Разработка и анализ алгоритмов

Лекция 14
Дерево Фенвика. Частичное каскадирование.

Сергей Леонидович Бабичев

Дерево Фенвика

- Дерево отрезков — чудесная структура данных.
- Недостаток: приходится писать достаточно много кода, если в узлах — сложные структуры данных.
- Легко написать задачу: дай сумму на отрезке.
- Сложнее: дай сумму на прямоугольники.
- Неприятно: дай сумму на параллелепипеде.
- Для некоторых операций есть красивая структура данных — дерево Фенвика, которая умеет переходить к многомерным случаям за пару строк.

Задача RSQ — Range Sum Query

Задача 1. Имеется массив $A_0, A_1, \ldots, A_{N-1}$. Нужно за время $O(\log N)$ исполнять запросы

1. Присвоение в точке K: $A_K \leftarrow X$.
2. Нахождение суммы на отрезке $[L, R]$.

- Задача хорошо решается деревом отрезком.
- Дерево Фенвика решит её короче и потребует меньше памяти.

Функции f и g

- Введём функции $f(x) = x\,\mathrm{bitand}(x+1)$ и $g(x) = x\,\mathrm{bitor}(x+1)$.
- Функция $f(x)$ заменяет заключительный *блок* единиц в двоичном представлении x на нули.
- Функция $g(x)$ заменяет самый правый нулевой бит в двоичном представлении x не единицу.
- $f(00100101011) = 00100101000$, $f(000111) = 000000$.
- $g(00100101011) = 00100101111$, $g(000111) = 001111$.
- $f(x) \leqslant x$, равенство достигается для чётного x.
- $g(x) \geqslant x$, равенство достигается для числа из всех единиц.

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$f(x)$	0	0	2	0	4	4	6	0	8	8	10	8	12	12	14	0	16
$g(x)$	1	3	3	7	5	7	7	15	9	11	11	15	13	15	15	31	17
$h(x)$	0	1	2	1	4	1	2	1	8	1	2	1	4	1	2	1	16

Построение дерева Фенвика

- Введём массив $S[0 \dots N) : S_x = \sum_{i=f(x)}^{x} A_i$.
- Отрезок $[f(x) \dots x]$ имеет ненулевую длину.
- Будем считать, что проводим его справа-налево, он начался в x и закончится в $f(x)$.
- Пусть $x' = f(x) - 1$. Проведём отрезок $x' \to f(x')$ и вычислим для него сумму. Затем Проведём отрезок $x'' \to f(x'')$ и так далее.

Lemma

Итеративная процедура $fiter : x_{i+1} \leftarrow f(x_i) - 1, x_i \geqslant 0$ *создаёт последовательность не более, чем из* K *членов, где* K *— количество бит в записи числа* x*.*

Доказательство.

Пусть в начале следующей итерации x содержит $t \geqslant 0$ заключительных единиц. Результат вычисления функции $f(x)$ будет содержать не менее $t + 1$ заключительных нулей. После вычитания единицы все заключительные нули превратятся в не менее, чем $t + 1$ заключительную единицу, который после операции f все получат нулевые значения. Таким образом, количество заключительных нулей монотонно увеличивается. □

Нахождение префикс-сумм

- Пусть массив $S[0..N)$ частичных сумм уже имеется.
- Требуется найти сумму всех A_i на отрезке $[0 \dots x]$ (префикс-сумму).
- Тогда применяя последовательно указанную процедуру получаем сумму за $O(\log N)$ итераций.

```
T prefixSum(int x) {
    T ret = 0;
    for (int i = x; i >= 0; i = (i & (i+1)) - 1) ret += S[i];
    return ret;
}
```

$$requestSum(L, R) = prefixSum(R) - prefixSum(L - 1).$$

Lemma

Итеративная процедура $giter : x_{i+1} \leftarrow g(x_i), x_i < N$ *создаёт монотонно возрастающую последовательность* x_i*, содержащую не более* K *членов, где* K *— количество бит в записи числа* x*.*

Доказательство.

Если x_i не содержит нулей в двоичном представлении, то алгоритм закончен. Иначе очередная итерация заменит ровно один самый правый нулевой бит в записи числа. Нулевых битов в числе не более K, что доказывает утверждение о количестве шагов. □

Операция обновления в точке x значением y: $assign(x, y)$.

- Что умеем: при предпосчитанных S_i находить сумму на префиксе.
- Что нужно: при изменении A_x изменить все суммы S_i, зависящие от A_x.
- Задача: найти такие и только такие i, для которых для заданных x $f(i) \leqslant x \leqslant i$.

Lemma

Все значения x_i, полученные итеративной процедурой $giter : x_{i+1} \leftarrow g(x_i), x_i < N$ с начальным значением $x_0 = x$, удовлетворяют условию $f(x_i) \leqslant x \leqslant x_i$.

Доказательство.

Пусть x_i заканчивается на ноль в позиции k и какой-то набор единиц. Тогда $f(x_i)$ обнулит эти единицы. По начальному условию x лежит внутри интервала $[f(x_i); x_i]$ и тоже содержит ноль в позиции k. Применение функции $g(x)$ приводит к замене какого-то нулевого бита. Если заменён бит k, то алгоритм завершён. Иначе полученное число всё ещё лежит в нужном интервале. □

$f(x_i)$	*	*	*	*	0	0	0	0	0	0
x	*	*	*	*	0	?	?	?	?	?
x_i	*	*	*	*	0	1	1	1	1	1

Дерево Фенвика: операция точечного изменения

```
void assign(int x, int y) {
    for (int i = x; i < n; i |= i+1) S[i] += y;
    return ret;
}
```

Дерево Фенвика: инициализация массива $S[N]$

1. Универсальный, медленный. Обнулить S и N раз $assign(i, A_i)$. $T = N \log N$.
2. Не универсальный, быстрый. Создать массив префиксных сумм, по нему вычислять S_i, удалить массив префиксных сумм.

K-мерное дерево Фенвика

Задача 2. Имеется двумерный массив размером $N \times M$. К нему подаются следующие запросы:

1. Присвоить значение в точке $A_{ij} = val$.
2. Выдать значение суммы элементов на прямоугольнике l_1, r_1, l_2, r_2.

Решение: Сводим решение задачи на произвольном прямоугольнике к решению задачи на префиксном прямоугольнике. PS_{ij} — префиксная сумма на прямоугольнике $(0,0) - (i,j)$.
$S_{(L_1,R_1)(L_2,R_2)} = PS_{(L_2,R_2} + PS_{L_1,R_1} - PS_{L_1,R_2} - PS_{L_2,R_1}$.

K-мерное дерево Фенвика

- Вычислим $S_{ij} = \sum_{u=f(i)}^{i} \sum_{v=f(j)}^{j} A_{uv}$.
- Это даст нам значения префикс-сумм аналогично одномерному варианту.
- По префикс-суммам вычисляем суммы на прямоугольниках.
- Обновление S_{ij} в точке аналогично одномерному варианту.
- Инициализация S предпочтительнее через массив префиксов.

Почему всё [иногда] работает

- При исполнении операции на отрезке существенно, что мы можем вычислить «лишнее».
- Мы заменяем операцию на отрезке на две операции на префиксах и вычитаем результаты.
- Мы пользуемся, что у операции $A + B$ есть обратная — $A - B$. Зная $A + B$ и B мы восстанавливаем A.
- Операция $\max$ не имеет обратной и не восстановима.
- Дерево Фенвика можно использовать и для некоторых необратимых операций.
- Но требуется исключить лишнее вычисление с последующим откатом.
- Для этого используется «обратное дерево Фенвика».

Обратное дерево Фенвика

- Реализуем дерево Фенвика на операции max.
- Вычислим два вспомогательных массива.
- Прямой $S_i = \sum_{u=f(i)}^{i} A_u$.
- Обратный $T_i = \sum_{v=i+1}^{g(i)} A_v$.
- По прямому массиву мы двигаемся как обычно, справа-налево.
- По обратному — слева-направо.

Обратное дерево Фенвика: операция на интервале

- Исполним $getmax(L, R)$.
- Двигаемся справа-налево традиционно, $r = f(r) - 1$ до тех пор, пока r больше l.
- Левее двигаться нельзя, у max нет обратной операции.
- После первого этапа r — самая левая из всех возможных точек перемещения r.
- Два инварианта:
  1. $r > l$
  2. $f(r) \leqslant l$.

Lemma

Если выполняются условия $r > l \wedge f(r) \leqslant l$, *то истинно* $g(l) \leqslant r$.

Доказательство.

l	*	*	*	*	0	?	?	?	?	?
r	*	*	*	*	1	?	?	?	?	?

Рассмотрим первые биты отличия l и r. Так как $l < r$, то у l это 0, у r — единица. Пусть это позиция k. Переход $r \leftarrow f(r)$ — зануление последнего блока единиц. Если перед блоком есть 0, то сохраняется условие $f(r) > l$ из-за ведущей единицы в l, не имеющей соответствия в r. Условие $f(r) \geqslant l$ выполняется только тогда, когда во всех позициях в r правее k содержатся единицы.

Функция $g(l)$ заменяет младший нуль в l на единицу. В тот момент, когда будет заменён бит k, произойдёт совпадение l с r.

□

Обратное дерево Фенвика: запрос максимума

```
T intervalMax(int l, int r) {
    T ret = 0;
    while (l < r && f(r) > l) { // Right-to-left pass
        ret = ret > S[r] ? ret : S[r];
        r = f(r) - 1;
    }
    while (l < r) { // Left-to-right pass
        ret = ret > T[l] ? ret : T[l];
        l = g(l);
    }
    return ret;
}
```

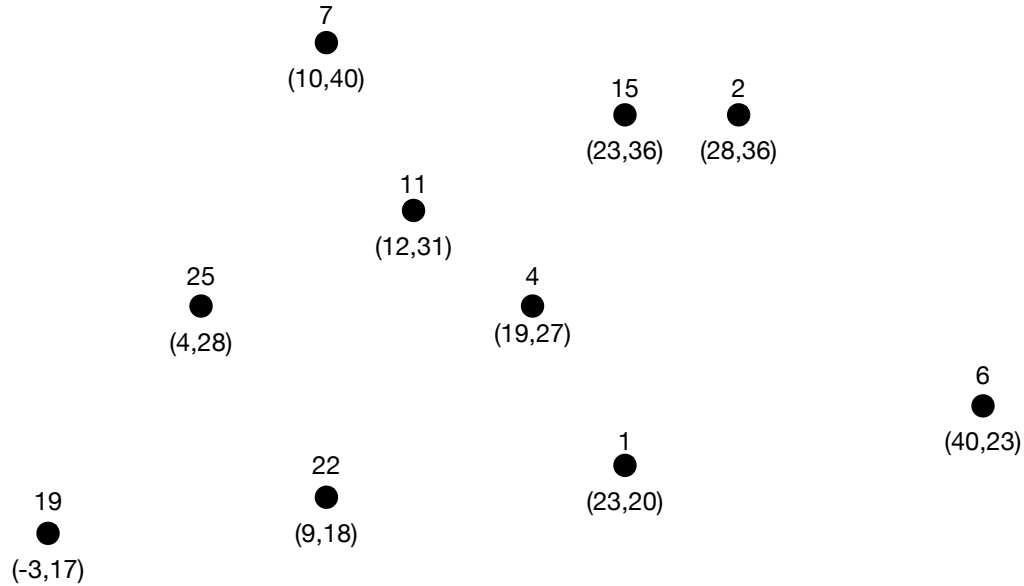
Согласно лемме отрезки не перекроются и за пределами $[l, r]$ вычислений не произойдёт.

Фенвик Фенвиков

Задача 3. Имеется N точек на плоскости с координатами (X_i, Y_i) и значением A_i. Требуется исполнять два типа запросов:

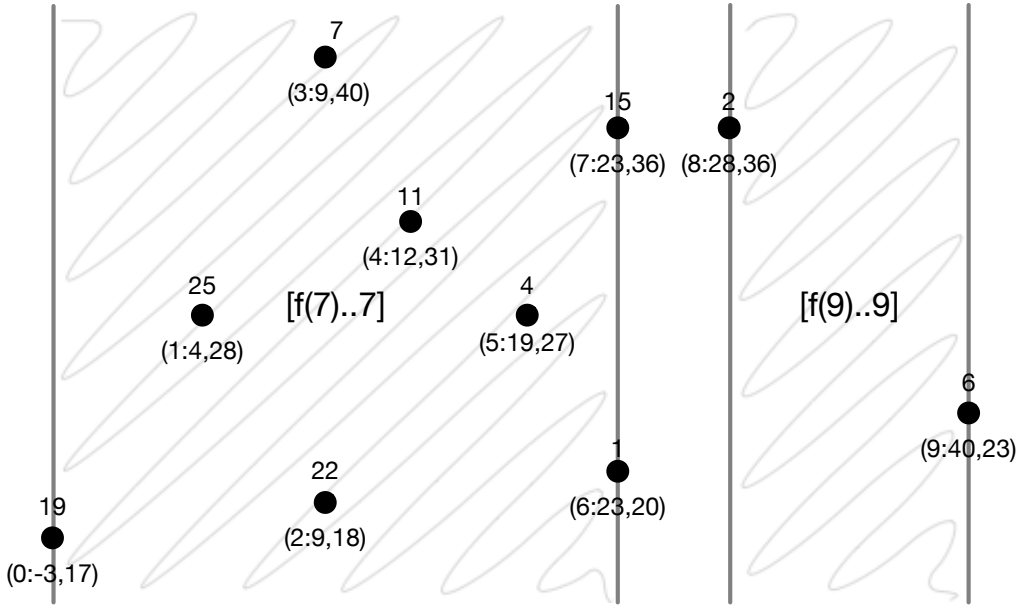
1. Изменить значение в $i-$й точке $A_i = val$.
2. Найти сумму A_i в прямоугольнике.

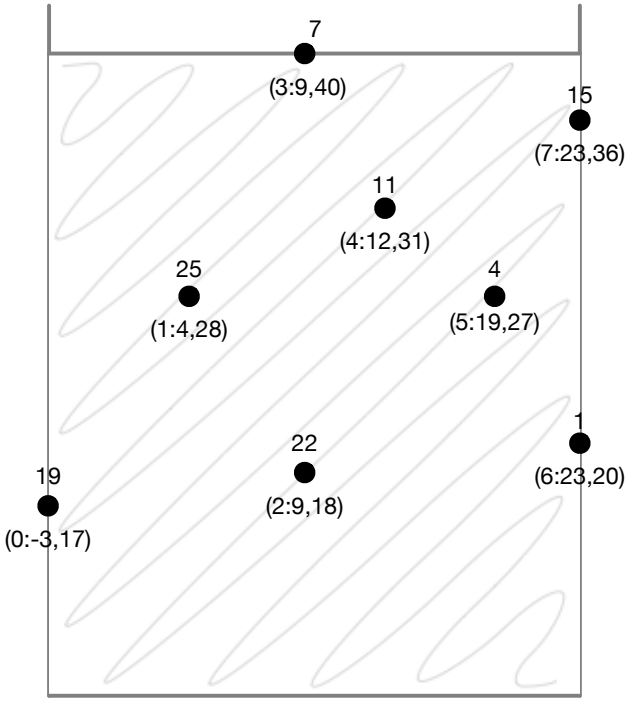
- Почему не классический Фенвик?
- $X_i, Y_i, A_i \leqslant 10^9$.
- Двумерный Фенвик потребует $O(10^{18})$ памяти и времени.


7
(10,40)
15
(23,36)
2
(28,36)
11
(12,31)
25
(4,28)
4
(19,27)
6
(40,23)
1
(23,20)
22
(9,18)
19
(-3,17)

Фенвик Фенвиков

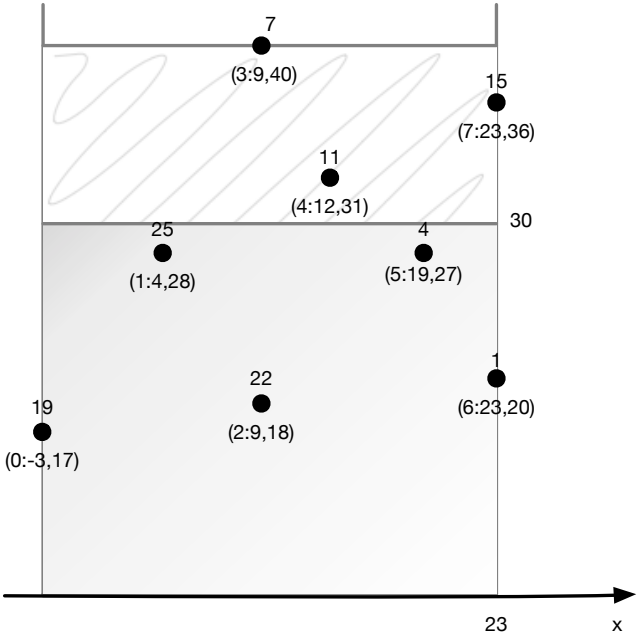
- Для решения требуется *сжатие координат*.
- Переведём систему координат $(X, Y), X, Y \leqslant 10^9$ в систему координат $(X', Y'), X', Y' < N$.
- Отсортируем массив пар X, Y сначала по X, затем по Y и удалим дубликаты. Отправим всё в A'.
- Для каждого значения i в A' построим полоски, содержащие точки с номерами от $f(i)$ до i.
- Каждая полоска — дерево Фенвика с операциями: 1) точечное обновление и 2) запрос суммы на отрезке.
- Для построения сортируем точки в полоске по Y и строим по ним деревья Фенвика.
- Всего полос — N.
- Матожидание количества точек в полосе — $\log N$.


7
(3:9,40)
15
(7:23,36)
2
(8:28,36)
11
(4:12,31)
25
(1:4,28)
[f(7)..7]
4
(5:19,27)
[f(9)..9]
6
(9:40,23)
22
(2:9,18)
1
(6:23,20)
19
(0:-3,17)


7
(3:9,40)
15
(7:23,36)
11
(4:12,31)
25
(1:4,28)
4
(5:19,27)
1
(6:23,20)
22
(2:9,18)
19
(0:-3,17)

Фенвик Фенвиков. Запрос на интервале

- Пришёл запрос: дай сумму в прямоугольнике (x, y).
- Находим бинпоиском первое число в A', не меньшее x. Пусть его индекс — k.
- Классическим Фенвиком дважды обрабатываем полоски $[f(k) \dots k]$, $[f(f(k)-1), f(k)-1], \dots$.
- Бинпоиском находим верхнюю точку полосы, ещё лежащую в
- Находим сумму на точек из множества, лежащих не выше максимального Y из полоски и не выше минимального Y из полоски.


7
(3:9,40)
15
(7:23,36)
11
(4:12,31)
30
25
(1:4,28)
4
(5:19,27)
1
(6:23,20)
22
(2:9,18)
19
(0:-3,17)
23
x

Фенвик Фенвиков: общая стоимость алгоритма

- Сжатие координат: $O(N \log N)$ по времени и $O(N)$ по памяти.
- Построение деревьев: $O(N \log^2 N)$ по времени и $O(N \log N)$ по памяти.
- Запрос на интервале:
  1. Поиск правой границы полосы: $O(\log N)$.
  2. Заход в каждую полосу $O(\log N)$ раз.
  3. В каждой полосе $O(\log N)$ поиск границы + $O(\log N)$ исполнение запроса.
  4. Итого по полосам $O(N \log^2 N)$. Это и есть время на запрос.

Частичное каскадирование.

Задача 4. Имеется K упорядоченных по неубыванию массивов произвольных размеров:

$$a_{11}, a_{12}, \ldots, a_{1N_1}$$
$$a_{21}, a_{22}, \ldots, a_{2N_2}$$
$$\ldots, \ldots, \ldots,$$
$$a_{K1}, a_{K2}, a_{KN_K}.$$

Требуется по данному числу X для каждого массива минимальный из элементов, не меньший, чем X.

Для $X = 14$

1	1	2	3	5	8	13	21	34	55	89

1	2	4	8	16	32	64	128

1	3	5	6	8	20	30	30	31	35	69	95	113

Задача массового поиска

- Пусть $N = \sum_{i=1}^{K} N_i$.

1. Легко найти решение задачи за время $O\left(\sum_{i=1}^{K} \log N_i\right) \leqslant O\left(K \log N\right)$. На это потребуется $O(N)$ памяти.
2. Построим BST из всех элементов всех массивов как ключей. В каждом узле храним множество из K элементов с ответами. Тогда время решения будет $O(\log N) + K)$, а память $O(KN)$.
3. Построим структуру данных, которая займёт память как первое решение, и будет требовать времени как второе решение — *слоистую таблицу частичного каскадирования.*

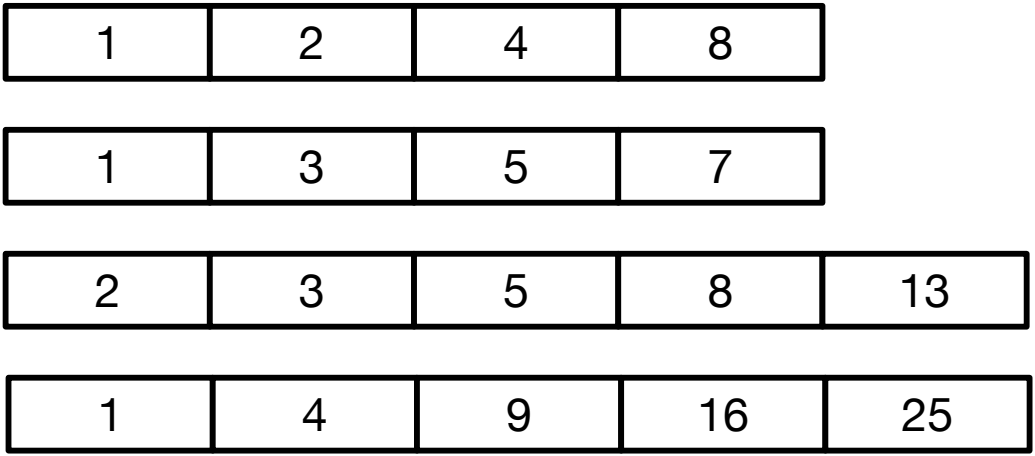
Частичное каскадирование: идеи

- Бинарный поиск в первом варианте в каждом массиве начинается с первого элемента и оканчивается последним.
- Мы можем завести ссылки между массивами, позволяющие сузить поиск.
- В массивы могут быть добавлены вспомогательные элементы. Будем называть модифицированные массивы *слоями*.

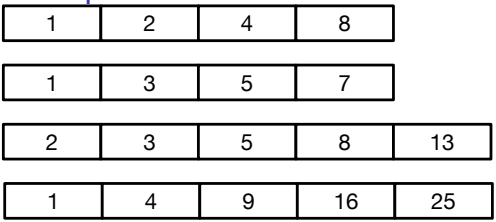
Частичное каскадирование: идеи

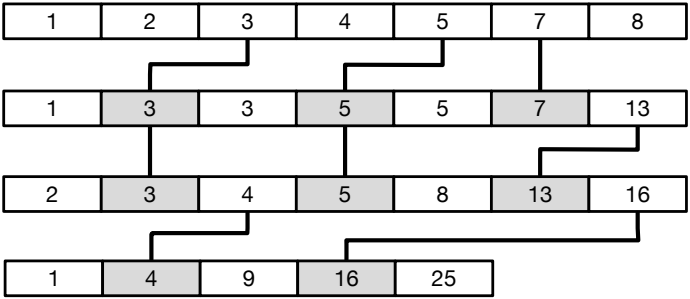
- Что за элементы можно добавлять? Элементы следующего или предыдущего слоя. Если сохранить их местонахождение в другом слое, можно сократить границы поиска.
- Сколько таких элементов добавить? Точно не все. Давайте ровно половину.
- Мы сливаем текущий слой с чётными элементами предыдущего, сохраняя оригинальные позиции элементов.
- При слиянии мы добавляем к элементу предпосчитанную информацию, где его можно найти в текущем слое и где — в предыдущем.
- Для реальных элементов будем сохранять ссылки на максимальный ссылочный, меньший текущего (слева) и на минимальный больше текущего (справа).
- Для ссылочных будем сохранять ссылки на реальные элементы слева и справа.

Подопытный кролик



Построение слоёв





Исполнение запроса

1. Часть 1. Верхний слой. Бинарным поиском находим в верхнем построенном слое нужный элемент.

1a. Этот элемент может быть реальным (есть ответ) или ссылочным.

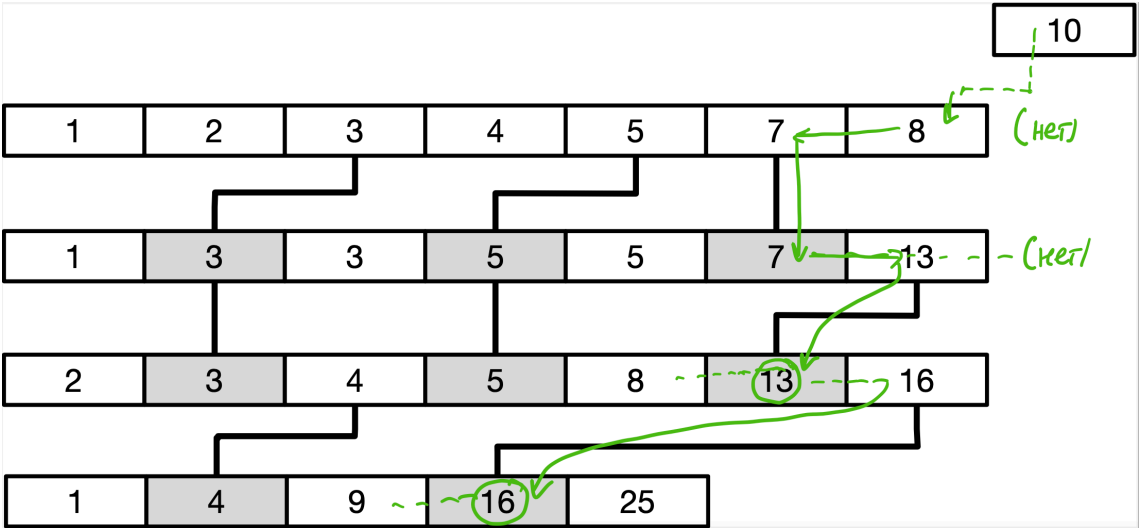
1b. Если элемент был ссылочный — идём по ссылке влево.

2. Этап 2. Спуститься вниз мы можем только из ссылочного. Идём влево и спускаемся.

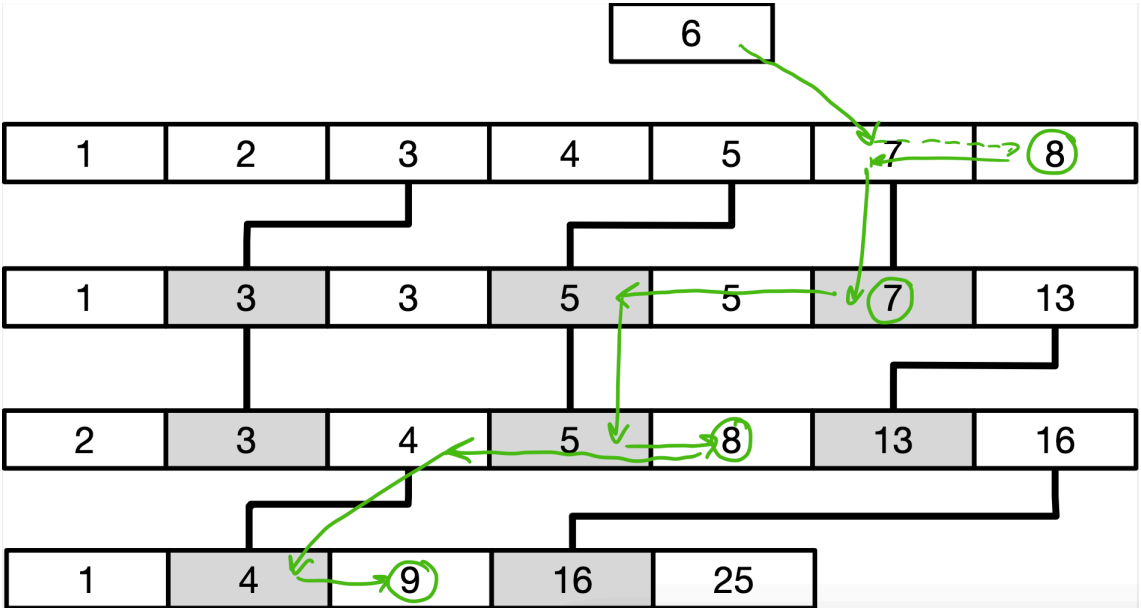
3. Если мы попали в реальный, то ответ может быть не далее, чем в двух элементах справа от спуска.

4. Если попали в ссылочный — отправляемся влево и повторяем этап 2 до нижнего слоя.

Поиск числа 10



Поиск числа 6



Сложность алгоритма

1. Построение слоёв требует $O(N)$ памяти.
   Последний слой $O(N_k)$, предпоследний $O(N_{k-1} + \frac{N_k}{2})$,
   первый: $O(N_1 + \frac{N_2}{2} + \frac{N_3}{4} + \cdots + \frac{N_k}{2^{k-1}})$.
   При суммировании слой k вносит не более $2N_k$ в сумму. Общая стоимость $O(N)$.
2. Поиск в первом слое требует $O(\log N)$, в каждом из остальных — $O(1)$. Итого на поиск $O(\log N + K)$.