - L > 1$, то её левый сын отвечает за $[L; (L + R)/2)$, а правый – за $[(L + R)/2; R)$.
- Иначе вершина – лист.

Вроде видно (мне лень проверять), что тогда дерево – полное бинарное и вообще может храниться как куча.

Зачем это надо? Ну, допустим, хочется нам уметь изменять элементы массива и спрашивать максимум на отрезке. Будем (потом будет рассказано, как) хранить в вершине максимум на отрезке, за который она отвечает. Тогда отрезок разбивается на сколько-то отрезков, за каждый из которых отвечает вершина дерева. Например, подойдёт множество листьев, отвечающих за элементы нужного отрезка. Их, правда, слишком много, поэтому будем брать минимальное по размеру множество походящих вершин.

Понятно, что это минимальное множество получается таким образом: сначала выберем все нужные листья, потом, пока можем, выбираем двух братьев из выбранных и заменяем их на их отца. Это тоже работает долго, зато позволяет доказать следующую лемму:

Lm 5.1.1. Размер минимального множества $\le 2\log_2 n$.

Доказательство. Пусть мы после скольких-то объединений получили все вершины на высоте не выше некоторого уровня. Выбранные на этом уровне вершины тогда образуют непрерывный отрезок вершин этого уровня. В непрерывном отрезке может быть не более двух вершин, братья которых не лежат в этом отрезке: самая левая и самая правая. Если же у вершины есть брат в том же отрезке, то она в разбиение не попадёт. □

Теперь научимся выбирать это минимальное множество. Способов это написать есть два: снизу и сверху.

5.1.1. Дерево отрезков снизу

Это способ реализовать именно то, что мы делаем в лемме 5.1.1.

Будем хранить дерево в массиве размером $2n$, листья имеют номера $n..2n-1$, родителем врешины v является вершина $v/2$.

1. Максимум на отрезке.

   Начинаем на уровне листьев (при таком способе они имеют одинаковую высоту, однако, стоит заметить, что мы храним ни разу не дерево, что будет объяснено чуть позже). Пока отрезок ещё не обработанных вершин не пуст, мы смотрим, если надо, на крайние вершины и поднимаемся на уровень выше ($L \to L/2$, $R \to R/2$). На крайнюю вершину надо смотреть, если её брат не лежит в отрезке (смотрим на чётность номера), после просмотра её нужно от отрезка отрезать (каламбур). Работает за $\mathcal{O}(\log(R-L))$.

2. Изменение в точке.

   Тут просто. Изменяем значение в листе, потом поднимаемся, пока можем, наверх, и пересчитываем максимум во внутренней вершине через значения её детей. $\mathcal{O}(\log n)$.

5.1.2. Дерево отрезков сверху

Храним вершины также, корень в единице, потомки вершины v – вершины $2v$ и $2v+1$.

Update Спускаемся из корня вниз, рекурсивно идём в нужного ребёнка, после чего пересчитываем значение в вершине.

GetMax Спускаемся от корня вниз рекурсивно. В аргументах функции будут числа vl, vr, l, r – границы отрезка, за который отвечает текущая вершина и границы отрезка запроса. Если отрезки не пересекаются, возвращаем $-\infty$, если отрезок вершины вложен в отрезок запроса, возвращаем значение в вершине, иначе рекурсивно запускаемся от двух детей. Просмотрим не больше 4 вершин на уровень.

Всё за $\mathcal{O}(\log n)$.

5.1.3. Сравнение последних двух

Преимущества запросов снизу:

1. Скорость. Нет рекурсии + быстрый ответ на запросы на небольших отрезках.

2. Всего $2n$ памяти.

Количество вершин при запросе сверху

Хорошие новости: не больше $4n$. Ну точно не больше чем $2\lceil\log_2 n\rceil$. Плохие новости: оценка достигается.

Преимущества запросов сверху

Их три.

1. Persistent. Умеет быть персистентным, если надо и работать за тот же $\log n$. Тогда нельзя хранить дерево как кучу и придётся хранить его на указателях.

2. Умеет в отложенные операции.

   Пусть нам нужно брать тот же максимум и делать += на отрезке. Отрезок также разбивается на log маленьких вершин. Для каждой запомним, что к её отрезку прибавилось число – это будет *модификатор* этой вершины. Далее перед тем, как перейти из вершины в вершину, будем *раздавать* её модификатор детям – изменять модификатор детей, а вершине ставить модификатор 0. Ну и когда возвращаем значение в вершине, помним про модификатор. Такое решение всё ещё работает за $\mathcal{O}(\log n)$.

3. Может быть динамическим.

   Пусть мы хотим хранить индексы до $M = 10^{18}$, но какие-то значения будут только в N индексах. Тогда это тоже можно сделать деревом отрезков, но уже на указателях. Просто не будем хранить вершину, если в ней нет значений. Пишется также.

   Time: $\mathcal{O}(\log M)$

   Memory: $\min\{2M - 1; k \log M\}$.

   Кроме того, это можно делать в оффлайне, если посортим все встречающиеся индексы и сделаем *сжатие кординат* – отображение из $[0; M)$ в $[0; 2k)$ с сохранением порядка. Это уже $\mathcal{O}(k \log k)$ времени и $\mathcal{O}(k)$ памяти.

5.2. Двумерные деревья

Давайте ещё что-нибудь
засунем в дерево отрезков

С. К.

5.2.1. ДО сортированных массивов

В вершине можно хранить ещё, например, сортированный массив всех значений на этом отрезке. Тогда у нас будет $O(n \log n)$ памяти – $\mathcal{O}(n)$ на каждом уровне. Строить это тоже можно за $\mathcal{O}(n \log n)$: если мы построили массив в детях, чтобы построить массив в вершине, нужно сделать `merge` за линейное время.

Запрос к этому будет, например, такой: сколько чисел на отрезке $[l; r]$ лежат в отрезке $[a; b]$. Нужно разбить отрезок на логарифм вершин и в каждой сделать два бинпоиска. $\mathcal{O}(\log^2 n)$ на запрос.

Кроме того, можно искать k-тую порядковую статистику на отрезке (k-тый в порядке сортировки элемент на отрезке). Пока $\mathcal{O}(\log^3 n)$ – будем делать бинпоиск по ответу (координаты сожмём заранее), а внутри запрос за $\mathcal{O}(\log^2 n)$.

5.2.2. ДО of ДД

Если в вершине хранить декартово дерево (да, в общем-то, любое бинарное дерево поиска), то ещё можно будет делать присвоение в точке. Нужно будет уметь удалять из середины и добавлять в середину – как раз умеем через ДД.

5.2.3. ДО of (ДО of Sorted Array)

Можно и трёхмерные запросы – в вершине хранить двумерную структуру.

А можно и k-мерные. Работать будет за $\mathcal{O}(n \log^{k-1} n)$ памяти и $\mathcal{O}(\log^k n)$ времени на запрос.

5.3. Scanline

> Любые две штуки, какими бы ни были эти две штуки, можно назвать точкой на плоскости
>
> С. К.

По сути двумерные запросы позволяют решать задачи про точки на плоскости ($X_1 \le x_i \le X_2$, $Y_1 \le y_i \le Y_2$). Однако некоторые из них, не требующие онлайн-запросов, можно решать быстрее.

5.3.1. Прямоугольники и точки на плоскости

В слудующих двух задачах у нас будет дано n точек на плоскости и m прямоугольников со сторонами, параллельными осям.

Для каждого прямоугольника найти количество точек внутри

Отсортируем точки сначала по x, потом по y. Пусть мы умеем считать количество точек в стакане $G(X, L, R)$ – множестве $\{(x, y) \mid x \le X, L \le y \le R\}$. Тогда количество точек в прямоугольнике с углами (левый нижний и правый верхний) (x_1, y_1) и (x_2, y_2) – разность количеств точек в стаканах $G(x_2, y_1, y_2)$ и $G(x_1, y_1, y_2)$. (здесь нужна большая аккуратность со знаками $\le$, но идея, вроде бы, ясна).

Стаканы отсортируем по X. Будем идти слева направо и встречать события двух типов – новая точка и стакан. Будем хранить ДО, в котором для каждого y будет хранится количество уже пройденных точек с таким y. Тогда количество точек в стакане, когда к нам придёт стакан, будет равно сумме на отрезке в ДО.

$\mathcal{O}((n + m) \log n)$.

Для каждой точки найти количество покрывающих её прямоугольников

Теперь изменять ДО будут стаканы, а спрашивать – точки. Если к нам пришёл стакан x_1, нужно сделать `+= 1` на отрезке, а если x_2, то `+= -1` на отрезке. Если пришла точка – взять значение в точке.

$O((n + m) \log n)$.

Возможно, к стаканам нужна картинка. Если да, свяжитесь со мной.

5.3.2. Переход к online

Если запросы заранее неизвестны, а объекты не меняются (во втором примере выше, например, объектами являются прямоугольники, а запросами – точки), то можно воспользоваться персистентностью – пройдём по объектам сканлайном, как если бы у нас были запросы, и запомним для каждого X персистентную копию ДО в момент времени X. После этого на запросы мы умеем отвечать в online. Памяти получится побольше, но ответ уже за $\mathcal{O}(\log n)$.

5.3.3. k-тая порядковая статистика на отрезке

> Мы к этому долго шли и, наконец, пришли
>
> С.К.

Заметим, что внутри бинпоиска по ответу в нашем предыдущем алгоритме (см. раздел 5.2.1) вполне себе двумерный запрос. Однако запросы нам в любом случае заранее неизвестны, так как мы делаем бинпоиск. Но в предыдущем разделе мы уже научились отвечать на них в online за $\mathcal{O}(\log n)$, поэтому k-тую порядковую статистику на отрезке мы уже умеем искать за $\mathcal{O}(\log^2 n)$.

Осталось совсем немного – избавиться от бинпоиска. Сейчас у нас есть два (персистентных) ДО из нулей и единиц и нам нужно найти k-тую единицу в их разности. В вершине будем хранить сумму. Давайте рекрсивно спускаться сразу по двум копиям. Мы всегда знаем, куда нам спускаться дальше – направо или налево исходя из суммы (количестве единиц) в их разности. Значит, спустившись до низа, мы сможем найти k-тую единицу за $\mathcal{O}(\log n)$.

RMQ и LCA

Лекция 17 мая 2017 года

В этой лекции новое обозначение: если структура строится за $\mathcal{O}(f)$ и отвечает на запрос за $\mathcal{O}(g)$, говорят, что она работает за $\langle f, g\rangle$.

6.1. Range Minimum Query

Эта задача уже возникала у нас в теме про корневые оптимизации и даже про деревья отрезков. Мы умеем её решать такими способами:

- Дерево отрезков. Работает за $\langle n, \log n\rangle$.
- Centroid Decomposition. Массив – тоже дерево (ну такое). Ну было у нас, да.

Сейчас мы сделаем ещё что-то.

6.1.1. Sparse table

Работает за $\langle n \log n, 1\rangle$. Умеет только в идемпотентные операции.

Вот бы нам можно было для каждой длины запомнить минимум на каждом отрезке! Но нет, это долго. Поэтому запомним только на отрезках длины 2^k, где k – целое.

Как узнать ответ на $[L; R)$? Пусть k – максимальное число, для которого $2^k \le R - L$. Тогда ответ для k – это минимум из ответов для $[L; L + 2^k)$ и $[R - 2^k, R)$. Действительно, эти отрезки перекрываются (тут пользуемся идемпотентностью). Эти ответы мы, к тому же, уже посчитали.

Осталось найти k. Предпосчитаем его для каждой длины заранее и обзовём `Log`(len).

6.1.2. SparseTable++

Как корневая, но не корневая.

Разобьём отрезок на отрезки длины $k = \log n$. На каждом посчитаем минимум и построим на них SparseTable. Предподсчёт работает за $(n/\log n) \cdot \log \frac{n}{\log n} \le n$ времени и памяти.

Как теперь отвечать на запросы? Снова отрезки бывают лежащие полностью в блоке и не лежащие в блоке полностью. С первыми всё понятно: посчитав минимумы на префиксах и суффиксах каждого блока, мы сможем отвечать на эти запросы за $\mathcal{O}(1)$.

С маленькими отрезками не всё так гладко, ~~как с матанализом~~. Пока есть два варианта: посчитать на них SparseTable и дерево отрезков. Первое сработает за $(n/\log n)(\log n)\log\log n = n\log\log n$ времени и памяти на предподсчёт ($\langle n\log\log n, 1\rangle$ в итоге), второе – за $\langle n, \log\log n\rangle$. Одна из целей этой лекции – побороть эти отрезки и сделать $\langle n, 1\rangle$.

6.2. LCA

LCA (Lowest Common Ancestor) – задача о нахождении самого нижнего общего предка двух вершин в подвешенном дереве. Она очень тесно связана с RMQ. Будем её решать.

6.2.1. Двоичные подъёмы

Та же SparseTable, только вертикальная.

Для каждой вершины посчитаем её 2^0-го предка, 2^1-го предка, 2^2-го предка, ..., $2^{\log n}$-го предка. Это делается в процессе DFS динамикой (`up[v][k] = up[up[v][k-1]][k-1]`) либо просмотром стека. Также посчитаем глубину каждой вершины.

Теперь мы можем за $\mathcal{O}(\log n)$ умеем прыгать из вершины на любую высоту k вверх. Для этого самое простое придумывательно (но не написательно) решение такое: выпишем двоичное представление числа k и попрынгаем по всем единицам в нём.

Пусть нам дали вершины u и v. Сначала сделаем так, чтобы они были на одной высоте. Умеем прыгать как в предыдущем абзаце. Теперь будем идти по $k = \log n$ до 0 вниз и сравнивать `up[v][k] == up[u][k]`, и если да, делать присваивание `v = up[v][k], u = up[u][k]`. В конце мы поднимемся ровно до вершин под LCA. Действительно, мы в точностью прыгаем на нужную нам высоту -1.

Есть более быстрый способ, требующий, однако, предподсчитать времена входа и выхода. О них подробнее.

Обойдём дерево `dfs`-ом и выпишем для каждой вершины время, в которое мы туда вошли и время (счётчик), в которое мы оттуда вышли. Тогда мы умеем за $\mathcal{O}(1)$ узнавать, является ли одна вершина предком другой. Для этого мы должны были в неё войти раньше, а выйти позже.

Теперь о способе написания LCA. Не будем никуда подниматься, а прыгать будем только из одной вершины. Проходимся по $k = \log n$ до 0 и прыгаем на 2^k если не попадаем в предка второй вершины. Работает точно потому же, почему работал предыдущий способ.

6.2.2. LCA → RMQ ± 1

У дерева есть *эйлеров обход* – обход в порядке dfs'а по рёбрам вниз и вверх. Выпишем его. Можем выписать сами рёбра, а можно выписать вершины и их высоты (при входе в дерево и каждый раз, когда спускаемся в новую вершину). Запомним для каждой вершины первое вхождение в обход.

Утверждается, что LCA – вершина минимальной высоты на отрезке между первыми вхождениями двух вершин. Действительно, она там появится, а вершины меньшей высоты – нет. Поэтому мы теперь умеем сводить LCA к RMQ.

Почему ± 1? Потому что высоты соседних в обходе вершин всегда отличаются на 1. Этим можно будет пользоваться, что мы потом и сделаем.

6.2.3. LCA-offline $\mathcal{O}(\alpha(n))$ (Тарьян)

Где множество, там и СНМ

С. К.

Сейчас мы научимся решать LCA для кучи запросов за очень быстро, если запросы известны заранее.

Обойдём дерево dfs'ом. Вершины, как помним, бывают белые, чёрные и серые. Отвечать на запрос будем в тот момент, когда посетим вторую по очереди вершину запроса.

Ответ – одна из серых вершин. Для каждой серой вершины будем поддерживать множество "вторых" вершин, для которых она является ответом, если "первая" вершина – текущая. Как изменяются эти множества? Если вершина становится серой (вошли в вершину), надо завести новое множество из одной вершины, а если становится чёрной (вышли из вершины), надо объединить её множество с множеством её родителя. Это в точности то, что умеет СНМ. Значит, на m запросов ответим за $\mathcal{O}((n+m)\alpha(n))$, что является весьма крутым временем.

6.3. Снова RMQ

6.3.1. RMQ± 1 за $\langle n, 1\rangle$

Асимптотика, конечно, говорит, но лучше бы она промолчала.

Рома Колганов

Помним, что в RMQ нам осталось научиться лишь считать ответ для маленьких отрезков за требуемое время. Воспользуемся идеей четырёх русских.

Если у нас есть ограничение ± 1, то можно воспользоваться такой идеей: принципиально (структурно) разных отрезков длины не более k у нас всего $\mathcal{O}(2^k)$. Имеется ввиду, что на каждом из $k-1$ переходов между соседними вершинами мы делаем либо $+1$, либо -1.

Так вот. Возьмём теперь $k = \frac{\log n}{2}$. Для каждого принципиально разного класса отрезков длины k и каждого его подотрезка хоть за линию посчитаем минимум. Это будет работать за $\mathcal{O}(2^k k^3) = \mathcal{O}(\sqrt{n}\log^3 n) = o(n)$. Теперь для каждого блока длины k поймём его класс. Всё, теперь мы умеем решать задачу RMQ± 1 за очень классное в теории время.

6.3.2. RMQ →LCA

Последний рывок к решению RMQ за очень быстро – научиться сводить RMQ к LCA.

Построим Cartesian tree (как декартово дерево, только без рандома, см. в соответствующей главе) на парах $\langle i, a_i \rangle$, где a – элементы исходного массива. Его мы умеем строить за $\mathcal{O}(n)$. Теперь ответ на RMQ – вершина с минимальным приоритетом на отрезке $[L, R]$, то есть вершина с минимальной высотой в Cartesian tree, то есть LCA.

Вот. Теперь мы зачем-то умеем решать RMQ за быстро. Но идея классная.

6.4. Level Ancestor

Самое сложное – это название.

С. К.

Тут был какой-то алгоритм Вишкина, который я очень фигово записал, поэтому тут пока пусто :(

6.5. Euler Tour Trees

Хочется уметь поддерживать дерево динамически – уметь добавлять и удалять рёбра. Ну надо.

Боюсь, что ничего умнее чем “давайте хранить эйлеров обход в дереве поиска по неявному ключу” я не напишу.

HLD и Link-Cut Trees

Лекция 24 мая 2017 года от Алексея Давыдова

7.1. Heavy-Light Decomposition

Вспомним идею про лёгкие и тяжёлые рёбра. Она у нас была в теме про Leftist Heap и Fibonacci Heap. Применим её к дереву.

Напомню. Ребро называется тяжёлым, если на нём висит больше половины поддерева. Иначе ребро называется лёгким. Есть замечательное свойство: количество лёгких рёбер на пути от любой вершины до корня не превосходит $\log_2 n$. Действительно, при проходе вверх по лёгкому ребру размер поддерева текущей вершины увеличивается хотя бы вдвое. Также из любой вершины торчит не более одного тяжёлого ребра.

Разобьём рёбра дерева на вертикальные пути из тяжёлых рёбер. Каждое лёгкое ребро будем считать отдельным путём. Тогда мы умеем считать почти что угодно на пути: на каждом пути построим структуру, умеющую в запросы на отрезке, найдём LCA и посчитаем требуемое на двух вертикальных путях за $\mathcal{O}(\log n)$ запросов к структурам: каждый вертикальный путь состоит из $\leq \log n$ тяжёлых вертикальных путей.

В частности, мы теперь умеем минимум и сумму на пути за $\mathcal{O}(\log n)$. Но и много крутого.

Лектор заботливо подсказывает, что вместо лёгких/тяжёлых рёбер проще считать тяжёлым ребро, на котором висит наибольшее поддерево в каждой вершине. Так оно будет, во всяком случае, не хуже.

7.2. Link-Cut trees

> А если мы навернём Splay на такую штуку, не будет лучше?
>
> — Миша Ютман

Уже умеем делать Euler-Tour Tree. Хочется большего: например, они не умеют в динамический минимум на пути. А вот в сумму умеют, но это уже другая история.

В минимум на пути умеет HLD (но вы упоролись, если ищете его с помощью HLD). Сейчас будем делать что-то похожее, но динамичное.

Будем всегда хранить дерево как набор вертикальных путей. Пофиг на тяжёлость. Каждый путь будет храниться в дереве по неявному ключу и иметь вершину-родителя. Все операции будем выражать через операцию `expose`, которая берёт вершину и делает так, чтобы её вертикальный путь начинался в корне. `expose` работает втупую и за линию: отрезает от путей на пути к корню лишнее и объединяет.

Th 7.2.1. `expose` сделает амортизированное $\mathcal{O}(\log n)$ операций с деревом поиска.

Доказательство. Введём потенциал $\varphi = \langle$количество тяжёлых рёбер, попавших в пути$\rangle$. Оценим количество операций с деревьями поиска: $a_i = t_i + \Delta\varphi$.

Пусть мы попрыгали по h тяжёлым рёбрам. Тогда потенциал увеличился на h. Пусть мы попрыгали по l лёгким рёбрам. Тогда потенциал уменьшился не более чем на l, но пусть бы он даже не уменьшился.

$a_i = t_i + \Delta\varphi \le (h + l) + (-h) = l \le \log n$, помним свойство про количество лёгких рёбер на пути до корня. □

Также полезна операция `to_root`. Она делает вершину корнем дерева. Она делает `expose`, после чего разворачивает (в BST умеем) путь, в котором лежит корень.

Th 7.2.2. `to_root` работает за амортизированный $\mathcal{O}(\log n)$.

Доказательство. Казалось бы, `expose` + $\mathcal{O}(\log n)$. Но надо понять, как разворот меняет потенциал.

Рёбра, лежащие не в пути корня, не изменились (в плане тяжести). Но могли изменится рёбра, там лежащие. Перед разворотом там был $\mathcal{O}(\log n)$ тяжёлых рёбер и после разворота там $\mathcal{O}(\log n)$ тяжёлых рёбер. Значит, $\Delta\varphi \le 2\log n$, чего и хотелось. □

Теперь мы умеем делать `link` и `cut`. В первом случае надо сделать `to_root` и подвесить путь, во втором – разрезать путь. На практике вроде как было доказано, что потенциалы от этого не умрут.

Corollary 7.2.1. Link-Cut Tree с произвольным BST внутри работает за $\mathcal{O}(\log^2 n)$.

Почему такая странная формулировка? Потому что `splay` магическим образом сработает за $\mathcal{O}(\log n)$. Это также доказано на практике.

7.3. MST за $\mathcal{O}(n)$

7.3.1. Проверка остовного дерева на минимальность

За $\mathcal{O}(E+V\log V)$. Была в теордз. Суть: берём ребро и проверяем, не станет ли лучше, если мы возьмём его вместо какого-то на цикле, который оно соединяет. Для этого надо взять максимум на пути и проверить, не больше ли он этого ребра. Доказывается элементарно от противного (типа ну возьмём тогда это ребро и будет норм).

Для подсчёта максимума на пути (нам нужны только вертикальные, так как умеем в LCA) предлагается юзать СНМ без ранговой эвристики. Амортизированная оценка на сжатие путей останется.

Кажется, я сам не понял, что происходит в прошлом абзаце.

Тарьян предлагает линейный алгоритм. Об этом нужно помнить.

7.3.2. Собственно Randomized BST

Для простоты предположим то же, что и Борувка: MST единственно. Алгоритм назовём $F(V, E)$. Делать будем следующее:

I. 3 шага из Борувки. Останется $\frac{n}{8}$ вершин.

II. От рандома выберем $\frac{m}{2}$ рёбер (множество E').

III. Запустим $F(V, E')$. Получим множество рёбер E_f, остальные рёбра из E' не могут лежать в MST.

IV. Остались рёбра $(E \backslash E')$. В нём есть рёбра, которые могут улучшить E_f (назовём их множество $(E \backslash E')^+$), и остальные $((E \backslash E')^-)$. Остальные уж точно не лежат в MST. Запустим $F(V, E_f \cup (E \backslash)^+)$.

Всё хорошо, только надо понять, почему за линию работает. Да, в третьем пункте мы стали снова искать максимум на пути какой-то Тарьяновской магией.

Теперь $F(V, E)$ обозначает время работы. По контексту должно быть понятно, где я поленился писать модули.

<u>Th</u> 7.3.1. $F(V, E) = \mathcal{O}(|V| + |E|)$.

Доказательство.

$$F(V, E) = (E + V) + (E) + F\left(\frac{V}{8}, \frac{E}{2}\right) + F\left(\frac{V}{8}, \left(\frac{V}{8} - 1\right) + |(E \backslash E')^+|\right)$$

Хотим показать, что $|(E \backslash E')^+| + \frac{V}{8} + \frac{E}{2} < E$.

Покажем, что матожидание $|(E \backslash E')^+|$ – примерно $\frac{V}{8} - 1$.

Пусть бы мы там запустили Краскала. Он бы нашёл то же остовное дерево, оно ведь единственно по предположению. Но вместо разбиения на два множества мы бы просто рандомно разрешали или не разрешали бы ему брать ребро. Он добавит $\frac{V}{8}$ рёбер в итоге. Но в среднем перед каждым просмотренным он ещё в среднем один раз обломается и не возьмёт хорошее ребро (матожидание, тут следовало бы быть построже). Значит, останутся в среднем $\frac{V}{8}$ невзятых рёбер. Они и лежат в $(E \backslash E')^+$. □

Вот, такие дела.