SPb HSE, 1 курс, весна 2024/25
Практика по алгоритмам #10

BST, AVL

9 апреля

Собрано 15 апреля 2025 г. в 14:42

Содержание

BST, AVL

1. **`BST` и $\mathcal{O}(1)$**

   Вспомните, как мы в `BST` за $\mathcal{O}(1)$ делаем `find`/`next`/`prev`/`del`.

   Вспомните, чем `BST` лучше хеш-таблицы?

2. **Быстрая вставка в `BST`**

   Сделайте вставку за $\mathcal{O}(\texttt{lower\_bound}) + \mathcal{O}(1)$.
   Докажите, что нельзя $\mathcal{O}(\texttt{lower\_bound})$ сделать быстрее $\log n$.

3. **Минимум на отрезке**

   Улучшите BST, хранящее пары $\langle x_i, y_i \rangle$: научите его находить $\min y_i \colon L \leqslant x_i \leqslant R$ за $\mathcal{O}(h)$.

4. **Прямой обход**

   Дан прямой обход BST: рекурсивная запись xLR. За $\mathcal{O}(n^2)$ восстановить BST. А за $\mathcal{O}(n)$?

5. **k-min**

   Придумать, как в AVL-дереве за $\mathcal{O}(k)$ возвращать k минимальных ключей?
   Что дополнительно нужно хранить? (есть два решения).

   За сколько работают оба способа в произвольном сбалансированном дереве, в BST?

6. **Операции над AVL**

   a) Удаление за $\mathcal{O}(\log n)$. Распишите, как меняются высоты при вращениях.
   b) Пусть `root.l.h = root.r.h + 3`. Как перебалансировать дерево за $\mathcal{O}(1)$?
   c) Пусть `root.l.h = root.r.h + k`. Как перебалансировать дерево за $\mathcal{O}(k)$?
   d) Merge за $\mathcal{O}(\log n)$. Любой способ. `(*)` Второй способ.
   e) Split за $\mathcal{O}(\log n)$. Без обоснования времени работы.
   f) `(*)` Доказательство времени работы $\mathcal{O}(\log n)$ для Split, полученного выше.
   g) `(*)` Уменьшите количество дополнительной информации до двух бит на вершину.

7. **Сила в split**

   Теперь, когда с нами сила `split` и `merge`, давайте выразим все операции через эти две: `add(x)`, `delete(x)`, `getMin(l,r)`, любой запрос на отрезке (см. задачу 2).

8. **Простые запросы**

   a) Запросы: добавить/удалить пару $\langle x, y \rangle$; посчитать сумму y по всем $\langle x, y \rangle \colon l \leqslant x \leqslant r$.
   b) То же самое и посчитать сумму x по всем $\langle x, y \rangle \colon l \leqslant y \leqslant r$.
   c) Дан массив, посчитать сумму на отрезке с l по r.
   d) Запросы: добавить x; посчитать сумму $x \colon l \leqslant x \leqslant r$; посчитать сумму x, добавленных в моменты времени с l по r.
   e) Предыдущая задача плюс запрос удаления x.

9. **Точка в стакане**

   Дано множество точек на плоскости. Нужно быстро обрабатывать запросы:

   - добавить/удалить точку,
   - вывести любую точку внутри области $d_i \leqslant y \leqslant u_i, x \leqslant r_i$.

10. **k-й элемент**

    Как в BST за $\mathcal{O}(h)$ спуститься к k-му по порядку элементу?

    Как по x найти его порядковый номер в BST?

11. **Повторение**

    Пусть у нас есть массив. Пусть ключ – позиция в массиве. Реализуйте вставку нового элемента на i-ую позицию: $[2, 7, 8, 3] \to [2, 7, \mathbf{4}, 8, 3]$.

12. **Модификации отрезка**

    a) Запросы: `insert(i,x)`, `del(i)`, `get_sum(l,r)`, `set(l,r,value)`.
    b) Добавим запросы `reverse(l,r)` и `rotate(k)`.
    c) `(*)` То же самое, но `add(l,r,value)` по модулю 5, а `get_sum` по-прежнему без модуля.

13. `(*)` **Монотонные пути**

    Дан взвешенный орграф. Найти самый **длинный** по суммарному весу рёбер путь, на котором рёбра строго возрастают и по номеру, и по весу.

14. `(**)` **Вставка ключевых значений**

    Изначально все ячейки пусты. Нужно обрабатывать запросы вида `Insert(i, x)`. При этом, если i-я ячейка занята, она и все прилегающие справа занятые ячейки сдвигаются вправо. Пример: $(5, 1); (5, 2); (5, 3); (1, 7); (1, 8); (2, 9) \to [8, 9, 7, 0, 3, 2, 1]$.

Разбор задач практики

1. **BST и $\mathcal{O}(1)$**

   `find`: hashMap из `int` в `node*`.

   `next`/`prev`: внутри `node*` храним ссылку.

   `del`: `find` + `next` + $\mathcal{O}(1)$.

   Тем что там есть порядок (`next`/`prev`) и `lowerBound`. А хуже тем, что в BST вставка за $\log n$.

2. **Быстрая вставка в BST**

   Вставляем x. Нашли v – `lower_bound(x)`. Если нет левого сына, вешаем туда x.
   Если есть, делаем x правым ребенком $u = \texttt{prev}(v)$ за $\mathcal{O}(1)$.
   Раз v – `lower_bound(x)`, всё его левое поддерево $< x$, x занял там максимальную позицию.
   В поддереве v всё ок. У предков v, у которых v слева, v меньше их, $x \leqslant v$.
   Предки v, у которых v справа, $< x$, поскольку v – `lower_bound`.
   P.S. Ещё можно вставить x в середину ребра между v и `v->left`, но так глубина больше.

   $Time(\texttt{sort}) = Time(\texttt{lower\_bound}) \cdot n + n \Rightarrow$ нет `lower_bound`-а быстрее $\log n$.

3. **Минимум на отрезке**

   Храним в каждой вершине $\min y$ в её поддереве.
   Также храним диапазон ключей, то есть, min и $\max x$ в поддереве.

```
int get(node* t, int L, int R):
    if (!t) return INF;
    if (R < t->min_key || t->max_key < L) // в поддереве t нет x из [L, R]
        return INF;
    if (L <= t->min_key && t->max_key <= R) // в поддереве t все x из [L, R]
        return t->min_y;
    return min(get(t->l, L, R), get(t->r, L, R), (L<=t->x<=R ? t->y : INF));
```

   На каждом уровне ветвимся только в двух вершинах, которые пересекают $[L, R]$.
   $\Rightarrow$ на каждом уровне смотрим только 4 вершины $\Rightarrow \mathcal{O}(h)$.

   Более хитрый подход: не хранить $[\texttt{min}, \texttt{max}]$ явно, а вычислять границы ключей в вершине в процессе рекурсии. Если для вершины с ключом x знали границы $[A, B]$ и перешли направо, то теперь границы станут $[x + 1, B]$. В корне границы равны $[-\infty, +\infty]$.

4. **Прямой обход**

   Алгоритм. Идем по массиву. Поддерживаем стек. Видим x, снимаем со стека всё, что $< x$. Делаем x правым ребенком последней снятой. Если ничего не сняли, делаем x левым ребенком вершины стека. Кладем x в стек.

   Время. $\mathcal{O}(n)$, каждое значение один раз положили и один раз сняли.

   Смысл. В стеке лежат все вершины, у которых еще не началось правое поддерево. Когда встречаем x, то у всех $> x$ ещё не кончилось левое поддерево, а у всех $< x$ кончилось.

5. **k-min**

   В произвольном BST можно прошивкой – поддерживать `next`, `prev` для всех вершин.

   Можно «взять минимум за $\mathcal{O}(1)$, от него $k-1$ раз перейти к следующему элементу проходом по дереву». Хранить отцов. Работает в общем случае за $\mathcal{O}(k + \log n)$, в AVL за $\mathcal{O}(k)$.

Доказательство времени $\mathcal{O}(k)$. Рассмотрим v – самую высокую вершину, пройденную алгоритмом. Когда мы поднимаемся вверх, мы всегда выписываем вершину, как очередной минимум, а вот спуститься вниз мы можем «вхолостую», выписать только самую нижнюю вершину $\Rightarrow$ время работы $= k + x$, где x – длина пути вниз от v до k-го минимума. Осталось заметить, что $x \leqslant h(v.r) \leqslant h(v.l) + 1$, а $h(v.l) \leqslant size(v.l) < k$, так как все они меньше v.

6. **Операции над AVL**

a) **Удаление.**

Сперва как в обычном BS: если нет правого сына, вешаем левого на место v, иначе сперва $\texttt{swap}(v, \texttt{next}(v))$, затем удаляем $\texttt{next}(v)$. Чтобы поддержать AVL-свойство, откатимся по предкам от удалённой вершины и перебалансируем их.

Пусть у вершины уменьшилась глубина $\Rightarrow$ после перебалансировки либо высота предка не изменится, тогда конец, либо уменьшится на один, тогда нужна перебалансировка выше. И далее до корня. В худшем случае вращений понадобится $\Omega(\log n)$.

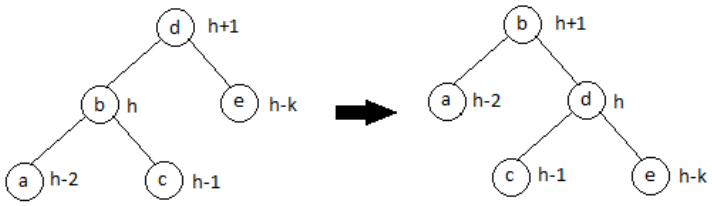
c) **Перебалансировка при дисбалансе** k

Индукция по k. По индукции для всех $i < k$ знаем, что за $\mathcal{O}(i)$ умеем исправить дисбаланс, и глубина корня или не изменится, или уменьшится на 1.

Пусть у нашей вершины d высота $h{+}1$, у левого ребенка b высота h, у правого e равна $h{-}k$. Дети левого ребенка – a и c. Делаем малое вращение вправо. Смотри картинку. Варианты:

1. $a.h = h - 2, c.h = h - 1 \Rightarrow$ высота корня не поменялась, в d дисбаланс $k{-}1$
2. $a.h = h - 1, c.h = h - 1 \Rightarrow$ высота корня не поменялась, в d дисбаланс $k{-}1$
3. $a.h = h - 1, c.h = h - 2 \Rightarrow$ высота корня уменьшилась на 1, в d дисбаланс $k{-}2$

В 1-м варианте в конце или высота d уменьшилась, или нужно ещё одно вращение для b.
Во 2-м если в конце если высота d уменьшилась, то и высота b уменьшилась.
В 3-м в конце если высота d уменьшилась, то высота b уже не уменьшится.



d) **Merge за** $\mathcal{O}(\log n)$

Способ #1. Удалим из L дерева max за $\mathcal{O}(\log n)$. Сделаем его корнем, подвесим к нему наши два дерева, перебалансируем за $\mathcal{O}(\log n)$, используя предыдущий пункт.

Способ #2. Смотри код.

```
1 node* Merge(node* l, node* r):
2     if (!l || !r) return l ? l : r;
3     if (l->h <= r->h):
4         r->l = Merge(l, r->l), Rebalance(r);
5         return r;
6     else:
7         l->r = Merge(l->r, r), Rebalance(l);
8         return l;
```

Можно показать по индукции `(*)`, что у `Merge(l, r)` высота $\leqslant \max(\texttt{l.h}, \texttt{r.h}) + 1$.

$\Rightarrow$ на каждом уровне рекурсии дисбаланс $\leqslant 3$ и `Rebalance` работает за $\mathcal{O}(1)$.
$\Rightarrow$ время равно глубине рекурсии, $\mathcal{O}(\log n)$.

P.S. Доказать `(*)` в случае `l.h` $=$ `r.h` не так уж просто.
Сложность только техническая, доказывается большим разбором случаев.

P.P.S. 3 достигается.

f) **Split за $\mathcal{O}(\log n)$**

Пусть `t->x` $< x$, тогда посплитим рекурсивно `t->r`.
Левую часть результата запишем обратно в `t->r`. Избавимся от дисбаланса в `t`.

```
TreePair Split(node* t, int x):
    if (!t) return {0, 0};
    if (t->x < x):
        TreePair p = Split(t->r, x);
        t->r = p.first; Rebalance(t);
        return {t, p.second};
    else:
        // симметричный случай
```

Каждый `Rebalance` работает $\mathcal{O}(\log n) \Rightarrow$ время работы `Split` $\mathcal{O}(\log^2 n)$.

На самом деле все `Rebalance` кроме одного отработают за $\mathcal{O}(1)$. Доказательство: после первого долгого `Rebalance` и далее (рисуем картинку), оба дерева-результата имеют высоту на $\mathcal{O}(1)$ отличающуюся от исходного.

g) **`(*)` Два бита на вершину**

В вершине вместо высоты поддерева храним разницу $l.h - r.h$. Это число из $\{-1, 0, 1\}$. Надо узнать, как $l.h - r.h$ изменилась после рекурсивной обработки ребёнка (и таким образом узнать, есть ли дисбаланс). Из рекурсивных функций, меняющих дерево с корнем t, дополнительно вернём, насколько изменилась высота t (опять же $\{-1, 0, 1\}$).

Как пересчитывать разности при поворотах? Вычислим *относительную* высоту всех узлов, участвующих в повороте, относительно корня поддерева. Повернём, получим относительные высоты после поворота, возьмём разности братьев.

7. **Сила в split**

`add(x)` $=$ `split(x)` $+$ `merge`

`del(x)` $=$ `split(x-1)` $+$ `split(x)` $+$ `merge`

`getMin(l,r)` $=$ `split(r)` $+$ `split(l-1)` $+ answer +$ `merge`

Ответе на запрос на отрезке: высплетить отрезок, ответ лежит в корне, вмерджить обратно.

8. **Простые запросы**

a) **Добавить/удалить $\langle x, y\rangle$; $\sum y \colon \langle x, y\rangle \colon l \leqslant x \leqslant r$.**

Дерево по ключу x с дополнительным полем y. В каждом узле храним сумму y, min и max x в поддереве, при изменении детей эти величины надо пересчитывать.

```
int count(node* t, int l, int r):
    if (!t) return 0;
    if (r < t->min_key || t->max_key < l) // в поддереве t нет x из [L, R]
        return 0;
    if (l <= t->min_key && t->max_key <= r) // в поддереве t все x из [L, R]
        return t->sum;
    return count(t->l, l, r) + count(t->r, l, r) + (l<=x && x<=r ? t->y : 0)
```

b) **Дан массив, посчитать сумму на отрезке с l по r.**

Ключ BST – позиция в массиве.

c) **Добавить $\langle x, y\rangle$; $\sum y\colon \langle x, y\rangle\colon l \leqslant x \leqslant r$; $\sum x\colon \langle x, y\rangle\colon l \leqslant y \leqslant r$.**

Два дерева: по ключу x с дополнительным полем y, и по ключу y с полем x.

de) **Добавить/удалить x; $\sum x\colon l \leqslant x \leqslant r$; $\sum x$, добавленных с l по r момент времени.**

Можно завести два дерева, как в (b), $y =$ времени добавления.
Если нет удаления, вместо второго дерева будем хранить массив префиксных сумм.

9. **Точка в стакане**

BST по y. В каждой вершине храним точку с минимальным x в поддереве.
Найдём точку с min x на отрезке $d_i \leqslant y \leqslant u_i$. Если она не в стакане, то стакан пуст.

10. **k-й элемент**

По k найти k-ый элемент дерева:
храним размеры поддеревьев; пересчитываем при `add`/`del`; если `v->L->size` $\geqslant k$, идём влево.

По x вернуть позицию в дереве:
размеры уже есть $\Rightarrow$ просто спускаемся; при повороте направо прибавляем `v->L->size+1`.

11. **Повторение**

Спускаемся к i-й позиции, вставляем рядом.
Ко всем позициям, которые правее (больше), нужно сделать `+=1`. На самом деле ключ-позицию можно не хранить, а восстанавливать, зная размеры поддеревьев, тогда дерево будет называться «деревом по неявному ключу».

12. **Модификации отрезка**

a) `add(i,x)`, `del(i)`, `get_sum(l,r)`, `set(l,r,value)`.

Дерево по неявному ключу на нашем массиве. В каждой вершине храним:

- `size` – размер поддерева (для неявного ключа),
- `sum` – сумма в поддереве,
- `pending` – число, которое мы лениво хотим присвоить в поддерево.
- `time` – отметка времени, когда присвоили `pending`.

Каждый раз, когда заходим в вершину, делаем `push` – проталкиваем `pending` вниз, ставя вершине корректный `sum`.
Каждый раз, когда меняем вершину, делаем `update` – пересчитываем `size` и `sum`.
Код приводится для случая, когда определен пустой `node* null`.

```
void push(node* t):
    if (t == null) return;
    for (node* ch : {t->l, t->r})
        if (ch->time < t->time)
            ch->pending += t->pending, ch->time = t->time;
    t->sum = t->size * t->pending;

void update(node* t):
    if (t == null) return;
    push(t->l), push(t->r);
    t->sum = t->x + t->l->sum + t->r->sum;
    t->size = 1 + t->l->size + t->r->size;
```

- `insert` и `del` делаются обычным образом, но с добавлением вызовов `push` и `update`.
- `get_sum(l, r)` как в прошлой задаче. Либо, если есть `Split` и `Merge`, вырезанием нужного куска, взятием `sum` корня и обратным слиянием.
- `set(l,r,value)` аналогично `get_sum`, но вместо возвращения `sum` происходит `pending = value, time = timer++`

b) `reverse(l,r)`, `rotate(k)`
Храним `rev` – нужно ли разворачивать поддерево. Лениво проталкиваем (как и `pending`).

```
def push(t):
    if t.rev == 1:
        t.l.rev ^= 1, t.r.rev ^= 1, t.rev = 0
        swap(t.l, t.r)
```

`rotate(k) = split(k) + merge` в обратном порядке.
`rotate(k) = reverse(0, n-k) + reverse(n-k, n) + reverse(0, n)`

c) `(*)` **Прибавление по модулю 5, сумма без модуля**
Как раньше, но вместо `sum` в вершине храним массив `count[5]`, в `count[i]` записано, сколько раз в поддереве встречается i. В `push` теперь будем циклически сдвигать `count` и пересчитывать `sum` через него.

13. `(*)` **Монотонные пути**

Добавляем рёбра в пустой граф в порядке возрастания номера. Новое ребро $e\colon a \to b$ веса w_e можно дописать в конец пути, заканчивающемся в a, если предыдущее ребро пути было меньше w_e. У каждого пути есть два параметра — длина d и вес последнего ребра x. Среди всех путей, заканчивающихся в a и с $x < w_e$ нужен максимальный по d. Если есть два пути (x_1, d_1) и $(x_2, d_2)\colon x_1 \geqslant x_2$ и $d_1 \leqslant d_2$, то первый хуже по обоим параметрам и не нужен.

$\Rightarrow \forall v$ поддерживаем массив пар $\langle x_1, d_1\rangle, \langle x_2, d_2\rangle, \ldots : x_i < x_{i+1},\ d_i < d_{i+1}$ – параметры путей, кончающихся в v. Храним эти пары, конечно, в дереве поиска $tree[v]$.
Заметим, спускаться по такому дереву можно как по x, так и по d.

Добавление ребра $e\colon a \to b$ веса w_e:
1. `<d,x> = prev(lower_bound(tree[a],`w_e`))`,
2. Выкинуть из `tree[b]` бесполезные из-за `<d+`w_e`,`w_e`>` пары,
3. `tree[b].add(<d+`w_e`,`w_e`>)`.

Пар всего m, каждая один раз пришла и один раз ушла.
Обработка ребра за амортизированное $\mathcal{O}(\log n)$. Итого время $\mathcal{O}(m \log n)$.

Домашнее задание

3.1. Обязательная часть

1. **(2) Площадь покрывающего прямоугольника**

   Придумать структуру данных, которая поддерживает множество точек $S \subseteq \mathbb{N}^2$ со следующими операциями, каждая за $\mathcal{O}(\log n)$, где $|S| = n$:

   - добавить/удалить точку (x, y);
   - найти min площадь прямоугольника со сторонами, параллельными осям координат, покрывающего все такие точки из S, что $x \in [l, r]$.

2. **(3) Глубины вершин в несбалансированном дереве**

   Рассмотрим процесс добавления ключей в обычное несбалансированное BST.
   Одна операция добавления может работать за $\Omega(n)$.
   В процессе мы поддерживаем `struct Node { Node *l, *r, *p; int depth, key; }`.
   Научитесь моделировать этот процесс за $\mathcal{O}(\log n)$ на запрос добавления.

3. **(3) Excel**

   Есть excel-табличка. Состоящая из $n \times n$ изначально пустых ячеек ($n \leqslant 10^5$). Научитесь за $\mathcal{O}(\log n)$ обрабатывать запросы

   a) Столбец j подвинуть влево-вправо на d.
   b) Строку i подвинуть вверх-вниз на d.
   c) Поменять/прочитать ячейку $[i, j]$.

   Hint: всё описанное делается обычным одномерным деревом по неявному ключу.

4. **(2) k минимальных на отрезке**

   Пусть дано произвольное сбалансированное BST по ключу x, хранящее пары $\langle x, y \rangle$.
   Можно сохранить в вершинах дерева дополнительную информацию, если её можно пересчитывать через детей. Находить k минимальных y на отрезке $l \leqslant x \leqslant r$ за $\mathcal{O}(k \log n)$.

   a) **(1)** В вершине можно хранить сколько угодно информации.
   b) **(1)** В вершине можно хранить $\mathcal{O}(1)$ информации.

3.2. Дополнительная часть

1. **(2) Изменений в AVL мало**

   Начинаем с пустого AVL дерева, делаем n операций `add`. Докажите, что высота вершин будет меняться суммарное всего лишь $\mathcal{O}(n)$ раз. Иными словами, одна операция `add` делает амортизированное $\mathcal{O}(1)$ операций записи в память.

2. **(2) k минимальных на отрезке нельзя извлекать быстро**

   Пусть дано произвольное Balanced Search Tree по ключу x, хранящее пары $\langle x, y \rangle$.
   Пусть так же это дерево уже построено за время $o(n \log n)$ (например, пары были исходно упорядочены по x). Можно сохранить в вершинах дерева дополнительную информацию, если её можно пересчитывать через детей. Докажите, что находить на отрезке $l \leqslant x \leqslant r$ k минимальных y и удалять их **нельзя** за $\mathcal{O}(k + \log n)$.

3. **(4) Недо-AVL-дерево**

   Постройте тест, на котором недо-AVL-дерево, которое делает только малые вращения (см. код), делает $\omega(n \log n)$ операций после n запросов `add`. Либо докажите, что такого теста нет.

   Формально: изначально дерево пусто, нужно n раз вызвать add(root, x_i) для некоторой последовательности x_i, что суммарное время работы $\omega(n \log n)$.

```
1  void add(node* &t, int x):
2      if (t == node::null)
3          t = new node(x);
4      else if (t->x == x)
5          return;
6      else if (x < t->x):
7          add(t->l, x);
8          if (t->l->h > t->r->h + 1)
9              t = rot_left(t);
10     else:
11         add(t->r, x);
12         if (t->r->h > t->l->h + 1)
13             t = rot_right(t);
14     t->calc();
```