Содержание

АТП Гос (ИВТ: Формалки + Оси)

Disclaymer: доверять этому конспекту или нет выбирайте сами

1. Алгоритмы и структуры данных

1.1. Динамический массив. Амортизационный анализ. Учётная оценка времени добавления элемента в динамический массив.

Определение 1.1.1: Пусть f, g – произвольные функции, тогда
$$f(x) = O(g(x)) \Leftrightarrow \exists M > 0 : \exists x_0 : \forall x \geq x_0 : |f(x)| \leq M|g(x)|$$

Определение 1.1.2: Пусть время n последовательных операций над некоторой структурой данных составит $t_1, ..., t_n$, тогда говорят, что **амортизированная стоимость** или **учётное время** операции составляет $t^* = \frac{1}{n}\sum_{i=1}^{n} t_i$.

Замечание 1.1.1: Далее, амортизированная сложность будет обозначаться со звездочкой, например, $O^*(1)$

Замечание 1.1.2 (Динамический массив): Интерфейс динамического массива:
- Обращение по индексу за $O(1)$
- Добавить элемент в конец за $O^*(1)$

Замечание 1.1.3 (Объяснение амортизационной сложности добавления): Очевидно, что при заполнении массива нам нужно делать реаллокацию. (Которая условно бесплатная, но копирование всех элементов на новое место – линейно).

Но, оказывается, если при каждой реаллокации мы будем увеличивать массив в два раза, то средняя стоимость добавления элемента в массив будет константной

Для доказательства будем использовать метод монеток.
- Пусть мы только что совершили реаллокацию – у нас есть $\frac{n}{2}$ свободного места и столько же уже добавленных элементов
- Пусть добавляем один из $\frac{n}{2}$ новых элементов – пусть его индекс $i \geq \frac{n}{2}$. Тогда, потратив монетку на добавление без реалокации, заложим по одной монетке на будущее копирование i-го и $i - \frac{n}{2}$-элемента.
- Таким образом, к заполнению буфера из n элементов, мы заложим по монетке за каждый элемент и после новой реалокации сможем условно бесплатно скопировать их на новое место, потратив монетки.

1.2. Связные списки. Стек, очередь, дека и их реализации

Замечание 1.2.1 (Список): Мы хотим от списка следующее:

Операция	**Время**
Вставка в известное место	O(1)
Удаление из известного места	O(1)
Поиск	O(N)
Обращение по индексу	O(N)

Наш список будет хранить цепочку из узлов, где каждый указывает на следующего за ним, а последний указывает в никуда.

Поиск и обращение по индексу требуют линейного прохода, но при этом вставка или удаление элемента – это создание узла и переприсвоение указателей.

Существует также двусвязный список – хранит в себе два указателя на узел после и позади нас. Благодаря дополнительному указателю, получаем возможность работы с обоими концами списка, не теряя в асимптотике.

Замечание 1.2.2 (Стек): Мы хотим от стека следующее:

Операция	**Время**
Вставка в начало	O(1)
Удаление из начала	O(1)
Узнать размер	O(1)

Стек можно реализовать на односвязном списке, однако для быстрого получения размера следует завести счётчик, изменяемый при вставке/удалении.

Замечание 1.2.3 (Очередь): Мы хотим от очереди следующее:

Операция	**Время**
Вставка в начало	O(1)
Удаление из конца	O(1)
Узнать размер	O(1)

Очередь тривиально реализуется на двухсвязном списке, однако есть способ реализовать её на двух стеках.

У нас будут два стека: входной и выходной. Вставка будет происходить в перывй, а удаление из второго. В случае удаление из пустого выходного стека требуется переложить все элементы из выходного стека в выходной (получим развёрнутый выходной стек).

Удаление из такой очереди уже будет $O^*(1)$, так как на каждый элемент хватит по 3 монетки – на его добавление, переброс в выходной стек и удаление.

Замечание 1.2.4 (Дека): Деку иногда называют двусторонним стеком или двусторонней очередью:

Операция	**Время**
Вставка в начало или в конец	O(1)
Удаление из начала или из конца	O(1)
Узнать размер	O(1)

Тривиально реализуется на двухсвязном списке.

1.3. Быстрая сортировка. Поиск порядковой статистики методом «Разделяй и властвуй»

Замечание 1.3.1 (QuickSort): Алгоритм быстрой сортировки:
- Случайно выбираем опорный элемент (pivot)
- Всё, что меньше опорного элемента перекинуть влево, а что больше – вправо
- Вызвать рекурсивно от правой и левой половины.

Минусы:
- Использует доппамять, так как нужно хранить стек рекурсии
- Под любую стратегию выбора опорного элемента можно построить контрданные так, чтобы работало квадратичное время с линейной доппамятью.

Теорема 1.3.1: Среднее время работы быстрой сортировки составляет $O(N \log N)$, если опорный элемент выбирается равновероятно.

Определение 1.3.1: **k-й порядковой статистикой** массива называют элемент, который после сортировки будет стоять на k-м месте.

Замечание 1.3.2 (QuickSelect): Алгоритм быстрого поиска:
- Выбрать опорный элемент
- Провести разбиение
- Если индекс опорного элемента окажется больше, чем k – то ищем слева k -ю статистику, иначе, справа – $k - i - 1$-ю.

Теорема 1.3.2: Среднее время работы алгоритма выше составляет $O(N)$, если опорный элемент выбирается равновероятно.

1.4. Сортировка слиянием. Поразрядные сортировки.

Замечание 1.4.1 (MergeSort): Воспользуемся стратегией разделяй и властвуй:
- Разбиить массив на две примерно равные половины
- Разбиваем массивы дальше пополам, пока не дойдём до массива из двух элементов, там всё тривиально
- Сольём два отсортированных массива в один большой

Два отсортированных массива сливаются методом двух указателей – $O(N)$ времени и доппамяти.

Для подсчёта общего времени работы рассмотрим дерево рекурсии. В нём $\log N$ уровней, при этом на каждом уровне после слияния отсортированная часть массива будет увеличиваться примерно в два раза.

Значит нам потребуется $\log N$ слияний – итоговое время $O(N \log N)$. А доппамять линейна.

Определение 1.4.1: Сортировка называется **стабильной**, если равные элементы относительно компаратора не поменяют своего взаимного расположения после сортировки.

Замечание 1.4.2 (Поразрядная сортировка): Рассмотрим сортировку подсчётом, считать будем в массиве P:

- Пройдёмся по исходному массиву и запишем в $P[i]$ число объектов с ключом (по чему сортируем) i
- Посчитаем префиксные суммы массива P, тогда мы знаем, начиная с какого индекса в сортированном массиве надо писать структуру с ключом i.
- Идём по изначальному массиву слева направо и вписываем каждый элемент в сортированный массив, согласно префиксным суммам.

Данная сортировка, очевидно, стабильна, однако при большом количестве разных чисел, расходы на массив подсчёта станут огромными.

Тогда научимся сортировать числа из большого диапазона! Рассмотрим массив произвольных uint32_t.

Давайте сортировать данные числа как двоичные строки, сначала сортируем по убыванию последнего байта, потом стабильно по убыванию второго байта etc. Работает это за $O(Nk)$ времени, где k - это число байтов, то есть константа. Массив подсчёта, очевидно, будет небольшим - 256.

1.5. Двойная куча и сортировка кучей. Слияние k отсортированных массивов с помощью кучи.

Определение 1.5.1: Дерево называется **подвешенным**, если у него есть выделенная вершина, называемая **корнем** дерева.

Определение 1.5.2: Дерево называется **бинарным**, если у каждой вершины степень не более трёх, то есть один родитель и не более двух детей.

Определение 1.5.3: Бинарное дерево называется **полным**, если все его уровни полностью заполнены.

Замечание 1.5.1 (Бинарная пирамида): Хотим такую структуру:

Операция	**Время**
Добавление элемента	$O(\log N)$
Удаление минимума	$O(\log N)$
Чтение минимума	$O(1)$

Бинарную пирамиду будем реализовывать на полном бинарном дереве, при этом будем поддерживать **свойство пирамиды** – все сыновья вершины строго больше её самой.

Тогда, очевидно, в корне всегда будет минимум, который и будем брать за константу.

Определим две вспомогательные операции – просеивание вниз и просеивание вверх. Просеивание вниз старается «утопить» элемент как можно ниже, меняя его местами с меньшим из сыновыей, не нарушая свойство пирамиды.

Для вставки будем хранить помимо указателя на корень ещё и указатель на самый правый узел, у которого меньше двух сыновый. Тогда подвесим новый элемент к этому узлу и потом просеим его вверх.

Если же таких узлов нет, то весь нижний уровень занят, а значит подвесим новый элемент к самому левому узлу и также просеим вверх.

Для удаления поменяем местами корень и крайний узел, а затем просеим вниз элемент в корне, а лист с минимумом просто удалим.

Замечание 1.5.2 (HeapSort): Для сортировки массива построим на нём пирамиду и N раз извлечём минимум. Итоговое время $O(N \log N)$.

Замечание 1.5.3 (Слияние k отсортированных массивов): Обобщение метода двух указателей:

- Из каждого из k массивов берём по первому элементу, добавляем в кучу
- Берём из кучи минимальный, он будет следующим элементов слитого массива, добавляем в кучу следующий элемент из исходного массива только что взятого элемента.
- Повторяем до опустошения всех массивов

Очевидно, сложность $O(Nk \log k)$

1.6. Хэш-таблица, полиномиальная хэш-функция

Определение 1.6.1: Пусть U – множество рассматриваемых объектов, тогда $h : U \to \{0, 1, ..., k-1\}$ называется **хэш-функцией**

Определение 1.6.2: Элементы $x, y \in U, x \neq y$ образуют **коллизию**, если $h(x) = h(y)$

Определение 1.6.3: Заведём массив размера k и скажем, что элемент v есть в множестве, если по индексу $h(v)$ уже занята ячейка.

Такая схема называется **прямой адресацией**.

Определение 1.6.4: **Бакетом** называют нечто, хранящее все элементы, образующие коллизию.

Замечание 1.6.1 (Хэш-таблица на цепочках): На практике, элементы, образующие коллизию, добавляются в бакет, например, список.

Давайте посчитаем средний размер бакета:

Считаем, что хэш-функция отображает из $\{0, 1, ..., n-1\}$ в $\{0, 1, ..., k-1\}$, а вероятность коллизиции равномерная:

$$P(\{h(x) = h(y)\}) = \begin{cases} 1, x = y \\ \frac{1}{k}, x \neq y \end{cases}$$

Тогда для произвольного бакета, соответствующего некоторому элементу $q \in \{0, ..., n\}$, матожидание его размера:

$$\mathbb{E}L_q = \mathbb{E}\sum_{i=1}^{n} \mathbb{I}(\{h(q) = h(i)\}) =$$

$$P(\{h(q) = h(q)\}) + \sum_{i=1, i\neq q}^{n} P(\{h(q) = h(i)\}) = 1 + \frac{n-1}{k} \leq 1 + \frac{n}{k}$$

Определение 1.6.5: **Коэффициентом загруженности** называют величину $\alpha = \frac{n}{k}$.

Заметим, что мы знаем эту величину уже при определении хэш-функции, а значит стоимость всех операций заведомо ограничивается некоторой константой, зависимой от α.

Замечание 1.6.2 (Полиномиальное хеширование): Приведём пример некоторой хэш-функции в отношении строк. Для строки p размера m такой хэш определён следующий образом:

$$h(p[1:m]) = \left(\sum_{i=1}^{m} p[i]x^{i-1}\right) \bmod q$$

где q – некоторое простое число, а x – число от 0 до $q-1$.

1.7. Динамическое программирование: общая идея, линейная динамика, матричная, динамика на отрезках

Определение 1.7.1: **Динамическое программирование** – способ решения алгоритмических задач, который основывается на переходе от частного к общему.

Замечание 1.7.1 (Нахождение наибольшей возрастающей подпоследовательности): Построим массив d, где $d[i]$ – это длина наибольшей возрастающей подпоследовательности, оканчивающейся в элементе с индексом i.

Пусть искомая последовательность – a.

Массив будет заполняться следующим образом:

$$d[i] = \begin{cases} 1, i{=}0 \\ 1 + \max_{j=0\ldots i-1; a[j]<a[i]} d[j] \end{cases}$$

Причём в случае максимума по пустому множеству будет возвращаться 0.

Данная задачка является примером линейного ДП с квадратичной сложностью.

Замечание 1.7.2 (Нахождение наибольшей общей подпоследовательности): Построим матрицу d, где $d[i, j]$ – это длина наибольшей общей подпоследовательности, оканчивающаяся в первой последовательности на i-м индексе, а во второй – на j-м.

Пусть искомые последовательность – s_1, s_2.

Тогда заполним матрицу:

$$d[n_1, n_2] = \begin{cases} 0, n_1{=}0 \vee n_2{=}0 \\ d[n_1{-}1, n_2{-}1]{+}1, s_1[n_1{-}1]{=}s_2[n_2{-}1] \\ \max(d[n_1{-}1, n_2], d[n_1, n_2{-}1]), \text{ else} \end{cases}$$

Данная задачка является примером матричного ДП с квадратичной сложностью.

Замечание 1.7.3 (ДП на подотрезках): В рассмотренных задачах у нас была «индукция» по индексам массива или матрицы, однако часто можно встретить «индукцию» по длинам подотрезков.

Например, пусть есть строка S и надо найти длину максимальной подпоследовательности палиндрома.

Пусть $f(l, r)$ – ответ для подстроки $S[l : r + 1]$, тогда сразу определим базу $f(l, l) = 1$ и $f(l, r) = 0, r < l$.

Сразу определимся, что ответом будет $f(0, |S| - 1)$.

Теперь научимся пересчитывать ответ. У нас могут быть две ситуации:

- Если $S[l] = S[r]$, то нам выгодно добавить эти символы к итоговому палиндрому, а значит $f(l, r) = f(l + 1, r - 1) + 2$.
- Если же $S[l] \neq S[r]$, то просто прокинем максимум из подстрок $f(l, r) = \max(f(l + 1, r), f(l, r - 1))$.

1.8. RMQ. Sparse table. Дерево отрезков.

Замечание 1.8.1 (Разреженная таблица): Разреженная таблица – двумерная структура данных $ST[i][j]$, построенная на бинарной, коммутативной, ассоциативной и идемпотентной операции F, для которой

$$ST[i][j] = F(...F(F(A[i], A[i+1]), ...), ..., A[i+2^j-1]); j \in [0...\log N]$$

Объём памяти, занимаемый таблицей, равен $O(N \log N)$, и заполненными являются только те элементы, для которых $i + 2^j \leq N$.

Теперь заметим, что для отрезка $[l, r]$ верно, что

$$F(A[l], ..., A[r]) = F(ST[l][j], ST[r-2^j+1][j]), j = \lfloor \log_2(r-l+1) \rfloor$$

Определение 1.8.1: **RMQ или range min query** – запрос минимума на подотрезке. Решается с помощью Sparse Table с $F = \min$

Замечание 1.8.2 (Дерево отрезков): Дерево отрезков способно за $O(\log n)$ получать на подотрезке результат любой операции, которая ассоциативна, коммутативна и имеет нейтральный элемент.

Опишем нерекурсивной построение дерева на массиве длины N. Для удобства будем считать, что $\log_2 N \in \mathbb{N}$, иначе дозаполним до степени двойки нейтральными элементами.

Теперь заведём массив длины $2N-1$ и будем его заполнять таким образом, что последние N элементов будут элементами исходного массива, а первые $N-1$ элементов заполним, как $t[i] = F(t[2i+1], t[2i+2])$.

Для обновления листа $i \geq \frac{N}{2}$ обновим его самого, а потом будем подниматься по дереву (На каждой итерации делаем $i' = \lfloor \frac{i}{2} \rfloor$) и пересчитаем все значения, зависимые от изменённого.

1.9. LCA. Двоичные подъёмы и сведение к RMQ

Определение 1.9.1: Пусть дано дерево T, подвешенную за вершину r.

Тогда назовём **наименьшим общим предком (LCA)** двух вершин u, v такую вершину $X = \text{LCA}(u, v)$, то она лежит на путях $u \to r, v \to r$, при этом такая вершина глубже всех подходящих.

Замечание 1.9.1 (Метод двоичных подъёмов): Сделаем предпосчёт двумерной матрицы $dp[v][v]$ – номер вершины, в которую мы придём, если пройдём из вершины v вверх по дереву 2^i шагов.

Пусть номер родители вершины v — $p[v]$, её глубина – $d[v]$, причём если v – корень, то $p[v] = v$.

Тогда для dp есть рекуррентная формула:

$$dp[v][i] = \begin{cases} p[v], i=0 \\ dp[dp[v][i-1]][i-1], i>0 \end{cases}$$

Заметим, что $i \leq \log_2 n$, так как иначе мы остаёмся в корне. Значит матрица занимает $O(N \log N)$ памяти.

Пусть $c = \text{LCA}(v, u), u \neq v$, тогда по определению $c \leq \min(d[v], d[u])$. Пусть $d[u] < d[v]$, тогда нам надо подняться вверх на $d[v] - d[u]$ шагов, для этого разобьём данную разность на степени двойки и поднимемся за $O(\log N)$.

Теперь будем считать, что $d[u] = d[v], u \neq v$. Остаётся тривиальный параллельный подъём до общего предка не более чем на $\log_2 N$ вверх.

Замечание 1.9.2 (Сведение LCA к RMQ): Заметим, что если w – LCA вершин u и v, то DFS из корня посетит сначала w, потом БОО зайдёт в u, затем снова посетит w и аналогично с v.

Поэтому нам достаточно знать для каждой вершины время её посещения обходом в глубину и её высоту:

- Массив Order, где $\text{Order}[i]$ – номер вершины, посещённой в момент времени i
- Массив высот h, где $h[i]$ – высота вершины $\text{Order}[i]$
- Массив First, где $\text{First}[v]$ – момент времени, когда вершина v была посещена впервые

Тогда для $\text{LCA}(u, v)$ нужно найти $\text{Order}[\min(h[\text{First}[u] : \text{First}[v]])]$, задачу поиска минимума на подотрезке как раз поможет решить RMQ.

1.10. Двоичное дерево поиска. Обходы в глубину и в ширину. Поиск ключа, наивные вставка и удаление ключа. АВЛ-дерево.

Определение 1.10.1: **Деревом поиска** называют дерево, в котором в поддереве левее все элементы не больше элемента в данном узле, а в поддереве правее – строго больше.

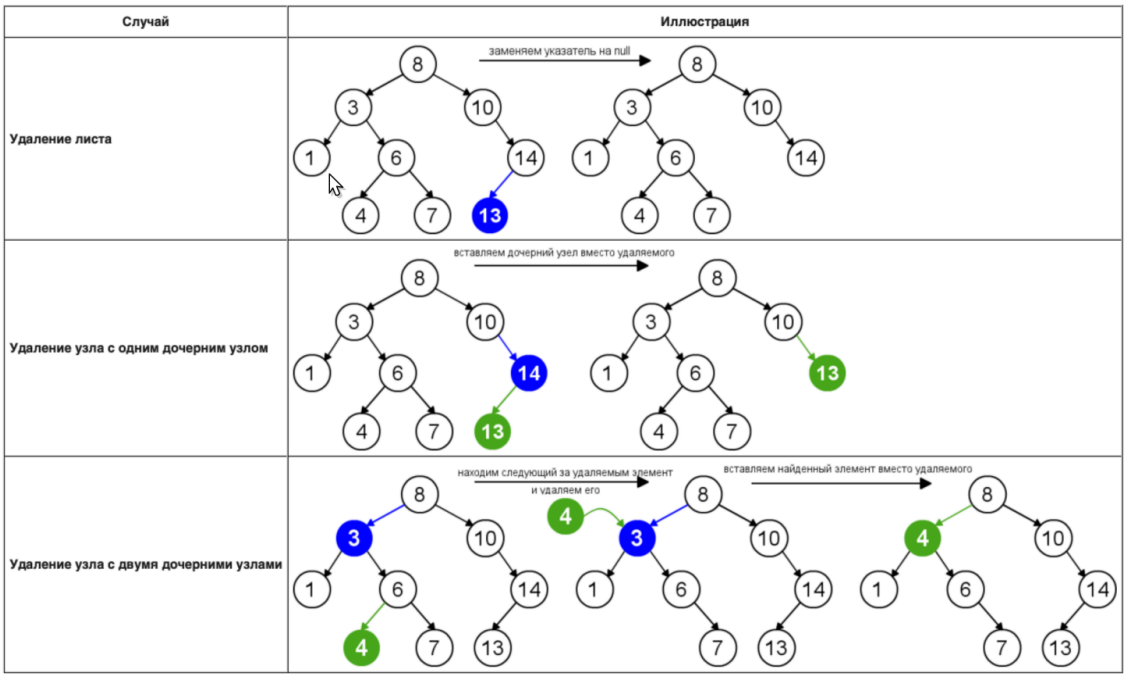
Пусть h – высота дерева, тогда мы хотим

Операция	Время
Вставка элемента	$O(h)$
Поиск элемента	$O(h)$
Удаление элемента	$O(h)$

Поиск в таком дереве тривиальный – начинаем с корня, если искомый ключ меньше – идём влево, иначе – вправо. Если элемент есть – мы его найдём, иначе – попытаемся зайти в несуществующий узел.

Для вставки мы пойдём аналогично поиску, но когда дойдём до несуществующего узла – то добавим его на это место.

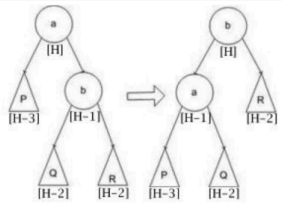
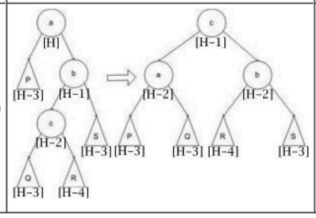
Иллюстрация удаления:



Определение 1.10.2: Дерево поиска является **AVL-деревом**, если для каждой вершины высота её правого и левого поддеревьев различаются не более, чем на единицу.

Теорема 1.10.1: Высота AVL-дерева равна $O(\log N)$.

Определение 1.10.3: **Балансировкой** AVL-дерева будем называть процесс, который из дерева поиска, в узле которого разных высот поддедревьев равна двум, переподвешивает детей и внуков так, чтобы выполнялось свойство AVL-дерева.

Тип вращения	Иллюстрация	Когда используется	Расстановка балансов
Малое левое вращение		$diff[a] = -2$ и $diff[b] = -1$ или $diff[a] = -2$ и $diff[b] = 0$.	$diff[a] = 0$ и $diff[b] = 0$ $diff[a] = -1$ и $diff[b] = 1$
Большое левое вращение		$diff[a] = -2, diff[b] = 1$ и $diff[c] = 1$ или $diff[a] = -2, diff[b] = 1$ и $diff[c] = -1$ или $diff[a] = -2, diff[b] = 1$ и $diff[c] = 0$.	$diff[a] = 0, diff[b] = -1$ и $diff[c] = 0$ $diff[a] = 1, diff[b] = 0$ и $diff[c] = 0$ $diff[a] = 0, diff[b] = 0$ и $diff[c] = 0$

Симметрично определяются два правых поворота.

После каждой операции изменения дерева, запускаем балансировку от нового узла до корня, причём баланс уже сбалансированных поддеревьев никак не изменится.