Конспект по алгоритмам и структурам данных
II семестр

Коченюк Анатолий

4 июля 2021 г.

Глава 1

Новые структуры данных

1.1 Дерево отрезков

Segment Tree, Range Tree, Interval Tree – можно наткнутся на другие структуры

Есть массив a на n элементов

1. set(i,v) $\quad a[i] = v$
2. sum(l,r) $\quad \sum_{i=l}^{r-1} a[i]$

$[3, 5, 2, 1, 6, 4, 8, 3]$

Построим дерево, где в каждом узле сумма двух под ним. Каждое число это сумма чисел на каком–то отрезке.

Заменим 4 на 7. set(5, 7). Нужно пересчитать все узлы, которые выше него. Их логарифм.

Теперь к сумме. Сумму от l до r мы будем раскладывать на суммы, которые мы уже знаем.

1. Как найти эти суммы: обойдём наше дерево рекурсивно, не заходя в плохие поддереве. Идём вниз: если в поддереве нет нужных элементов, выходим, если полностью содержится в нужном, берём эту сумму, иначе идём дальше вниз.
2. Мы обошли не так много (порядка логарифма) узлов. Посмотрим на узлы, в которых не сработало отсечение. Тогда наш отрезок содержит одну из границ отрезка. Отрезков, которые содержат границу (правую или левую) не более $2 \log n$.

Запуск рекурсии: узлов, в которых не сработало отсечение – $2 \log n$, всего узлов $\approx 4 \log n$.

Будем хранить полное двоичное дерево. У узла i дети $2i + 1$ и $2i + 2$

```
set(i, v, x, lx, rx):
    if rx - lx = 1:
        tree[x] = v
    else:
        m = (lx+rx)/2
        if i < m:
            set(i,v,2x+1,lx,m)
        else:
            set(i,v,2x+2,m, rx)
        tree[x] = tree[2x+1] + tree[2x+2]

sum(l, r, x, lx, rx):
    if l >= rx || lx >= r:
        return 0
    if lx >= l && rx <= r:
        return tree[x]
    m = (lx+rx)/2
    s1 = sum(l, r, 2x+1, lx, m)
    s2 = sum(l, r, 2x+2, m, rx)
    return s1 + s2
```

Пусть мы теперь хотим посчитать минимум. Делаем то же самое, только в каждом узле храним не сумму на отрезке, а минимум. Только в случае пустого дерева вместо 0 в сумме надо вернуть $+\infty$ (нейтральный элемент по операции)

Если нужно сделать операцию $a \bigotimes b$. Если у неё есть ассоциативность $(a \otimes b) \otimes c = a \otimes (b \otimes c)$, то её можно встроить в дерево отрезков.

$max, +, \cdot, \&, |,$ НОД, НОК

1.1.1 Снизу вверх

Занумеруем также, но будем идти снизу вверх и без рекурсии.

```
set(i, v): // n = 2^k
    x = i + n - 1
    delta = v - tree[x]
    while x >= 0:
        tree[x]+=delta
        x = (x+1)/2-1

sum(l, r):
```

```
l = n-1+r
r = n-2+r
res = 0
while r >= l:
    if l % 2 = 0:
        res += tree[l]
    l = l / 2
    if r % 2 = 1:
        res += tree[r]
    r = r/2-1
return res
```

Если нужно посчитать другую функцию, поменяется нейтральный элемент. Если не коммутативная функцию, можно сначала считать левую, потом правую, а потом их сложить.

Такая реализация чуть лучше работает как $o(\log(l-r))$. Каждый раз разница между l и r уменьшается в 2 раза.

1.2 Персистентное дерево отрезков

```
   15
  8   7
 3 5 6 1
```

set(2,8) Сделаем новый узел рядом с 6. Рядом с узлом 7 на верхнем слое приделаем ещё один узел 9. А к корню приделаем узел 17. Так у нас появилось две параллельные версия дерева отрезков.

1.3 Дерево отрезков v.2

Теперь мы хотим:

1. $add(l, r, v) \quad a[i] += v \quad i = l \dots r-1$
2. $get(i) \quad return\ a[i]$

Построим дерево отрезков над массивом как в прошлый раз. В начале везде запишем нолики. Хотим добавить 3 к отрезку. Разобъём его на кусочки и в верхние узлы над нужным отрезком запишем тройки.

```
add(l, r, v, x, lx, rx):
    if (lx >= r) || (l >= rx):
        return
    if lx >= l && rx <= r:
        tree[x] += v
        return
    m = (lx+rx)/2
```

```
    add(l, r, v, 2x+1, lx, m)
    add(l, r, v, 2x+2, m, rx)
```

Сделаем $modify(l, r, v)$ $a[i] = a[i]\bigstar v$, где $\bigstar$ ассоциативно и коммутативно.

Как жить с некоммутативными операциями? Записывать порядок. Точнее при получении значения пропихиваем верхние значения вниз справа.

```
propogate(x):
    tree[2x+1]+=tree[x]
    tree[2x+2]+=tree[x]
    tree[x] = 0
```

Пример (Присваивание)**.** set – задаёт значение на отрезке. код такой же

```
propogate(x):
    if tree[x] != NO_OPERATION
```

Теперь сделаем две операции: прибавление и минимум, оба на отрезке.

Сначала сделаем дерево на минимум. Дальше при добавлении находим нужные отрезки и запоминаем, что нужно добавить 3 в узлах. минимумы снизу мы пересчитывать не будем

Когда доходим до узла добавляем v в два массива: добавочный и минимумов + обвнояляем минимум, при спуске: $tm[x] = min(tm[2x+1], tm[2x+2]) + ta[x]$

```
propogate

min(l, r, x, lx, rx):
    ...
    ...
    ...
    s1 = min(l, r, 2x+1, lx, m)
    s2 = min(l, r, 2x+2, m, rx)
    return min(s1,s2) + ta[x]
```

Здесь мы пользовались дистрибутивностью при пересчёте. На самом деле, можно для недистрибутивных (например двух плюсов) сделать её такой, или помнить как именно нужно изменить

1.4 Дерево Фенвика

Задача 1. Есть массив

Нужно:

1. inc(i, v) $a[i] = +v$
2. sum(l,r) $\sum_{i=l}^{r-1} a[i]$

Оба за логарифм

Казалось бы всё это умеет дерево отрезков.. Но можно лучше! (всё ещё за логарифм, но)

$f(i) = \sum_{j=p(i)}^{i} a[j]$

$sum(l, r) = sum(l) - sum(r)$

Как считается $sum(i)$: В последнем элементе лежит какая-то сумма. Добавим её к ответу. В элементе до этой суммы (префикса) лежит какая-то сумма. Добавим её к ответу... Так, пока не дойдём до начала массива.

inc(i, v): ищем все суммы, содержащие i и увеличиваем их на v

По итогу нам нужна хорошая $p(x)$, чтобы и отрезков в $sum(i)$ было поменьше (логарифм), и отрезков, содержащих i было тоже логариф.

$x: \quad \max k : (x+1) \vdots 2^k \quad p(x) = x + 1 - 2^k$ (В терминах двоичной записи все единички идущие подряд в конце числа становятся нулями.

$p(x)$ довольно просто найти, это $x\&(x+1)$

```
sum(l,r):
    return sum(r) - sum(l)

sum(r):
    x = r - 1
    res = 0
    while x >= 0:
        res += f[x]
        x = (x&(x+1))-1
    return res
```

Для инкремента нужно найти все такие $j \quad p(j) \leqslant i \leqslant j$

j – префикс, 0 и последовательность единиц. $p(j)$ – префикс, 0 и последовательность нулей.

Тогда i, между ними, – префикс, ноль и что-угодно. Соответственно такие j из i можно получать заменяя последние n бит числа i на единицы.

```
inc(i, v):
    j = i
    while (j < n):
        f[j] += v
        j = j | (j+1)
```

1.5 Разреженная таблица

Хотим. Структуру. Данных.

Задан массив и он не меняется. Хотим посчитать минимум на любом отрезке за 1.

Можем предпосчитать за квадрат на всех отрезках и просто выдавать, но это долго.

$m[i,j] = \min\left(a\left[i \dots i + 2^j - 1\right]\right) \qquad i = 0 \dots n-1 \quad j = 0 \dots \log n$

```
for i = 0 .. n m[i,0] = a[i]
for j = 1 .. log n
    for i = 0 .. n-2^j
       m[i,j] = min(m[i,j-1], m[i+2^(j-1), j-1])
```

Теперь собственно как искать минимум:

$d = r - l \quad \max k : 2^k \leqslant d$

$res = \min\left(m[l,k], m[r-2^k, k]\right)$

Нужно искать к-шки за O(1). Вообще с помощью *битовых извращений* можно найти за $\log\log$, а с помощью *страшных битовых извращений* вообще за $O(1)$, но мы пока не будем выпендриваться и просто предпосчитаем к-шки для всех d

Ура! всё работает. Но вся эта схема сильно использует тот факт, что мы считаем минимум (использует свойство идемпотентности $min(x,x) = x$). Таких функций не прям чтоб много, а хотелось бы и с другими аналогичные вещи делать.

Давайте поделим отрезок пополам и посчитаем префиксные суммы справа и слева от середины. Тогда большой отрезок в запросе можно разбить на два: до и после середины, сложить (применить функцию) и получить ответ.

Проблема: если отрезок в одной половине, то его в общем случае не посчитать. Решение: поделим половины отрезков так же, как делили целый. В итоге мы найдём $2n$ сумм. n на всё отрезке и по $\frac{n}{2}$ на половинах. С каждым таким делением добавляется n. Всего делить можем $\log n$ раз, значит сумм всего будет $n \log n$

Отвечаем на запрос: находим границу, которую отрезок от l до r пересекает (первый бит числа $l\&r$)

1.6 Чё можно делать в двумерном случае

Задача 2. Есть прямоугольнички, они пересекаются. Ещё есть набор точек. Для каждой точки надо найти сколько прямоугольников покрывает эту точку

Решение. Метод заметающей прямой. Поставим прямую и будем двигать её направо. Когда встретим точку, ответим на запрос про неё.

Для одной линии считаем сколько покрывает каждую точку.

Выделим все у кординаты, в которых есть вершина прямоугольника.

Встретили границу, делаем +=1 или -=1

■

Задача 3. Есть прямоугольники. Надо найти площадь их объединения

Структура:

1. $a[i] \pm 1$ на отрезка
2. $\sum len(i) : a[i] > 0$

Дерево отрезков:

1. sum(i,v)
2. sum(l,r)

Прямой аналог:

1. set(i,j,v) a[i,j] = v
2. sum(l,r,t,b)

Сделаем дерево отрзков для строчек. Каждая вершина отвечает за полосу из строчек. В каждой вершине хранится дерево отрезков по столбцам.

1.7 Дерево Фенвика 2D

$$f(i) = \sum_{j=p(i)}^{i} a[j]$$

$$f(i,j) = \sum_{x=p(i)}^{i} \sum_{y=p(j)}^{j} a[x,y]$$

План: сделать всё точно также

```
x = i-1
while x >= 0:
    rest += f(x)
    x = x&(x+1)-1

x = i
while x >= 0:
    y = j
    while y >= 0:
        rest += f(x,y)
        y = (y+1)&y-1
    x = (x+1)&x-1
```

1.8 Разреженные таблицы 2D

$m[i,j] = min\left(a[i, i + a^j - 1]\right)$

$m[i_1, j_1, i_2, j_2] = min\left(a[i_1, i_1 + 2^{j_1} - 1, i_2, i_2 + 2^{j_2} - 1\right)$

precalc $O(n^2 \lg^2)$

Запрос: $O(1)$

Задача 4. Есть двумерные точки. Хотим вывести все точки в прямоугольнике.

Спроектируем точки на одну координату и построим на них дерево отрезков. В вершинах сортированные по другой координате точке.

1.9 Бинпоиск.каскадирование

Делаем один и тот же бинпоиск на многих массивах. – $O\left(n \log m\right)$

Если знать в нашей последней задаче где бинпоиск в корне, можно легко понять как будет в рекурсии, элементы только пропадают и позиции однозначно соответствуют

Частичное каскадирование, Fractional Cascading

Чтобы не появлялось много лишних элементов, запихаем массива наверх. Чтобы не было слишком много, запихаем половину (каждый второй).

При переходе смотрим на ближайшие запихнутые элементы. X между ними, проверем две позиции.

А теперь у нас много элементов.

1.10 Дерево поиска

set, map. Что умеет и то, и то? Хэш-таблица

add, remove, contains, put(k,v), get(k)

Хотим иметь линейный порядок. ($x < y$)

Выполняется свойство: Все элементы в левом поддереве элементы меньше узла. Справа – больше.

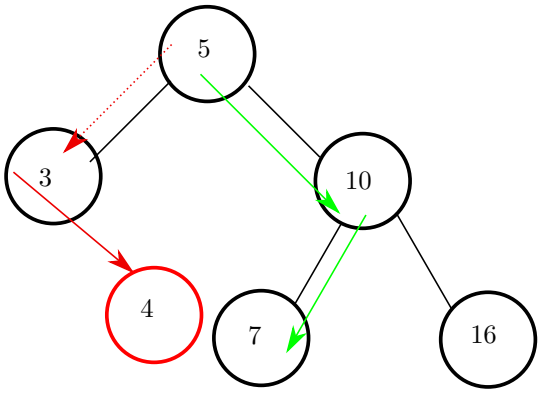

Рис. 1.1: $contains(7) \quad add(4)$

h – высота дерева. за $O(h)$ можем получать и проверять на наличие

Поищем что-нибудь. Например lower bound – $\min y \geqslant x \quad y \in Tree$

Стартуем в корне. Если элемент меньше x, идём вправо. Иначе запоминаем и идём влево поискать что-нибудь получше.

Если дерево – бамбук, то его высота n. Оптимальное дерево – бинарное, чтобы $\log n$ было (меньше не сделать)

1.11 АВЛ-дерево

H_1, H_2 – высоты поддеревьев (правого и левого) $|H_1 - H_2| \leqslant 1$

$H \leqslant c \cdot \lg n$

$n \geqslant 2^{\frac{H}{c}} = \underbrace{\left(2^{\frac{1}{c}}\right)}_{\alpha}{}^{H}$

$n \geqslant \alpha^H$

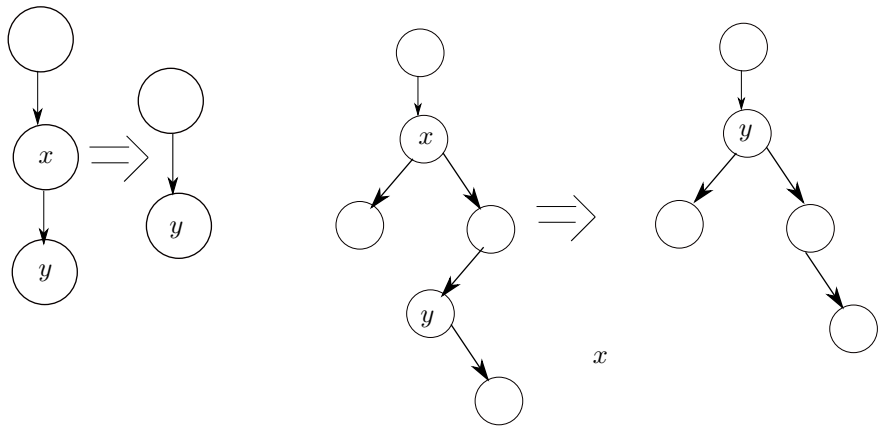

Рис. 1.2: removing (дерево поиска)

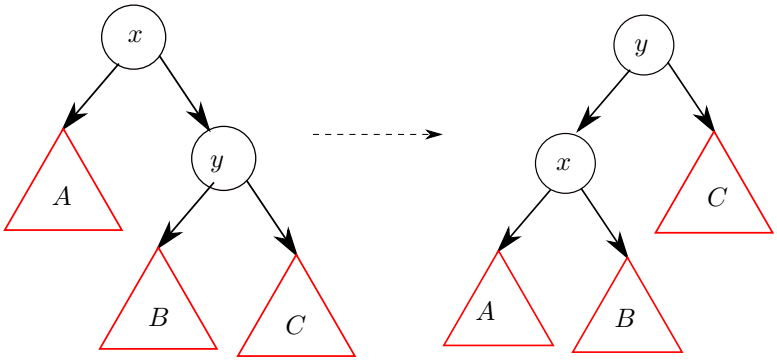

Рис. 1.3: rotation

$F(H) = \min$ число вершин в дереве высоты H

$F(H) = 1 + F(H-1) + F(H-2) > 2F(H-2)$

$F(H) \geqslant \frac{\sqrt{2}^H}{2}$

Утверждение 1. Если добавить один элемент в дерево, высота любого дерева изменится максимум на 1

Было $(H, H+1)$ стало $(H, H+2)$. Сломалось АВЛ-дерево

Сделаем операцию балансировки.

Определение 1 (RR-поворот)**.** Новый элемент добавился в правое поддерево правого поддерева. Крутим ребро xy

LL – аналогично.

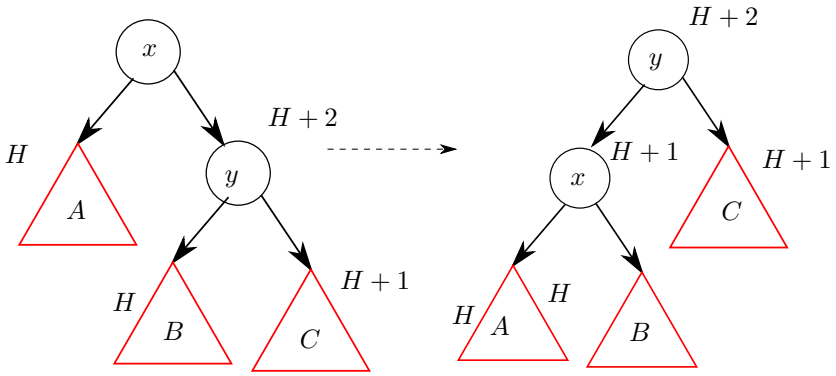

Рис. 1.4: RR

Определение 2 (RL)**.** LR также

```
1     class Node{
2        Node left, right;
3        int key;
4     }
5
6     rotate(x, y, p)
7        if (y == x.right):
8            x.right = y.left
9            y.left = x
10       else:
11           x.left = y.right
12           y.right = x
13       x.recalc()
14       y.recalc()
15
16    recalc():
17       H = max(left.H, right.H)+1
18
19    Node add(Node x, Node nw)
20       if (x == null)
```

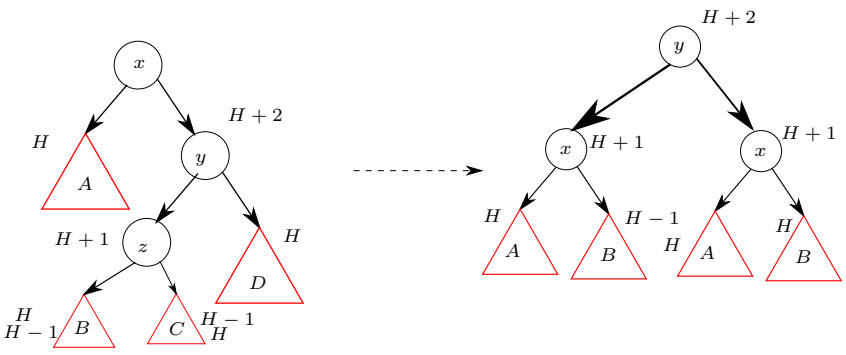

Рис. 1.5: RL

```
21              return nw
22          if nw.key > x.key
23              x.right = add(x.right, nw)
24              x.recalc()
25              y = x.right
26              n = x.left.H
27              if y.right == h+1:
28                  rotate(x,y)
29                  return y
30              else if y.leftH = h+1:
31                  z = y.left
32                  rotate(y,z)
33                  rotate(x,z)
34                  return z
35          else:
36              x.left = add(x.left, nw)
37              x.recalc()
38              y = x.left
39              h = x.right.H
40              if y.left.H = h+1
41                  rotate(x,y)
42                  return y
43              else if y.right.H == h+1
44                  z = y.right
45                  rotate(y,z)
46                  rotate(x,z)
47                  return z
48
49
```

Утверждение 2. А это всё тоже персистентным можно делать

Можно всякие функции считать на отрезке ключей.

1.12 Декартово Дерево

1. split(x). Из дерева делает два дерева, одно в котором все элементы $< x$, второе – $\geqslant x$
2. merge(T1, T2). Берёт два дерево, которые соответственно $< x$ и $\geqslant x$ и склеивает их.

За $\log n$ научимся делать обе эти операции.

Итак, в каждом узле нашего дерева будет находится два ключа: x, y и выполняется свойство: В правом поддереве: $(< x; < y)$, в левом – $(> x; < y)$

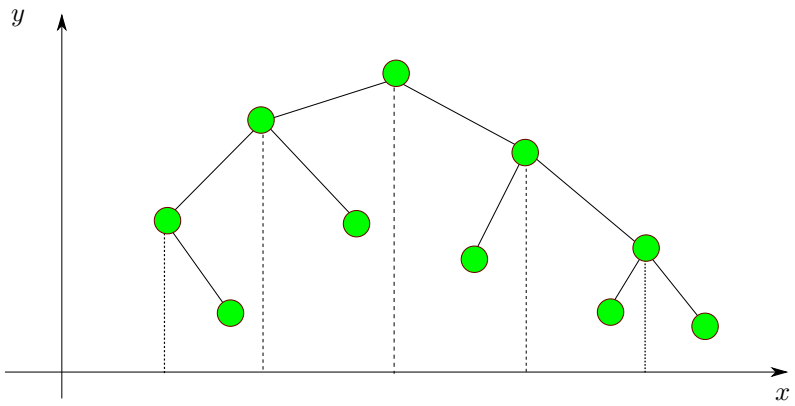

Рис. 1.6: dektree

Если посмотреть на дерево x, получится нормальное дерево поиска. Если посмотреть на ключи y, мы получим кучу.

В английском это всё называется Treap = Tree + Heap

$y = rand\,()\quad E(n) = O\,(\log n)$

Как строить по набору точек на плоскости:

1. Находим точку с наибольшим y, она, очевидно, корень.
2. Дальше всё, что правее неё, лежит в правом поддереве, всё, что левее – в левом. Соответственно мы рекурскивно находим корни в этих поддеревьях и соединяем их с корнем на предыдущем шаге.

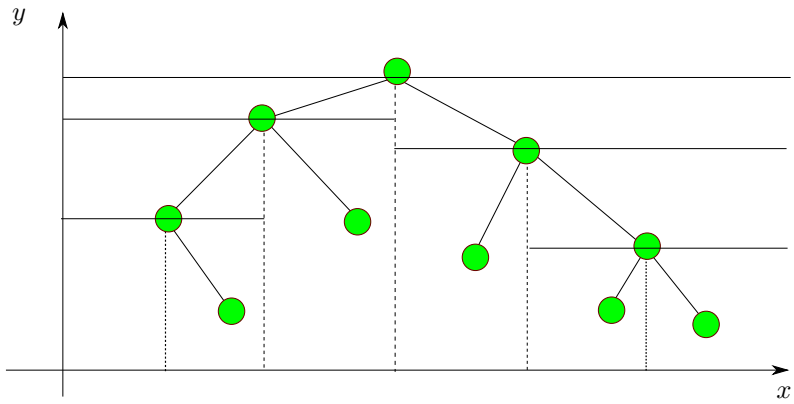

Рис. 1.7: treapbuilding

Эта процедура очень похожа на quick sort, по той же причине эта штука делается за логарифм.

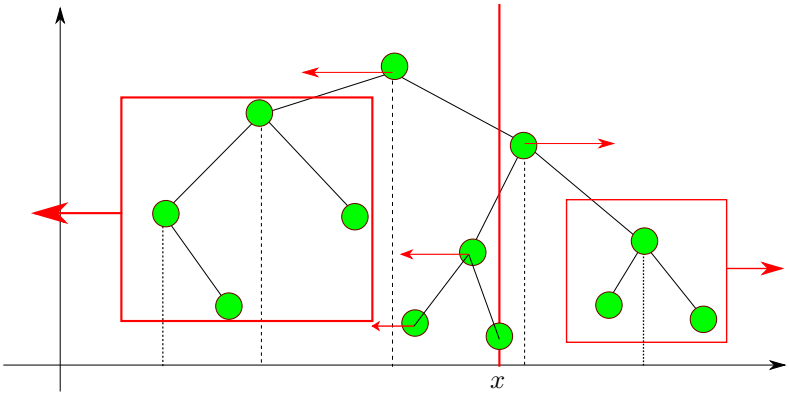

Рис. 1.8: split

Split(node, x):

1. смотрим с какой стороны стоит корень. Так мы можем сразу сделать вывод об одном из поддеревьев
2. Запускаемся рекурсивно от второго поддерева

3. Возвращаем пару деревьев (два корня)

```
split(node, x):
    if node = null:
        return {null, null}
    if node.x < x:
        p = split(node.right, x)
        node.right = p.first
        return {node, p.second}
    else:
        p = split(node.left, x)
        node.left = p.second
        return {p.first, node}
```

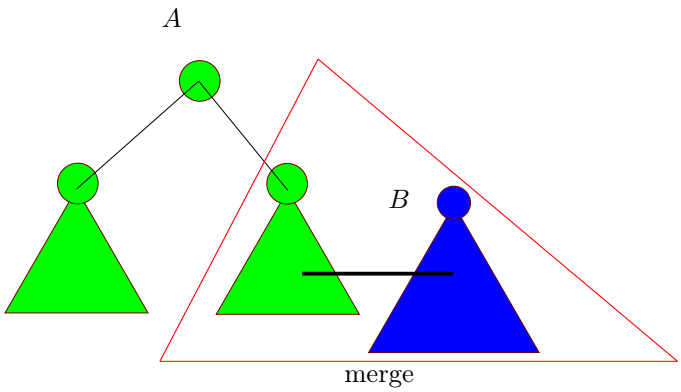

Рис. 1.9: merge

Merge(T1, T2): Все узлы первого меньше всех узлов второго

1. Определяем кто из двух корней будет итоговым корнем.
2. Делаем merge от второго дерева и ближайшего к нему поддерева главного.
3. Возвращаем корень выбранный на первом шаге

```
if A = null:
    return B
if B = null:
    return A
if A.y > B.y:
    A.right = merge(A.right, B)
    return A
else:
    B.left = merge(A, B.left)
    return B
```

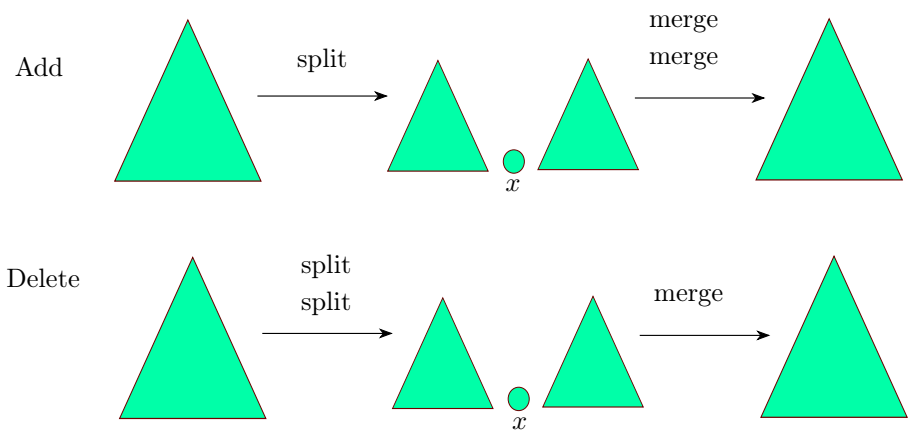

Рис. 1.10: add

Add(A, x):

1. Сплиттим на до x и после x, получается три дерева: эти и с одним элементом – x
2. Два раза мёрждим, чтобы получить итоговое дерево
3. Возвращаем результат последнего мёрджа

```
add(A, node):
    if node.y > A.y:
        p = split(A, node.x)
        node.left = p.first
        node.right = p.second
    if node.x < A.x:
        A.left = add(A.left, node)
    else:
        A.right = add(A.right, node)
    return A

remove(A, x):
    if A.x = x:
        return merge(A.left, A.right)
    if x < A.x:
        A.left = remove(A.left, x)
    else:
        A.right = remove(A.right, x)
    return A
```

На этом всём можно делать действия на отрезке как обычно, в том числе с propogate

Чтобы сделать персистентным, нужно найти все места, где мы присваиваем

и не присваивать. Пример:

```
    Ap.right = merge
    Ap.left = A.left
    return Ap
```

такая копия, у которой одна половинка такая же, а другая изменённая

Воспоминание 1. Были списки [G F W O] + [P C Q] = [G F W O P C Q]

Хотим два списка объектов обеъединять в один список. А также хотим делать сплит по позиции.

split(3) -> [G F W], [O P C Q]

План: хранить последовательности в дереве поиска

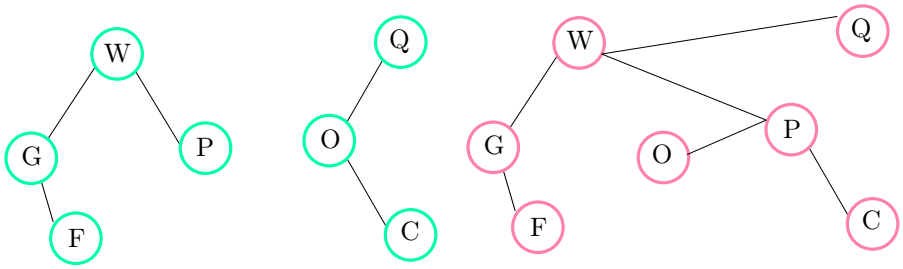

Рис. 1.11: Пример merge'а

$merge\,(W, Q) \to merge\,(W, P) \to merge\,(O, P) \to merge\,(O, null)$

$split\,(Q, 5) \to split\,(W, 5) \to split\,(P, 2) \to split\,(C, 0)$

1.13 Splay дерево

Нет никаких вариантов, просто дерево. Любое дерево это валидное Splay дерево. Балансируем с помощью магии, о которой дальше

Есть операция $splay(x)$, которое последовательностью поворотов ставит x в корень

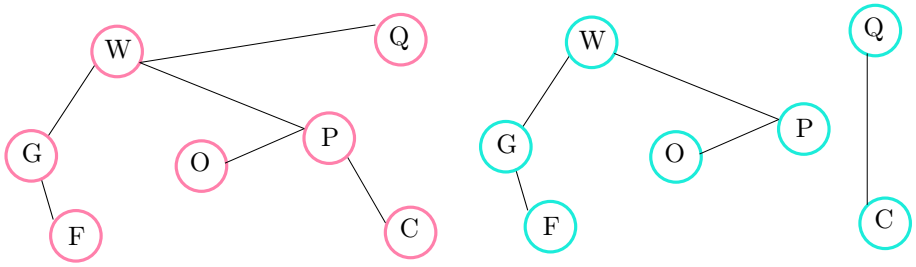

Рис. 1.12: Пример split'a

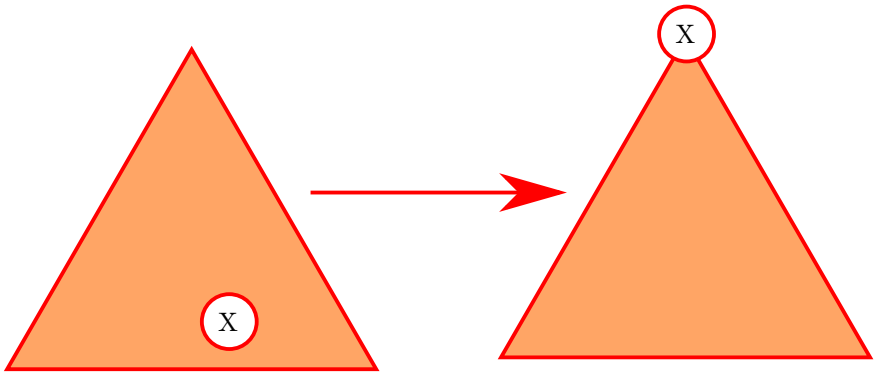

Рис. 1.13: операция Splay

Что делать со всеми остальными операциями? Ко всем операция, которые лезут вглубь дерева мы будем в конце приписывать Splay

Хотим доказать, что $\widetilde{T}(splay) = O(\lg n)$, из этого будет следовать, что и $\widetilde{T}(find) = \lg n$

Вот и весь алгоритм. Давайте ещё раз:

1. Есть путь до x и мы хотим сделать операцию *splay*

z

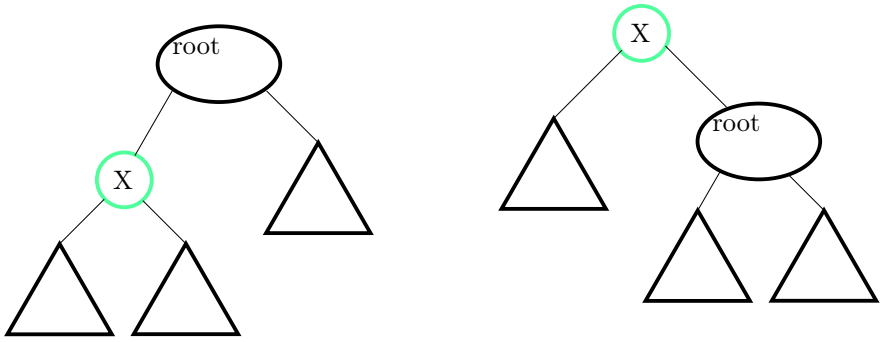

Рис. 1.14: Операция zig

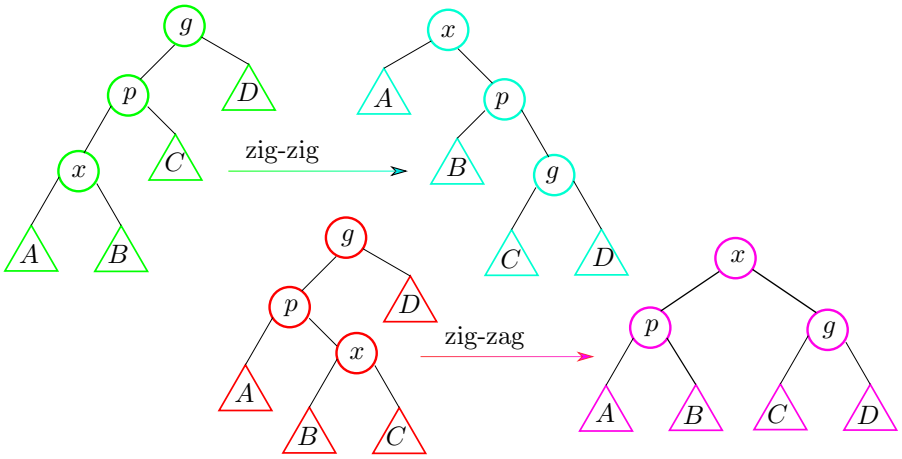

Рис. 1.15: zigzag-zigzig

2. Смотрим на два уровня вверх от x и применяем zig-zig или zig-zag соответственно

3. Если остаётся только один родитель, делаем zig

$T(find) = 2T(splay)$

$\widetilde{T}(splay) = O(\log n)$

$\sum T(splay) \leqslant k \log n \implies \sum T(find) \leqslant 2k \log n \implies \widetilde{T}(find) = O(\log n)$

Слейтор, Тарьян. Доказывать амортищированные оценки будем с помощью метода потенциалов.

Ф – потенциал $\geqslant 0$ $\qquad \widetilde{T} = T + \underbrace{\Delta\Phi}_{\geqslant 0} \geqslant \sum T$

Потенциал здесь – мера того насколько дерево плохое. Хотим такой потенциал, чтобы $\widetilde{T} = O(\log n)$

$w(x)$ – вес узла x. $w(x) = 1$ пока что все веса единицы

$s(x)$ – суммарный вес поддерева x

$r(x) = \log_2 s(x)$

$\Phi = \sum r(x)$

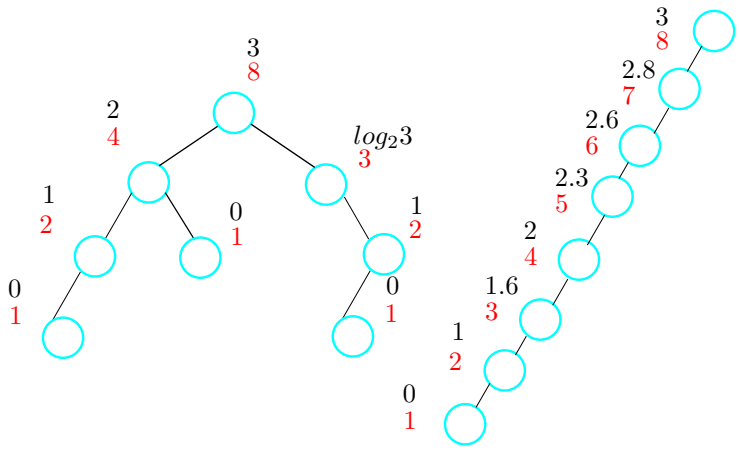

Рис. 1.16: proof-splay

$\Phi \approx 8.6$ $\qquad \Phi = \approx 15.3$

$\widetilde{T}(splay(x)) \leqslant 1 + 3\,(r'(x) - r(x))$

Доказать это – весь план

zig $\widetilde{T} \leqslant 1 + 3\,(r'(x) - r(x))$

zig-zag, zig-zig $\widetilde{T} \leqslant 3\,(r'(x) - r(x))$

Если сложить последовательность применений этих команд получится $3(r_2 - r_1) + 3\,(r_3 - r_2) + \ldots + 3\,(r_{n-1} - r_{n-2}) + 1 + 3\,(r_n - r_{n-1}) = 1 + 3\,(r_n - r_1)$

Замечание. Единицы в сумме может не оказаться hence знак $\leqslant$

Доказательство.

zig $\widetilde{T} = \overbrace{T}^{=1} + \Delta\Phi = 1 + \cancel{r'(x)} + \overbrace{r'(root)}^{\leqslant r'(x)} - r(x) - \cancel{r(root)} \leqslant 1 + (r'(x) - r(x)) \leqslant$
$\leqslant 1 + 3\,(r'(x) - r(x))$

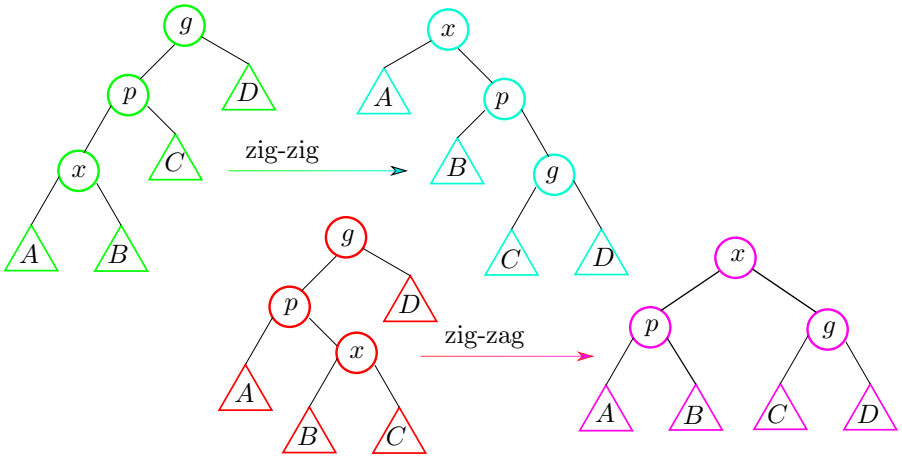

Рис. 1.17: zigzag-zigzig

zig-zag

$$\widetilde{T} = 2 + r'(x) + r'(p) + r'(g) - r(x) \overbrace{-r(p)}^{\leqslant -r(x)} - \overbrace{r(g)}^{=r'(x)}$$

$$\leqslant 2 + r'(p) + r'(g) - \cancel{2r(x)} \overset{?}{\leqslant} 2\,(r'(x) - \cancel{r(x)})$$

$$r'(p) + r'(g) - 2r'(x) \leqslant -2$$

$$\log_2 \left(\frac{s'(p) \cdot s'(g)}{s'(x) s'(x)} \right) \leqslant -2$$

$$\underbrace{\frac{s'(p)}{s'(x)}}_{A} \cdot \underbrace{\frac{s'(g)}{s'(x)}}_{B} \leqslant \frac{1}{4}$$

.

$A + B \leqslant 1$ (размер p и g вместе меньше чем размер вершины над ними – x)

$A \cdot B - - \max \iff A = B (= \frac{1}{2}) \qquad A \cdot B = \frac{1}{4}$

$A =: \frac{\alpha}{\gamma} \quad B =: \frac{\beta}{\gamma}$ (обозначим соответствующие части итогового дерева как α, β и γ)

$A + B = \frac{\alpha+\beta}{\gamma} \quad \alpha + \beta < \gamma$

zigzig

$$\widetilde{T} = 2 + \cancel{r'(x)} + \overbrace{r'(p)}^{\leqslant r'(x)} + r'(g) - r(x) \overbrace{-r(p)}^{\geqslant -r(x)} - \cancel{r(g)}$$
$$\leqslant 2 + r'(x) - 2r(x) + r'(g) \leqslant 3\left(r'(x) - r(x)\right)$$

$r(x) + r'(g) - 2r'(x) \leqslant -2.$

Доказывается той же логикой

$$\widetilde{T}(splay) \leqslant 1 + 3\left(\underbrace{r'(x)}_{\leqslant \log n} - \underbrace{r(x)}_{\leqslant 0}\right) \leqslant 1 + \log n = O\left(\log n\right) \qquad \blacksquare$$

Итого не хуже логарифма, но может быть веселее, если случай не худший.

Обычно при оценке времени работы алгоритма, мы смотрим на худший случай. А теперь мы хотим посмотреть на время работы во всех случаях.

Как понять худший ли наш случай или нет, когда мы делаем $find(x_1)\ find(x_2)\ \ldots\ find(x_k)$?

Пусть делаем $find(x) \ldots find(x) \quad k$ раз

АВЛ-дерево потратит $k \log n$, а Splay-дерево – $k + \log n$. Этот случай явно не худший и Splay-дерево это “просекло”

Теперь смотрим на последовательность $find(x_1), \ldots, find(x_k) \quad T_{opt}$ – минимальное время для обработки (сделаем специально дерево, которое хорошо крутится, подбирает коэффициенты, чтобы получилось минимальное время обработки)

Интересно отношение $\frac{T}{T_{opt}}$, т.е. на каком тесте оптимальное дерево действует быстро, а наше долго.

Замечание. Есть гипотеза, что Splay-дерево динамически оптимально и отношение $\frac{T}{T_{opt}} \leqslant const$???

Никто не знает правда это или нет. Есть несколько важных неравенств про разные случаи. На разное смотрели, везде получалось такое, какое надо.

Если вдруг окажется, что это правда, то можно будет делать такие штуки: Есть задача и в ней нужно дерево поиска. Мы можем доказывать, что существует какое-то дерево, которое быстро работает. Тогда мы сможем пихнуть Splay-дерево и оно тоже будет быстро работать, потому что магия.

Такие случаи в целом бывают. Есть реализации, про которые известно, что они лучше, чем любые другие реализации. Верно ли это про Splay-дерево – неизвестно.

1.14 Scapegoat tree

- Построение: $n \log n$
- Зарос $\log^2 \quad (\log n)$
- Изменение $\log^2$

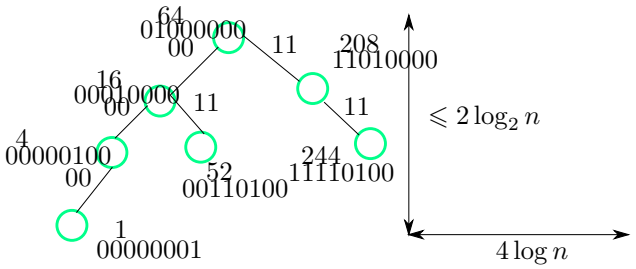

Рис. 1.18: scapegoat

Проблема: если просто добавлять элементы в дерево поиска получится бамбук. Значит не хватает балансировки.

$\alpha = const \quad \frac{1}{2} \leqslant \alpha < 1$. Скажем, что размер ребёнка должен быть не больше αw

$\alpha = 0.75$

Пусть добавили вершину. Найдём тех, кому стало больно. Возьмём поддерево того, кому плохо. Берём и полностью его строим с нуля. Если несколько берём максимальное.

Теперь смотрим за сколько всё это счастье работает.

Утверждение 3. Амортизированное время работы не очень большое

Доказательство. Всё, кроме перестроения работает за логарифм (потому что высота)

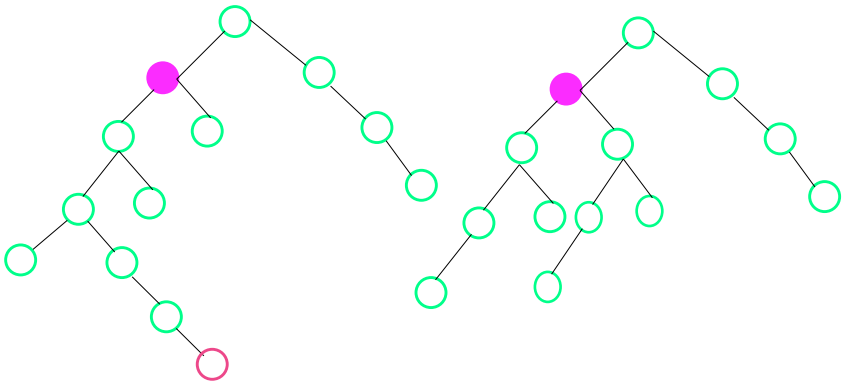

Рис. 1.19: goatex

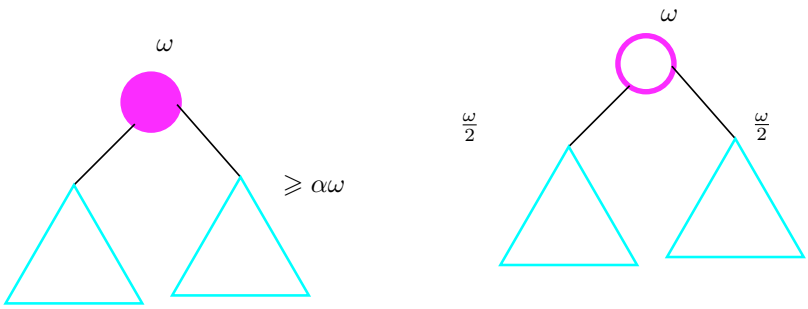

Рис. 1.20: rebuild

$$\begin{aligned}
\frac{\omega}{2} + x &> \alpha\,(\omega + x) \\
\omega + 2x &> 2\alpha\,(\omega + x) \\
2\,(1-\alpha)\,x &> (2\alpha - 1)\,\omega \\
x &> \omega\,\underbrace{\frac{2\alpha - 1}{2(1-\alpha)}}\,.
\end{aligned}$$

$T = c_1\omega$

$\widetilde{T} = \frac{c_1\omega}{c_2\omega} = O(1)$

Но добавляя элемент, мы добавляем во все поддеревья, поэтому $\widetilde{T}(add) = O(\log n)$ ■

А теперь будем не АВЛ делать, а scapegoat.

Будет $T = c_1\omega \log\omega \quad \widetilde{T}(add) = O\left(\log^2 n\right)$

Будем в каждом узле хранить мапу сколько какой элемент встречается. Можно ещё выпуклую оболочку.

1.15 ListOrderMaintenance

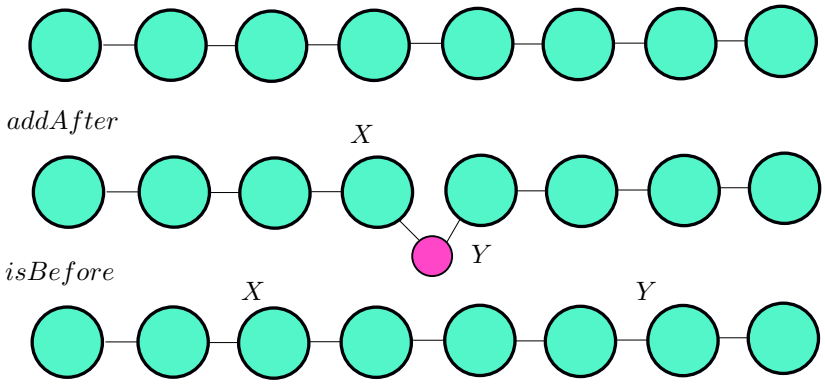

Рис. 1.21: lom

1. $addAfter(x, y)$ – добавить y после x-а
2. $isBefore\,(x, y)$– проверяет, что x находится левее, чем y

При добавлении между x и z будем записывать $\frac{x+z}{2}$

В какой-то момент закончатся целые или точность дробных..

1. Храним числа, когда добавляем делаем $\frac{x+z}{2}$. Работает, если немного элементов.

   Если $\approx k = \log M$ чисел (где M макс число), то всё норм. Но при этом обе операции работают за $O(1)$

2. План: идём направо добавляем 11 иначе 00

   Теперь мы можем сравнивать за $O(1)$

   Итог:

   - $addAfter \quad O(\log n)$
   - $isBefore \quad O(1)$

Один логарифм убился, осталось убить ещё один логарифм.

Разобьём список на блоки не больше, чем по логарифму штук (но хотя бы $\frac{\log n}{2}$). И один элемент будет описанием этого блока

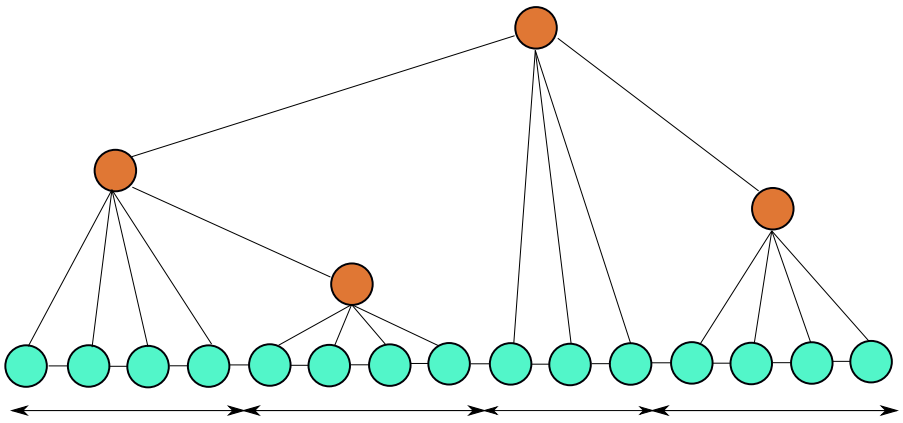

Рис. 1.22: blocklog

Добавили в блок. Блок мог стать больше логарифма. Сделам два блока, каждый размером $\frac{\log n}{2}$

Амортизированно за $O(1)$

Итого штука, которая делает обе операции за единицу.

1.16 LCA

lowest common ancestor

1.16.1 Двоичные подъёмы

<todo> 20 минут лекции

$dist(u, v) = (d(u) - d(w)) + (d(v) - d(w))$

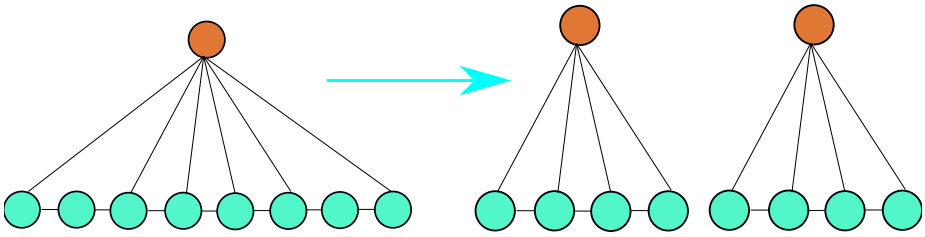

Рис. 1.23: lomgoatadd

$jump[v, k]$ – вершина на расстоянии 2^k от v

v не больше n значений, k – $\log n$

Всего $n \log n$ значений, утверждается, что можно построить за $O(n \log n)$

$jmp[v, 0] = v.parent$

```
    for k = 1 .. lg n
        jmp[v,k] = jmp[jmp[v,k-1],k-1]
```

Теперь, собственно, ищем $LCA(u, v)$

1. Возьмём u, v
2. (пусть u нижу v) Находим u' на той же глубине, что и v. Делается за log прыжков
3. Двоичный поиск с указателями:
   - Прыгаем на 2^k, находим общего предка
   - Прыгаем на поменьше.
   - Если попали в разные, значит lca повыше, запускаемся от полученных вершин
   - Если в одинаковые, значит либо это lca, либо есть пониже, уменьшаем степень

```
    if d(u) < d(v)
        swap(u,v)
    for k = lg n .. 0
        up = jmp[u,k]
```

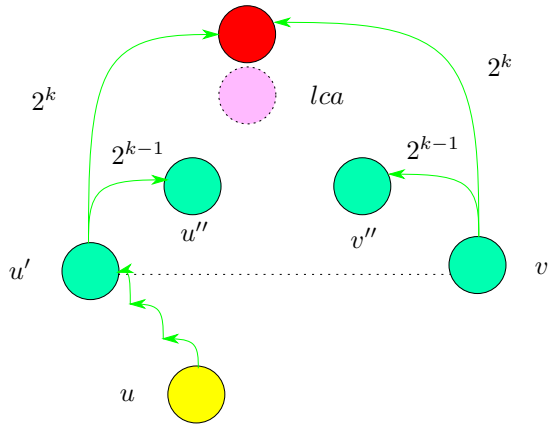

Рис. 1.24: lcapic

```
5           if d[up] >= d(v)
6               u = up
7       if u = v
8           return u
9       for k = lg n .. 0
10          up = jmp[u,k]
11          vp = jmp[v,k]
12          if up != vp
13              u = up
14              v = vp
15      return u.parent
```

В конце мы имеем две вершины, такие что как бы мы не прыгали наверх всегда встретимся в одно, значит если прыгнуть на 1, то получим lca.

Асимптотика логарифм $O(\log n)$

Дерево можно расписать в виде линии:

a =[1 2 4 2 5 6 5 2 1 3 7 5 1 8 1]

b =[0 1 2 1 2 3 2 1 0 1 2 1 0 1 0]

$b[i] = d[a[i]$

Пусть считаем $lca(3, 6)$

Смотрим на кусок массива глубин между ними. Это какой-то путь между ними. Там точно есть lca, мимо него не пройти. Утверждается, что lca – вершина с минимальной глубиной. Если из lca выйти, то обратно мы не зайдём.

$\min b[i]$ на отрезке – lca

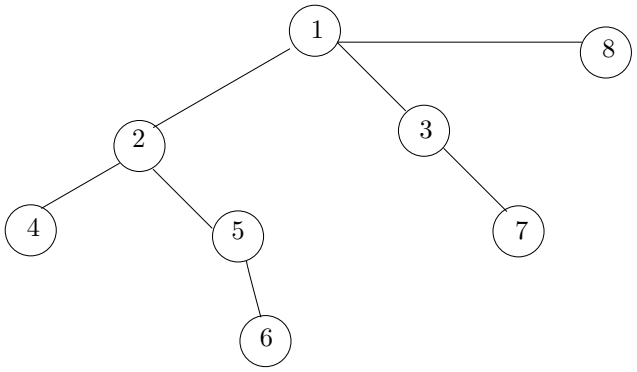

Рис. 1.25: derevo-davno-ne-bylo

Задача 5. Есть много запросов на lca

	предпосчёт	запрос
Двоичный подъём	$n \log n$	$\log n$
Э.О. + Д.О.	n	$\log n$
Э.О. + Р.Т.	$n \log n$	1
ФКБ	n	1

План – отвечать на вопросы про минимум за 1 не потратив $\log n$ на предпосчёт. Выберем кусочки длины $B = 3$. В каждом куске выберем минимум и его запомним

$b' =$[0 1 0 1 0] – длина $\frac{n}{b}$

На b' построим Разреженую таблицу

Начнём с простого: как это нам поможет?

Эту таблицу мы строим за $\frac{n}{b} \log \frac{n}{b}$

Если взять $b = \log n$, то $\frac{n}{b} \log \frac{n}{b} = \frac{n}{\log n} \log \frac{n}{\log n} \leqslant \frac{n}{\log n} \log n = O(n)$

Объвим блоки [2 3 4 3 4 3] и [5 6 7 6 7 6] эквивалентными (можем добавить 3)

Сколько есть классов эквивалентности? Будем вычитать первое число из блока.

2^{B-1} блоков бывает, если длина блока B

Класс эквивалентности можно кодировать так: Если следующее число на

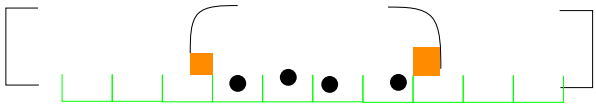

Рис. 1.26: minfind

1 больше, то 1, если меньше, то 0. Так составляется двоичное число для нумерации классов.

Переберём все классы эквивалентности. Для каждого класса эквивалентности посчитаем все возможные вопросы за $O(2^B \cdot B^2)$. Если $B = \log n$, то это $O(n \log^2 n$

Возьмём поменьше $B = \frac{\log n}{2}$ $\quad O\left(2^B B^2\right) = O(\sqrt{n} \log^2 n) = O(n)$

Остался обещанный запрос за $O(1)$. Для блоков, которые вошли целиком, лезем в таблицу и узнаём минимум на них. Для двух крайних тоже уже всё предпосчитано.

Утверждение 4. С помощью такой прекрасной техники можно вообще в любом массиве искать минимум за единичку (здесь мы пользовались, что соседние элементы отличаются на 1)

a =[5 4 8 6 2 5 3 7]

Построим Декартого дерево. Возьмём минимум и сделаем его корнем

Утверждение 5. Минимум это *lca* вершин. Если минимум дерева не на отрезке, то обе вершины лежат в одном поддереве. Если минимум на отрезке, то это корень и он разделяет вершины. Это единственная вершина, которая разделяет эти вершины.

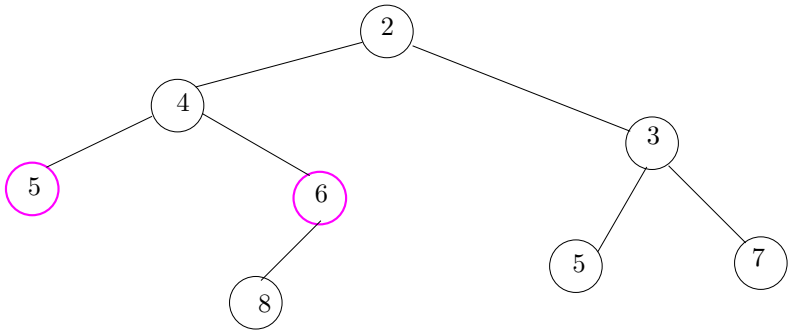

Рис. 1.27: anymin

1.17 Что ещё можно делать с деревьями

Замечание. Двоичные подъёмы работают, когда граф не меняется.

- calc(u,v) = sum на пути от u до v
- set(u,v) = задать значение

Перед деревом подумаем про простой случай. Например наше дерево – бамбук. Тогда нужно решать задачу на отрезке и решение этой задачи есть – дерево отрезков.

1.17.1 HLD

План:

1. возьмём дерево
2. Выбираем для каждой вершины ребро к поддереву с наибольшим весом (самый тяжёлый)
3. Выделяем пути таких рёбер
4. Каждый путь тяжёлых рёбер – дерево отрезков
5. Смотрим на путь от u до v. Разбиваем его до двух отрезков до lca

Предпосчёт – $O(n)$

Кусков логарифм, в каждом дерево отрезков – $O(\log^2)$ на calc

Засунуть значение – найти дерево отрезков и изменить его – $O(\log n)$

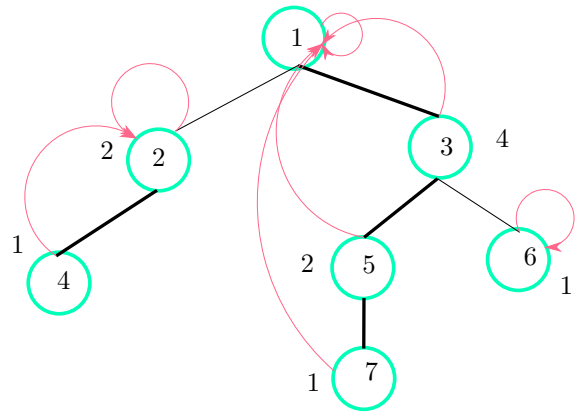

Рис. 1.28: hlddef

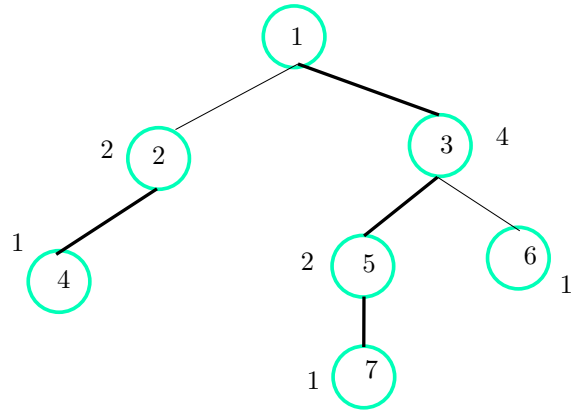

Рис. 1.29: smalltreewithnumbers

```
    dfs(x):
        p.push_back(x)
        if heavy (x) == nul:
            return
        dfs(heavy(x))
        for y : children(x)
            if y != heavy(x)
                dfs(y)
```

Обход – $\boxed{1, 3, 5, 7, \underline{6}, 2, \underline{4}}$

Второе упрощение в жизни – lca не нужно сразу считать, можно по ходу. Для каждой вершины можно храниты ссылочку в верхнюю вершину её тяжёлого пути. Глубины тоже посчитать можно, если нужны.

1. Смотрим на две вершины и их тяжёлые пути
2. Если они совпадают, то считаем между ними расстояние.

```
    x = top(u)
    y = top(v)
    if x = y:
        res += sum(u,v)
        return red
    # y пониже. y ниже, чем lca, кусок с y можно добавить
    if d[y] >= d[x]:
        res += sum(v,y)
        v = y.parent # если( на рёбрах, то это либо надо учеть, либо
    сама добьётся, включившись)
    else:
        rest += sum(u,x)
        u = x.parent
```

1.18 Бонус:

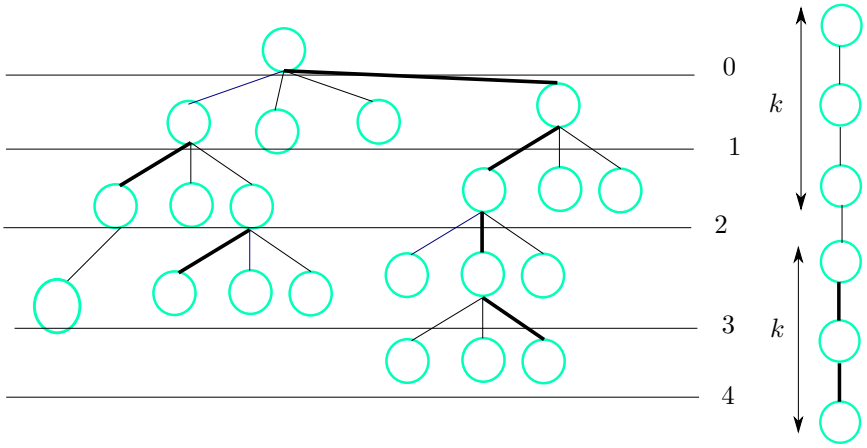

Рис. 1.30: bonus-derevo

Задача 6. Есть у нас дерево

Нужно находить $la(v, l)$ – предок v на уровне l

$D = (d[v] - l)$

1. Выберем длинные пути

2. Из этих путей делаем лесенки: возьмём длинный путь, в нём k вершин. Найдём следующие k вершин сверху (или до корня). Всё это вместе назовём лесенкой. Каждая лесенка – массив

3. $\sum$ длина лесенок $= n$. $\sum$ длина лесенок $\leqslant 2n$. Лесенка строится за линию от размера лесенки. Сумма размеров путей – n, сумма времени на лесенки – $O(n)$

Считаем $la(v, l)$:

1. v лежит на длинном пути. Берём лесенку этого пути и поднимаемся по ней.

2. Возьмём новый длинный путь и новую лесенку. Если мы перешагнули нужный уровень, значит где-то на пути находится

3. Каждый раз мы увеличиваем длину длинного пути как минимум в два раза (потому что у вершине, в которую мы перешли, есть уже путь длинной 2· предыдущюю длину. Значит длинный путь у новой вершины как минимум такой длины, по выбору длинного пути)

4. Значит таких прыжков не больше, чем логарифм штук.

```
x = top(v)
if d(x) > l:
    v = x
else:
    return ladde[v][l-d[x]]
```

Замечание. у нас есть два подхода: двоичные подъёмы и лесенки. Давайте их скрестим

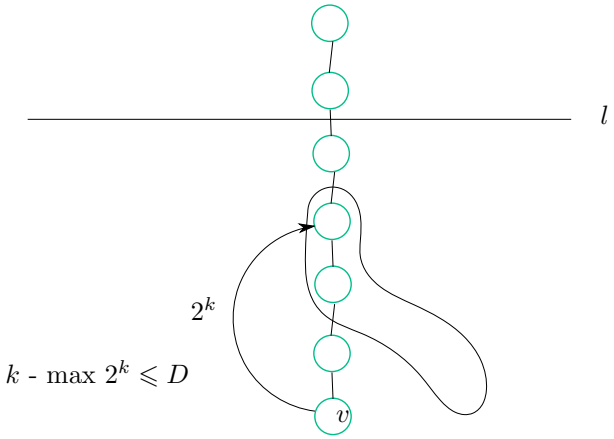

Рис. 1.31: ezh-x-ezh

Будем делать двоичные подъёмы только в листьях.

Возьмём лист в поддереве v. Оттуда делаем двоичный подъём, а оттуда по лесенке . Это делается за константу.

Случай: ёжик, одна вершина с n детьми

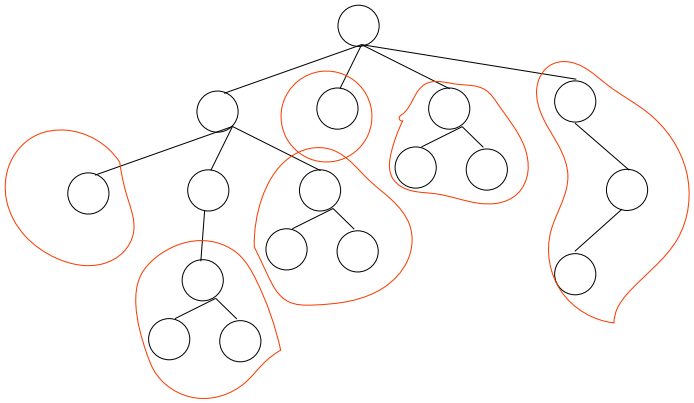

Рис. 1.32: ezh-real

Отрежем все деревья тяжелее, чем $B(=3)$

Если $B \sim \log n$, то появится время на двоичные подъёмы. Листов станет $\leqslant \frac{n}{B}$

Случаи:

1. v не в листе. Делаем как раньше, в лист, двоичный подъём, лесенка

2. v в отрезанном листе. У неё есть ссылка на верхнюю вершину листа. Берём вершинку выше (родителя), а дальше алгоритм из пункта 1)

3. v внутри и предок тоже внутри. Число типов $\leqslant 2^{2B} \leqslant \sqrt{n}.\lhd B = \frac{\log n}{4}$

   Смотрим на одно дерево конкретного типа, считаем на нём все возможные запросы (табличка mem[type,v,l]) – $O(\sqrt{n}\log^2 n) = O(n)$

1.19 TODO

1.20 Euler TODO

1.21 Цетроидная декомпозиция

Назовём близкими вершины, для которых $dist(u, v) \leqslant D$

Найти число таких пар можно бинпоиском, префиксными суммами, двумя указателями ... но это на бамбуке

Выберем вершины, поделим пути на те, которые содержат и которых не содержат вершину

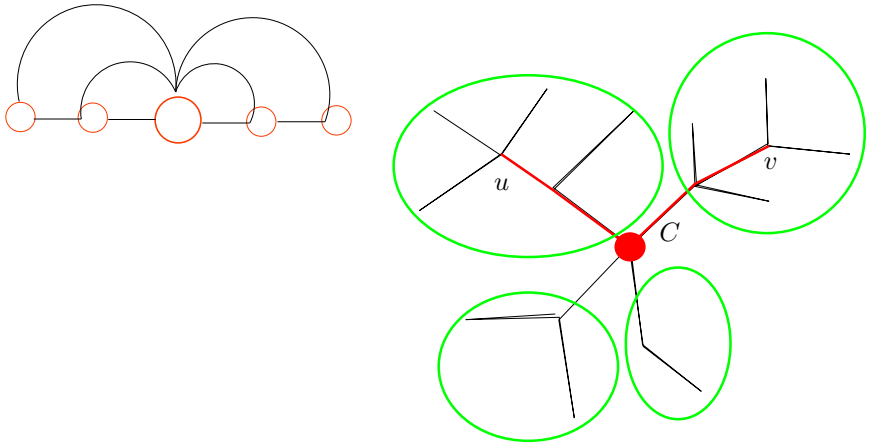

Рис. 1.33: deli

Когда у нас несколько деревьев, одно выбираем, а все другие запихиваем в большое дерево поиска и ищем там

$d[v] = dist(v, c)$

```
    BST
    for t in trees:
        for v in t:
            find(bst, D - d(u))
        for v in t:
        insert(bst, v)
```

Для хорошести рекурсии нужно, чтобы самый большой кусок при раздлении был поменьше

Лемма 1. В каждом дереве есть вершина, такая что при разделении по ней куски получаются $\leqslant n$

Доказательство. Подвесим за любую вершину. Посчитаем все веса поддеревьев.

Если какое-то дерево слишком большое, то пойдём в ребёнка большего половины.

При переходе мы знаем что всё вне ребёнка $\leqslant \frac{n}{2}$, значит нужно проверить , что у детей детей $\leqslant \frac{n}{2}$, если нет, то идём дальше. За один обход находим нужную вершины, называемую центроидом ■

Алгоритм:

1. Делим по центроиду
2. В кусках рекурсивно высчитываем пути
3. В путях меньше деревьями считаем процедурой выше.

Задача 7. Есть сеть дорог.

[картинка] меня можно пнуть её сделать

Картинка делеает за $n \log n$

Доказательство. Найдём первую оющую компоненту двух вершин

Предпосчитаем расстояния от всех центродов к их детям

```
        for c in centroids[v]:
            i = binsearch(component[c], d - dist(c, v))
            res = min(res, prefmin[c][i])

```

■

Если считать что-то менее хорошее, чем минимум, например сумму, то можно посчитать два раза если зайти в то же поддерево из которого мы вышли.

Задача: нам нужно посчитать сумму по всех компонентам, кроме нашей

Человеческий код:

```
    build(v):
        # строит декомпозицию
        c = find_centroid(v)
        mark[c] = True # Чтобы по ней больше не ходить
        prepare(c) # посчитать расстояни, посортировать, ... all the fun
     stuff
        for u in g[c]: # рёбрасоседи-
```

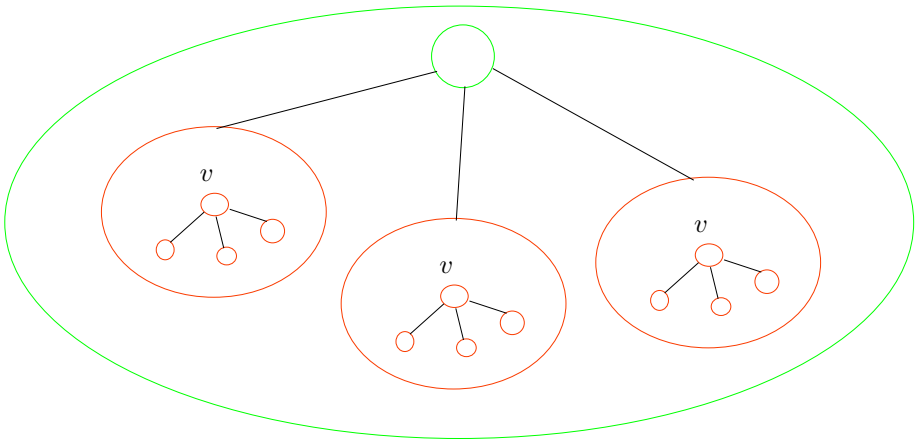

Рис. 1.34: kartinka

```
7             if (not mark[u]):
8             build(u)
9
10    prepare(v):
11        dfs(v, -1)
12
13    dfs(v, p, d):
14        for u in g[u]:
15            if !mark[u] and u != p:
16                dfs(u,v,d+1)
17        centroids[v].add({c, d})
18
19    for v = 1 ... n:
20        for (c, d) in centroids[v]:
21            comp[c].add({d, r})
22            comp2[c].add({dp, v}) # в comp2 вершину сортируются по
    расстоянию до предудущго центроида
23            dp = d # dp -- расстояние до предыдущего центроида
24
25    for c = 1 ... n:
26        sort(comp(c))
27        sort(comp2(c))
28
29
30    calc(v, D):
31        for (c, d) in centroid[v]:
32        i = binsearch(comp[c], D - d)
33        res += prefsum[c][i]
34        i = binsearch(comp2[c], D - dp)
35        res -= prefsum2[c][i]
36        dp = d
37
38    calc(v, D):
39    k = centroids[v].size
```

```
    for i = 0 .. k-1:
        c = centroids[v][i]
        x = binsearch(comp[c], D - d)
        res += prefsum[c][x]
        if i < k-1:
            cp = centroids[v][i+1]
            x = binsearch(comp2[cp], D - d)
            res  = prefsum2[c'][x]
```

Итоговая асимптотика: $O\left(n \log^2 n\right)$ на предпосчёт. $O\left(\log^2\right)$ на запрос

1.22 Алгоритмы во внешней памяти

RAM-модуль: доступ за $O(1)$

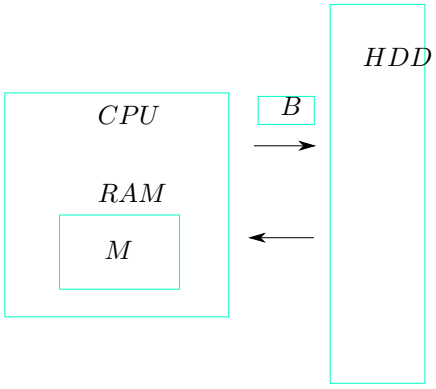

Рис. 1.35: sistema

M, B – параметры системы. Время работы = число чтений/записей блоков размера B

Эта модель может применяться к много чему, например к облачному хранилищу с долгим временем запроса и большими блоками запроса соответственно.

Тупое решение: пусть у нас есть массив и нам нужно посчитать на нём сумму. Разделим его на блоки размера B. $T(Scan(N)) = \left\lceil \frac{N}{B} \right\rceil \approx \frac{N}{B} + 1$

Отсортируем массив: хотим работать с блоками, MergeSort

$\frac{N}{B} \log \frac{N}{B}$

так мы используем только 3 Мб памяти. Давайте сливать сразу много блоково

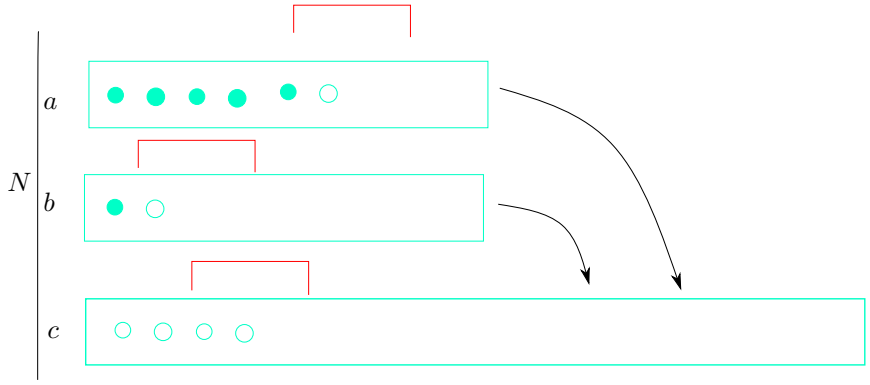

Рис. 1.36: morzh

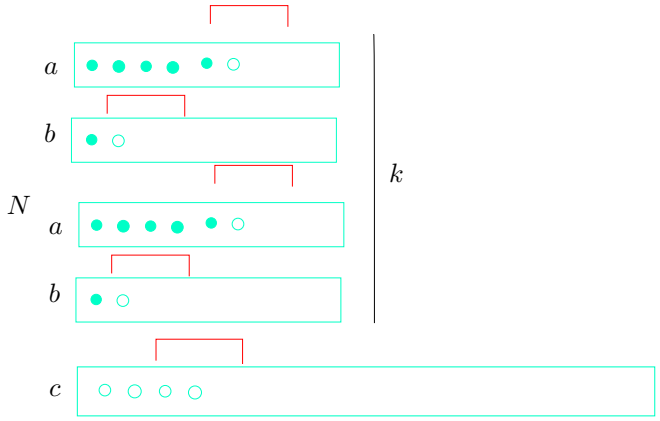

Рис. 1.37: big-morzh

$k \approx \frac{M}{B} \quad M \geqslant kB$

$Sort(N) = \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$

Утверждение 6. Это время оптимально.

Доказательство. N! всех перестановок. Выбираем одну, нужно $\log(N!)$ сравнений (примерно $N \log N$)

Когда мы мёрджим блоки размера M и B.

$\log C_m^B \approx \log \frac{M(M-1)(M-2)\ldots(M-B+1)}{B(B-1)(B-2)\ldots 1} \approx B \log \frac{M}{B}$ ■

Стэк во внешней памяти:

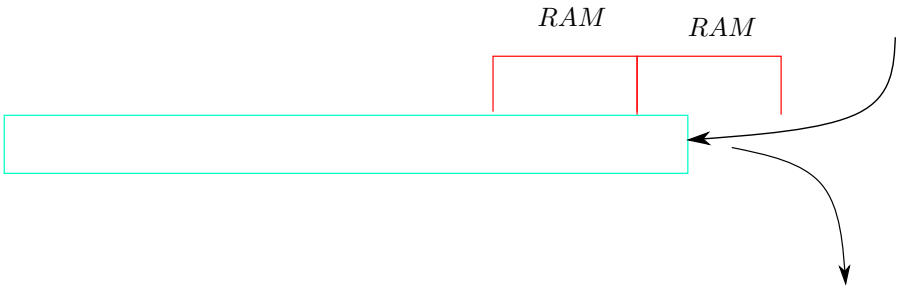

Рис. 1.38: stack

$O(1)$ – нехорошо

$O(\frac{1}{B})$ – правильная оценка. 1 запрос на B операций

Дерево поиска:

B-дерево. В каждом узле хранится B ключей и $B+1$ ребёнка. Время работы $O\left(\log_b^N\right)$

С деревом посика нельзя запросы меньше 1, потому что дерево поиска может выдать любой свой элемент, а держать их все в оперативной памяти оно не может, значит всегда есть запрос, который заставит его что-то подгрузить.

Хотим обратную перестановку. Если бы не было внешней памяти, то

```
1     for i = 1 .. n:
2         r[p(i)] = i
```

С памятью запишем все такие пары $(i, p[i])$ и отсортируем их по второму элементу

35214 $(1,3),(2,5)(3,2)(4,1)(5,4) \implies (4,1)(3,2)(1,3)(5,4)(2,5)$ – выбираем первые элементы пар и это будет обратная перестановка.

Есть два массива. Хотим посчитать третий, чтобы $c[i] = b[a[i]] \quad a[i] \in [1..N]$

Сделаем массив пар $(i, a[i])$ И $(i, b[i])$

Отсортируем первый по $a[i]$, а второй по i. Тогда переход между ними будет приятный (можно идти указателями по сортированным массивам)

Задача 8. Есть массив $next$, в котором написан какой элемент следующий после данного. $[5, 7, -, 3, 2, 4] \quad 1 \to 5 \to 3 \to 6 \to 2 \to 7 \to 4$

Доказательство. Безыдейно:

```
        x = 1
        for i = 1 .. n:
            print x
            x = next(x)

```

Переделаем задачу: Есть элементики

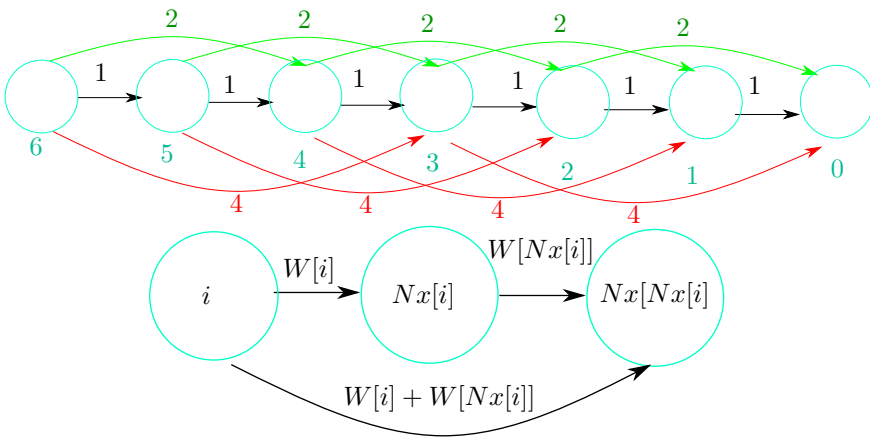

Рис. 1.39: elementiki

Если делать все двоичные подъёмы, то будет $O\left(\log N \cdot \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$

Можно делать не все подъёмы.

Поставим в элементы рандомные битики. В местах, где с 1 переходим на 0 будем делать переход через 2 (удалять тот, на котором 1)

```
    if renoved[Next[i]:
        Next[i] = Next[Next[i]
        W[i] = N[i] + W[Next[i]]
```

■

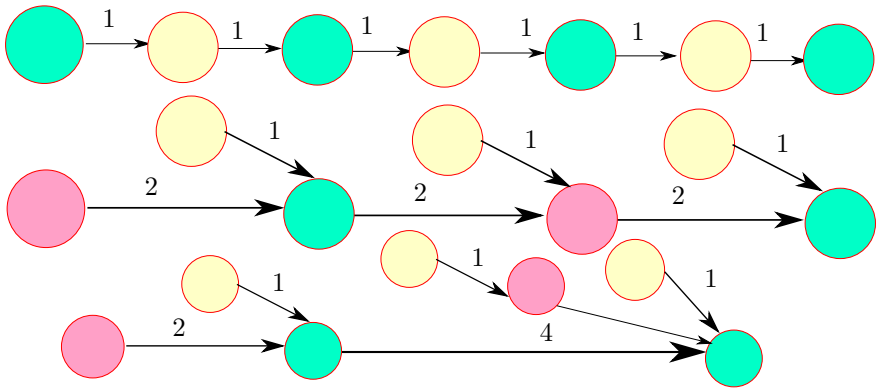

Рис. 1.40: ne-vse-podъёmы

1.23 Классы сложности

$O(n^2)$ $O\left(n^3 \cdot \log n\right)\ldots$

Класс P, решается за Полином $O(n^k)$

Хорошие задачи, они решаются

Сам полином зависит от того где решаем. (1 и $\log n$ в зависимости от RAM-модели или Pointer Machine)

Но это если за полином на одной машине, то за полином и на другой

Есть тупая машина под название Машина Тьюринга

Замечание. У машины Тюринга есть большаая лента, на которой хранятся данные

Есть одно место, текущая ячейка, на которой хранятся данные

Нет переменных, есть только состояние

$(state, x) \to (x', staete', \leftarrow\rightarrow\downarrow$

Локальных переменных нет, их нужно где-то хранить на ленте

```
1     for i = 0 .. n-1
```

Есть задачи, которые не решаются за полином

Пример. Есть шахматная доска $n \cdot x$, на которой стоят фигуры. Вопрос: выиграют ли белые?

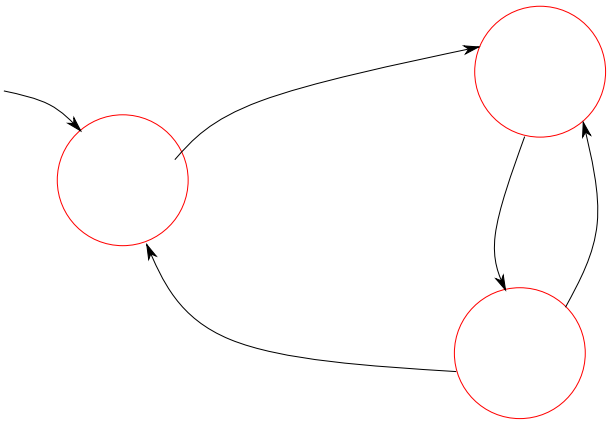

Рис. 1.41: turing-auto

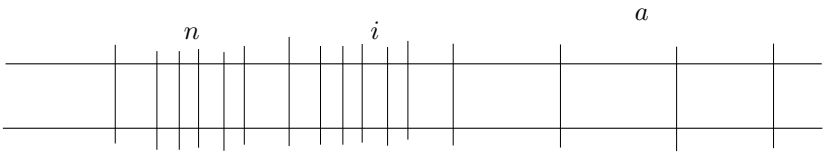

Рис. 1.42: for-turing

задача решается, но за экспоненциальное время

$O\left(e^{n^k}\right)$

$P \quad O\left(n^k\right)$

$EXP \quad O\left(e^n\right)$

RO (что-нибудь)

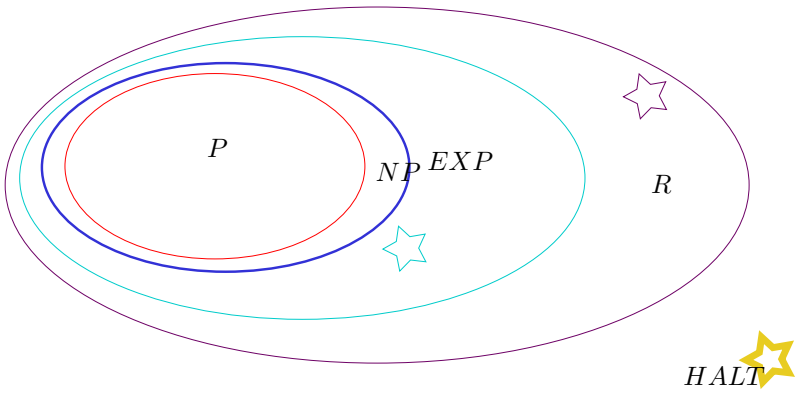

Рис. 1.43: classes-PNP

$Halt\,(p, x)$ p – программа (М.Т.) x – ввод

Правда ли, что программа p остановится на вводе x (не будет работать бесконечно)

Доказательство.

```
bool f(p):
    if HALT(p, p):
        while True:
            pass
    else:
        return True

```

$f(f)$

1. $f(f)$ Остановится, первая ветка $f(f)$ зациклиться
2. $f(f)$ зациклиться, значит вторая ветка, значит остановится

Резюме: анализировать чужие программы сложно ■

Если изучаем что-то не сильно проще, чем Машина Тьюринга, скорее всего она неразрешима

Пример. Игра “жизнь”

бесконечное (уже повод задуматься) поле. Есть клеточки, живые и мёртвые.

Есть начальное состояние

Анализировать сложно, примерно потому что можно съэмулировать машину тьюринга

Если бы могли, то решилась бы и машина Тююринга (был бы ответ "что будет в конце")

Пример. Клеточный автомат с состоянием на прямой. Каждая клеточка знает предыдущую и следующую.

Функции от 3 перемнных, их 256, не так много, можно перебрать

$x\underline{0}1 \to 1$

$1\underline{1}1 \to 0$

Вёл себя странно, через некоторое время останавливался, похоже на машину Тюринга

Пацаны потом доказали, что действитель Тьюринг-Полная

Пример. Есть квадратики, на сторонах написаны числа, есть несколько типов квадратиков

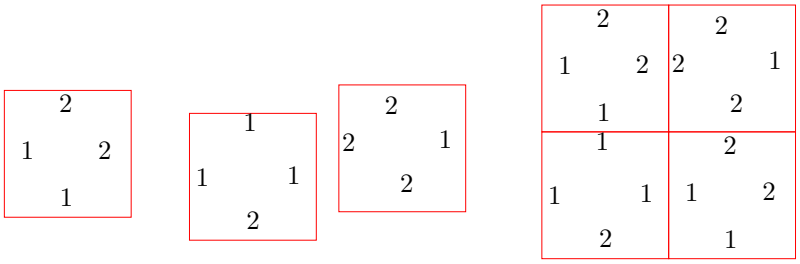

Рис. 1.44: kvadratiki

Соединяется по равным рёбрам.

Проверить, заполняет ли оно плоскость. Неразрешима

Замечание. Если бы подобные задачи решались, было бы скучно, большая Теорема Ферма доказывалась бы HALT'ом по программе, которая ищет контрпример к ней

Посмотрим на границу между P и EXP. Ребята выделили ещё один класс – NP

NP – решается за $O\left(n^k\right)$ на недетрминированной машине

ω -сертификат

Ответ “да” $\iff \exists\omega : f(x,\omega) = 1$

Например есть граф и нужно проверить есть ли в нём Гамильтонов Цикла. Она из NP

Доказательство. ω – набор подсказок – куда идти

Задача из NP, когда её ответ можно проверить. По набору подсказок куда идти понять правильно ли это. ■

Co_NP – легко проверить, что ответа нет, крайне сложно проверить, что ответ есть.

Не знаем можем ли реать задачи из $CoNP$ за полином и оказывается, что там довольно много задач

Хочется сравнивать сложности задач A не проще B. Если A научимся, то и B тоже

Хотим B свести к A

$B(x) = A(f\,(x))$

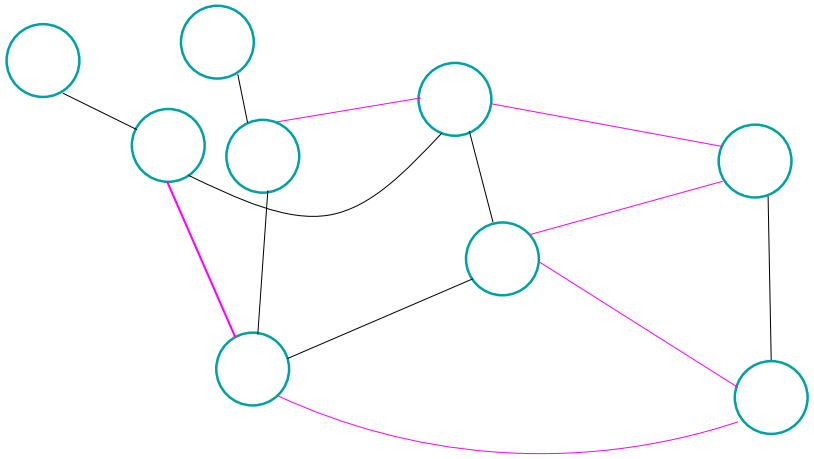

Рис. 1.45: mal-graph

Пример. Можем свести Гамльтонов цикл к Гамильтонову пути.

В NP есть задачи, к которым сводятся все задачи NP, они называются NP-трудными.

NP-полная $\iff$ NP-трудная и $\in NP$

Таких задач тоже довольно много.. Если научиться решать их за полином, то решатся все за полином

Периодически смотрят на задачу внутри NP и P. Тыкают в неё долго и выясняют, что она либо P, либо NP-полная

Если $P = NP$, всё немного странно в мире.

(p, x, n^k), права ли, что p остановится на вводе x за не более, чем n^k действий. Если решить такую, решим всё из NP

Эту НМС (недерминированную машину тюринга) можно эмулировать разными штукми

Пример. n переменных $x_1, x_2, \ldots, x_n$

$1 = (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (x_2 \vee \overline{x}_5 \vee x_8) \wedge \ldots$

NP, можно сказать значения всех переменных

3-SAT

Пример. Сведём Гамильтонов граф и 3-SAT

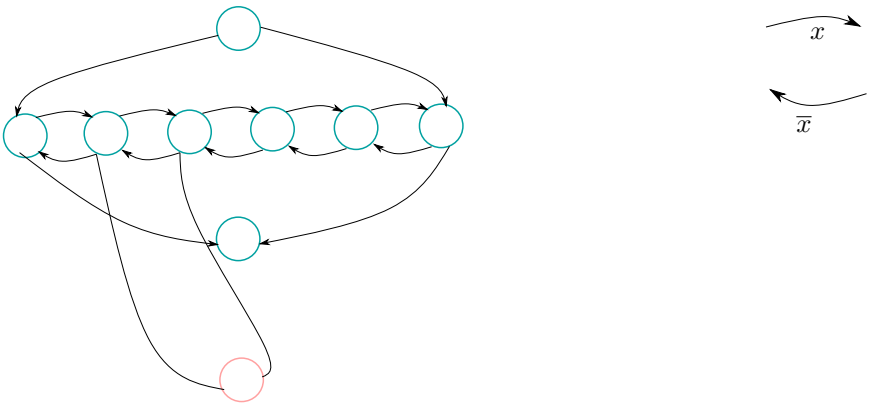

Рис. 1.46: gamil

Пример. Рюкзак. Есть w_i. Хотим набрать $\sum w_i = S$. Задача на да/нет. Она NP-полная

есть сертификат, проверит просто.

$(x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_3 \vee \overline{x}_4) \wedge (x_1 \vee x_2 \vee \overline{x}_3) \ldots$

x_1 $101\ldots$

$\overline{x}_1$ 010

x_2 001

$\overline{x}_2$ 100

x_3 110

$\overline{x}_3$001

Выберем x_1 $\overline{x}_2$ x_3 – 311 (поразрядно складываем)

Запретим брать одновременно x_i и $\overline{x}_i$. Сделаем ещё один бит спецаильно для этого. У первой переменной будет $100\ldots$, у второй $010\ldots$

Скажем, что сумма этих дополнительных разрядов должны быть с 1 максимум везде

Теперь у нас есть 311, а хочется, чтобы разряды были одинаковые. Добьём всё до 7. К каждому рахряду ещё предметы $00600.., 00500.., 00400..$

$S = 777\ldots77111\ldots11$