

Содержание

1. Алгоритмы и структуры данных	2
1.1. Динамический массив. Амортизационный анализ. Учётная оценка времени добавления элемента в динамический массив.	2
1.2. Связные списки. Стек, очередь, дека и их реализации	3
1.3. Быстрая сортировка. Поиск порядковой статистики методом «Разделяй и властвуй»	4
1.4. Сортировка слиянием. Поразрядные сортировки.	5
1.5. Двойная куча и сортировка кучей. Слияние k отсортированных массивов с помощью кучи.	6
1.6. Хэш-таблица, полиномиальная хэш-функция	7

АТП Гос (ИВТ: Формалки + Оси)

Disclaimer: 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, \dots, 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 реаллокации, 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 реаллокации 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 первый, а удаление из второго. В случае удаления 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 допамять, так как нужно хранить стек рекурсии
- Под любую стратегию выбора опорного элемента можно построить контр-примеры так, чтобы работало квадратичное время с линейной допамятью.

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:

- Разбить 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 допамяти.

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 допамять линейна.

Определение 1.4.1: Сортировка называется **стабильной**, если равные элементы относительно компаратора не меняют 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 отсортированном массиве надо писать структуру с ключом i .
- Идём по изначальному массиву слева направо и вписываем каждый элемент в отсортированный 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 сыновей, не нарушая свойство пирамиды.

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

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

Для удаления поменяем местами корень и крайний узел, а затем просеем 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 $h : U \rightarrow \{0, 1, \dots, k-1\}$ называется **хэш-функцией**

Определение 1.6.2: Элементы $x, y \in U, x \neq y$ образуют **коллизия**, 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 коллизия.

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

Давайте посчитаем средний размер бакета:

Считаем, что хэш-функция отображает из $\{0, 1, \dots, n-1\}$ в $\{0, 1, \dots, k-1\}$, а вероятность коллизии 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 $q \in \{0, \dots, n\}$, 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 загрузки** 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, зависящей от α .

Замечание 1.6.2 (Полиномиальное хеширование): Приведём пример некоторой хэш-функции в отношении строк. Для строки p размера m такой хэш определён следующим 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 $q-1$.