Оглавление

1. Понятие алгоритма **5**
1.1. Машина с произвольным доступом к памяти . . . 5
1.2. Алгоритмический язык . . . 6
1.3. Определение алгоритма . . . 7
1.4. O-символика . . . 8
1.5. Числа Фибоначчи . . . 10

2. Структуры данных **15**
2.1. Массивы . . . 15
2.1.1. Расширяемый массив с аддитивной схемой выделения . . . 16
2.1.2. Расширяемый массив с мультипликативной схемой выделения. . . . 17
2.2. Списки . . . 18
2.2.1. Односвязный список . . . 18
2.2.2. Двусвязный список . . . 18
2.2.3. Сравнение с массивами . . . 19
2.3. Абстрактные типы данных . . . 19
2.3.1. Стек . . . 20
2.3.2. Очередь . . . 21
2.4. Двухсторонняя очередь . . . 23
2.5. Амортизационный анализ . . . 23
2.5.1. Метод учётных стоимостей . . . 23
2.5.2. Метод предоплаты . . . 28

3. Разделяй и властвуй **29**
3.1. Переход к подзадачам . . . 29
3.2. Двоичный поиск . . . 29
3.3. Умножение n-битовых чисел . . . 30
3.4. Сортировка слиянием . . . 30
3.5. Основная теорема о рекуррентных соотношениях . . . 32
3.5.1. Как побороть аддитивные константы? . . . 32
3.6. Умножение матриц . . . 32

4. Сортировка **33**
4.1. Квадратичные сортировки . . . 33
4.2. Нижняя оценка на сортировку сравнениями . . . 34
4.3. Сортировка кучей . . . 35
4.4. Быстрая сортировка . . . 36
4.4.1. Анализ времени работы . . . 36
4.4.2. Сортировка массива с повторяющимися элементами . . . 37
4.5. Линейные сортировки . . . 37

4.5.1. Сортировка подсчётом . . . 37
4.5.2. Цифровая сортировка . . . 38
4.5.3. Блочная сортировка . . . 38

5. Задачи RMQ и LCA **39**
5.1. Задача о минимуме на отрезке . . . 39
5.1.1. Динамическая задача RMQ . . . 39
5.1.2. Статическая задача RMQ . . . 45
5.2. Задача о наименьшем общем предке . . . 46

6. Деревья поиска **51**
6.1. АТД с быстрым поиском . . . 51
6.2. Представление корневых деревьев в памяти . . . 52
6.3. Двоичные деревья поиска . . . 53
6.3.1. Операции поиска . . . 53
6.3.2. Добавление элемента . . . 54
6.3.3. Удаление элемента . . . 55
6.4. АВЛ-дерево . . . 57
6.4.1. Сбалансированность . . . 57
6.4.2. Вращения . . . 57
6.5. Splay-дерево . . . 59
6.5.1. Запросы к splay-дереву . . . 59
6.5.2. Операция `splay` . . . 61
6.5.3. Анализ сложности . . . 64
6.6. Декартово дерево . . . 67
6.6.1. Определение и теорема существования . . . 67
6.6.2. Эффективное построение . . . 68
6.6.3. Операции . . . 69
6.6.4. Дуча . . . 71
6.7. B-деревья . . . 71

7. Хеширование **73**
7.1. Прямая адресация . . . 73
7.2. Хеш-таблица . . . 73
7.2.1. Хеш-функция . . . 74
7.2.2. Методы разрешения коллизий . . . 74
7.2.3. Примеры хеш-функций . . . 75
7.2.4. Детали реализации . . . 77
7.3. Гипотеза равномерного хеширования . . . 77
7.4. Универсальное хеширование . . . 79
7.5. Совершенное хеширование . . . 82

8. Числовые алгоритмы **85**
8.1. Арифметические операции с n-битовыми числами . . . 85
8.2. Модульная арифметика . . . 89

9. Альтернативные модели вычисления **93**
9.1. Работа с большими данными . . . 93
9.1.1. Модель внешней памяти . . . 93
9.1.2. Модель cache-oblivious . . . 95
9.2. Параллельные и распределённые вычисления . . . 96

9.2.1. Модель PRAM 96
9.2.2. Модель BSP 97
9.3. Другие модели 101

A. Математические факты **103**
A.1. Суммирование последовательностей 103

Глава 1

Понятие алгоритма

Перед тем, как начать говорить об алгоритмах, нужно сначала разобраться с самим понятием алгоритма. Все мы имеем какие-то общие представления о том, что собой представляют алгоритмы обычной в жизни. Наиболее яркий и распространённый пример алгоритма — это кулинарные рецепты, которые описывают процесс приготовление блюд, разбивая его не *базовые операции*, вроде «посолить», «нарезать», «вскипятить». Для того, чтобы сформулировать понятие алгоритма нам нужно договориться о том, какие операции мы будем называть базовыми. Другими словами, нам нужно описать то вычислительное устройство, для которого мы будем разрабатывать алгоритмы.

1.1. Машина с произвольным доступом к памяти

Математическое описание вычислительного устройства называется *моделью вычисления*. Вы уже могли слышать про такие известные модели, как «машина Тьюринга», «лямбда-исчисление», «алгорифмы Маркова» и другие. В рамках этого курса мы почти всё время будем работать с моделью вычисления *машина с произвольным доступом к памяти (random access memory model, RAM-машина)*. Если не сильно увлекаться формализацией, то можно считать, что речь идёт о компьютере с одноядерным процессором и прямым доступом к оперативной памяти. Для нашего удобства и простоты мы даже откажемся от устройства ввода-вывода и будем предполагать, что входные данные изначально записаны в оперативной памяти, и решение задачи тоже нужно записать в заранее определённое место в памяти. Оперативная память в этой модели состоит из ячеек одинакового размера, пронумерованных целыми числами. Слова «произвольный доступ» в названии этой модели означают, что процессор может обратиться к любой ячейке оперативной памяти по её номеру, причём это всегда делается за одну операцию (один такт процессора), т.е. время доступа к ячейке не зависит от её номера. [1]

Одна ячейки оперативной памяти современного компьютера обычно имеет размер 32 или 64 битов. Так как большинство результатов в теории алгоритмов являются асимптотическими, то нам будет удобно считать, что размер ячейки не является фиксированным числом, а зависит от некоторого параметра n, соответствующего размеру решаемой задачи (обычно в качества n будет выступать

[1] В современных процессорах всё устроенно несколько сложнее. Для оптимизации доступа к оперативной памяти в процессоре имеется несколько уровней кешей. Обращение к оперативной памяти происходит не напрямую, а через кеш: для того, чтобы обратиться к ячейке, которой в кеше нет, её нужно сначала загрузить в кеш. Поэтому доступ к ячейкам, которые в данный момент присутствуют в кеше, происходит значительно быстрее, чем к тем, которых в данный момент в кеше нет.

размер входных данных). Это может показаться странным, т.к. получается, что модель вычисления зависит от задачи, которую мы решаем. На это лучше смотреть следующим образом: если параметр задачи равен n, то нам гарантируется, что ячейка памяти имеет размер $w \geq \lceil \log_2(n) \rceil$ битов. Такого количества битов будет достаточно, чтобы представить любое целое число от 0 до n.[2]

Следующий набор операций в этой модели мы будем называть *базовыми* и считать, что каждая из них выполняется 1 такт процессора.

1. Обращение к ячейке памяти по её номеру (чтение или запись).

2. Логические операции с булевыми значениями.

3. Управляющие операции: условный переход, вызов функции, возврат из функции, прерывание из цикла и т.п.

4. Арифметические операции с w-битовыми целыми числами: сравнение, сложение, вычитание, умножение, деление и взятие остатка по целому модулю.

5. Битовые операции w-битовыми целыми числами: инверсия всех битов, побитовые AND, OR и XOR, а также битовые сдвиги влево и вправо на любое число $k \leq w$.

6. Операции с вещественными числами с ограниченной точностью: сравнение, сложение, вычитание, умножение, деление, а также приведение (округление) к целому числу.[3]

В предложенной модели мы будем называть *указателем* переменную, хранящую номер ячейки памяти. Кроме того, мы договоримся, что есть специальное значение для указателя (нулевой указатель), которое не соответствует никакой ячейке памяти.

1.2. Алгоритмический язык

Для записи алгоритмов нам потребуется выбрать некоторый довольно богатый язык программирования. В данном курсе по причине простоты синтаксиса в качестве такого языка выбрано некоторое подмножество языка Python, которое содержит базовые синтаксические конструкции и работу с массивами

Важная оговорка. Из соображений простоты синтаксиса мы будем предполагать, что встроенные списки языка Python на самом деле являются массивами, т.е. следующий код определяет массив из 10 элементов.

```
primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

[2] В литературе такую версию машины с произвольным доступом называют также называют “word RAM” (word random access machine).

[3] Тут нужно быть аккуратным и сразу договориться, что на самом деле мы будем работать не с вещественными числами, а с их приближениями с ограниченной точностью, например, представляя вещественные числа как числа с плавающей точкой или числами с фиксированной точностью. Если разрешить работать с настоящими вещественными числами, то получившаяся модель вычисления будет позволять, например, сохранить данные неограниченного размера в одной ячейке памяти, представив их как дробную часть вещественного числа.

Мы будем считать, что `primes` является “классическим” массивом, т.е. элементы находятся в памяти последовательно друг за другом и, соответственно, по номеру элемента массива легко вычислить номер ячейки памяти, где хранится этот элемент. На практике интерпретаторы Python этого не гарантируют. Поэтому, если в реальной программе требуется работа с “настоящими” массивами, то приходится использовать внешние библиотеки, например, библиотеку `numpy`.

1.3. Определение алгоритма

Определение 1.3.1. *Алгоритмом* для решения задачи P будем называть программу, записанную на алгоритмическом языке, которая получает на вход условие задачи P и вычисляет ответ для это задачи. Мы будем говорить, что алгоритм *корректен*, если для любого условия задачи P алгоритм находит верное решение.

Так как в описании модели вычисления мы отказались от устройства ввода-ввывода, то все алгоритмы мы будем записывать как функции, которые получают входные данные в качестве аргументов и возвращают ответ.

Для каждого алгоритма, рассматриваемого в этом курсе, мы будем пытаться отвечать на следующие три вопроса.

1. Корректен ли этот алгоритм?
2. Какова сложность этого алгоритма?
3. Можно ли решить эту задачу быстрее?

В тех случаях, когда ответ на третий вопрос будет отрицательным, мы будем говорить, что рассматриваемый алгоритм является «оптимальным».

Пример: проверка наличия элемента в массиве. Давайте разработаем алгоритм, который проверяет, встречается ли заданный элемент в некотором массиве. Договоримся, что алгоритм получаем искомый элемент `el` и массив `arr`, как аргументы функции. Наиболее естественный подход к решению этой задачи — последовательно проверять все элементы массива `arr` на равенство `el` до первого совпадения. Эта идея реализуется функцией `has_element`.

```
# проверка наличия элемента el в массиве arr
def has_element(arr, el):
    for x in arr:
        if x == el:
            return True

    return False
```

Будем предполагать, что массив `arr` и переменная `el` содержат целые числа, кроме того мы обозначим длину массива `arr` через n. Попробуем оценить количество базовых операций, которое совершает данный алгоритм. Количество итераций в цикле не превосходит n. На каждой итерации происходит обращение к двум ячейкам памяти, сравнение, условный переход, увеличение счётчика для перехода к следующей ячейки или возвращение значения. Таким образом, общее число операций не превосходит $5n + 1$ (единичка добавляется для возврата значения в случае, если элемент в массиве отсутствует).

Такой точный анализ числа базовых на практике имеет очень мало смысла. Дело в том, что при реализации алгоритма на настоящем компьютере реальное количество тактов процессора на входе размера n будет зависеть от множества дополнительных факторов, таких как архитектура процессора, язык программирования, версия и параметры компилятора и/или интерпретатора, используемой операционной системы и т.д. Поэтому нас интересует какой-то способ сравнивать эффективность алгоритмов, который бы не зависел от подобных факторов. Поэтому вместо точной оценки количества операций мы будем оценивать *скорость роста* количества операций в зависимости от размера входных данных. В данном случае в качестве размера входных данных выступает число n, поэтому количество операций *линейно* возрастает при увеличении n. В таких случаях мы будем говорить, что мы имеем дело с *линейным алгоритмом* и обозначать его сложность $O(n)$.

Перед тем, как начать разбираться с формальными определениями для *O-символики*, нужно отметить ещё один важный момент. При оценке сложности функции `has_element` мы оценивали количество операций в самом *худшем случае*, т.е. тогда, когда в массиве `arr` нет элемента `el`, и функции придётся проверить весь массив, чтобы убедиться в этом. В дальнейшем мы всегда, если не указано обратное, под *сложностью алгоритма* будем понимать количество операций в худшем случае.

1.4. O-символика

В следующих определениях функции $f, g : \mathbb{N} \to \mathbb{R}_{>0}$ (т.е. функции,)

Определение 1.4.1. Будем говорить, что функция f *растёт не быстрее* g и обозначать $f(n) = O(g(n))$ или $f \preceq g$, если существует такая константа $c > 0$, что $f(n) \leq c \cdot g(n)$ для всех натуральных n.

Пример 1.4.1. $3n^2 + 4n + 5 = O(n^2)$. Для того, чтобы это показать, нам нужно предъявить подходящую константу c. В данном случае достаточно выбрать $c = 12$, поскольку при всех $n \geq 1$ выполнено $3n^2 + 4n + 5 \leq 3n^2 + 4n^2 + 5n^2 = 12n^2$.

Свойства

1. Для любых константы $a > 0$ верно $a \cdot f(n) = O(f(n))$, т.е. можно отбрасывать константные множители.

   Пример 1.4.2. $10n + 1500 = O(n)$.

2. Для любой константы $a \geq 0$ верно $a = O(1)$.

   Пример 1.4.3. $10^{100} = O(1)$.

3. Для любых $a \leq b$ выполняется $n^a = O(n^b)$.

   Пример 1.4.4. $n^2 = O(n^3)$.

4. Все логарифмы с основаниями больше единицы растут одинаково быстро: для любых $a, b > 1$ выполняется $\log_a n = O(\log_b n)$. Поэтому, основание логарифма в O-символике обычно не указывают.

   Пример 1.4.5. $\log_{16} n = O(\log n)$.

5. Любой полилогарифм растёт медленнее любого полинома: для любых $a, b > 0$ выполняется $\log^a n = O(n^b)$.

   Пример 1.4.6. $\log^{100} n = O(n^{0.01})$.

6. Любой полином растёт медленнее любой экспоненты: для любых $a > 0$ и $b > 1$ выполняется $n^a = O(b^n)$.

   Пример 1.4.7. $n^{100} = O(1.01^n)$.

7. Сумма функций растёт не быстрее, чем самое большое слагаемое: если $f(n) = O(g(n))$, то $f(n) + g(n) = O(g(n))$.

   Пример 1.4.8. $18n^3 + 12n^2 + 5n + 100 + 35\log_2(n) = O(n^3)$.

Упражнение 1.4.1. Приведите формальные доказательства для каждого из этих свойств.

Замечание 1.4.1. Обозначение $f(n) = O(g(n))$ не совсем корректно с точки зрения математики. Дело в том, что знак равенства в таком выражении не обозначает равенство в его обычном смысле. Например, из того, что $2n = O(n)$ и $3n = O(n)$ не следует, что $2n = 3n$. Для того, чтобы разобраться, почему так происходит, нужно задаться вопросом, а какие математические объекты стоят по обеим сторонам «равенства» $f(n) = O(g(n))$? Следуя определению слева стоит функция $f : \mathbb{N} \to \mathbb{R}_{>0}$. А какой математический объект скрывается за $O(g(n))$? Под $O(g(n))$ понимается класс всех функций, которые растут не быстрее g. Таким образом, математически корректно было бы написать $f(n) \in O(g(n))$, т.е. функция f принадлежит классу (множеству) всех функций, которые растут не быстрее g. Однако в теории алгоритмов (и не только в ней) принято писать $f(n) = O(g(n))$ и мы тоже будем этому следовать.

Преимущества O-символики. Использование O-символики позволяет охарактеризовать зависимость времени работы алгоритма от размера входных. Получаемые оценки устойчивы к изменению таких факторов, как язык программирования или параметры компиляции. Кроме того, сами утверждения про сложность алгоритмов носят более простой характер (сложность алгоритма оценивается как $O(n^2)$ вместо $3n^2 + 4n + 5$), и значительно упрощается анализ (не нужно скрупулёзно подсчитывать количество операций, достаточно выделить наиболее трудоёмкие).

Недостатки O-символики. При использовании O-символики алгоритмы с одинаковыми оценками сложности на практике могут иметь значительно различающуюся производительность. Представим себе два алгоритма, имеющие сложность $O(n)$, такие что первый делает порядка $2n$ операций, а второй — $2000n$ операций. В терминах O-символики эти алгоритмы имеют одинаковую сложность, но на практике второй алгоритм значительно проигрывает первому. Более того, нередко бывает, что, например, для некоторой задачи существует линейный алгоритм, но на практике используются простой алгоритм с оценкой $O(n \log n)$, который имеет лучшую производительность (это объясняется тем, что константа скрытая в $O(n)$ значительно больше, чем $\log n$ для характерных значений n, встречающихся в реальных задачах).

Связанные определение. Введённое выше определение O-символики позволяет сравнивать две функции на «меньше или равно». В некоторых случаях нам потребуются следующие связанные определения, которые задают все оставшиеся сравнения между функциями.

Определение 1.4.2. Будем говорить, что функция f *растёт не медленнее* g, и обозначать $f(n) = \Omega(g(n))$ или $f \succeq g$, если существует такая константа $c > 0$, что $f(n) \geq c \cdot g(n)$ для всех натуральных n.

Определение 1.4.3. Будем говорить, что функция f *растёт так же как* g, и обозначать $f(n) = \Theta(g(n))$ или $f \asymp g$, если существуют две константы $c_1, c_2 > 0$, такие что $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ для всех натуральных n. Другими словами, $f(n) = \Theta(g(n))$, если одновременно выполняется $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.

Следующие два определения задают строгие отношения, соответствующие «меньше» и «больше».

Определение 1.4.4. Будем говорить, что функция f *растёт медленнее* g и обозначать $f(n) = o(g(n))$, или $f \prec g$, если $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Определение 1.4.5. Будем говорить, что функция f *растёт быстрее* g и обозначать $f(n) = \omega(g(n))$, или $f \succ g$, если $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$, т.е. $g(n) = o(f(n))$.

Применимость такого анализа на практике.

1.5. Числа Фибоначчи

Определение 1.5.1. *Числа Фибоначчи* — это бесконечная числовая последовательность целых чисел $F_0, F_1, F_2, F_3, \ldots$, определённая по следующим правилам:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}.$$

Таким образом $F_2 = F_1 + F_0 = 1$, $F_3 = F_2 + F_1 = 2$, $F_4 = F_3 + F_2 = 3$ и т.д.

Можно выписать начало этой последовательности:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Насколько быстро члены этой последовательности возрастают? Следующие две леммы показывают, что числа Фибоначчи возрастают экспоненциально.

Лемма 1.5.1. *Для любого* $n \geq 6$, $F_n \geq 2^{n/2}$.

Доказательство. Докажем индукцией по n.

1. **База.** $F_6 = 8 \geq 2^3 = 2^{6/2}$, $F_7 = 13 \geq 12 = 2^3 \cdot 1.5 > 2^3 \cdot \sqrt{2} = 2^{7/2}$.
2. **Предположение.** Предположим, что $F_n \geq 2^{n/2}$ для всех $n < k$ для некоторого $k \geq 2$.
3. **Переход.** Докажем, что $F_k \geq 2^{k/2}$. По предположению $F_{k-1} \geq 2^{(k-1)/2}$ и $F_{k-2} \geq 2^{(k-2)/2}$, следовательно

$$F_k = F_{k-1} + F_{k-2} \geq 2^{(k-1)/2} + 2^{(k-2)/2} \geq 2 \cdot 2^{(k-2)/2} = 2^{k/2}.$$

□

Лемма 1.5.2. *Для любого $n \geq 0$, $F_n \leq 2^n$.*

Доказательство. Докажем индукцией по n.

1. **База.** $F_0 = 0 \leq 2^0$, $F_1 = 1 \leq 2^1$.
2. **Предположение.** Предположим, что $F_n \leq 2^n$ для всех $n < k$ для некоторого $k \geq 2$.
3. **Переход.** Докажем, что $F_k \leq 2^k$. По предположению индукции $F_{k-1} \leq 2^{k-1}$ и $F_{k-2} \leq 2^{k-2}$, следовательно,

$$F_k = F_{k-1} + F_{k-2} \leq 2^{k-1} + 2^{k-2} \leq 2 \cdot 2^{k-1} = 2^k.$$

□

Следствие 1.5.1. *Для любого $n \geq 6$ выполняется*

$$2^{n/2} \leq F_n \leq 2^n.$$

Числа Фибоначчи очень хорошо изучены, в том числе известна и точная оценка $F_n = \Theta(\varphi^n)$, где $\varphi = \frac{1+\sqrt{5}}{2}$ — золотое сечение. Эта оценка следует из формулы Бине для вычисления чисел Фибоначчи:

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} = \frac{\varphi^n - (-\varphi)^{-n}}{\varphi - (-\varphi)^{-1}} = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}.$$

Доказывать эту формулу не входит в наши цели. Более того, если мы захотим использовать эту формулу для вычисления чисел Фибоначчи, то нам потребуются работать с числами с плавающей точкой, а следовательно придётся как-то оценивать погрешность. В результате окажется, что использовать данную формулу можно только для относительно небольших n.

Вместо это давайте напишем простую функцию, которая вычисляет F_n по определению. Так как числа Фибоначчи определяются через рекуррентно, то естественно вычислять числа их при помощи рекурсии.

```python
# рекурсивная реализация вычисления чисел Фибоначчи
def fib_rec(n):
    if n == 0:
        return 0

    if n == 1:
        return 1

    return fib_rec(n-1) + fib_rec(n-2)
```

Структура этой функции дословно повторяет определение чисел Фибоначчи, поэтому более-менее понятно, что алгоритм корректен. Тем не менее, давайте приведём *формальное доказательство корректности*. В данном случае корректность алгоритма означает, что для целого положительного n функция `fib_rec(n)` возвращает F_n.

Лемма 1.5.3. *Для любого целого $n \geq 0$ функция* `fib_rec(n)` *возвращает F_n.*

Доказательство. Докажем индукцией по n.

1. **База.** Для $n = 0$ и $n = 1$ функция возвращает соответственно F_0 и F_1.
2. **Предположение.** Предположим, что для всех n меньших некоторого $k \geq 2$ функция `fib_rec(n)` возвращает F_n.
3. **Переход.** Докажем, что `fib_rec(k)` возвращает F_k. По предположению `fib_rec(k-1)` и `fib_rec(k-2)` возвращают соответственно F_{k-1} и F_{k-2}. Следовательно `fib_rec(k)` возвращает $F_{k-1} + F_{k-2} = F_k$.

□

Разобравшись с корректностью этого алгоритма мы можем попробовать оценить время его работы. Пусть $T(n)$ — это количество операций, которые которые требуются для вычисления `fib_rec(n)`. Тогда $T(0) = 1$ (одно сравнение), $T(1) = 2$ (два сравнения), а $T(n) = T(n-1) + T(n-2) + 3$ (два сравнения и сложение). Полученное соотношение очень похоже на определение чисел Фибоначчи. Покажем по индукции, что $T(n) \geq F_n$ для любого n.

1. **База.** $T(0) \geq F_0$, $T(1) \geq F_1$.
2. **Предположение.** Предположим, что это верно для всех n меньше некоторого $k \geq 2$.
3. **Переход.** Докажем, что $T(k) \geq F_k$. По предположению $T(k-1) \geq F_{k-1}$ и $T(k-2) \geq F_{k-2}$, следовательно

$$T(k) = T(k-1) + T(k-2) + 3 \geq F_{k-1} + F_{k-2} + 3 > F_k.$$

Таким образом получается, что количество операций, которые нужны функции `fib_rec` для вычисления F_n не меньше F_n, т.е. $T(n) = \Omega(2^{n/2})$. Это время работы совершенно неприемлемо, т.к. для вычисления всего лишь F_{100} нам потребуется более $2^{50} > 10^{15}$ операций (более 10 дней на процессоре с частотой 1GHz).

Замечание 1.5.1. Мы здесь оценили время работы предложенного алгоритма *снизу*. Можно аналогичным способом по индукции доказать, что $T(n) \leq 2F_{n+1}$ и таким образом получить верхнюю оценку $T(n) = O(2^n)$, но для данного алгоритма это не очень интересно, т.к. мы уже знаем, что он очень медленный.

Для того, чтобы разобраться, почему этот естественный алгоритм работает так медленно, давайте нарисуем схему рекурсивных вызовов функции `fib_rec`.

На этой схеме видно, что для вычисления F_n мы сначала рекурсивно вычисляем F_{n-1} и F_{n-2}. При этом для вычисления F_{n-1} мы снова вычисляем F_{n-2}. Здесь и кроется проблема этого алгоритма: мы не запоминаем, что уже однажды вычислили F_{n-2}, и вычисляем его заново. На схеме видно, что F_{n-3} вычисляется уже трижды, F_{n-4} — пять раз, и т.д. Из этого анализа возникает естественная идея, что мы должны запоминать те числа Фибоначчи, которые мы уже вычислили. Это приводит нас к следующему алгоритму вычисления чисел Фибоначчи с сохранением результатов в массиве.

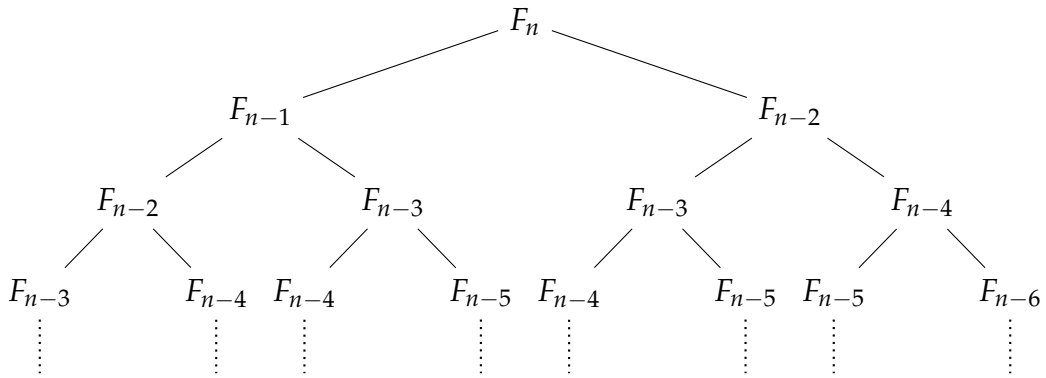


Рис. 1.5.1. Схема рекурсивных вызовов `fib_rec(n)`.

```
# вычисления числа Фибоначчи с сохранением результатов в массиве
def fib_array(n):
    # изначально в массиве хранятся F_0 и F_1
    a = [0, 1]
    for i in range(2, n+1):
        # добавляем в массив F_i = F_{i-1} + F_{i-2}
        a.append(a[i-1] + a[i-2])

    # возвращаем F_n
    return a[n]
```

Если более детально проанализировать алгоритм `fib_array`, то можно заметить, что на каждой итерации мы пользуемся только двумя самыми последними элементами массива. Это приводит нас к мысли, что достаточно сохранить только два последних элемента. В этом случае вместо массив мы заведём две дополнительные переменные.

```
# вычисления числа Фибоначчи с двумя дополнительными переменными
def fib_vars(n):
    if n == 0:
        return 0

    # изначально переменные хранят F_0 и F_1
    a, b = [0,1]
    for i in range(2, n+1):
        a, b = b, b + a

    return b
```

Полученный алгоритм так же работает за $O(n)$, но в отличие от `fib_array` использует константное количество ячеек памяти.

Замечание 1.5.2. Вычисление можно ещё ускорить, если записать вычисление числа Фибоначчи на языке матриц и воспользоваться алгоритмом быстрого возведения в степень. В результате можно получить алгоритм со сложностью $O(\log n)$ использующий $O(1)$ ячеек памяти.

Глава 2

Структуры данных

Для эффективной работы с данными используются различные способы организации их хранения в памяти — *структуры данных*. В данной главе мы поговорим про самые базовые структуры данных, а так же научимся оценивать амортизированную сложность *запросов* к ним[1].

2.1. Массивы

Наиболее простой структурой данных для RAM-машины является массив. Массив представляется собой *непрерывную* область памяти, разбитую на ячейки одинакового размера. Зная адрес начала массива можно легко вычислить адрес произвольной ячейки. Языки программирования обычно уже имеют готовые функции для работы с массивами, но нам было бы полезно разобраться, как они устроены.

Если нам изначально известно, сколько элементов будет хранится в массиве, или же у нас есть некоторая оценка сверху на количество элементов, то мы можем сразу же *выделить* массив достаточного размера (т.е. договориться, что некоторая непрерывная область памяти будет использоваться для хранения элементов массива). Для добавления элемента в конец заранее выделенного массива достаточно просто скопировать его в ещё незаполненную ячейку и увеличить счётчик, который хранит количество заполненных ячеек. Таким образом, сложность запроса на добавление элемента — $O(1)$ операций. Запрос на удаление последнего элемента реализовать ещё проще, т.к. нам достаточно просто уменьшить счётчик числа заполненных ячеек.

Замечание 2.1.1. Здесь и далее при оценке сложности запросов к структурам данных мы всегда будем предполагать, что размер элементов структуры данных фиксирован и не зависит от размера входа. Другими словами, мы будем предполагать, что сложность копирования любого элемента структуры данных константна, т.е. стоит нам $O(1)$ операций. Если это не так, то в большинстве случаев полученные оценки сложности легко скорректировать, домножив их на сложность копирования одного элемента. Например, если в массиве хранятся строки длины $\sqrt{n}$, то оценки сложности всех запросов, которые приводят к копированию элементов, нужно умножить на $\sqrt{n}$: добавление элемента в конец будет иметь сложность $O(\sqrt{n})$, а удаление последнего элемента — $O(1)$, т.к. не требует копирования элементов.

[1] Для того, чтобы избежать путаницы мы будем говорить об *операциях* процессора и *запросах* к структурам данных.

Если мы захотим удалить или добавить элемент в начале или где-то в середине массива, то нам потребуется "сдвинуть" все последующие элементы, поэтому эти запросы будут иметь сложность $O(n)$, где n — это количество элементов в массиве.

Однако, мы далеко не всегда заранее знаем, сколько элементов будет хранится в массиве. В этом случае можно начать с небольшого массива и расширять его постепенно, добавляя новые ячейки тогда, когда они потребуются. Предположим, что все ячейки массива заполнены, а нам нужно добавить к массиву ещё один элемент? Вполне возможно, что ячейки памяти, которые расположены сразу же после конца массива, свободны. Тогда мы могли бы расширить массив за счёт присоединения к нему необходимого количество ячеек памяти (например, это может сделать функция `realloc` в языке C). Но что делать, если ячейки памяти после конца массива уже заняты или наш язык программирования не поддерживает расширение массивов? В этом случае вместо расширения существующего массива нам придётся создать новый массив большего размера и скопировать туда все данные из исходного массива (массив обязательно должен лежать в непрерывной области памяти, поэтому мы не можем расположить дополнительные ячейки где-то отдельно). Давайте рассмотрим две различные реализации этой идеи.

2.1.1. Расширяемый массив с аддитивной схемой выделения

Предположим, что нам нужно добавить один новый элемент в конец массива размера s, который уже полностью заполнен, т.е. хранит s элементов. В этом случае мы можем выделить новый массив размера $s + c$, где c — некоторая целая положительная константа, скопировать туда все элементы из исходного массива и добавить новый элемент в первую из добавленных пустых ячеек (см. рис. 2.1.1). Это называют *расширяемым массивом с аддитивной схемой выделения*. Какую слож-

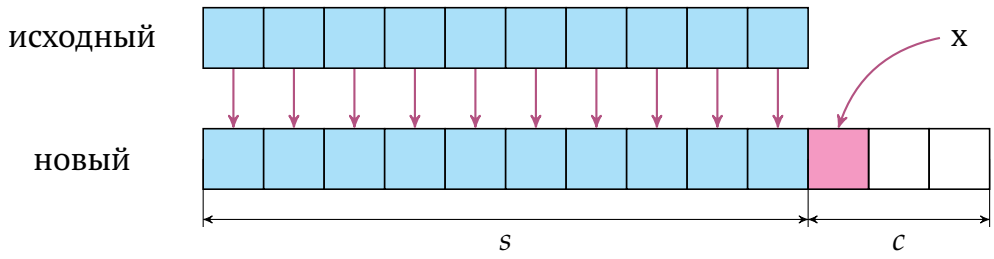


Рис. 2.1.1. Добавление элемента x в заполненный расширяемый массив с аддитивной схемой выделения при $c = 3$.

ность будет иметь запрос на добавление элемента в при такой реализации расширения? Пусть в массиве n элементов. Если свободных ячеек нет, то при добавлении нового элемента нам потребуется скопировать n элементов из исходного массива и один новый элемент. Таким образом запрос на добавление элемента будет иметь сложность $\Theta(n)$.

Полезно также оценить суммарные накладные расходы на расширение массива. Пусть мы начали с пустого массива, и в процессе работы он увеличился до размера n. Сколько дополнительных операций потребовалось? Для простоты будем учитывать только операции копирования элементов. При первом нетривиальном расширении массива было скопировано c элементов, при втором — $2c$ элементов, и так далее. В сумме получается, что на все расширения массива по-

требовалось

$$c + 2c + 3c + \cdots + (n - c) = \frac{n(n/c - 1)}{2} = \Theta(n^2)$$

дополнительных копирований (см. лемму A.1.1). Получается, что при такой реализации расширяемого массива любой алгоритм, который использует массив размера n, имеет сложность $\Omega(n^2)$.

2.1.2. Расширяемый массив с мультипликативной схемой выделения.

Вместо увеличения размера массива на константу, можно увеличивать размер массива *в константу раз*. Эта идея называется *расширяемым массивом с мультипликативной схемой выделения*. Выберем некоторое $\alpha > 1$ (наиболее типичным значением является $\alpha = 2$). Если в текущем массиве размера s все ячейки заполнены, то выделим новый массив размера $\lceil \alpha s \rceil$ (см. рис. 2.1.2).

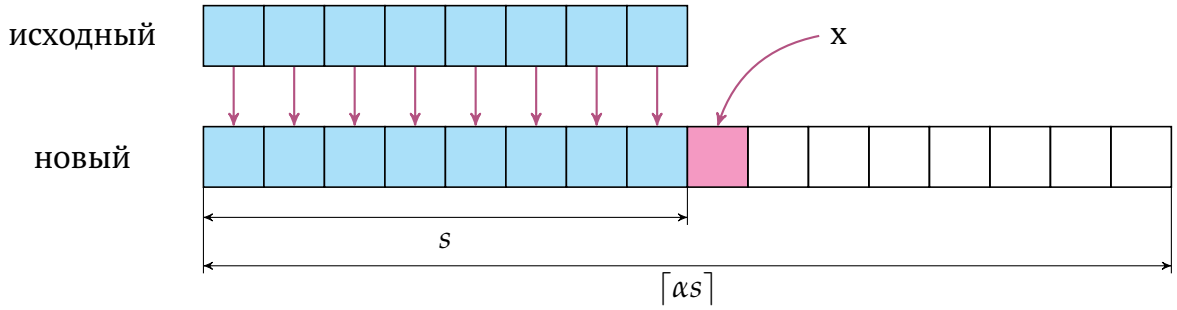


Рис. 2.1.2. Добавление элемента x в заполненный расширяемый массив с мультипликативной схемой выделения при $\alpha = 2$.

Давайте оценим также суммарные накладные расходы на расширение массива. Пусть мы начали с массива некоторого фиксированного размера, и в процессе работы размер увеличился до n. При последнем расширении массива было скопировано $\lfloor n/\alpha \rfloor$ элементов, при предпоследнем — $\lfloor n/\alpha^2 \rfloor$, и так далее. В сумме на расширения массива потребовалось $\lfloor n/\alpha \rfloor + \lfloor n/\alpha^2 \rfloor + \cdots + c$ дополнительных копирований, где c — начальный размер массива. Эту сумму можно оценить сверху суммой бесконечной убывающей геометрической прогрессии (см. лемму A.1.3)

$$\lfloor n/\alpha \rfloor + \lfloor n/\alpha^2 \rfloor + \cdots + c \le \sum_{k=1}^{\infty} \lfloor n/\alpha^k \rfloor < \sum_{k=1}^{\infty} n/\alpha^k = n \cdot \frac{1}{1 - 1/\alpha} = \Theta(n).$$

Таким образом при мультипликативной схемой выделения накладные расходы линейные. Это можно переформулировать следующим образом: в среднем на каждый элемент массива требуется $O(1)$ дополнительных операций на расширение массива. Стоит отметить, что не смотря на это, запрос на добавление в массив будет иметь сложность $\Theta(n)$ в худшем (сложность в худшем оценивает количество операций в самом “плохом” случае).

Упражнение 2.1.1. Как реализовать запрос на удаление элемента из массива так, чтобы в среднем на каждый элемента массива дополнительно требовалось $O(1)$ операций, и при этом свободное место использовалось бы экономно (т.е., например, если из большого массива удалить почти все элементы, то зарезервированное свободное место должно пропорционально уменьшиться)?

2.2. Списки

Другая базовая структура данных — это список. В отличие от массива, где все данные хранятся в непрерывной области памяти, элементы списка могут быть разбросаны по памяти произвольным образом. Выделяют два основных вида списков: *односвязный* и *двусвязный*.

2.2.1. Односвязный список

Каждый элемент *односвязного списка* хранится в отдельной структуре с двумя полями — *узле списка*. Кроме самого элемента каждый узел хранит *указатель* на следующий узел — номер ячейки памяти, где находятся данные следующего узла списка (см. рис. 2.2.1). Односвязный список задаётся парой указателей: на первый узел списка, *голову списка*, и на последний узел списка, *хвост списка*.

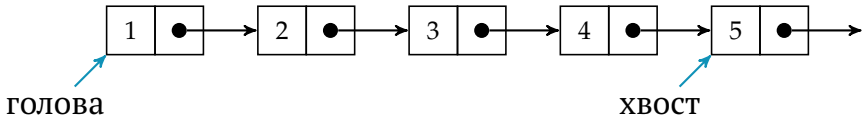


Рис. 2.2.1. Односвязный список.

Такая организация хранения данных позволяет перебирать элементы от головы к хвосту списка, причём переход к следующему элементу выполняется за $O(1)$, удалять и добавлять элементы, находящиеся в начале списка, за $O(1)$, добавлять элементы в конец списка за $O(1)$, а так же удалять и добавлять элементы в произвольном месте списка за $O(1)$ по ссылке на предыдущий элемент см. рис. 2.2.2. При этом обращение по индексу и удаление из конца списка будут работать за $\Theta(n)$ (для удаления последнего элемента нужно найти предпоследний узел, а для этого придётся пройти весь список от головы до хвоста).

Кроме того, списки позволяют эффективно осуществлять некоторые более сложные операции. Например, можно «вырезать» часть списка и выделить её в отдельный список или наоборот «вклеить» один список внутрь другого, причём обе операции будут работать за $O(1)$, т.к. они требуют изменения всего лишь пары указателей.

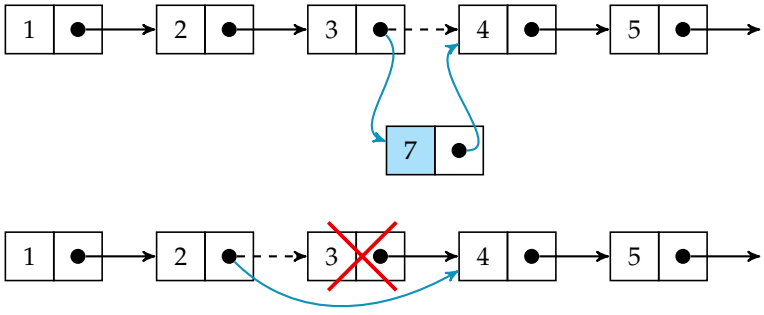


Рис. 2.2.2. Добавление и удаление элемента односвязного списка.

2.2.2. Двусвязный список

Если есть необходимость перебирать элементы списка в обоих направлениях, то применяется *двусвязный список*. Элементы двусвязного списка хранятся в уз-

лах вместе с указателями на следующий и предыдущий узлы списка (см. рис. 2.2.3). За счёт хранения двух указателей двусвязный список позволяет добавлять и удалять элементы в произвольном месте списка за $O(1)$. При этом доступ к элементу по индексу так же осуществляется за $\Theta(n)$.

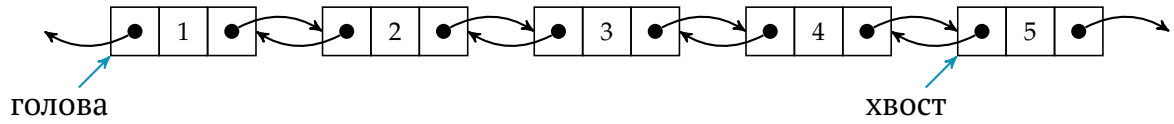


Рис. 2.2.3. Двусвязный список.

По функциональности двусвязный список ни в чём не уступает односвязному, в т.ч. так же позволяет эффективно реализовать операции «вырезания» и «вклеивания». Тем не менее, в тех приложениях, где память очень ограничена (например, внутри кода драйверов или во встроенных системах), предпочитают использовать односвязные списки, если их возможностей достаточно для решения поставленной задачи.

Упражнение 2.2.1. Придумайте, как можно хранить двусвязный список так, чтобы в каждом узле хранился только один указатель, но при этом мы могли бы перебирать элементы списка в обоих направлениях. (Подсказка: нужно использовать битовые операции с целыми числами.)

2.2.3. Сравнение с массивами

В следующей таблице представлены сложности основных запросов к массивам и спискам.

Запрос	**Массив**	**Односвязный список**	**Двусвязный список**
Обращение по номеру	$O(1)$	$O(n)$	$O(n)$
Добавление в начало	$O(n)$	$O(1)$	$O(1)$
Удаление из начала	$O(n)$	$O(1)$	$O(1)$
Добавление в конец	$O(n)$	$O(1)$	$O(1)$
Удаление из конца	$O(1)$	$O(n)$	$O(1)$
Добавление в середину	$O(n)$	$O(1)$	$O(1)$
Удаление из середины	$O(n)$	$O(1)$	$O(1)$

Таблица 2.1. Сравнение сложности запросов к массивам и спискам.

2.3. Абстрактные типы данных

Определение 2.3.1. *Абстрактный тип данных (АТД)* — это математическое описание структуры данных в терминах реализуемых ей запросов.

Понятия абстрактного типа данных и структуры данных соотносятся так же, как в объектно-ориентированном программировании соотносятся понятия интерфейса и класса. Когда мы определяем структуру данных, то мы подробно описываем то, как эта структура устроена, и какие запросы к этой структуре можно реализовать эффективно. При определении абстрактного типа данных мы описываем только те запросы, которые этот тип данных должен реализовывать, но

ничего не говорим про его устройство в памяти и реализацию запросов. Таким образом, у одного абстрактного типа данных может быть несколько реализаций, основанных на разных структурах данных. В этом разделе мы рассмотрим два базовых абстрактных типа данных и поговорим о том, как их можно реализовать на основе массивов и списков.

2.3.1. Стек

Абстрактный тип данных *стек* определяется следующими запросами:

- `push(x)` — добавить элемент в стек,
- `pop()` — вытолкнуть элемент стека.

Порядок, в котором элементы выталкиваются из стека, является обратным порядку добавления, т.е. первым выталкивается тот элемент, который был добавлен последним (см. рис. 2.3.1). Про стек говорят, что он реализует концепцию «последним пришёл — первым ушёл» («last in, first out», LIFO). В обычной жизни такой порядок можно встретить, например, если положить несколько книг в стопку на стол: сверху будет лежать та книга, которую мы положили последней. В соответствии с этой аналогией про элемент, который добавлен последним, говорят, что он находится *на вершине* стека, а про тот, что был добавлен первым — что он находится *на дне* стека.

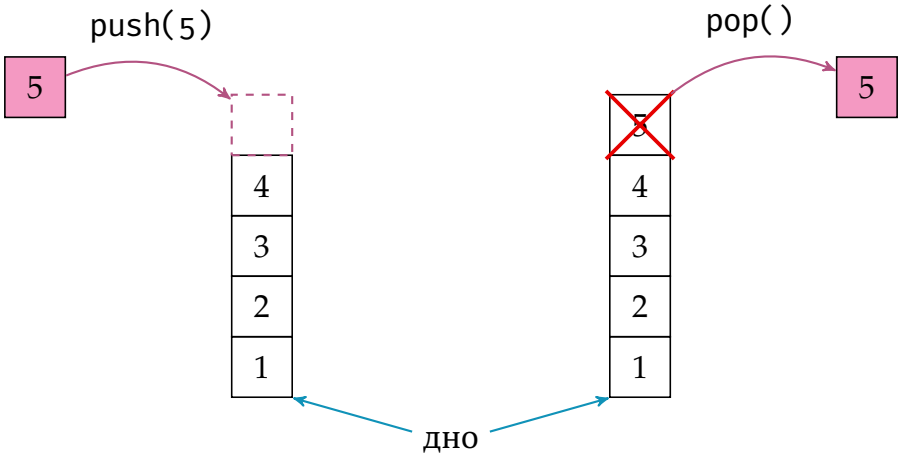


Рис. 2.3.1. Операции со стеком: добавление и выталкивание элемента.

Реализация на списках

Стек эффективно реализуется на односвязном списке, а следовательно и на двухсвязном: элементы добавляются и выталкиваются из начала списка.

Реализация на массиве

При эффективной реализации стека на массиве элементы добавляются в конец массива и выталкиваются из конца массива. Если массив заполняется полностью, то происходит его расширение по мультипликативной схеме.

2.3.2. Очередь

Абстрактный тип данных *очередь* определяется следующими запросами:

- `enqueue(x)` — добавить элемент `x` в очередь,
- `dequeue()` — извлечь элемент из очереди.

Как и в обычной жизни, порядок, в котором элементы извлекаются из очереди, должен соответствовать порядку, в котором они очередь добавляются. Про очередь говорят, что она реализует концепцию «первым пришёл — первым ушёл» («first in, first out», FIFO).

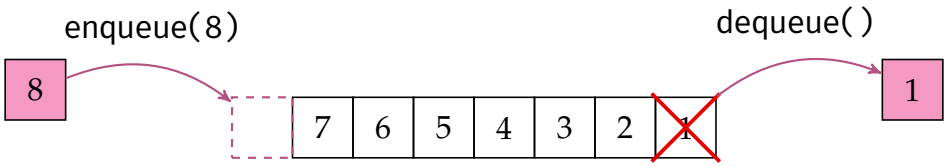


Рис. 2.3.2. Операции с очередью.

Реализация на списках

Очередь можно эффективно реализовать на двусвязном списке, т.к. он позволяет добавлять элементы в начало и удалять из конца за $O(1)$. Если немного подумать, то становится ясно, что очередь можно эффективно реализовать и на односвязном списке — для этого нужно добавлять элементы в конец списка, а удалять из начала.

Реализация на массиве

Для эффективной реализации очереди на массиве требуется использовать идею *кольцевого буфера*. Для этого в массиве поддерживаются два указателя: на начало очереди и на конец очереди (см. рис. 2.3.3). Элементы добавляются по указателю на конец очереди, выталкиваются по указателю на начало. При этом, если указатель достигает конца массива, то он «перескакивает по циклу» на начало массива. Если после добавления элемента в очередь указатели на начало и конец встретились, т.е. весь массив заполнен элементами очереди, то массив расширяется по мультипликативной схеме, и элементы очереди копируются в новый массив от начала до конца.

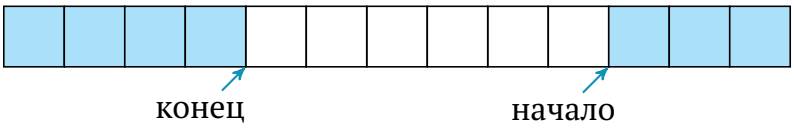


Рис. 2.3.3. Реализация очереди на массиве с использованием кольцевого буфера.

Реализация на двух стеках

В некоторых случаях можно описать реализацию одного абстрактного типа данных через другой абстрактный тип данных. В частности в некоторых алгоритмах используется следующий способ реализации очереди на двух стеках. Пусть далее s_1 и s_2 — это два стека, на основе которых мы должны реализовать запросы enqueue и dequeue. Будем предполагать, что стеки реализуют запросы push, pop и также запрос empty, который возвращает True, если стек пуст. Реализация запроса enqueue будет затрагивать только первый стек.

```
def enqueue(x):
    s1.push(x)
```

При запросе dequeue проверяется, есть ли элементы во втором стеке. Если второй стек пуст, то мы перекладываем элементы первого стека во второй, и после этого выталкиваем верхний элемент из второго стека (см. рис. 2.3.4).

```
def dequeue(x):
    if s2.empty():
        while not s1.empty():
            s2.push(s1.pop())

    return s2.pop()
```

При такой реализации очереди запрос dequeue будет иметь сложность $\Theta(n)$ в худшем (при условии, что стек реализован эффективно). Однако, можно показать, что суммарные накладные расходы на все запросы dequeue так же будут линейными. Для этого нужно заметить, что для каждого элемента очереди будет сделано не более двух запросов push и не более двух запросов pop. Следовательно, сложность любой последовательности запросов к такой очереди для n элементов ограничена $O(n)$.

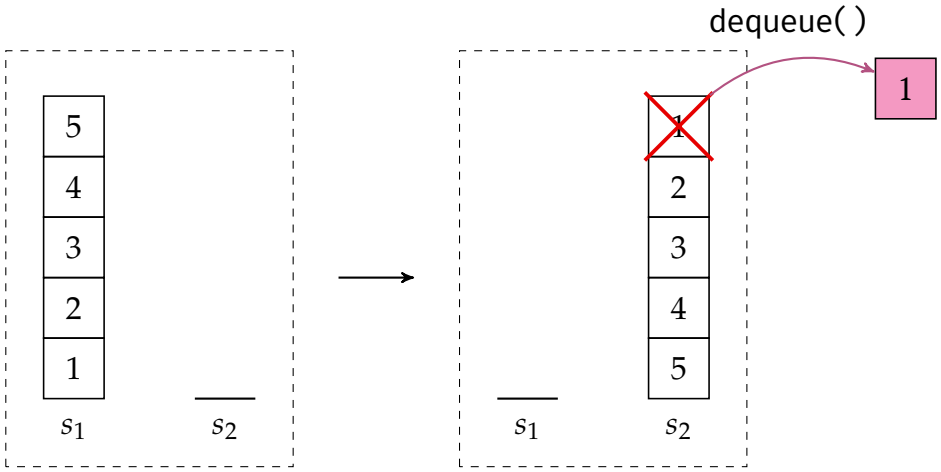


Рис. 2.3.4. Если при запросе dequeue( ) стек s_2 пуст, то содержимое стека s_1 поэлементно перекладывается в стек s_2.

Упражнение 2.3.1. Объясните, почему не получится реализовать стек при помощи нескольких (константного количества) очередей.

2.4. Двухсторонняя очередь

Двухсторонняя очередь позволяет добавлять и выталкивать элементы с обеих сторон, т.е. определяется следующими запросами:

- `enqueue_back(x)` — добавляет элемент `x` в конец очереди,
- `enqueue_front(x)` — добавляет элемент `x` в начало очереди,
- `dequeue_front()` — вытаскивает элемент из начала очереди.
- `dequeue_back()` — вытаскивает элемент из конца очереди.

Реализация на списках

Двухстороннюю очередь можно эффективно реализовать на двусвязном списке, но не на односвязном, т.к. он не позволяет эффективно удалять из конца списка.

Реализация на массиве

Эффективная реализация двухсторонней очереди на массиве использует ту же идею кольцевого буфера, только теперь указатели могут двигаться в обоих направлениях. Соответственно, если указатель на начало массива нужно сдвинуть влево, то он «перепрыгивает по циклу» на последний элемент массива.

2.5. Амортизационный анализ

Амортизационный анализ (англ. amortized analysis) — это способ подсчёта средней сложности запроса к структуре данных. Он позволяет оценить *амортизированную стоимость*[2] *запроса* — средняя сложность (в худшем случае), где усреднение берётся по всем запросам к структуре данных. Мы рассмотрим два метода получения таких оценок.

2.5.1. Метод учётных стоимостей

Опишем технику оценки амортизированной стоимости запросов к структурам данных. Пусть к некоторой структуре данных, содержащей не более n элементов, выполняются m запросов, и пусть $c_1, \ldots, c_m$ — *истинные* стоимости этих запросов (т.е. реальное количество операций, потраченных на их выполнение). Если (индивидуальную) сложность каждого запроса нельзя оценить лучше, чем $O(f(n))$, то суммарная стоимость m таких запросов можно оценить так

$$\sum_{i=1}^{m} c_i = O(m \cdot f(n)).$$

[2] В амортизационном анализе принято использовать термин «стоимость» как синоним «сложности»

Пусть $S = \{S_0, S_1, \ldots, S_m\}$ — это множество состояний структуры данных, где S_0 — это исходное состояние, а S_i — состояние после запроса с номером i. Для любой функции $\Phi : S \to \mathbb{R}_+$ можно определить *учётную стоимость запроса i относительно* Φ, обозначаемую $\tilde{c}_i$, следующим образом:

$$\tilde{c}_i = c_i + \Phi(S_i) - \Phi(S_{i-1}).$$

Для удобства введём обозначение $\Phi_i = \Phi(S_i)$. В таких обозначениях

$$\tilde{c}_i = c_i + \Phi_i - \Phi_{i-1}.$$

Просуммируем это соотношение по всем i:

$$\sum_{i=1}^{m} \tilde{c}_i = \sum_{i=1}^{m} c_i + \sum_{i=1}^{m} \Phi_i - \sum_{i=1}^{m} \Phi_{i-1} = \sum_{i=1}^{m} c_i + \underbrace{\Phi_m - \Phi_0}_{\Delta\Phi}.$$

Основная идея метода

Пусть про Φ дополнительно известно, что $\Delta\Phi \geq 0$ (можно потребовать, например, чтобы $\Phi(S_0) = 0$). Тогда мы можем оценить сумму истинных стоимостей как

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} \tilde{c}_i - \Delta\Phi \leq \sum_{i=1}^{m} \tilde{c}_i.$$

Заметим, что у нас есть большая свобода в выборе функции Φ, т.к. от неё почти ничего не требуется. Если нам удастся подобрать Φ такую, что учётная стоимость[3] любого запроса будет ограничена $O(g(n))$, т.е. $\tilde{c}_i = O(g(n))$ для любого i, и при этом $g(n) = o(f(n))$, то в результате мы получим лучшую оценку на общее число операций

$$\sum_{i=1}^{m} c_i \leq \sum_{i=1}^{m} \tilde{c}_i = O(m \cdot g(n)) = o(m \cdot f(n)).$$

В таком случае мы будем говорить[4], что *амортизированная стоимость запроса* не более $O(g(n))$.

Пример: амортизация стоимости печи в пекарне

Слово «амортизация» в контексте анализа алгоритмов используется в том же смысле, в котором оно используется в экономике. Попробуем разобрать идею метода учётных стоимостей на следующей несложной экономической задаче. Представим, что вам в наследство досталась пекарня, и теперь вы разрабатываете для неё бизнес-план. Для простоты будем считать, что вы планируете выпекать только один вид булочек. Вы оценили затраты на производство одной булочки в 40 рублей — сюда входят затраты на ингредиенты, оплату труда пекаря, аренду и прочее. Осталось учесть только одну «мелочь»: печь имеет ограниченный ресурс

[3] Учётная стоимость используется как удобная абстракция, которая не имеет “физического” смысла — в зависимости от разных Φ мы будем получать разные учётные стоимости, которые ничего не говорят про истинную стоимость запросов.

[4] Тут важно не забывать, что мы оценили амортизированную стоимость запроса относительно последовательности из m запросов. Сама по себе эта оценка ничего не говорит про сложность одного запроса.

и её требуется заменить после выпекания 10 тысяч булочек. Стоимость новой печи 50 тысяч рублей. Таким образом, первые 9999 тысяч булочек действительно будут стоить вам по 40 рублей за каждую. А вот булочка с номером 10000 обойдётся вам уже в 50040 рублей. Вместо того, чтобы требовать с «несчастливого» покупателя булочки с номером 10000 дополнительные 50 тысяч рублей, вы решаете разделить эту сумму между всеми покупателями. Для этого вы добавляете 5 рублей к стоимости каждой булочки (т.е. считаете, что затраты на производство равны 45 рублям), и в результате накапливаете 50 тысяч непосредственно к тому моменту, когда печь нужно будет заменить.

В наших обозначениях, истинная стоимость производства i-ой булочки $c_i = 40$ для всех $i < 10000$ и $c_{10000} = 50040$. В качестве потенциала выберем функцию

$$\Phi(i) = 5i \mod 50000,$$

которая отражает количество отложенных денег после производства i булочки. Если после производства булочки мы накопили заветные 50 тысяч, то мы их сразу же тратим на обновление печи. Тогда для всех $i < 10000$ учётная стоимость производства булочки равна 45 рублям:

$$\tilde{c}_i = 40 + \Phi(i) - \Phi(i-1) = 40 + 5i - 5(i-1) = 45.$$

В то же время учётная стоимость производства булочки с номером 10000 так же получится равной 45 рублям:

$$\tilde{c}_{10000} = 50040 + \Phi(10000) - \Phi(9999) = 50040 + 0 - 5 \cdot 9999 = 45.$$

Таким образом, не смотря на то, что истинная стоимость производства некоторых булочек очень высокая (т.к. включает в себя стоимость замены печи), за счёт амортизации мы ограничили затраты на каждую булочку всего 45 рублями. Аналогично мы поступаем в применении к анализу алгоритмов — мы разделяем истинную стоимость «дорогих» операций между всеми операциями и оцениваем получившиеся учётные стоимости.

Замечание. Для этого простого примера нам удалось придумать естественные интерпретации для функции Φ и учётных стоимостей. В дальнейшем мы столкнёмся со значительно более сложными потенциалами, которым будет сложно придать какой-то физический смысл.

Пример: анализ расширяющегося массива

Применим метод учётных стоимостей для оценки сложности запроса добавления элемента в расширяющийся массив. Мы уже знаем, что вне зависимости от схемы выделения памяти в худшем случае сложность добавления нового элемента в массив $O(n)$. Однако мы также показали, что при мультипликативной схему выделения памяти *в среднем* на добавление каждого нового элемента мы потратим $O(1)$. Давайте докажем это при помощи метода учётных стоимостей.

Будем предполагать, что мы начинаем с пустого массива и последовательно добавляем в него n элементов, кроме того коэффициент расширения $\alpha = 2$. Для получения оценки нам достаточно оценить количество копирований как самой трудоёмкой части запроса.

Для доказательства нам нужно предъявить функцию потенциала, относительно которой учётная стоимость запросов будет константной. Чтобы подобрать такую функцию, давайте посмотрим, сколько копирований происходит при добавлении нового элемента в конец массива в зависимости от его номера. Если это первый элемент, то требуется одно копирование: выделяется массив единичного размера и в него копируется первый элемент. Для добавления второго элемента требуется выделить массив размера два и скопировать туда первый и второй элементы. И так далее. Заметим, что для добавления четвёртого элемента нам требуется только одно копирование, т.к. к этому моменту мы уже выделили массив размера 4, одна из ячеек которого свободна. Аналогично, только одно копирование требуется для пятого, шестого, седьмого и восьмого элементов (см. рис. 2.5.1).

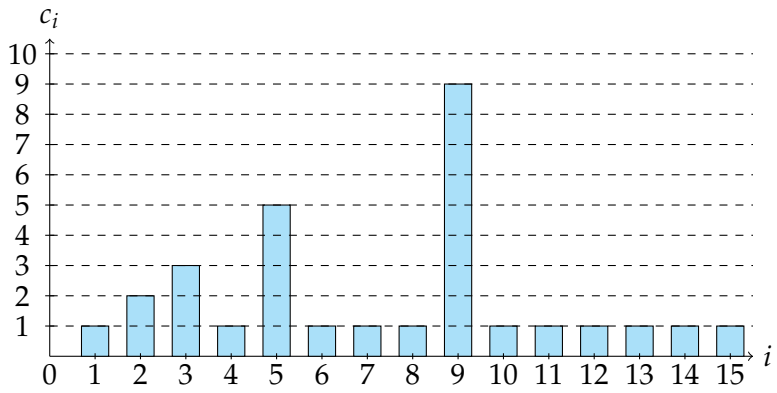


Рис. 2.5.1. Стоимость добавления элемента в расширяющийся массив.

Искомая функция потенциала в должна вести себя в некотором смысле противоположно тому, как ведут себя стоимости операций c_i: в те моменты, когда c_i большое, функция потенциала должна резко убывать, компенсируя таким образом c_i, а в тех случаях, когда для добавления требуется только одна операция, то потенциал должен возрастать не сильно. Для того, чтобы определить такую функцию, нам нужно как-то описать состояние расширяющегося массива. Давайте за l обозначим размер массива, а за s — количество заполненных ячеек (см. рис. 2.5.2).

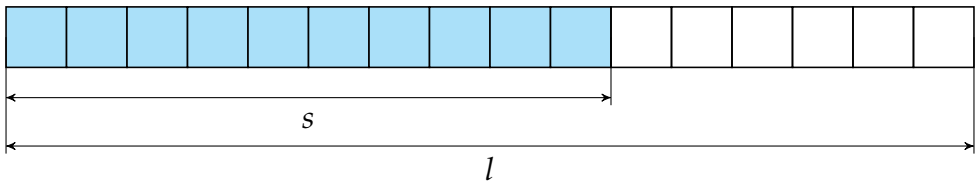


Рис. 2.5.2. Массив размера l с s элементами.

Заметим, что s с каждым добавленным элементом возрастает на 1, а l либо не изменяется, либо увеличивается в 2 раза, если массив был полностью заполнен, т.е., когда для добавления нового элемента требуются дополнительные копирования. На основе этих наблюдений можно догадаться, что нам может подойти линейная функция потенциала, которая положительно зависит от s и отрицательно от l — тогда эта функция будет медленно возрастать, пока в массиве есть свободные ячейки, и резко падать, если массив заполнился и произошло расширение. Осталось подобрать подходящие коэффициенты. Исходя из того, что при расширении массива уменьшение функции потенциала должно компенсировать

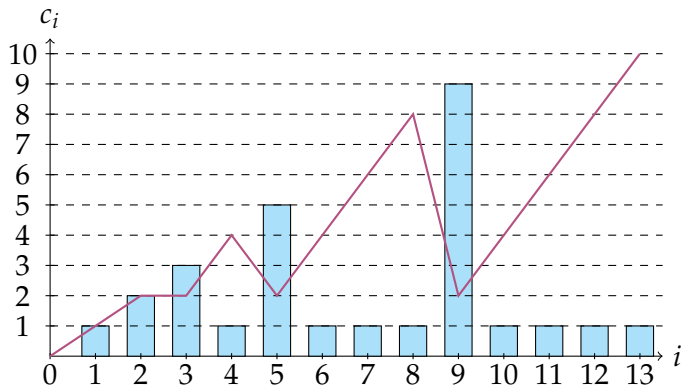


Рис. 2.5.3. Потенциал Φ и стоимость добавления элемента массива.

дополнительные затраты на копирования, предлагается использовать функцию $\Phi(s,l) = 2s - l$ (см. рис. 2.5.3).

Нам осталось показать, что учётные стоимости запросов на добавление действительно ограничены константой относительно предложенной функции потенциала $\Phi(s,l) = 2s - l$. По определению учётная стоимость равна

$$\tilde{c_i} = c_i + \Phi_i - \Phi_{i-1}.$$

Пусть l и s обозначают соответственно размер массива и количество элементов в нём до добавления элемента с номером i. Нам нужно рассмотреть два случая.

1. В массиве есть пустые ячейки, т.е. $s < l$. Тогда для добавления нам потребуется только одно копирование, т.е. $c_i = 1$, следовательно

$$\tilde{c_i} = c_i + \Phi(s+1,l) - \Phi(s,l) = 1 + (2(s+1) - l) - (2s - l) = 3.$$

2. Массив полностью заполнен, т.е. $s = l$. В этом случае нужно скопировать все s элементов в новый массив и потом скопировать туда же новый элемент, т.е. $c_i = s + 1$. Поэтому

$$\tilde{c_i} = c_i + \Phi(s+1,2l) - \Phi(s,l) = (s+1) + (2(s+1) - 2l) - (2s - l) = 3$$

(тут мы пользуемся тем, что $s = l$).

Мы показали, что относительно такой функции потенциала учётные стоимости запросов ограничены константой. Осталось заметить, что потенциал пустого массива $\Phi_0 = \Phi(0,0) = 0$. Это позволяет нам заключить, что запрос на добавление элемента в конец массива при использовании мультипликативной схемы расширения имеет амортизированную оценку сложности $O(1)$.

Упражнение 2.5.1. Подберите функцию потенциала для произвольного коэффициента расширения $\alpha > 1$.

Упражнение 2.5.2. Рассмотрим двоичный счётчик, который реализует запрос на увеличение значения на единицу. Если в счётчике хранится число n, то для увеличения его на единицу в худшем случае необходимо изменить значения $1 + \lfloor log_2 n \rfloor$ битов, т.е. сложность этого запрос равна $O(\log_2 n)$. Покажите, что амортизированную стоимость запроса на увеличение не превосходит $O(1)$.

2.5.2. Метод предоплаты

Метод предоплаты значительно менее формален, чем метод учётных стоимостей, и за счёт этого позволяет упростить некоторые рассуждения. Метод основан на следующей идее: для того, чтобы показать, что суммарное количество операций невелико, можно каждому элементу структуры данных «выдать» некоторое количество монеток, которыми этот элемент сможет «оплачивать» операций с ним. Если показать, что каждому элементу хватит выданных монеток, и что в сумме на m запросов к структуре данных, хранящей n элементов, нам потребуется не более $O(m \cdot g(n))$ монет, то можно заключить, что амортизированную стоимость запросов ограничена $O(g(n))$.

Пример: анализ очереди на двух стеках

В разделе 2.3.2 мы рассматривали реализацию очереди на двух стеках. Воспользуемся методом предоплаты для того, чтобы показать, что запросы к такой очереди имеют амортизированную стоимость $O(1)$. Для этого каждому элементу, который попадает в очередь, выдаётся четыре монетки на оплату операций со стеками. Легко увидеть, что четырёх монеток достаточно: каждый элемент должен оплатить две операции `push` и две операции `pop`. В сумме на оплату любых m запросов нам потребуется не более $4m$ монеток. Следовательно амортизированная стоимость запросов к такой очереди $O(1)$.

Глава 3

Разделяй и властвуй

3.1. Переход к подзадачам

В этом курсы мы будем изучать не только алгоритмы для конкретных задач, но и общие методы построения алгоритмов. Начнём с одного из самых естественных подходов, который получил название «разделяй и властвуй» («divide and conquer»). Идея этого метода заключается в следующих трёх шагах:

- разделить вход алгоритма на кусочки меньшего размера,
- решить *подзадачи* для этих кусочков рекурсивно,
- вычислить решение для исходной задачи на основании решения подзадач.

Далее мы проиллюстрируем работу этого метода на нескольких примерах и в завершение докажем общую теорему, которая поможет нам оценивать сложность алгоритмов, построенных на этой идее.

3.2. Двоичный поиск

Рассмотрим задачу поиска заданного элемента в упорядоченном массиве. Те, кто когда-либо пытался найти слово в словаре (имеется в виду настоящая бумажная книга), уже интуитивно знают как нужно действовать. Не стоит пытаться найти слово последовательно просматривая все страницы словаря. Нужно действовать умнее и открыть словарь примерно на середине. Посмотреть, какие слова находятся на открытой странице, и в зависимости от этого продолжить поиск в первой или второй половине книги.

Используя эту же идею мы построим эффективный алгоритм для этой задачи. Давайте возьмём средний элемент массива и сравним его с искомым. Если искомый элемент меньше, то он может быть только в левой половине. Если искомый элемент больше, то он, соответственно, может быть только во второй половине массива. А если нам повезло и искомый элемент равен среднему элементу, то мы можем завершить работу.

```
# ищем самое левое вхождение x в array[left:right]
def binary_search(array, x, left, right):
    #
    if right == left:
        return None

```

```
    if right == left + 1:
        if array[left] == x:
            return left
        else:
            return None

    mid = left + (right - left) // 2

    if array[mid] < x:
        return binary_search(array, x, mid + 1, right)
    elif array[mid] == x and array[mid - 1] < x:
        return mid
    else:
        return binary_search(array, x, left, mid)
```

3.3. Умножение n-битовых чисел

3.4. Сортировка слиянием

Алгоритм был изобретён Джоном фон Нейманом в 1945 году.

```
878 # сортировка слиянием
879 def merge_sort(array):
880     n = len(array)
881 
882     # пустой и одноэлементный массивы сортировать не нужно
883     if n < 2:
884         return array
885 
886     # рекурсивно сортируем левую и правую половины массива
887     array1 = merge_sort(array[:n/2])
888     array2 = merge_sort(array[n/2:])
889 
890     # возвращаем результат слияние двух упорядоченных массивов
891     return merge(array1, array2)
892 
893 # слияние двух упорядоченных массивов
894 def merge(array1, array2):
895     n, m = len(array1), len(array2)
896 
897     # переменные цикла
898     i, j = 0, 0
899 
900     # массив для результата слияния
901     result = []
902 
903     # перебираем элементы массивов
904     while i < n and j < m:
905         # на каждой итерации выбираем минимальный из текущих
906         # и записываем его в массив с результатом
907         if array1[i] < array2[j]:
908             result.append(array1[i])
909             i = i + 1
910         else:
911             result.append(array2[j])
912             j = j + 1
913 
914     # добавляем в конец оставшиеся элементы
915     while i < n:
916         result.append(array1[i])
917         i = i + 1
918 
919     while j < m:
920         result.append(array2[j])
921         j = j + 1
922 
923     return result
```

3.5. Основная теорема о рекуррентных соотношениях

Теорема 3.5.1. *Пусть $T(n)$ удовлетворяет следующему рекуррентному соотношению:*

$$T(1) = O(1), \quad T(n) = a \cdot T(\lceil n/b \rceil) + O(n^d)$$

3.5.1. Как побороть аддитивные константы?

Рекуррентное соотношение в теореме 3.5.1 задаётся в очень удобной форме. На практике соотношение может выглядеть менее красивым. Рассмотрим два случая.

1. Пусть дано следующее соотношение: $T(n) = 2 \cdot T(\lceil n/2 \rceil - 4) + O(n)$. В этом случае мы можем сказать, что $T(n)$ — возрастающая функция, и поэтому

$$T(n) = 2 \cdot T(\lceil n/2 \rceil - 4) + O(n) \leq 2 \cdot T(\lceil n/2 \rceil) + O(n),$$

   что позволяет применить теорему о рекуррентных соотношениях и получить оценку $T(n) = O(n \log n)$.

2. Пусть теперь задано соотношение: $T(n) = 2 \cdot T(\lceil n/2 \rceil + 4) + O(n)$. Предудущие рассуждения здесь уже не сработают, т.к. $\lceil n/2 \rceil + 4 > \lceil n/2 \rceil$. Мы могли бы повторить доказательство теоремы 3.5.1 для того, чтобы показать, что она верна для $T(n) = a \cdot T(\lceil n/b \rceil + c) + O(n^d)$ с прозвольной константой c. Это же можно объяснить при помощи следующего трюка с заменой переменной. Введём переменную $m = n - 8$, т.е. $n = m + 8$, и заменим переменную в заданном рекуррентном соотношении. Получим

$$T(m+8) = 2 \cdot T(\lceil (m+8)/2 \rceil + 4) + O(m+8) = 2 \cdot T(\lceil m/2 \rceil + 8) + O(m).$$

   Теперь рассмотрим функцию $T'(m) = T(m+8)$. Для неё выполняется следующее рекуррентное соотношение:

$$T'(m) = 2 \cdot T'(\lceil m/2 \rceil) + O(m),$$

   для которого по теореме 3.5.1 получается оценка $O(m \log m)$. Нам осталось снова пройти по чепочке замен и получить требуемую оценку

$$T(n) = T(m+8) = T'(m) = O(m \log m) = O((n-8)\log(n-8)) = O(n \log n).$$

Упражнение 3.5.1. Покажите, что теорема 3.5.1 верна для рекуррентных соотношений вида $T(n) = a \cdot T(\lceil n/b \rceil + c) + O(n^d)$ с прозвольной константой c.

3.6. Умножение матриц

Глава 4

Сортировка

4.1. Квадратичные сортировки

```python
def swap(array, i, j):
    array[i], array[j] = array[j], array[i]
```

```python
def selection_sort(array):
    n = len(array)

    for i in range(n):
        min_pos = i
        for j in range(i + 1, n):
            if array[min_pos] > array[j]:
                min_pos = j

        swap(array, i, min_pos)

    return array
```

```python
def insertion_sort(array):
    n = len(array)

    for i in range(n):
        for j in range(i, 0, -1):
            if array[j - 1] > array[j]:
                swap(array, j - 1, j)

    return array
```

```python
def bubble_sort(array):
    n = len(array)

    for i in range(n - 1):
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                swap(array, j, j + 1)

    return array
```

4.2. Нижняя оценка на сортировку сравнениями

Мы уже знаем, что массив можно отсортировать за $O(n \log n)$ при помощи сортировки слиянием. А можно ли быстрее? Оказывается, что если при сортировке использовать *только сравнения* элементов, то сортировать быстрее не получится. Будем говорить, что сортировка *основана на сравнениях*, если она не использует никаких других операций с ключами, кроме сравнений (т.е., например, не использует арифметических операций с ключами).

Теорема 4.2.1. *Любой алгоритм сортировки массива длины n требует $\Omega(n \log n)$ сравнений.*

Для доказательства этой теоремы мы сначала докажем её для более слабой модели вычислений.

Определение 4.2.1. *Деревья сравнений* — это модель вычислений, для которой каждая программа представляется в виде двоичного дерева, в котором внутренние вершины соответствуют сравнениям элементов, а листья — перестановкам элементов.

Для объяснения того, как работает эта модель, давайте рассмотрим следующую программу для сортировки трёх элементов. На вход программе подаётся

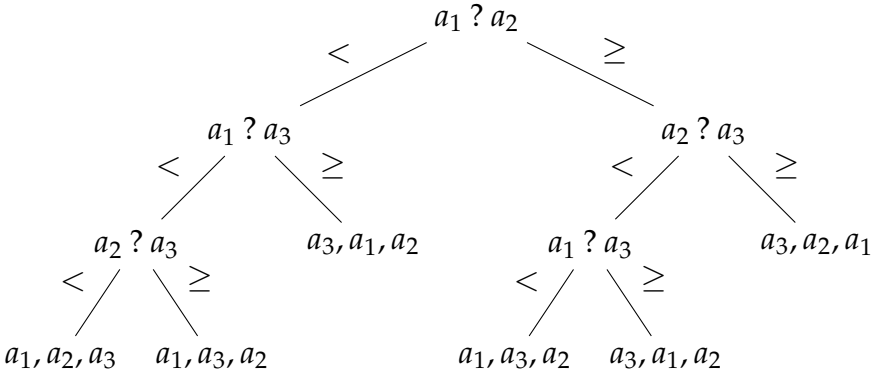


Рис. 4.2.1. Дерево сравнений для сортировки трёх элементов.

набор из трёх чисел: a_1, a_2, a_3. В каждый момент времени программа находится в одной из вершин дерева. Один шаг программы состоит в выполнении сравнения двух элементов, которые указаны в текущей вершине, и переходе в одного из сыновей текущей вершины, в зависимости от результата сравнения. Выполнение программы начинается в корне дерева и завершается в одном из листьев, результат выполнения — перестановка элементов, записанная в соответствующем листе.

В данной программе на первом шаге выполняется сравнение a_1 и a_2. В зависимости от результата сравнения исполнение переходит в одного из сыновей: если оказывается, что $a_1 < a_2$, то исполнение переходит в левого сына, а иначе — в правого. Выполнение продолжается, пока не достигнет листа. Проверьте, что данная программа действительно корректно сортирует любой набор из трёх элементов.

Теперь мы докажем, что в глубина любого дерева сравнений, упорядочивающее n элементов, имеет глубину $\Omega(n \log n)$.

Лемма 4.2.1. *Любое дерево сравнений корректно сортирующее n элементов имеет глубину $\Omega(n \log n)$.*

Доказательство. Если дерево сравнений сортирует n элементов корректно, то среди его листьев должны встречаться все возможные $n!$ перестановок из n элементов. Таким образом, количество листьев у любого такого дерева не меньше $n!$. Теперь нам потребуется воспользоваться следующим свойством двоичных деревьев: в дереве глубины h не более 2^h листьев. Действительно, если дерево высоты h —полное, т.е. в него нельзя добавить вершин, не увеличив высоты, то в нём ровно 2^h листьев, следовательно в произвольном дереве глубины h листьев не больше 2^h. Отсюда получаем следующее соотношение на глубину дерева h:

$$2^h \geq n! = 1 \cdot 2 \cdot \dots \cdot n.$$

Логарифмируя обе части неравенство получаем

$$h \geq \log(n!) = \log(1) + \log(2) + \dots + \log(n).$$

Оставим только половину слагаемых справа.

$$h \geq \log(\lceil n/2 \rceil) + \dots + \log(n) \geq \frac{n}{2} \cdot \log(\lceil n/2 \rceil) = \Omega(n \log n).$$

□

4.3. Сортировка кучей

4.4. Быстрая сортировка

```
def quick_sort(array, left, right):
    if right <= left + 1:
        return

    pivot = random.randint(left, right - 1)

    mid = partition(array, pivot, left, right)

    quick_sort(array, left, mid)
    quick_sort(array, mid + 1, right)

    return array
```

```
def partition(array, pivot, left, right):
    x = array[pivot]

    swap(array, pivot, right - 1)

    m = left

    for p in range(left, right - 1):
        if array[p] < x:
            swap(array, p, m)
            m += 1

    swap(array, m, right - 1)
    return m
```

4.4.1. Анализ времени работы

4.4.2. Сортировка массива с повторяющимися элементами

```
def quick_sort3(array, left, right):
    if right <= left + 1:
        return

    pivot = random.randint(left, right - 1)

    (m1, m2) = partition3(array, pivot, left, right)

    quick_sort3(array, left, m1)
    quick_sort3(array, m2, right)

    return array
```

```
def partition3(array, pivot, left, right):
    x = array[pivot]

    swap(array, pivot, right - 1)

    m = left
    p = left
    q = right - 1

    while p < q:
        if array[p] == x:
            q -= 1
            swap(array, p, q)
        else:
            if array[p] < x:
                swap(array, p, m)
                m += 1
            p += 1

    for i in range(q, right):
        swap(array, m + i - q, i)

    return m, m + right - q
```

4.5. Линейные сортировки

4.5.1. Сортировка подсчётом

```
def counting_sort(array, max_value):
    counters = [0 for k in range(max_value + 1)]

    for k in array:
```

```
        counters[k] += 1

    result = []
    for k in range(max_value + 1):
        for i in range(counters[i]):
            result.append(k)

    return result
```

```
def counting_sort_stable(array, max_value):
    counters = [0 for k in range(max_value + 1)]

    for (k, v) in array:
        counters[k] += 1

    idx = 0
    indices = []
    for k in range(max_value + 1):
        indices.append(idx)
        idx += counters[k]

    result = [None for k in range(len(array))]
    for (k, v) in array:
        result[indices[k]] = (k, v)
        indices[k] += 1

    return result
```

4.5.2. Цифровая сортировка

4.5.3. Блочная сортировка

Глава 5

Задачи RMQ и LCA

В данном разделе мы изучим две задачи, которые неожиданным образом оказываются связанными: *задачу о минимуме на отрезке* и *задача о наименьшем общем предке*.

5.1. Задача о минимуме на отрезке

Определение 5.1.1. *Задача о минимуме на отрезке (range minimum query, RMQ)*: по последовательности $A = (a_1, \ldots, a_n)$ и паре индексов (i, j), $i \leq j$, найти такой индекс $k \in [i, j]$, что $a_k = \min\{a_i, \ldots, a_j\}$.

В такой постановке эта задача имеет сложность $\Theta(n)$: её можно решить за $O(n)$ *прямым вычислением*, т.е. перебрав элементы $a_i, \ldots, a_j$, и при этом нельзя решить быстрее, т.к. для пары индексов $(1, n)$ нам потребуется перебрать все элементы последовательности. Задача становится более интересной, если к одной и той же последовательности совершается множество запросов. В этом случае мы можем вычислить какую-то вспомогательную информацию или использовать структуры данных, которые позволят отвечать на такие запросы значительно быстрее.

5.1.1. Динамическая задача RMQ

Разрешим дополнительно кроме запроса минимума на отрезке также изменять элементы последовательности $A = (a_1, \ldots, a_n)$ (но не их количество) при помощи запроса `change(i, x)`, который заменяет элемент a_i на x. В таких случаях принято говорить о *динамической постановке задачи RMQ*. Давайте подумаем, какой сложности для таких запросов следует ожидать в лучшем случае. Предположим, что на вычисление вспомогательной информации и на построение структур данных мы хотим потратить не более $O(n)$ (на меньшее надеяться не стоит, т.к. нам требуется прочитать все элементы последовательности). Можно доказать следующую оценку.

Теорема 5.1.1. *Если сложность вычисления вспомогательной информации и построения структур данных не более $O(n)$, то максимальная из сложностей запросов* `minimum` *и* `change` *в динамической постановка задачи RMQ не меньше $\Omega(\log n)$.*

Доказательство. Будем доказывать от обратного. Предположим, что существует алгоритм, позволяющий выполнять запросы `minimum(i,j)` и `change(i, x)` на массиве длины n за $o(\log n)$ после предобработки массива функцией `rmq-preprocess`, сложность которой не более $O(n)$. Покажем, что в таком случае мы можем отсортировать произвольных массив длины n сравнениями за $o(n \log n)$, что противоречит нижней оценке $\Omega(n \log n)$ на сортировку сравнениями. Пусть нам дан

массив A длины n, и пусть M — это некоторое значение, которое больше любого элемента массива A. Рассмотрим следующую функцию, которая упорядочивает массив A используя алгоритм для RMQ.

```
# сортирует массив используя алгоритм для RMQ
def sort-using-rmq(A):
    n = len(A)

    # предобработка для RMQ, не более O(n)
    rmq-preprocess(A)

    # массив для результата
    result = []

    for i in range(n):
        # вычисляем минимум для всего массива, o(log n)
        k = minimum(0, n - 1)
        # сохраняем минимум
        result.append(A[k])
        # заменяем элемент k на значение M, o(log n)
        change(k, M)

   return result
```

Сложность этой функции складывается из сложности предобработки $O(n)$, сложности n запросов `minimum` и сложности n запросов `change`. По нашему предположению сложность любого запроса не более $o(\log n)$, следовательно суммарная сложность ограничена $o(n \log n)$. □

Таким образом, самая оптимистичная оценка — это $O(\log n)$ на запрос, при условии, что на вычисление вспомогательной информации и на построение структур данных тратится не более $O(n)$ операций. Такую оценку позволяет достичь специальная структура данных, называемая *деревом отрезков*.

Определение 5.1.2. *Дерево отрезков* для массива длины n — это двоичное дерево, с каждой вершиной которого ассоциирован отрезок массива по следующему правилу:

- с корнем r ассоциирован отрезок $S(r) = [1, n]$,
- для каждой вершины v с отрезком $S(v) = [i, j]$,
  - если $i = j$, то v — лист,
  - в противном случае v имеет два потомка u и w, причём

$$S(u) = [i, m] \quad \text{и} \quad S(w) = [m + 1, j], \quad \text{где} \quad m = \lfloor (i + j)/2 \rfloor.$$

Легко проверить, что у дерева отрезков для массива длины n ровно n листьев — по одному на каждый элемент массива. Кроме того, так как отрезки во внутренних вершинах при переходе к потомкам уменьшаются вдвое, то высота такого дерева отрезков ограничена $\lceil \log n \rceil$. Будем называть отрезки, которые соответствуют вершинам дерева, *каноническими* и докажем следующую лемму.

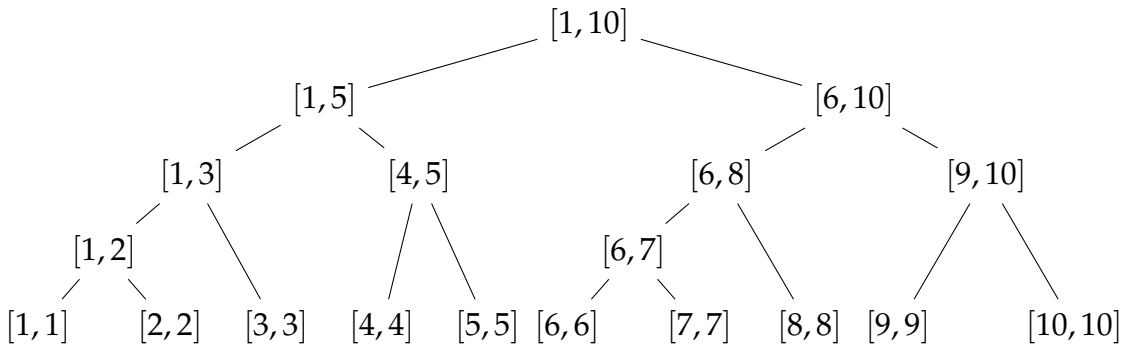


Рис. 5.1.1. Дерево отрезков для массива длины 10.

Лемма 5.1.1. *Любой отрезок $s \subseteq [1, n]$ может быть представлен в виде объединения непересекающихся канонических отрезков $s_1, \ldots, s_k$, где $k = O(\log n)$.*

Доказательство. Будем задавать отрезки парами целых чисел и в каждой вершине двоичного дерева дополнительно хранить поле `segment` — ассоциированный отрезок. Нам будет удобно воспользоваться следующей вспомогательной функцией для пересечения двух отрезков.

```
# вычисляет отрезок — пересечение двух отрезков
def segment_intersect((l1,r1), (l2, r2)):
    # левая граница — это максимум из левых границ
    l = max(l1, l2)

    # правая граница — это минимум из правых границ
    r = min(r1, r2)

    # если окажется, что l > r, то пересечение пустое
    return (l, r)
```

Рассмотрим следующую рекурсивную процедуру, которая по отрезку $[l, r]$ возвращает список канонических отрезков, задающий разбиение $[l, r]$.

```
# разбивает отрезок s на множество канонических отрезков
def decompose((l,r), v = root):
    # обрабатываем пустой отрезок
    if r < l:
        return []

    # возвращаем канонический отрезок
    if (l,r) == v.segment:
        return [v.segment]

    # разбиваем s
    s_left  = segment_intersect((l,r), v.left.segment)
    s_right = segment_intersect((l,r), v.right.segment)

    # выполняем рекурсивные вызовы и объединяем результаты
    return decompose(s_left, v.left) + decompose(s_right, v.right)
```

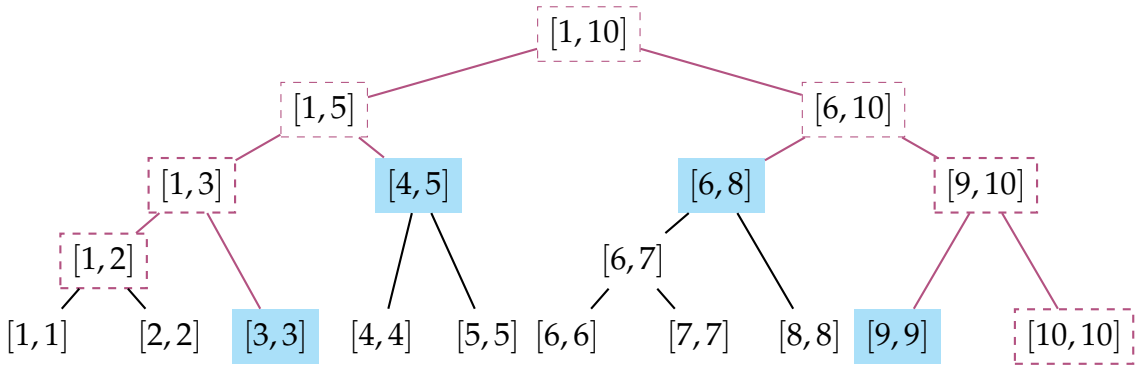


Рис. 5.1.2. Разбиение отрезка $[3, 9]$ функцией `decompose` для массива длины 10. В результате отрезок разбивается на канонические отрезки $[3, 3]$, $[4, 5]$, $[6, 8]$, $[9, 9]$. Остальные просмотренные вершины выделены пунктиром.

Сначала докажем корректность, а именно покажем, что данная процедура действительно задаёт разбиение отрезка на канонические. Заметим, что при каждом рекурсивном вызове `decompose(s, v)` поддерживается следующий инвариант: отрезок `s` всегда является подотрезком `v.segment`. Будем доказывать корректность этой процедуры по индукции по длине отрезка, ассоциированного с вершиной `v`. База индукции: процедура `decompose` корректна для листьев. Теперь предположим, что процедура корректна для всех вершин с отрезками длины менее k, и покажем корректность для вершины с отрезком длины k. Если мы попали в один из **`if`**-ов, то процедура корректна. Если же произошли рекурсивные вызовы, то по предположению индукции оба вызова вернут корректное разбиение для левой и правой частей отрезка `s`, что в сумме даст корректное разбиение для `s`.

Осталось доказать, что в разбиении получится не более $O(\log n)$ отрезков. Сначала рассмотрим частный случай, когда один из концов отрезка `s` совпадает с соответствующим концом отрезка `v.segment` . Для определённости давайте предположим, что у `s` и `v.segment` общее начало. Тогда могут быть три варианта выполнения `decompose(s, v)`:

- если `s == v.segment`, то вызов завершится без рекурсивных вызовов,
- если `s_left == v.left.segment`, то внутри `decompose(s_left, v.left)` не будет рекурсивных вызовов, а у `s_right` и `v.right` будет общее начало,
- если `s_left != v.left.segment`, то отрезок `s_right` — пустой, следовательно внутри `decompose(s_right, v.right)` не будет рекурсивных вызовов, а у `s_left` и `v.left.segment` будет общее начало.

Другими словами, при каждом вызове `decompose` рекурсия будет продолжаться не более, чем в одной ветке. Поэтому всего будет не более $2\lceil \log n \rceil$ вызовов `decompose`, и следовательно, `decompose(s, v)` вернёт не более $O(\log n)$ отрезков.

Теперь предположим, что у `s` и `v.segment` нет общих концов. В этом случае есть два варианта выполнения `decompose(s, v)`:

- если один из отрезков `s_left` и `s_right` пустой, то рекурсия продолжится только в одном из рекурсивных вызовов `decompose`,
- если оба отрезка `s_left` и `s_right` непустые, т.е. в этой вершине произошло нетривиальное разбиение отрезка `s` на две части, то теперь в обоих рекурсивных вызовах `decompose` один из концов отрезка совпадает с концом отрезка, ассоциированного с вершиной.

Пока мы находимся в условиях первого варианта, рекурсия продолжается только в одном из рекурсивных вызовов. Как только происходит нетривиальное разбиение отрезка, то в обоих рекурсивных вызовах мы попадаем в условия частного случая, который мы рассмотрели выше, и про который мы доказали оценку $O(\log n)$ на количество отрезков. Поэтому в сумме в общем случае `decompose` вернёт не более $O(\log n) + O(\log n) = O(\log n)$ канонический отрезков. □

Замечание 5.1.1. Может показаться, что и время работы функции `decompose` ограничено $O(\log n)$. Действительно, мы доказали, что функция `decompose` посещает $O(\log n)$ вершин дерева. Однако, эта функция возвращает список, и на объединение списков этих списков в Python дополнительно потратится $O(\log^2 n)$.

Упражнение 5.1.1. Перепишите функцию `decompose` так, чтобы её сложность была не более $O(\log n)$.

Применим дерево отрезков для решения поставленной задачи. Пусть дан некоторый массив A длины n. Построим дерево отрезков для этого массива и в каждой вершине запишем индекс минимального элемента на соответствующем отрезке. Отметим, что построение дерева и заполнение индексов можно организовать за $O(n)$: будем заполнять индексы в вершинах от листьев к корню, индекс в листе — это номер соответствующего элемента массива, а индексом во внутренней вершине выбирается один из индексов её сыновей — тот, что соответствует меньшему элементу массива.

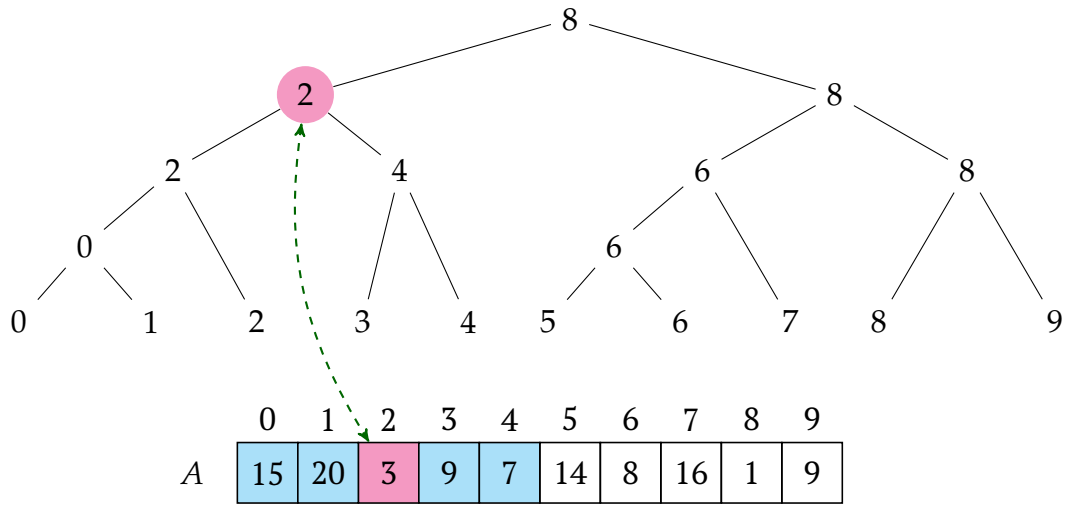


Рис. 5.1.3. Дерево отрезков с индексами минимумов для массива длины 10. Выделенный элемент хранит индекс минимального элемента среди первых пяти элементов массива

Пришло время реализовать процедуры `minimum` и `change`. Будем предполагать, что в каждой вершине хранится значение `value`, которое равно минимальному элементу на соответствующем отрезке. Процедура `minimum` реализуется аналогично процедуре `decompose`, посещает те же узлы дерева отрезков, и, следовательно, имеет сложность $O(\log n)$.

```
# действуем аналогично процедуре decompose
def minimum((l,r), v = root):
    # обрабатываем пустой отрезок
    if r < l:
        return None
```

```
    # если отрезок совпал с отрезком вершины
    if (l,r) == v.segment:
        # возвращаем минимум на этом отрезке
        return A[v.index]

    # разбиваем s
    s_left  = segment_intersect((l,r), v.left.segment)
    s_right = segment_intersect((l,r), v.right.segment)

    # выполняем рекурсивные вызовы
    left  = minimum(s_left)
    right = minimum(s_right)

    # объединяем результаты
    if left == None:
        return right
    if right == None:
        return left

    # если оба вызова вернули не None
    if A[left] < A[right]:
        return left
    else
        return right
```

Теперь опишем процедуру изменения элемента массива. При изменении элемента нам нужно обновить минимумы во всех вершинах-предках соответствующего листа, количество таких вершин не больше высоты дерева, т.е. $O(\log n)$.

```
# изменяем элемент массива
def change(i, x, v = root):
    # если мы пришли в соответствующий лист
    if v.segment == (i,i):
        v.value = x
        return

    # рекурсивно спускаемся до соответствующего листа
    if i <= v.left.segment[1]:
        change(i, x, v.left)
    else
        change(i, x, v.right)

    # обновляем значение в вершине
    v.value = min(v.left.value, v.right.value)
```

Таким образом мы научились решать задачу RMQ в динамической постановке за $O(n)$ на построение дерева отрезков и $O(\log n)$ на каждый запрос. Заметим, что наша конструкция не завязана на какие-либо свойства операции минимума, кроме ассоциативности. Поэтому аналогичный подход может быть использован,

например, для максимума, суммы, произведения и других ассоциативных операций.

5.1.2. Статическая задача RMQ

Теперь предположим, что массив для задачи RMQ задан и никогда не меняется. В этом случае принято говорить *о статической постановке задачи RMQ*. Мы уже умеем отвечать на запрос RMQ за $O(\log n)$ при помощи дерева отрезков. В статическое постановке мы можем получить дополнительное ускорение за счёт *предобработки* — можно вычислить какие-то дополнительные данные о массиве, когда уже известно, что дальше он изменяться не будет, и использовать их для более быстрого ответа на запрос. В дальнейшем, говоря о сложности решения задачи RMQ, мы будем оперировать парами оценок $(f(n), g(n))$, где $f(n)$ — это сложность предобработки, а $g(n)$ — сложность ответа на запрос RMQ. В этих обозначениях сложность прямого вычисления $(O(1), O(n))$.

Перед тем как перейти к решению статической задачи RMQ, рассмотрим более простую задачу RSQ.

Определение 5.1.3. *Задача о сумме на отрезке (range sum query, RSQ)*: по последовательности $A = (a_1, \ldots, a_n)$ и паре индексов (i, j), $i \le j$, вычислить сумму $S_{i,j} = a_i + \cdots + a_j$.

В статической постановке эту задачу легко решить за $(O(n), O(1))$. Давайте вычислим вспомогательные *частичные суммы*

$$S_0 = 0, \quad \forall k \in [n],\ S_k = a_1 + \ldots + a_k.$$

Это легко сделать за $O(n)$. Теперь для ответа на запрос $RSQ(i, j)$ нам достаточно вычислить разность соответствующих частичных сумм:

$$\mathrm{RSQ}(i, j) = S_j - S_{i-1}.$$

Для RMQ это не сработает, т.к. операция минимума не имеет обратной.

Полное предвычисление. Начнём с наиболее простого подхода: раз уж мы отдельно учитываем сложность предобработки и сложность запроса, то давайте сделаем все вычисления на этапе предобработки. Для этого вычислим RMQ для всех возможных пар индексов (i, j) и запишем ответы в таблицу размера $n \times n$. Такое решение будет иметь сложность $(O(n^2), O(1))$. (Если действовать „в лоб“ и заполнять эту таблицу прямым вычислением для каждого запроса, то получится и вовсе $(O(n^3), O(1))$, но нетрудно заметить, что такую таблицу можно заполнить за $O(n^2)$, если действовать немного умнее.)

Метод разреженной таблицы. Назовём отрезок *стандартным*, если его длина равна некоторой степени двойки. На этапе предобработки вычислим ответы на запросы RMQ только для стандартных отрезков. Это потребует $O(n \log n)$ времени и памяти (опять же такая оценка достигается, только если заполнять соответствующую табличку от меньших отрезков к большим). Как воспользоваться этими данными для ответа на запрос RMQ для произвольного отрезка? Воспользуемся следующим наблюдением.

Лемма 5.1.2. *Любой отрезок является либо стандартным, либо может быть представлен как объединение двух пересекающихся стандартных отрезков.*

Доказательство. Пусть отрезок $[l, r]$ не является стандартным. Выберем k такое, что

$$2^{k+1} > r - l + 1 > 2^k.$$

Такое k обязательно существует, иначе отрезок был бы стандартным. Тогда $[l, r]$ можно представить как объединение двух стандартных отрезков длины 2^k:

$$[l, r] = [l, l + 2^k - 1] \cup [r - 2^k - 1, r].$$

(Можно проверить, что $l + 2^k - 1 \geq r - 2^k - 1$, т.е. $l - r + 1 \geq 2^{k+1} - 1$.) □

Теперь для ответа на запрос нам нужно либо вернуть значение из таблицы, если отрезок оказался стандартным, либо представить отрезок в виде объединения двух стандартных и взять минимум из минимумов на этих отрезках.

Замечание 5.1.2. В методе разреженной таблицы мы пользуемся свойством *идемпотентности* операции минимума, т.е. тем, что $\min\{a, a\} = a$. Это позволяет получать ответ как минимум из минимумов на пересекающихся отрезках. Поэтому этот подход не будет работать для операций, которые этим свойством не обладают. Например, метод разреженной таблицы не подходит для задачи RSQ, т.к. сумма на отрезке не равна сумме на двух пересекающихся отрезках, которые его покрывают. К счастью, для RSQ у нас есть более эффективное решение.

Замечание 5.1.3. При оценке сложности этого метода предполагалось, что мы умеем находить ближайшую степень двойки (подходящее число k из леммы 5.1.2) за $O(1)$. Этого можно достичь, заполнив таблицу размера n, но на практике это обычно реализуется при помощи битовых операций, время работы которых в этом случае (число n ограничено размером памяти) можно считать константным.

5.2. Задача о наименьшем общем предке

Теперь пришло время поговорить о второй задаче этого раздела.

Определение 5.2.1. *Задача о наименьшем/нижайшем общем предке (least/lowest common ancestor, LCA)*: по паре вершин (u, v) корневого дерева T найти их самого нижнего общего предка. Другими словами, требуется найти наиболее удалённую от корня вершину, лежащую одновременно и на пути от u к корню и на пути от v к корню. Поэтому, если v является предком u, то $\mathrm{LCA}(u, v) = v$.

Для решения этой задачи мы применим *сведе́ние* задачи LCA к задаче RMQ, т.е. научимся по условию задачи LCA строить условие для задачи RMQ, а потом из ответа восстанавливать ответ для исходной задачи. Существует множество различных видов сведений, некоторые из которых мы изучим, когда будем говорить о NP-трудных задачах. Следующее определение довольно неформально, но вполне достаточно для наших целей.

Определение 5.2.2. $(O(n), O(1))$*-сведением задачи* A *к задаче* B будем называть тройку алгоритмов (F, G, H) таких, что

- алгоритм F по условию для задачи A размера n строит условие для задачи B размера $O(n)$ за время $O(n)$,

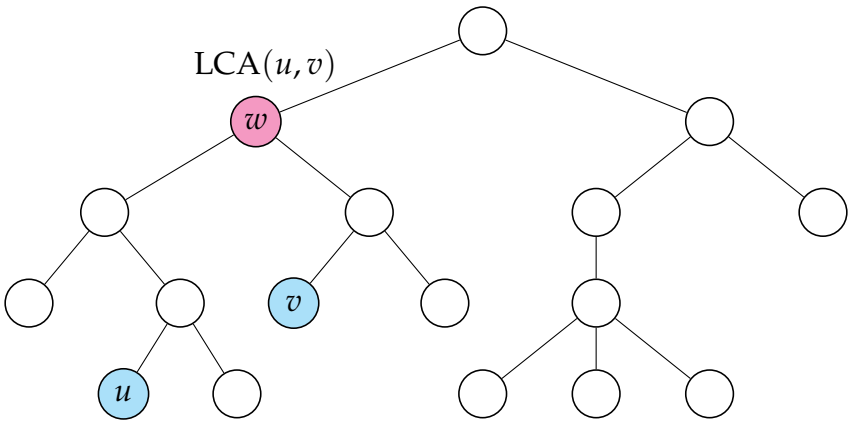


Рис. 5.2.1. Вершина w является ближайшим общим предком u и v.

- алгоритм G по запросу для задачи A строит соответствующий запрос для задачи B за $O(1)$,
- алгоритм H по ответу для запроса к задаче B восстанавливает ответ к исходному запросу для задачи A за время $O(1)$.

Утверждение 5.2.1. *Если существует $(O(n), O(1))$-сведение задачи LCA к задаче RMQ, то задача LCA решается за время $(O(n \log n), O(1))$.*

Доказательство. Для решения задачи LCA на дереве T с n вершинами на стадии предобработки применим алгоритм F, который по T построит массив A размера $O(n)$ за линейное время. Для массива A построим разреженную таблицу за время $O(n \log n)$. На стадии запросов для каждого запроса LCA будем применять алгоритм G, вычислим полученный RMQ-запрос методом разреженной таблицы, и в завершение применим алгоритм H, который вернёт нам ответ для исходного LCA-запроса. В результате на предобработку мы потратим $O(n \log n)$, а каждый запрос будет обрабатываться за $O(1)$. □

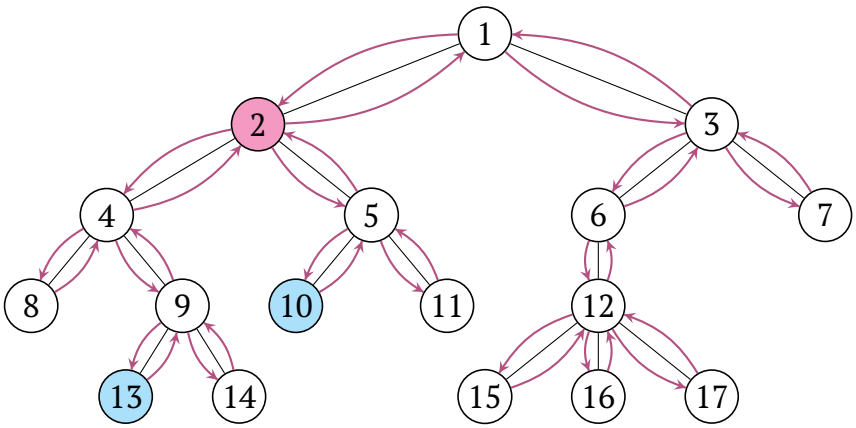
Сведение LCA к RMQ. Построим сведение LCA к RMQ. Пусть нам дано корневое дерево T с n вершинами. Выполним *эйлеров обход* дерева T, т.е. рекурсивно обойдём дерево, проходя по каждому ребру ровно два раза (см. рис. 5.2.2), и выпишем для вершины на пути её номер и глубину в дереве.

```
# построение эйлерова обхода
def euler-traversal(v = root, depth = 0, result = None):
    # создаём список для хранения результата
    if result == None:
        result = []

    # добавим в обход текущую вершину
    result.append((v, depth))

    # рекурсивно вызовем для каждого ребёнка и объединим результаты
    for c in v.childen:
        euler-traversal(c, depth + 1, result)
        result.append((v, depth))

    return result
```



вершина	1	2	4	8	4	9	13	9	14	9	4	2	5	10	5	11	···
глубина	0	1	2	3	2	3	4	3	4	3	2	1	2	3	2	3	···

Рис. 5.2.2. Эйлеров обход дерева T и пример вычисления $\mathrm{LCA}(13, 10)$ через RMQ.

В результате обхода мы получим массив пар длины $2n - 1$. Заполним вспомогательный массив first длины n, где для каждой вершины v в ячейке с номером v запомним индекс её первого вхождения в эйлеров обход. Теперь по эйлерову обходу можно построить задачу RMQ, в которой глубина вершины будет ключом, а номер вершины — дополнительной информацией. Докажем следующую теорему.

Теорема 5.2.1. $\mathrm{LCA}(v, u)$ *равно вершине, на которой достигается* $\mathrm{RMQ}(\mathrm{first}[v], \mathrm{first}[u])$.

Давайте для начала проверим это утверждение на примере. Сначала найдём на рис. 5.2.2 первое вхождение вершин 13 и 5. Потом на отрезке между этими двумя ячейками найдём вершину с минимальной глубиной — это будет вершина 2, и действительно она является ближайшим общим предком вершин 13 и 5. Теперь перейдём к доказательству теоремы.

Доказательство. Давайте переформулируем условие теоремы в терминах дерева: ближайший общий предок u и v — это вершина с минимальной глубиной, которая встречается в эйлеровом обходе между первым вхождением u и первым вхождением v. Покажем сначала, что ближайший общий предок обязательно встретится между первым вхождением u и первым вхождением v. Действительно, в дереве между парой вершин есть только один простой путь, и ближайший общий предок всегда лежит на этом пути, поэтому по пути из u в v мы обязательно посетим ближайшего общего предка. Теперь покажем, что на этом пути не будет других вершин с меньшей глубиной. Предположим, что такая вершина есть. Следовательно, по пути из u в v эйлеров обход прошёл через ближайшего общего предка и поднялся выше, т.е. поднялся по ребру, по которому до этого когда-то спустился. Но ведь для того, чтобы дойти до v, необходимо будет снова спуститься по этому ребру, а такого не бывает, т.к. каждое ребро проходится ровно два раза. □

Для завершения построения сведения нам осталось описать, как будут устроены алгоритмы F, G, H. Алгоритм F по дереву T строит его эйлеров обход и заполняет массив first. Алгоритм G преобразует запрос $\mathrm{LCA}(v, u)$ в запрос $\mathrm{RMQ}(\mathrm{first}[v], \mathrm{first}[u])$.

Алгоритм H возвращает номер вершины, записанный в ячейке массива, на которой достигается минимум на соответствующем отрезке. Нетрудно проверить, что эти алгоритмы имеют требуемую сложность.

Замечание 5.2.1. Получившаяся в результате сведения задача RMQ обладает важными дополнительными свойствами, а именно любые два соседних значения отличаются на единицу. Задачи RMQ с таким свойством называют ±1-RMQ. Для статической задачи ±1-RMQ существует алгоритм Фарака-Колтона — Бендера [1], который решает её за $(O(n), O(1))$. Таким образом и задачу LCA можно решить за $(O(n), O(1))$ при помощи построенного нами сведения. Но оказывается, что можно пойти дальше, и построить сведение задачи RMQ к LCA (по массиву для RMQ можно построить декартово дерево и на нём запускать алгоритм для LCA). В результате композиции этих двух сведений можно построить сведение задачи RMQ к ±1-RMQ, а следовательно решить задачу RMQ в общем случае за $(O(n), O(1))$. Однако на практике этот алгоритм не используется, т.к. асимптотическое ускорение нивелируется большими константами и сложностью реализации.

Упражнение 5.2.1. Постройте $(O(n), O(1))$-сведение задачи RMQ к задаче LCA.

Глава 6

Деревья поиска

6.1. АТД с быстрым поиском

Определение 6.1.1 (множество, set)**.** Абстрактный тип данных *множество* позволяет хранить набор однотипных элементов и реализует следующие запросы.

- `insert(k)` — добавить k в множество.
- `delete(k)` — удалить k из множество.
- `find(k)` — найти k в множестве.

Если разрешается хранить несколько одинаковых значений, то такой АТД называется *мультимножеством*.

Определение 6.1.2 (отображение, словарь, map, dictionary)**.** Абстрактный тип данных *отображение* позволяет хранить набор пар "ключ-значение" и реализует следующие запросы.

- `insert(k, v)` — добавить значение v с ключом k.
- `delete(k)` — удалить запись с ключом k.
- `find(k)` — найти запись с ключом k.

Если разрешается хранить несколько пар с одинаковыми ключами, то такой АТД называется *мультиотображение*.

Заметим, что имея реализацию множества, мы можем реализовать и отображение: будем хранить в множестве пары и сравнивать их только по первому полю. Поэтому в дальнейшем мы будем говорить только о реализации множества, понимая при этом, что все полученные результаты справедливы и для отображения.

Нас будет интересовать эффективная реализация множества в случае, если на элементах множества задан порядок (т.е. элементы множества можно сравнивать). Для эффективной реализации множества применяются деревья поиска.

Замечание 6.1.1. Если множество или отображение не изменяется после начального заполнения (т.е. сначала выполняются все запросы `insert( )`, а потом поступают только запросы `find( )`), то можно реализовать эти структуры на упорядоченном массиве: отсортируем массив после заполнения и будем искать элементы двоичным поиском.

6.2. Представление корневых деревьев в памяти

Перед тем, как говорить о деревьях поиска, полезно обсудить, как можно хранить корневые деревья в памяти компьютера. Существует два основных подхода.

- **Хранение в массиве.** Такой способ мы применяли для хранения двоичной кучи. Этот способ хорошо подходит для хранения двоичных деревьев, которые являются полными или почти полными. В этом случае корень хранится в ячейке с номером 1, его сыновья в ячейках 2 и 3, и в общем случае для вершины в ячейке i номера ячеек сыновей и предка вычисляются по формулам:

$$\text{left}(i) = 2i, \quad \text{right}(i) = 2i + 1, \quad \text{parent}(i) = \lfloor i/2 \rfloor.$$

  Такой подход можно обобщить на деревья любой ограниченной арности. Преимущество данного подхода в отсутствии необходимости хранить ссылки и отсутствии фрагментации памяти. Однако, если дерево сильно отличается от полного, то в данном представлении будет слишком много пустых ячеек, т.е. память будет использовать неэкономно.

- **Хранение в виде набора записей.** При таком способе хранения каждая вершина задаётся записью со ссылками на потомков, родителя и другими данными. Само дерево задаётся ссылкой на корень. При таком подходе не важно, является ли дерево полным, — количество используемой памяти пропорционально количеству вершин в дереве. Более того, данный подход позволяет хранить деревья произвольной арности — для этого в вершинах можно хранить массив или список потомков. К минусам данного подхода стоит отнести возможную фрагментацию памяти (зависит от способа хранения записей). Кроме того, могут потребоваться дополнительные затраты на управление памятью.

Представление «левый ребёнок — правый сосед». Как мы уже отметили, представление корневого дерева в виде набора записей позволяет хранить деревья произвольной арности. Однако, в некоторых случаях желательно, чтобы каждая вершина имела фиксированный размер. В таких случаях можно использовать представление «левый ребёнок — правый сосед». В этом случае каждая вершина хранит две ссылки: на левого ребёнка и на правого соседа. Такое представление, с одной стороны, сохраняет фиксированным размер вершины, а с другой стороны позволяет эффективно перемещаться по дереву, как если бы мы хранили потомков вершины в виде списка.

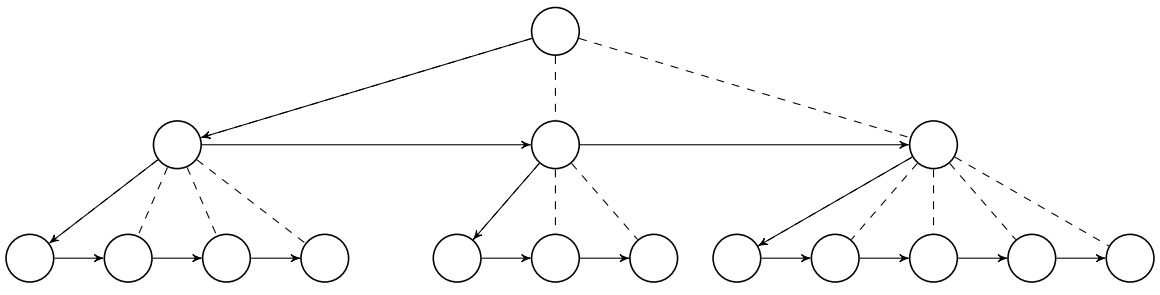


Рис. 6.2.1. Представление «левый ребёнок — правый сосед».

6.3. Двоичные деревья поиска

Определение 6.3.1. *Двоичное дерево поиска* — это структура данных, которая представляет корневое двоичное дерево, состоящее из однотипных узлов. Для значений, которые хранятся в узлах, поддерживается следующий инвариант: элемент в узле не меньше всех элементов в его левом поддереве и не больше всех элементов в его правом поддереве.

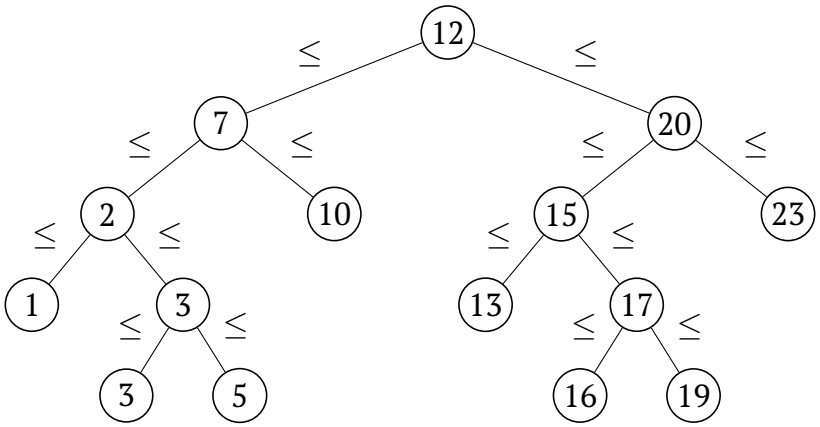


Рис. 6.3.1. Двоичное дерево поиска.

Для реализации в каждом узле будем хранить следующие поля:

- `key` — элемент множества,
- `left` — левый потомок,
- `right` — правый потомок,
- `parent` — родитель.

Само же дерево будет задаваться указателем на корень.

6.3.1. Операции поиска

Начнём с рекурсивного обхода дерева, при котором элементы будут перебираться в порядке возрастания (это т.н. *inorder обхода дерева*).

```
# inorder обход дерева
def inorder-traverse(v = root):
    if v == None:
        return
    inorder-traverse(v.left)
    print v.key
    inorder-traverse(v.right)
```

Утверждение 6.3.1. *Дерево T является деревом поиска тогда и только тогда, когда его inorder обход печатает ключи в неубывающем порядке.*

Тогда поиск элемента можно описать следующим псевдокодом.

```
# Поиск элемента в дереве поиска
def find(k, v = root):
    if v == None or k == v.key:
        return v
    if k < v.key:
        return find(k, v.left)
    else
        return find(k, v.right)
```

Структура дерева поиска позволяет так же легко найти, например, минимальный или максимальный элемент.

```
# Поиск минимального элемента
def min_element(v = root):
    if v.left == None:
        return v
    return min_element(v.left)

# Поиск максимального элемента
def max_element(v = root):
    if v.right == None:
        return v
    return max_element(v.right)
```

Заметим, что предыдущие функции возвращают не элемент множества, а узел дерева. Имея ссылку на элемент дерева, можно перейти к следующему или предыдущему элементу.

```
# Поиск следующего элемента
def succ(v):
    if v.right != None:
        return min_element(v.right)
    while v.parent != None and v.parent.left != v:
        v = v.parent
    return v.parent

# Поиск предыдущего элемента
def pred(v):
    if v.left != None:
        return max_element(v.right)
    while v.parent != None and v.parent.right != v:
        v = v.parent
    return v.parent
```

6.3.2. Добавление элемента

Будем всегда добавлять новый элемент в качестве листа дерева. Поэтому нам нужно запустить поиск по дереву и найти подходящего родителя, т.е. такой узел, у которого нет одного из сыновей и его значение находится в правильном соотношении с добавляемым элементом.

```
# Добавление нового элемента
# NB: если одинаковые элементы запрещены,
# то наличие элемента нужно проверить отдельно вызовом find
def insert(k):
    newnode = node(k)
    if root != None:
        p = root
        while True:
            if k < p.key:
                if p.left == None:
                    p.left = newnode
                    newnode.parent = p
                    return newnode
                p = p.left
            else # k >= p.key
                if p.right == None:
                    p.right = newnode
                    newnode.parent = p
                    return newnode
                p = p.right
    else
        root = newnode
        return newnode
```

6.3.3. Удаление элемента

Заметим, что никаких проблем с удалением листа нет — у листа нет потомков, которые нужно куда-то перевешивать. Теперь предположим, что у удаляемой вершины нет одного из потомков, например левого. Тогда после удаления этой вершины её место займёт её правый потомок.

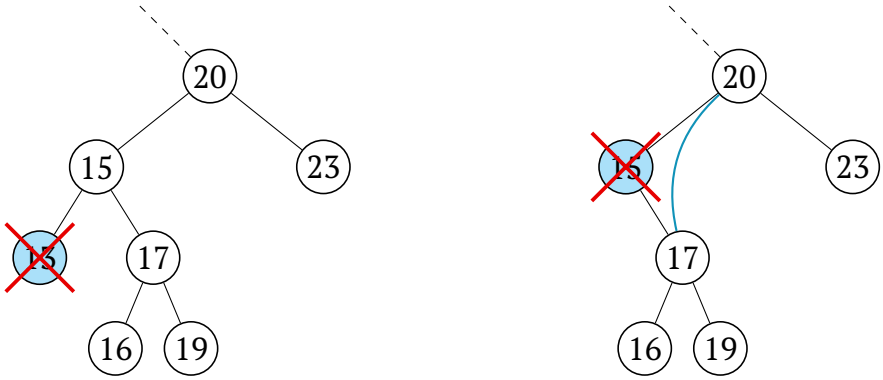


Рис. 6.3.2. Удаление листа и удаление вершины с одним потомком.

Осталось понять, что делать в случае удаления вершины, у которой есть оба потомка. Пусть мы хотим удалить некоторую вершину v, у которой есть оба потомка. Тогда в поддереве правого потомка обязательно будет вершина `u = succ(v)`, причём у вершины u гарантированно не будет левого потомка (следующая за v вершина — это минимальная вершина в поддереве её правого потомка). Обменяем вершины u и v местами и удалим v одним из описанных выше способов

(теперь у вершины v нет левого потомка). Нетрудно заметить, что в результате дерево останется деревом поиска.

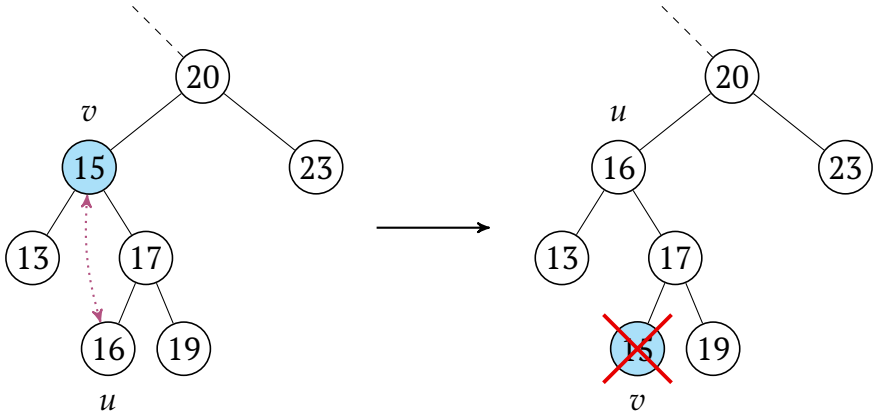


Рис. 6.3.3. Удаление вершины с двумя потомками.

Упражнение 6.3.1. Написать соответствующий код для удаление вершины.

Утверждение 6.3.2. *Все рассмотренные операции с деревом работают за $O(h)$, где h — высота дерева.*

Действительно, все рассмотренные выше операции делают не более одного спуска и не более одного подъёма по дереву. При этом в худшем случае высота дерева с n элементами может достигать $n-1$ (например, если последовательно добавлять в дерево поиска целые числа от 1 до n, то мы получим „бамбук“ длины $n-1$), т.е. операции с таким деревом будут работать за $O(n)$.

Для того, чтобы использование дерева поиска было разумным, нам нужно позаботиться о том, чтобы дерево имело высоту поменьше, тогда и операции будут выполняться быстрее. Минимальная высота двоичного дерева достигается, если оно *полное*, то есть все уровни дерева, кроме последнего, заполнены. Высота полного двоичного дерева с n вершинами равна $\lceil \log(n+1) \rceil - 1$, т.е. время работы всех рассмотренных операций будет не более $O(\log n)$.

Однако, поддерживать дерево в таком состоянии очень затратно (можно придумать примеры, когда при удалении или добавлении вершины нам придётся перевесить множество других вершин, чтобы снова сделать дерево полным). Вместо полноты двоичного дерева можно накладывать более слабые требования — требования *сбалансированности*, которые позволяют получить такую же асимптотическую оценку $O(\log n)$ на время работы операций, но при этом можно поддерживать дерево в сбалансированном состоянии, тратя на это не более $O(\log n)$ на каждую операцию. Существует несколько реализаций *сбалансированных деревьев поиска*, наиболее известные из которых — *красно-чёрные деревья* и *АВЛ-дерево*. Описание красно-чёрных деревьев можно посмотреть в [2] — на практике их использую чаще, но эта структура данных значительно более сложна в описании, нежели АВЛ-дерево.

Можно заметить, что оценка $O(\log n)$ на стоимость *всех* операции для дерева поиска — это лучшее, чего можно добиться. Действительно, пусть мы изобрели какое-то особенное дерево, для которого все операции работают за $o(\log n)$. Тогда мы можем построить дерево для произвольного набора элементов $a_1, \ldots, a_n$ за $o(n \log n)$ (n раз добавить вершину за $o(\log n)$). Если мы теперь обойдём это дерево (это можно сделать за $O(n)$ операций), то таким образом получим элементы

$a_1, \ldots, a_n$ в порядке возрастания, т.е. мы научились сортировать произвольные элементы за $o(n \log n)$, что противоречит нижней оценке на сортировку сравнениями.

6.4. АВЛ-дерево

6.4.1. Сбалансированность

АВЛ-дерево [3] придумано советскими учёными Г.М. Адельсоном-Вельским и Е.М. Ландисом. Будем называть двоичное дерево *сбалансированным*, если у любой его вершины высота её левого поддерева отличается от высоты её правого поддерева не более чем на 1.

Утверждение 6.4.1. *В сбалансированном дереве высоты h не более $2^h - 1$ вершины.*

Доказательство. Равенство достигается в случае полного двоичного дерева. □

Утверждение 6.4.2. *В сбалансированном дереве высоты h не менее 1.6^h вершин.*

Доказательство. Пусть M_h — это минимальное количество вершин в сбалансированном дереве высоты h. Тогда $M_h = 1 + M_{h-1} + M_{h-2}$. Заметим, что это рекуррентное соотношение очень похоже на соотношения для чисел Фибоначчи: $F_n = F_{n-1} + F_{n-2}$. Легко проверить, что $M_h = F_{h+3} - 1$. Используя оценку для чисел Фибоначчи $F_n \geq ((\sqrt{5}+1)/2)^n > 1.6^n$, получаем $M_h \geq F_{h+3} - 1 \geq 1.6^{h+3} - 1 > 1.6^h$. □

Следствие 6.4.1. *Сбалансированное дерево с n вершинами имеет высоту $\Theta(\log n)$.*

Доказательство. Прологарифмируем $2^h > n > 1.6^h$. □

Мы показали, что свойства сбалансированности достаточно для того, чтобы высота дерева была логарифмической, а следовательно, операции поиска, добавления и удаления вершины работали бы за $O(\log n)$. Осталось разобраться, как поддерживать дерево сбалансированным после добавления или удаления вершины.

6.4.2. Вращения

Для поддержания дерева в сбалансированном состоянии применяются локальные коррекции дерева, называемые *вращениями*. Вращения применяются, если после добавления или удаления вершины, дерево перестало быть сбалансированным. Вращение позволяет восстановить нарушенный баланс и при этом дерево остаётся деревом поиска.

Утверждение 6.4.3. *При добавлении (удалении) вершины в сбалансированное дерево, баланс мог нарушиться только у тех вершин, которые лежат на пути от корня к добавленной (удалённой) вершине. При этом разница высот поддеревьев в таких вершинах не может быть больше 2.*

Пусть x — первая вершина на пути от добавленной/удалённой вершины к корню, в которой нарушился баланс. Для восстановления баланса к ней нужно применить одно из двух вращений.

Малые вращения. Применяются в случае, когда левое поддерево левого потомка имеет высоту на 2 больше, чем правое поддерево x, и в симметричном случае. Пусть y — левый потомок x. При *малом правом вращении* y занимает место x, а x в свою очередь становится её правым потомком и забирает одно из поддеревьев, см. рис. 6.4.1 (высота поддерева β может совпадать с высотой α или быть на 1 меньше). При этом дерево остаётся деревом поиска. Для того, чтобы проверить это, выпишем отношения вершин этого дерева до и после вращения и в обоих случаях мы получим одинаковый порядок $\alpha \le y \le \beta \le x \le \gamma$.

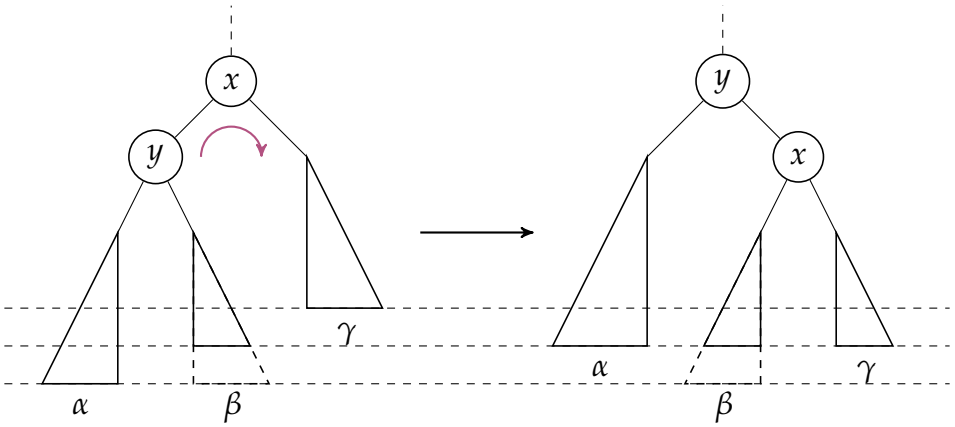


Рис. 6.4.1. Малое правое вращение АВЛ-дерева.

Большое вращение. Большие вращения применяется в случаях, когда малые вращение не могут исправить баланс, т.е. когда правое поддерево левого потомка x имеет высоту на 1 больше, чем высота правого поддерева x, и в симметричном случае. Большое вращение складывается из двух малых вращений, см. рис. 6.4.2 (высота одного из поддеревьев β и γ должна совпадать с высотой α, а у второго может быть такой же или на 1 меньше). Можно проверить, что в этом случае одним малым вращением не исправить баланс в вершине x. Большое вращение — это композиция двух малых вращений, каждое из которых сохраняет дерево деревом поиска, поэтому в результате большого вращения дерево так же остаётся деревом поиска.

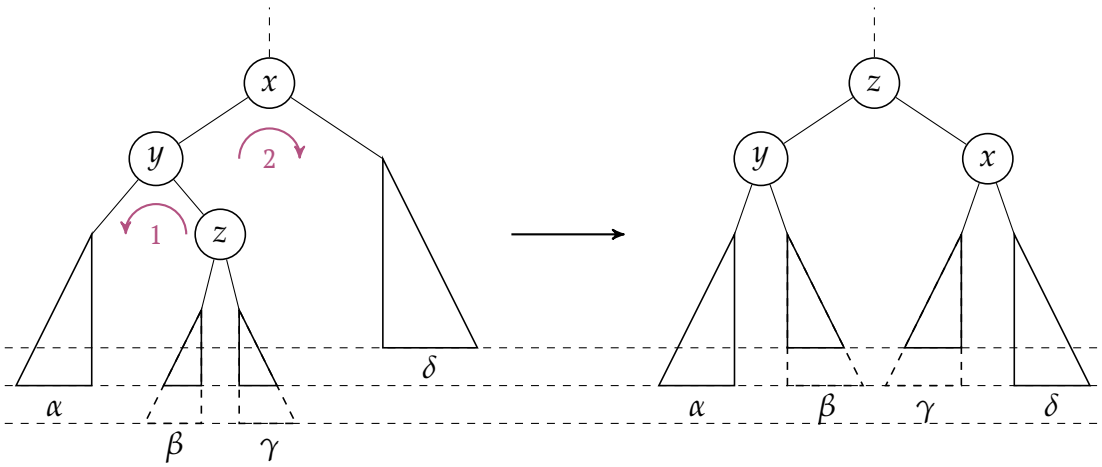


Рис. 6.4.2. Большое правое вращение АВЛ-дерева.

Утверждение 6.4.4. *Малое и большое вращения покрывают все возможные случаи нарушения баланса, когда высота поддеревьев отличается на 2.*

Восстановление сбалансированности. Теперь опишем как происходит восстановление сбалансированности при помощи вращений. Предположим, что мы добавили или удалили какую-то вершину из дерева. Будем подниматься от этой вершины к корню, пока не встретим какую-нибудь вершину, в которой баланс нарушен. Пусть x — такая первая вершина. Так как мы добавили или удалили только одну вершину, то разница высот поддеревьев x не может быть больше 2, а значит, мы можем применить вращение, чтобы восстановить баланс. При этом вращения не могут нарушить баланс ниже по дереву, поэтому если после восстановления баланса в x в дереве и остались несбалансированные вершины, то они находятся выше по дереву на пути к корню. Будем продолжать подниматься и исправлять баланс во всех встреченных несбалансированных вершинах.

Замечание 6.4.1. В данном рассуждении пропущен важный момент: мы не объяснили, почему не может случиться так, что после нескольких вращений баланс в одной из вершин-предков нарушится более чем на 2. Для того, чтобы это объяснить, нужно сделать следующие наблюдения.

- Вращения могут только уменьшить высоту какого-то поддерева. Поэтому интересующая нас ситуация, когда у какой-то вершины выше баланс нарушился более чем на 2, могла произойти только при удалении.
- При удалении вершины баланс нарушается только в одной вершине (при добавлении может нарушиться в нескольких). И в дальнейшем, каждое вращение, которое исправляет это дисбаланс может нарушить баланс только в одной вершине выше, т.к. может только уменьшить высоту.

Из этого можно сделать вывод, что такая ситуация невозможна.

Оценка времени работы. Количество вращений не больше, чем вершин на пути от листа к корню, т.е. $O(\log n)$. Каждое вращение можно выполнить за $O(1)$, т.к. оно затрагивает константное количество вершин. В сумме получается, что на восстановление баланса после изменяющей дерево операции нам потребуется $O(\log n)$.

Замечание 6.4.2. Для эффективной реализации АВЛ-дерева в каждой вершине следует хранить и поддерживать высоту её поддерева.

6.5. Splay-дерево

Splay-дерево — это самобалансирующееся двоичное дерево поиска, изобретённое Слеатором и Тарьяном [4, 1] в 1983. В отличие от АВЛ-дерева, которое обеспечивает оценки *в худшем*, splay-дерево имеет *амортизированную* оценку $O(\log n)$ на время работы основных операций. При этом реализовать splay-дерево немного проще, чем АВЛ-дерево, т.к. не нужно хранить и поддерживать высоту поддерева в каждой вершине, а также не нужно рассматривать разные случаи при удалении вершины.

6.5.1. Запросы к splay-дереву

Ключевую роль в splay-дереве играет операция `splay(v)`, которая перестраивает дерево при помощи вращений так, чтобы вершина `v` оказалась в корне.

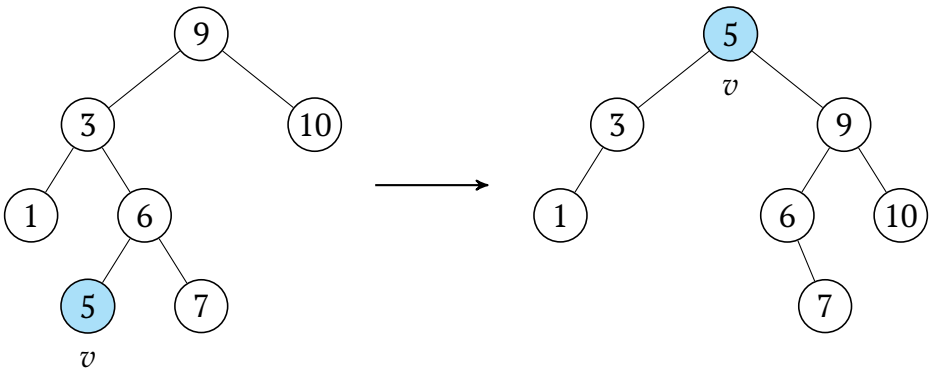


Рис. 6.5.1. Работа операции `splay(v)`.

Более подробно устройство этой операции мы рассмотрим позже, а пока с её помощью научимся реализовывать другие операции.

Операция `splay` вызывается для каждой вершины, которая была найдена, добавлена или удалена.

```
# Поиск элемента в splay-дереве
def splay-find(k, v = root):
    if k == v.key:
        splay(v)
        return v
    if k < v.key:
        if v.left == None:
            splay(v)
            return None
        return splay-find(k, v.left)
    else
        if v.right == None:
            splay(v)
            return None
        return splay-find(k, v.right)

# Добавление вершины в splay-дерево
def splay-insert(k):
    # вызов insert для обычного дерева поиска
    v = insert(k)
    splay(v)
    return v
```

Для реализации удаления давайте введём две вспомогательные операции — split и merge. Операция `split(k)` дерево на три части: дерево с элементами $\leq k$, вершина с ключом k, дерево с элементами $\geq k$.

```
# Разбиение дерева относительно ключа k
# предполагается, что ключ k в дереве есть
def splay-split(k):
    v = splay-find(k)
    T1 = v.left
    T2 = v.right
```

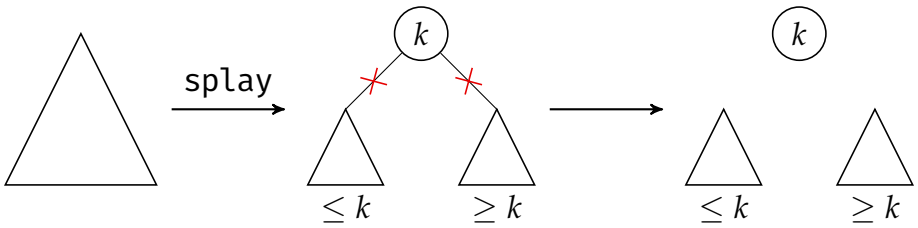


Рис. 6.5.2. Работа операции `splay-split(k)`.

```
    v.left = None
    v.right = None
    return (T1, v, T2)
```

Операция merge объединяет два дерева в одно (требуется, чтобы все ключи первого дерева были меньше ключей второго дерева).

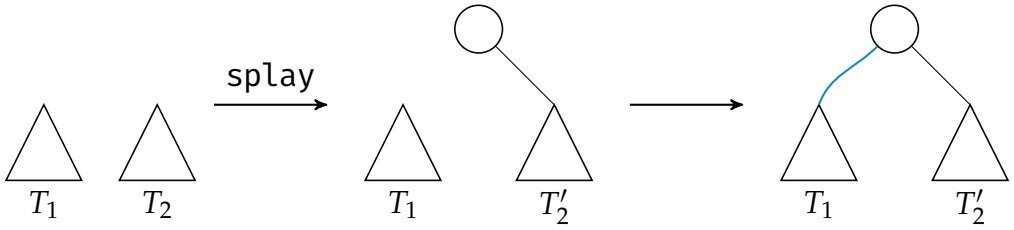


Рис. 6.5.3. Работа операции `splay-merge(T1, T2)`.

```
# Слияние двух деревьев
# ключи T1 меньше ключей T2
def splay-merge(T1, T2):
    v = min_element(T2)
    splay(v)
    v.left = T1
    return v
```

Теперь можно описать операцию удаления вершины.

```
# Удаление вершины с ключом k
def splay-remove(k):
    (T1, v, T2) = splay-split(k)
    return splay-merge(T1, T2)
```

Как мы увидим далее, сложность операции `splay` ограниченна $O(h)$, где h — высота дерева, соответственно, верно следующее утверждение.

Утверждение 6.5.1. *Все рассмотренные операции имеют сложность не более* $O(h)$.

6.5.2. Операция `splay`

Мы научились реализовывать все необходимые операции со splay-деревом, кроме самой операции `splay`, которая используется в каждой из рассмотренных операций как подпроцедура. Теперь мы рассмотрим устройство операции `splay` и докажем амортизированную оценку $O(\log n)$ на её сложность. Начнём со следующих двух утверждений.

Лемма 6.5.1. *Истинная стоимость* `splay(v)` *пропорциональна глубине вершины* v.

Эта лемма сама по себе не является особенно примечательной, т.к. реализация операции `splay` „в лоб“ при помощи малых вращений для АВЛ дерева имела бы такую же оценку (каждое малое вращение уменьшает глубину вершины на 1).

Вторая лемма значительно более интересная.

Лемма 6.5.2. *Существует целочисленная функция потенциала* Φ*, определённая на двоичных деревьях и принимающая неотрицательные значения, такая, что*

(a) для любого дерева с n *вершинами* $\Phi(T) = O(n \log n)$*,*

(b) учётная стоимость любой операции `splay` *относительно* Φ *не более* $O(\log n)$*,*

(c) добавление листа увеличивает потенциал не более чем на $\log n$*,*

(d) сумма потенциалов деревьев, получившихся из T *удалением корня, меньше исходного потенциала* T *не более чем на* $\log n$*,*

(e) при подсоединении одного дерева к корню другого потенциал получившегося дерева не более чем на $\log n$ *превосходит сумму потенциалов исходных деревьев.*

Вместе лемма 6.5.1 и пункты (a) и (b) леммы 6.5.2 позволяют сделать следующее неожиданное заключение. Предположим, что мы начали с некоторого splay-дерева с n вершинами и $m \geq n$ раз выполнили следующую операцию: нашли самую глубокую вершину и вызвали для неё `splay`. По лемме 6.5.2 суммарная (истинная) стоимость этих операций не будет превосходить $O(m \log n) + O(n \log n) = O(m \log n)$. С другой стороны мы ведь каждый раз вызывали `splay` для самой глубокой вершины, а значит, по лемме 6.5.1 количество операций было пропорционально высоте дерева. Получается, что и высота дерева в среднем была $O(\log n)$, т.е. за счёт вызовов `splay` дерево самобалансируется и „в среднем“ имеет небольшую высоту.

Теперь покажем, как можно обобщить это умозаключение на m произвольных (из рассмотренных выше) операций со splay-деревом. Все рассмотренные операции устроены более-менее одинаково: мы сначала спускаемся по дереву до некоторой вершины, а потом вызываем для неё `splay`. Рассмотрим подробнее операцию `splay-find`: в поиске ключа мы спускаемся от корня вниз пока не найдём подходящую вершину или не убедимся, что её нет, а потом вызываем `splay` от вершины, на которой поиск завершился. Заметим что если мы спустились на глубину d, то splay совершит $\Theta(d)$ операций, чтобы поднять вершину в корень (по лемме 6.5.1). Таким образом, сколько бы операций спуска по дереву мы не сделали, примерно столько же операций подъёма по дереву произойдёт внутри `splay` (тут важно, что если мы не нашли вершину, то мы всё-равно вызываем `splay` последней вершины в поиске). Получается, что стоимость `splay-find` лишь в константу раз больше стоимости вызова `splay` внутри неё. Соответственно, если над деревом размера n выполнить $m \geq n$ операций, некоторые из которых `splay`, а некоторые — `splay-find`, то суммарная сложность получится не более $O(m \log n)$, т.е. в среднем $O(\log n)$ на каждую операцию.

Можно ли перенести это рассуждение и на другие операции? Рассмотрим, например, `splay-insert`. Аналогичные рассуждения показывают, что истинная стоимость `splay-insert` лишь в константу раз больше, чем истинная стоимость `splay` внутри неё. Однако тут мы не учли, что `splay-insert`, в отличие от `splay-find`,

изменяет дерево, а значит, изменяет и его потенциал, и таким образом учётная стоимость `splay-insert` может оказаться значительно больше, чем учётная стоимость `splay`. Тут нам нужно воспользоваться пунктом (c) леммы 6.5.2 о том, что добавление листа изменяет потенциал не более чем на $\log n$, а значит, учётная стоимость `splay-insert` тоже ограничена $O(\log n)$.

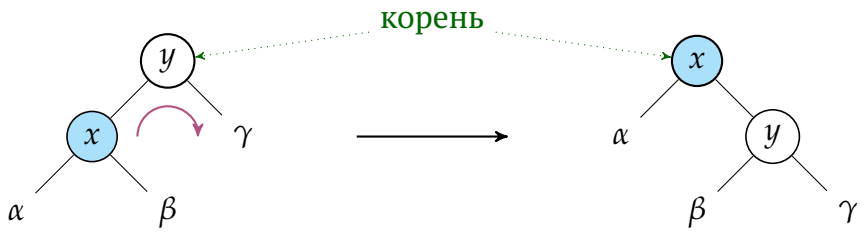
Аналогичные рассуждения позволяют показать, что учётная стоимость операции `splay-remove` не превышает $O(\log n)$. Удаление вершины состоит из двух спусков по дереву, из двух вызовов `splay` для соответствующих вершин, удалении корня и подсоединении одного дерева к корню другого. Осталось сказать, что сложность спусков не более чем в константу раз превышает сложность соответствующих операций `splay`, а операции удаления корня (пункт (d) леммы 6.5.2) и подсоединения одного дерева к корню другого (пункт (e) леммы 6.5.2) суммарно изменяют потенциал не более чем на $\log n$.

Таким образом, если над splay-деревом выполнить $m \geq n$ произвольных операций, в процессе которых в дереве никогда не будет более n элементов, то суммарная сложность будет не более $O(m \log n)$, т.е. в среднем $O(\log n)$ на каждую операцию.

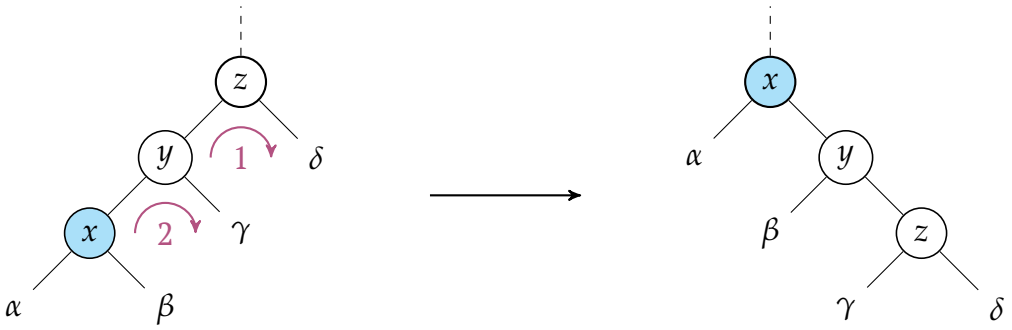
Осталось разобраться в устройстве функции и доказать леммы 6.5.1 и 6.5.2.

Вращения. Как уже было сказано, операция `splay` основана на вращениях. Опишем её работу для вершины x. Если x является корнем, то ничего делать не нужно. В противном случае применяется один из трёх шагов-вращений (или один из симметричных им).

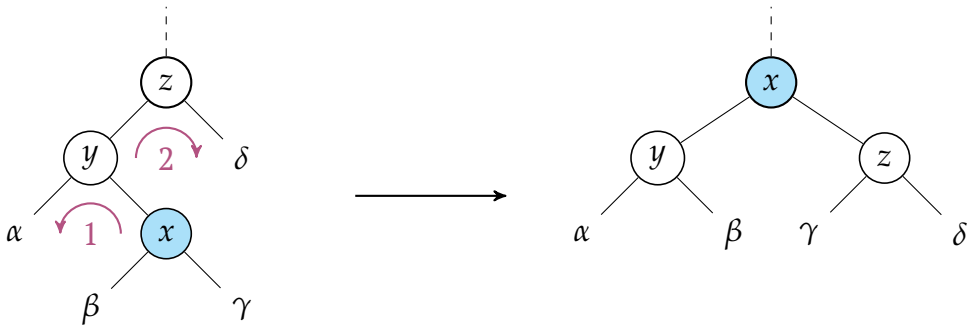
- **zig-шаг.** Применяется только в том случае, если предок вершины x является корнем дерева.



- **zigzig-шаг.** Применяется, если x является левым сыном и левым внуком одновременно.



- **zigzag-шаг.** Применяется, если x является правым сыном и левым внуком одновременно.



Замечание 6.5.1. При реализации zigzig и zigzag-шагов **важен порядок вращений**. Если бы в zigzig-шаге можно было бы поменять вращения местами, то не было бы необходимости рассматривать zigzig и zigzag-шаги отдельно, ведь они соответствовали бы двум zig-шагам. Но в этом случае получится другая структура данных, про которую лемму 6.5.2 уже доказать не получается.

6.5.3. Анализ сложности

Из описания операции `splay` следует лемма 6.5.1 (количество вращений пропорционально глубине, а каждое вращение требует не более $O(1)$ операций). Докажем лемму 6.5.2.

Доказательство леммы 6.5.2. Для каждой вершины x определим *вес* $w(x)$ равным количеству вершин в поддереве с корнем x. Теперь мы можем определить функцию *потенциала для вершины* $\Phi(x)$ равной $\lfloor \log w(x) \rfloor$. И наконец, определим потенциал для всего дерева как сумму потенциалов его вершин.

Потенциал каждой вершины не превосходит $O(\log n)$, соответственно, потенциал дерева не превосходит $O(n \log n)$. Таким образом Φ удовлетворяет требованию пункта (a).

Рассмотрим один вызов функции `splay(x)` для дерева с корнем r. После вызова функции `splay(x)` потенциал вершины x изменится на $\Phi_0(r) - \Phi_0(x)$, где Φ_0 обозначает функцию потенциала до вызова `splay`. Мы покажем, что учётная стоимость этой операции относительно Φ не превосходит $3(\Phi_0(r) - \Phi_0(x)) + 1$. Поскольку потенциал любой вершины не превосходит $\log n$, то и учётная стоимость `splay` будет ограничена $3 \log n + 1 = O(\log n)$.

Разобьём операцию `splay` на шаги в соответствии с её определением и оценим учётную стоимость каждого шага по-отдельности. Пусть Φ обозначает функцию потенциала до совершения шага, а Φ' — после. Тогда утверждается, что

1. учётная стоимость zig-шага не превосходит $3(\Phi'(x) - \Phi(x)) + 1$.
2. учётная стоимость zigzig-шага не превосходит $3(\Phi'(x) - \Phi(x))$.
3. учётная стоимость zigzag-шага не превосходит $3(\Phi'(x) - \Phi(x))$.

После вызова `splay` вершина x станет корнем дерева, поэтому суммируя все эти оценки по все шагам мы получаем, что учётная стоимость `splay(x)` не превосходит $3(\Phi_0(r) - \Phi_0(x)) + 1$ (мы пользуемся тем, что в `splay` не более одного zig шага и тем, что потенциал корня не изменяется).

Начнём с оценки zig-шага. Истинная стоимость любого вращения $O(1)$. Для простоты будем считать, что она равна 1. Тогда учётная стоимость zig-шага складывается из его истинной стоимости и приращения потенциала дерева, т.е. равна

$$\tilde{c} = 1 + \Phi'(x) + \Phi'(y) - \Phi(x) - \Phi(y).$$

Здесь $\Phi'(x) = \Phi(y)$ — это потенциал корня, поэтому $\Phi'(y) \le \Phi'(x)$. Подставляем и получаем требуемую оценку

$$\tilde{c} = 1 + \Phi'(y) - \Phi(x) \le 1 + \underbrace{\Phi'(x) - \Phi(x)}_{\ge 0} \le 1 + 3(\Phi'(x) - \Phi(x)).$$

Теперь докажем оценку для zigzig-шага — для zigzag-шага оценка доказывается аналогично. Выпишем учётную стоимость шага: $\tilde{c} = 1 + \Delta\Phi$, где

$$\Delta\Phi = \Phi'(x) + \Phi'(y) + \Phi'(z) - \Phi(x) - \Phi(y) - \Phi(z).$$

Воспользуемся тем, что потенциал корня не изменяется, т.е. $\Phi'(x) = \Phi(z)$, а также тем, что

$$\Phi'(z) \le \Phi'(y) \le \Phi'(x) \quad \text{и} \quad \Phi(y) \ge \Phi(x). \tag{6.5.1}$$

Получаем, что

$$\Delta\Phi = \Phi'(y) + \Phi'(z) - \Phi(x) - \Phi(y) \le 2(\Phi'(x) - \Phi(x)).$$

Рассмотрим два случая. Пусть сначала $\Phi'(x) > \Phi(x)$, тогда

$$\tilde{c} \le 1 + 2(\Phi'(x) - \Phi(x)) \le 3(\Phi'(x) - \Phi(x)) \tag{6.5.2}$$

(здесь мы пользуемся целочисленностью потенциала), и утверждение доказано.

Теперь предположим, что $\Phi'(x) = \Phi(x)$ (меньше быть не может, т.к. после этого шага x стал корнем поддерева). В этом случае неравенство (6.5.2) уже неверно. Однако это не значит, что доказываемое утверждение неверно, т.к. неравенство (6.5.2) получилось применением неравенств (6.5.1), и в этот момент мы могли что-то потерять. Действительно, покажем, что в этом случае

$$\Delta\Phi < 0 = 2(\Phi'(x) - \Phi(x)) = 3(\Phi'(x) - \Phi(x)),$$

а следовательно,

$$\tilde{c} = 1 + \Delta\Phi \le 0 = 3(\Phi'(x) - \Phi(x)),$$

Проведём доказательство от обратного. Предположим, что $\Phi'(x) = \Phi(x)$ и $\Delta\Phi = 0$, т.е. что все неравенства (6.5.1) на самом деле являются равенствами. Тогда получается, что все шесть потенциалов $\Phi'(x), \Phi'(y), \Phi'(z), \Phi(x), \Phi(y), \Phi(z)$ равны между собой и равны какому-то числу k. Покажем, что так не бывает. Пусть w и w' обозначают веса вершин до и после шага. Заметим, что после шага веса поддеревьев $\alpha, \beta, \gamma, \delta$ не изменились — могли измениться только веса вершин x, y, z. Тогда

$$\begin{aligned} w'(x) &= w'(\alpha) + w'(\beta) + w'(\gamma) + w'(\delta) + 3 \\ &= (1 + w(\alpha) + w(\beta)) + (1 + w'(\gamma) + w'(\delta)) + 1 \\ &= w(x) + w'(z) + 1 \ge 2^k + 2^k + 1 \ge 2^{k+1}, \end{aligned} \tag{6.5.3}$$

а значит, $\Phi'(x) \ge k + 1$, что противоречит нашему предположению.

Осталось доказать пункты (c)–(e).

(c) Добавление листа увеличивает вес всех вершин на пути от корня к листу. У скольких из них при этом может увеличиться потенциал? Только у тех, вес которых стал равен 2^k для некоторого целого k. Так как вес вершин на пути от листа к корню строго возрастает в промежутке 0 до n, то таких вершин не может быть более $\log n$.

(d) Удаление корня уменьшает суммарный потенциал деревьев на потенциал корня, т.е. не более чем на $\log n$.

(e) Подсоединение одного дерева к корню другого увеличивает потенциал корня, а он не может превышать $\log n$. □

Упражнение 6.5.1. Докажите оценку для zigzag-шага.

Упражнение 6.5.2. Проверьте, что если изменить порядок вращений в zigzig-шаге, то желаемое утверждение не получится доказать.

Если немного обобщить функцию потенциала в этой лемме, то можно доказать интересное свойство splay-деревьев.

Лемма 6.5.3. *Пусть в splay-дереве хранятся n ключей $k_1, k_2, \ldots, k_n$. Рассмотрим набор из $m \geq n$ запросов* `splay-find` *таких, что каждый ключ запрашивается хотя бы один раз. Для каждого $i \in [n]$ обозначим через $q_i \geq 1$ количество запросов ключа k_i. Тогда суммарное количество операций на выполнение всех запросов будет ограничено $O\left(m + \sum_i q_i \log(m/q_i)\right)$.*

Доказательство. Давайте посмотрим, как изменится доказательство леммы 6.5.2, если каждой вершине x назначить некоторый вес $\rho(x) \geq 1$ и определить $w(x)$ как сумму весов всех вершин в поддереве с корнем x. Тогда потенциал $\Phi(x)$ вершины x будет ограничен $O(\log(\sum_{i=1}^{n} \rho(i)))$. При таком определении w неравенство (6.5.3), остаётся верным:

$$\begin{aligned} w'(x) &= w'(\alpha) + w'(\beta) + w'(\gamma) + w'(\delta) + \rho(x) + \rho(y) + \rho(z) \\ &= (\rho(x) + w(\alpha) + w(\beta)) + (\rho(z) + w'(\gamma) + w'(\delta)) + \rho(y) \\ &\geq w(x) + w'(z) + 1 \geq 2^k + 2^k + 1 \geq 2^{k+1}, \end{aligned}$$

Таким образом учётная стоимость операции `splay` для вершины x будет не превосходить

$$3(\Phi_0(r) - \Phi_0(x)) + 1 \leq 3\left(\log \sum_{i=1}^{n} \rho(i) - \log \rho(x)\right) + 1.$$

Отсюда суммарное количество операций на m запросов не будет превосходить $3\sum_{i=1}^{n} q_i \cdot (\Phi_0(r) - \Phi(i)) + m$. Положим теперь $\rho(i) = q_i$ для всех $i \in [n]$. При таком выберем весов получаем финальную оценку

$$O\left(m + \sum_{i=1}^{n} q_i \cdot \log \frac{m}{q_i}\right) = O\left(m + m \cdot H\left(\frac{q_1}{m}, \ldots, \frac{q_n}{m}\right)\right).$$

□

Следствие 6.5.1 (Статическая оптимальность splay-дерева)**.** *Суммарное количество операций на выполнение m запросов поиска в splay-дереве не больше, чем в любом статическом двоичном дереве поиска.*

Доказательство. Заметим, что оптимальное количество операций достигается на соответствующем дереве кода Хаффмена с частотами $\frac{q_1}{m}, \frac{q_2}{m}, \ldots, \frac{q_n}{m}$. □

6.6. Декартово дерево

6.6.1. Определение и теорема существования

Определение 6.6.1. *Декартово дерево (cartesian tree)* — это бинарное дерево, в каждой вершине которого хранится пара значений (ключ, приоритет), и которое является одновременно деревом поиска по ключу и кучей по приоритету. Впервые рассмотрены Вулемином [5] в 1980.

Совершенно не очевидно, что декартово дерево существует для любого набора пар ключей и приоритетов. Поэтому мы начнём со следующей теоремы.

Теорема 6.6.1. *Для любого набора входных пар* $(k_1, p_1), (k_2, p_2), \ldots, (k_n, p_n)$ *существует декартово дерево.*

Доказательство. Мы предложим алгоритм построения декартова дерева. Предположим, что пары уже отсортированы по ключу, т.е. $k_i < k_{i+1}, \forall i \in [n-1]$ (в противном случае отсортируем их). Теперь найдём пару с максимальным приоритетом, пусть это пара (k_i, p_i). Назначим вершину (k_i, p_i) корнем дерева и рекурсивно вызовем алгоритм на $(k_1, p_1), \ldots, (k_{i-1}, p_{i-1})$ и $(k_{i+1}, p_{i+1}), \ldots, (k_n, p_n)$, получая таким образом левое и правое поддерево соответственно. Заметим теперь, что в корне выполняются одновременно свойство дерева поиска и свойство кучи. Следовательно, в результате мы получим декартово дерево (см. 6.6.1). □

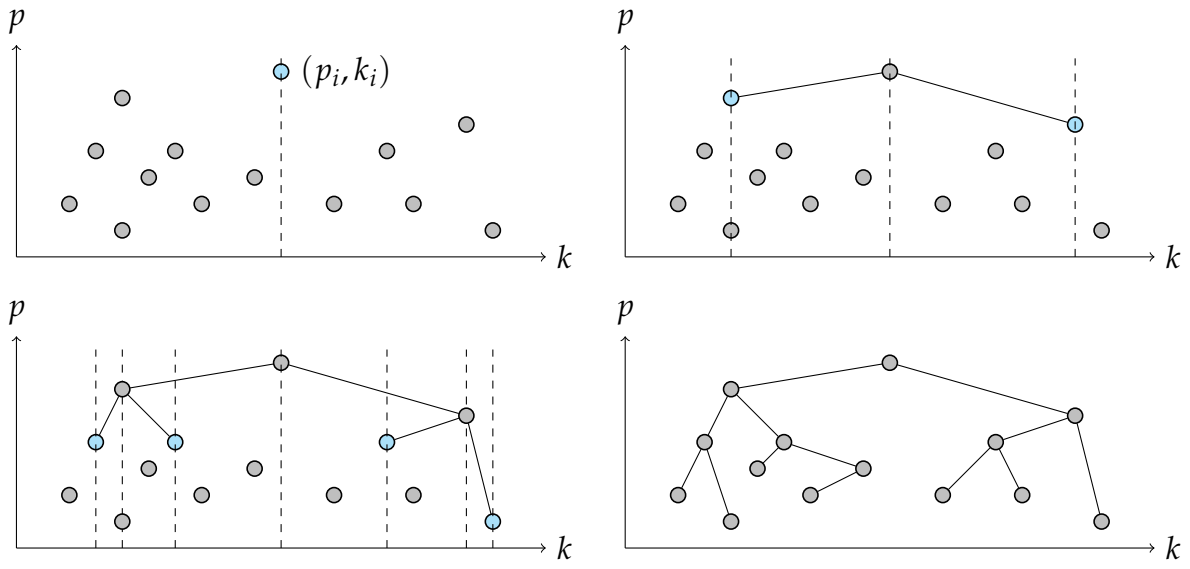


Рис. 6.6.1. Геометрическая интерпретация построения декартова дерева.

Давайте теперь оценим время полученного алгоритма: мы n раз ищем максимум, т.е. получается $O(n^2)$. Легко заметить, что этот алгоритм действительно будет работать за квадратичное время, если, например, приоритеты тоже отсортированы по возрастанию. Кроме того, этот алгоритм построения позволяет доказать следующее утверждение.

Утверждение 6.6.1. *Если все ключи и приоритеты различны, то декартово дерево определено однозначно.*

Действительно, в этом случае на каждой итерации корнем будет выбираться вершина с наибольшим приоритетом, и только эта вершина может быть корнем, иначе нарушится свойство кучи. Аналогично, уникальность ключей гарантирует, что после выбора корня каждая вершина однозначно будет отнесена либо к левой, либо к правой части.

6.6.2. Эффективное построение

Нам уже известно, что в общем случае построить дерево поиска быстрее чем за $\Theta(n \log n)$ не получится. Кроме того, мы умеем строить АВЛ-дерево и splay-дерево за $O(n \log n)$. Для декартова же дерева у нас пока есть только квадратичный в худшем случае алгоритм. В этом разделе мы разработаем алгоритм, который будет работать за линейное время при условии, что входные данные уже отсортированы по ключу (в противном случае нам придётся предварительно отсортировать входные данные, и алгоритм построения будет работать за $O(n \log n)$).

Будем добавлять вершины в дерево в порядке возрастания ключа. Каждая следующая вершина имеет ключ не меньше предыдущих, а следовательно, её можно будет добавить самой последней вершиной в самой правой ветке. Однако при этом мы не должны нарушить свойство кучи. Пусть $v_1, v_2, \ldots, v_t$ — это вершины правой ветки в порядке от корня вниз (т.е. v_1 — это корень). Предположим, что мы добавляем вершину $v = (k_i, p_i)$. Имеет место один из трёх случаев (см. рис. 6.6.2). Если p_i меньше приоритета v_t, то добавим v правым потомком вершины v_t. Если, напротив, p_i больше приоритета корня, то сделаем вершину v новым корнем, а v_1 — её левым потомком. Остался случай, когда p_i меньше приоритета корня, но больше приоритета v_t. Найдём пару таких вершин v_j и v_{j+1}, что p_i меньше приоритета v_j, но больше v_{j+1}. Тогда v становится правым потомком v_j, а v_{j+1} — левым потомком v.

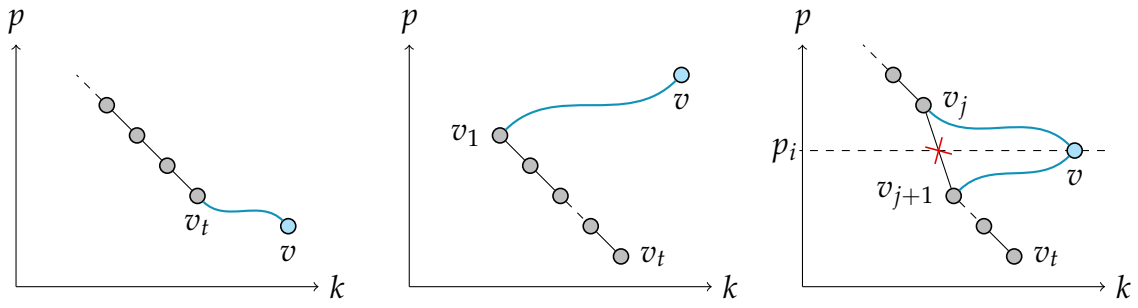


Рис. 6.6.2. Три случая добавления следующей вершины.

Оценим время работы этого алгоритма. Если при добавлении новой вершины мы будем каждый раз перебирать все вершины $v_1, \ldots, v_t$, то в худшем случае алгоритм получится квадратичным (действительно, квадратичное время достигается, например, если на вход алгоритму подать следующий набор пар $(1, n), (2, n-1), \ldots, (n, 1)$). Кажется, что мы ничего не выиграли по сравнению с первым рекурсивным алгоритмом построения. Однако, если изменить порядок и перебирать вершины $v_1, \ldots, v_t$ с конца (т.е. дополнительно хранить указатель на последнюю вершину в правой ветви и перебирать вершины от последней к корню), то алгоритм будет иметь линейное(!) время работы.

Давайте разберёмся, почему так происходит. Когда мы добавляем новую вершину v между вершинами v_j и v_{j+1} (третий случай на рис. 6.6.2), то вершины

$v_{j+1}, \ldots, v_t$ попадают в левое поддерево v и перестают принадлежать правой ветке. Поэтому вершины $v_{j+1}, \ldots, v_t$ никогда больше не будут встречаться при поиске места для новой вершины. Другими словами, если мы ищем подходящую пару вершин в правой ветке с конца, то каждое просмотренное на этой итерации ребро (v_k, v_{k+1}) больше никогда не будет просмотрено — оно либо будет удалено при добавлении новой вершины, либо перестанет принадлежать правой ветке. При этом на каждой итерации мы добавляем не более одного ребра в правую ветвь. Значит в сумме при поиске подходящей пары вершин мы рассмотрим не более n рёбер, а значит сложность полученного алгоритма не будет превышать $O(n)$.

Замечание 6.6.1. В предыдущем параграфе на самом деле описана амортизационная оценка для операции добавления новой вершины с ключом больше, чем все ключи в дереве. Подобные рассуждения будут ещё неоднократно встречаться в курсе. Неформально можно сформулировать следующий принцип: *„если на каждой итерации некоторого алгоритма мы увеличиваем некоторую величину t не более чем на единицу, и совершаем $k \leq t$ операций, попутно уменьшая t на k, то суммарное количество операций не превосходит количества итераций“*. Этот нефомальный принцип можно доказать «методом ростовщика»: на каждой итерации мы откладываем 1 рубль, а каждая операция тратит 1 рубль, причём нельзя потратить больше, чем к этому моменту отложено. Так как каждая операция в результате оплачена, то количество операций не превосходит количество отложенных рублей, т.е. количества итераций.

Упражнение 6.6.1. Докажите этот принцип при помощи метода потенциалов.

6.6.3. Операции

Осталось научиться добавлять и удалять вершины. Нам потребуются две вспомогательные операции — `cartesian-split` и `cartesian-merge`.

```
# Разделяет дерево на два по ключу x
def cartesian-split(x, v = root):
    if v == None:
        return (None, None)

    if key(v) < x:
        (T1, T2) = cartesian-split(x, v.right)
        v.right = T1
        return (v, T2)
    else
        (T1, T2) = cartesian-split(x, v.left)
        v.left = T2
        return (T1, v)
```

```
# Слияние двух деревьев
# ключи T1 меньше ключей T2
def cartesian-merge(T1, T2):
    if T1 == None:
        return T2
    if T2 == None:
        return T1

```

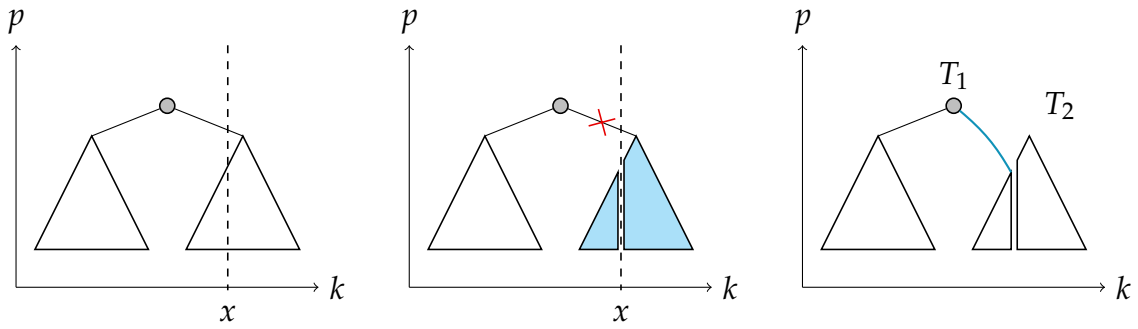


Рис. 6.6.3. Работа операции `cartesian-split(k)`.

```
    # корень правого дерева имеет больший приоритет
    if pri(T1) < pri(T2):
        T = cartesian-merge(T1, T2.left)
        T2.left = T
        return T2
    else
        T = cartesian-merge(T1.right, T2)
        T1.right = T
        return T1
```

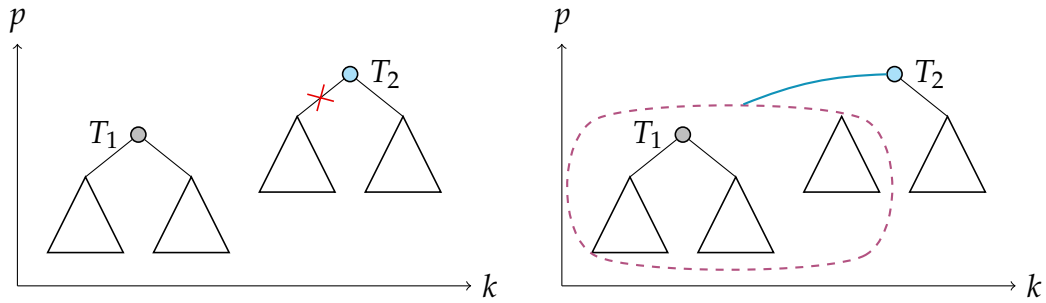


Рис. 6.6.4. Работа операции `cartesian-merge(T1, T2)`.

На основе этих двух операций легко реализовать операции добавления и удаления вершин.

```
# Добавление вершины
def cartesian-insert((k,p), T):
    v = node(k, p)
    (T1, T2) = cartesian-split(k, T)
    T1 = cartesian-merge(T1, v)
    return cartesian-merge(T1, T2)

# Удаление всех вершин с ключом k
def cartesian-remove(k, T):
    (T1, T2)   = cartesian-split(k,   T)
    (T2l, T2r) = cartesian-split(k+1, T2)
    return cartesian-merge(T1, T2r)
```

Как и операции для других деревьев, все рассмотренные операции с декартовым деревом работают за время $O(h)$, где h — высота дерева. Однако, как мы уже

знаем, у декартова дерева с n вершинами высота может достигать n. Кроме того, нет никакой возможности балансировать декартово дерево, т.к. легко построить пример входных данных, на которых декартово дерево определено однозначно и имеет линейную глубину. Что же делать?

6.6.4. Дуча

Определение 6.6.2. *Дуча (treap, курево, дерамида)* — это дерево поиска построенное на декартовом дереве, в котором приоритеты выбираются независимо и случайно.

Идея использовать случайные значения для приоритетов в декартовом дереве появилась только в 1996 году и принадлежит Сиделю и Арагону [6].

Теорема 6.6.2. *Математическое ожидание высоты дучи ограничено* $O(\log n)$.

Доказательство. Мы здесь воспользуемся теоремой о математическом ожидании глубины рекурсии для быстрой сортировки. Какая связь у декартова дерева с быстрой сортировкой? Давайте ещё раз посмотрим на квадратичный алгоритм построения декартова дерева, при помощи которого мы доказали теорему существования. Предположим, что на вход нам дали только ключи, а приоритеты мы выбираем независимо и случайно. С какой вероятностью каждая вершина будет выбрана корнем? Каждая вершина имеет одинаковую вероятность стать корнем, и это свойство сохраняется для рекурсивных вызовов: на каждом подотрезке, соответствующем рекурсивному вызову алгоритма, все вершины имеют одинаковую вероятность стать корнем соответствующего поддерева. Этот процесс очень похож на то, что происходит в быстрой сортировке: каждая вершина имеет одинаковую вероятность стать опорным элементом при первом вызове, и это свойство сохраняется для рекурсивных вызовов. Эти процессы не просто похожи, а верно и более сильное свойство: для любого фиксированного набора ключей существует взаимнооднозначное соответствие между всеми возможными дучами и деревьями рекурсии быстрой сортировки на этом наборе. Более того, это соответствие сохраняет вероятности (т.е. некоторая конкретная дуча будет иметь ту же вероятность, что и соответствующее ей дерево рекурсии быстрой сортировки). Таким образом оценка на математическое ожидание глубины рекурсии быстрой сортировки влечёт за собой такую же оценку на математическое ожидание высоты декартова дерева. □

Следствие 6.6.1. *Операции* `find`, `cartesian-insert` *и* `cartesian-remove` *для дучи работают за* $O(\log n)$ *в среднем.*

6.7. B-деревья

Другой подход к реализации деревьев поиска — это так называемые *B-деревья*, у которых вершины могут иметь произвольное количество потомков. B-деревья часто используются для поиска по данным, хранящихся во внешней памяти (например, в файловых системах или базах данных): за счёт увеличения арности внутренних вершин, происходит уменьшение глубины дерева, что позволяет уменьшить количество запросов на чтение ко внешней памяти (стоимость которых обычно значительно больше, чем стоимость операций с внутренней памятью). Хорошо известный всем пример B-дерева — это система каталогов файловой системы:

в каждом каталоге может быть произвольное число файлов (листьев) и подкаталогов (внутренних вершин). В различных реализация B-деревьев, элементы мо-

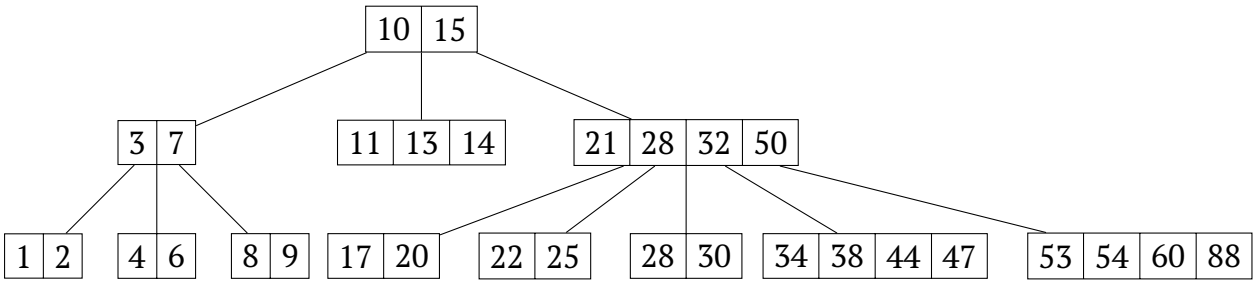


Рис. 6.7.1. B-дерево.

гут храниться как во всех вершинах, так и только в листьях (тогда в вершинах хранятся ссылки на максимальные или минимальные элементы в соответствующих поддеревьях). «Компромиссным» вариантом между двоичными деревьями и B-деревьями является *2-3 дерево*, реализующее идею сбалансированного B-дерева, у которого все внутренние вершины имеют два или три потомка.

Глава 7

Хеширование

В этом разделе мы снова поговорим про реализации абстрактных типов данных: множество и словарь (стр. 51). Ранее мы предполагали, что на ключах задан линейный порядок, и этого было достаточно, чтобы выполнять сортировку и применять двоичный поиск или реализовывать дерево поиска. Для разговора о хешировании нам потребуется другое предположение — мы будем предполагать, что существует функция, сопоставляющая каждому ключу некоторое неотрицательное целое число.

7.1. Прямая адресация

Начнём с ситуации, когда ключи сами по себе являются целыми числами из отрезка $[0, m-1]$. Если число m не очень большое, то мы можем хранить элементы в массиве длины m, или, другими словами, применить *таблицу с прямой адресацией (direct-address table)*. Для реализации множества в каждой ячейке будем хранить булево значение, которое определяет, присутствует элемент в множестве или нет. Для реализации словаря в каждой ячейке будем хранить ссылку на соответствующее значение или пустую ссылку, обозначающую отсутствие ключа. При такой реализации используемая память будет пропорциональна m, а все операции поиска, добавления и удаления будут работать за $O(1)$. Подобную конструкцию мы уже встречали в алгоритме сортировки подсчётом — там в каждой ячейке хранился счётчик, обозначающий сколько раз элемент встречается в массиве. Узким местом данного подхода является избыточное использование памяти: например, при хранении множества 32-битных ключей, нам в любом случае потребуется зарезервировать $m = 2^{32}$ ячеек памяти вне зависимости от размера множества, а для хранения 64-битных ключей данный метод и вовсе неприменим.

7.2. Хеш-таблица

Основной проблемой прямой адресации было то, что общее количество ячеек m в таблице никак не зависело от количества элементов, которые в этой таблице хранятся. Для эффективного использования памяти мы хотели бы, чтобы размер таблицы был не более $O(n)$, где n — количество элементов, которые хранятся в таблице. Но как этого достичь, если диапазон ключей очень большой или ключи и вовсе не являются целыми числами?

7.2.1. Хеш-функция

Мы будем предполагать, что на множестве ключей K задана некоторая *хеш-функция (hash function)*

$$h : K \to \{0, \ldots, m-1\}.$$

Значение $h(k)$ будем называть *хеш-кодом (hash code)* ключа k. Более того, мы будем предполагать, что мы умеем определять хеш-функции для разных значений m и таким образом влиять на количество пустых ячеек в таблице. В дальнейшем при анализе сложности мы потребуем, чтобы хеш-функции обладали некоторыми дополнительными свойствами, которые позволят получить хорошие оценки в среднем. Пока же мы будем считать, что „хорошая“ хеш-функция должна хорошо „перемешивать“ результаты, т.е., например, значение хеш-функции от двух близких по значению ключей могут быть произвольно далеки друг от друга.

7.2.2. Методы разрешения коллизий

Наличие хеш-функции h позволяет нам определить *хеш-таблицу* для ключей из множества K. Заведём массив H размера m. В этом массиве каждому ключу $k \in K$ будет соответствовать ячейка с номером $h(k)$. По принципу Дирихле, если размер множества ключей больше, чем m, то обязательно найдутся два ключа, которым будет соответствовать одна и та же ячейка. Каким образом хранить два элемента в одной ячейке? Такая ситуация, при которой $h(k_1) = h(k_2)$ при $k_1 \neq k_2$, называется *коллизией*. Коллизии неизбежно случаются, и поэтому хеш-таблицы обязательно имеют *метод разрешения коллизий*. Существуют множество подходов к этой проблеме, мы рассмотрим только два наиболее популярных из них.

Метод цепочек. *Метод цепочек (chaining)* предлагает в каждой ячейке таблицы H хранить список и записывать в него все ключи, которые попали в эту ячейку. При такой реализации метода разрешения коллизий время поиска в ячейке

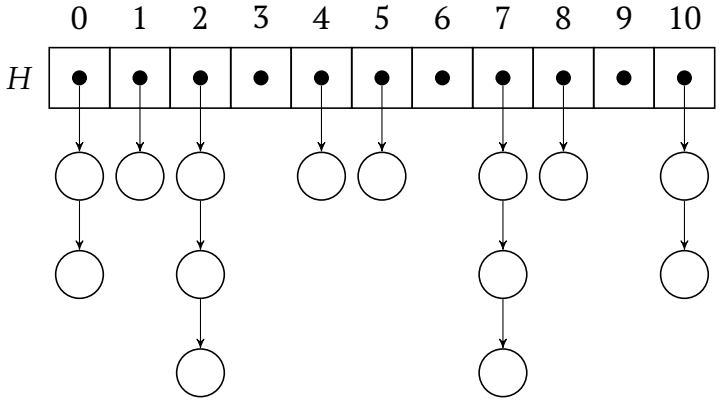


Рис. 7.2.1. Хеш-таблица с разрешения коллизий при помощи метода цепочек, $m = 11$.

составляется $O(l)$, где l — длина цепочки. Операция добавления элемента так же занимает время $O(l)$, если нам нужно проверить, нет ли уже этого ключа в списке, и $O(1)$, если в хеш-таблице разрешаются повторяющиеся ключи. Удаление занимает $O(l)$. Становится ясно, что в дальнейшем нас будут интересовать хеш-функции, которые имеют небольшое количество коллизий, и тем самым в

хеш-таблице будут не очень длинные цепочки. Как мы увидим дальше, этот метод асимптотически оптимален при правильном выборе хеш-функции. Однако у него есть небольшой недостаток — для хранения списка требуется дополнительное место для хранения ссылок.

Открытая адресация. Для полноты картины рассмотрим общую схему методов, альтернативных методу цепочек. При *открытой адресации* в каждой ячейке может храниться только один ключ, поэтому требуется $n \leq m$, но каждому ключу соответствует не одна ячейка, а последовательность ячеек. Для этого выбирается хеш-функция от двух аргументов:

$$h : K \times [0, m-1] \to [0, m-1],$$

где второй аргумент задаёт номер в последовательности. Таким образом ключу k соответствуют ячейки с номерами

$$h(k,0), h(k,1), \ldots, h(k, m-1).$$

Разумно потребовать, чтобы при фиксированном k эта последовательность была бы некоторой перестановкой чисел $\{0, \ldots, m-1\}$. Для того, чтобы найти ключ k в хеш-таблице с открытой адресацией, нужно последовательно проверять ячейки $h(k,0), h(k,1), \ldots$ до тех пор, пока не встретится либо ключ k, либо пустая ячейка. Добавление ключа реализуется так же: ячейки $h(k,0), h(k,1), \ldots$ последовательно проверяются, пока не встретится пустая ячейка. При удалении элемента возникают трудности, т.к. удаление может нарушить цепочку, соответствующую некоторому ключу. Это можно обойти, храня в каждой ячейке дополнительный флаг, обозначающий, что значение из ячейки было удалено — таким образом можно различать пустые ячейки, и ячейки, значение из которых было удалено. Однако при такой реализации длина цепочек не уменьшается даже при удалении элементов, что отрицательно сказывается на производительности.

Такую хеш-функцию с двумя аргументами можно сконструировать из „обычной“ хеш-функции. Самая простая реализация, задающая *линейный поиск* в таблице, строит h по хеш-функции h_1 следующим образом:

$$h(k,i) = (h_1(k) + i) \bmod m.$$

Однако этот способ обладает серьёзным недостатком — цепочки для близких ячеек таблицы могут „склеиваться“. В более сложном методе *двойного хеширования* функция h строится на основе двух „обычных“ хеш-функций h_1, h_2 следующим образом:

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

Для этого метода цепочки различных ключей будут „разбросаны“ по таблице с разным шагом, и поэтому не будут склеиваться. Нужно только как-то обеспечить, чтобы для фиксированного ключа элементы последовательности не повторялись. Для этого удобно выбрать m простым, и тогда нам подойдёт любая хеш-функция $h_2(k) : K \to [1, m-1]$.

7.2.3. Примеры хеш-функций

В общем случае для того, чтобы выбрать хорошую хеш-функцию, нам нужно что-то знать про распределение ключей. В этом разделе мы рассмотрим некоторые конструкции, которые могут давать неплохие результаты.

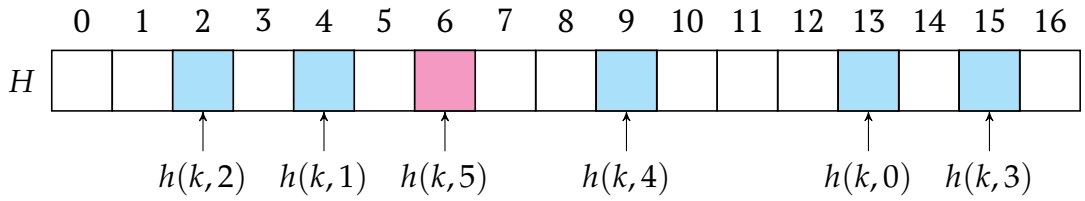


Рис. 7.2.2. Просмотр ячеек при открытой адресации, $m = 17$.

- **Равномерно распределённые вещественные числа**. Будем считать, что вещественные числа лежат в полуинтервале $[0,1)$. Если это не так, то воспользуемся масштабированием. Тогда в качестве хеш-функции можно использовать
$$h(k) = \lfloor mk \rfloor.$$
Такая хеш-функция просто разбивает $[0,1)$ на m полуинтервалов одинаковой длины и сопоставляет каждому числу номер интервала, в который оно попадает.

- **Целые числа.** Самый простой вид хеш-функция для целых чисел даёт метод *деления с остатком*:
$$h(k) = k \bmod m.$$
Более сложный *мультипликативный метод* обладает значительно лучшими перемешивающими свойствами:
$$h(k) = \lfloor m \cdot \{ck\} \rfloor,$$
где $\{\cdot\}$ обозначает дробную часть числа, а c — некоторая вещественная константа. Для лучшего результата константу c нужно выбирать иррациональной, например, $c = \sqrt{2}$, тогда дробная часть ck не равна нулю при любом k (в памяти компьютера можно задать только некоторое рациональное приближение иррационального числа, но и этого будет достаточно). Этот метод можно обобщить и на пару значений, выбрав две различные константы c_1 и c_2 и определив
$$h(k_1, k_2) = \lfloor m \cdot \{c_1 k_1 + c_2 k_2\} \rfloor.$$
(константы c_1 и c_2 лучше выбирать линейно независимыми над $\mathbb{Q}$ иррациональными числами, например, $\sqrt{2}$ и $\sqrt{3}$.)

- **Последовательности целых чисел, строки**. Для вычисления хеш-кода последовательности целых чисел $a = (a_0, a_1, \ldots, a_{n-1})$ хорошие результаты может дать *полиномиальный метод*: выберем некоторое взаимнопростое с m число x, называемое *основанием*, и определим
$$h(a) = (a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_{n-1} \cdot x^{n-1}) \bmod m.$$

- **Кортежи разнотипных значений.** Довольно частая задача, которая возникает при написании программ, — это придумать хеш-функцию для некоторой структуры, поля которой имеют разные типы. В этом случае разумно сначала вычислить хеш-коды для каждого из полей, а потом скомбинировать их при помощи мультипликативного или полиномиального методов.

Замечание 7.2.1. Вы могли также встречать или даже применять для каких-то задач *криптографические хеш-функции*, например, такие как MD5 или SHA-1. Как следует из названия, такие хеш-функции предназначены для криптографических протоколов и не используются в хеш-таблицах. Во-первых потому, что они имеют очень большое пространство хеш-кодов, значительно больше, чем нужно для хеш-таблиц (128 битов для MD5 и 160 битов для SHA-1). А во-вторых, так как на криптографические хеш-функции накладываются более сложные требования, то они довольно сложно устроены, и как следствие этого, долго вычисляются (значительно дольше, чем рассмотренные выше конструкции). С другой стороны, в некотором приближении можно считать, что схема хранения файлов в репозитории системы контроля версий Git — это хеш-таблица, основанная на криптографической хеш-функции SHA-256 (если говорить точнее, то там вместо таблицы используется дерево каталогов файловой системы, т.е. B-дерево).

В дальнейшем в этом разделе мы для простоты будем предполагать, что сложность вычисления хеш-функции равна $O(1)$. Это верно для чисел, но неверно, например, для строк и последовательностей. Поэтому в случаях, когда вычисление хеш-функции требует неконстантного времени, в оценке нужно будет дополнительно учесть, сколько раз тот или иной алгоритм вычисляет хеш-функцию.

7.2.4. Детали реализации

При реализации хеш-таблицы часто заранее неизвестно, сколько ключей в ней будет храниться. Поэтому разумно начинать с некоторой небольшой константы, а потом расширять таблицу аналогично тому, как мы делали это для расширяющегося массива: если коэффициент заполнения $\alpha = n/m$ стал больше некоторого порога, то нужно создать новую таблицу размера γm для некоторой константы $\gamma > 1$, выбрать новую хеш-функцию $K \to \{0, \dots, \gamma m - 1\}$ и добавить в неё все ключи из старой. При таком подходе будут верны оценки, полученные нами для расширяющегося массива, т.е. на расширение таблицы за всё время работы мы дополнительно потратим не более $O(n)$ копирований и $O(n)$ вычислений хеш-функции. Увеличивать таблицу можно не только при слишком большом значении α, но и если длинна самой длинной цепочки стала больше некоторого порога.

7.3. Гипотеза равномерного хеширования

Для удобства изложения введём обозначение для *коэффициента заполнения* хеш-таблицы $\alpha = \frac{n}{m}$. Как мы увидели ранее, сложность поиска ключа k в хеш-таблице пропорциональна длине цепочки, соответствующей k. Поэтому желательным свойством хеш-функции является „равномерность“ распределения хеш-кодов — тогда в среднем длина цепочки будет равна коэффициенту заполнения, и, соответственно, сложность поиска в среднем будет ограничена $O(1 + \alpha)$. Однако на самом деле равномерность распределения хеш-кодов зависит не только от функции, но и от распределения ключей. Для любой хеш-функции можно подобрать такое распределение на ключах, для которого распределение хеш-кодов будет очень далеко от равномерного: выберем некоторое $j \in \{0, \dots, m - 1\}$ и рассмотрим равномерное распределение на ключах с хеш-кодом j. Рассуждая в терминах злонамеренного противника мы могли бы сказать, что противник всегда может заставить хеш-таблицу работать медленно — для этого ему нужно записы-

вать в хеш-таблицу только ключи с хеш-кодом j для некоторого фиксированного j.

На практике же иногда у нас могут быть причины предполагать, что на имеющемся распределении ключей выбранная нами хеш-функция задаёт распределение хеш-кодов, близкое к равномерному. Давайте оценим сложность операций с хеш-таблицей в этом предположении. Для его формализации мы будем пользоваться следующей гипотезой, которая избавит нас от необходимости делать какие-то предположения про распределение ключей.

Предположение 7.3.1 (Гипотеза равномерного хеширования). Значение хеш-функции $h : K \to \{0, \dots, m-1\}$ от ключа k является случайной величиной, равномерно распределённой на множестве $\{0, \dots, m-1\}$. При этом значения хеш-функции на различных ключах k_1 и k_2 независимы.

Замечание 7.3.1. На практике значение хеш-функции не является случайной величиной, т.к. хеш-функции не используют случайных битов. Тем не менее, мы могли бы попробовать запрограммировать хеш-функцию, удовлетворяющую этой гипотезе. Такая функция могла бы быть устроена следующим образом: каждый раз, когда хеш-функция вычисляется от нового ключа k, значением $h(k)$ выбирается случайный элемент множества $\{0, \dots, m-1\}$, и эта информация запоминается. Если такую функцию реализовать на практике, то возникнет проблема с тем, что для эффективного хранения информации о выбранных значениях нам потребуется хеш-таблица.

Теорема 7.3.1. *В предположении гипотезы равномерного хеширования среднее время безуспешного поиска при использовании метода цепочек равно* $\Theta(1+\alpha)$.

Доказательство. В процессе безуспешного поиска ключа k мы должны полностью просмотреть цепочку в ячейке $h(k)$. Для каждого $i \in \{0, \dots, m-1\}$ обозначим длину цепочки в ячейке i как l_i. По гипотезе равномерного хеширования значение $h(k)$ с равной вероятностью равно каждому из индексов $\{0, \dots, m-1\}$. Следовательно, математическое ожидание длины цепочки в ячейке $h(k)$ равно

$$\mathbb{E}[l_{h(k)}] = \sum_{i=0}^{m-1} \frac{1}{m} \cdot l_i = \frac{n}{m} = \alpha.$$

Отсюда среднее время поиска равно $\Theta(1+\alpha)$ (единица возникает, т.к. мы всегда сделаем хотя бы одну операцию, даже если цепочка пустая.) □

В предыдущей теореме нам достаточно было применить гипотезу равномерного хеширования только для ключа, который мы ищем. В следующей теореме нам нужно будет применить эту гипотезу ко всем ключам в таблице.

Теорема 7.3.2. *В предположении гипотезы равномерного хеширования среднее время успешного поиска при использовании метода цепочек равно* $\Theta(1+\alpha)$.

Замечание 7.3.2. В данной теореме идёт речь о поиске ключа, который хранится в таблице, поэтому среднее время вычисляется не только по вероятностному пространству, соответствующему хеш-функции, но и по множеству ключей, находящихся в таблице.

Доказательство. Пусть в хеш-таблице хранятся ключи $k_1, k_2, \cdots, k_n$, и пусть нумерация ключей соответствует порядку, в котором ключи добавлялись в хеш-таблицу. Для каждой пары $i, j \in [n]$ определим случайную величину $X_{i,j}$, равную 1,

если случилась коллизия $h(k_i) = h(k_j)$, и равную 0 в противном случае. Заметим, что $E[X_{i,i}] = 1$, и по гипотезе равномерного хеширования $E[X_{i,j}] = \frac{1}{m}$ для $i \neq j$ (здесь мы пользуемся независимостью $h(k_i)$ и $h(k_j)$).

Предположим теперь, что мы ищем ключ k_i. Сколько элементов нам нужно просмотреть для поиска k_i в среднем? Будем считать, что при добавлении элемента в ячейку он добавляется в начало списка. Тогда количество элементов, которые мы просмотрим при поиске k_i, равно $X_{i,i} + \ldots + X_{i,n}$. Тогда среднее время поиска ключа в таблице (среднее по всем ключам присутствующим в таблице и по распределению случайных величин $\{X_{i,j}\}$) равно

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n} \mathbb{E}\left[\sum_{j=i}^{n} X_{i,j}\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \mathbb{E}[X_{i,j}]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right) \\
&= 1 + \frac{1}{mn}\sum_{i=1}^{n}(n-i) \\
&= 1 + \frac{1}{mn}\left(n^2 - \frac{n(n+1)}{2}\right) \\
&= \Theta(1+\alpha).
\end{aligned}$$

□

7.4. Универсальное хеширование

Гипотеза равномерного хеширования позволяет объяснить, почему при «хорошем» (т.е., равномерном) распределении хеш-кодов поиск в хеш-таблице будет работать быстро $O(1+\alpha)$. Однако в общем случае не понятно, будет ли распределение хеш-кодов для конкретной хеш-функции действительно равномерным, т.к. не известно распределение на ключах. Эту проблему можно решить, если рассматривать не одну хеш-функцию, а целое семейство хеш-функций с дополнительным свойством.

Определение 7.4.1. Множество функций $\mathcal{H}$ из множества ключей K в $\{0, \ldots, m-1\}$ называется *универсальным семейством хеш-функций (УСХФ)*, если для любой пары различных ключей k_1 и k_2

$$\Pr_{h \leftarrow \mathcal{H}}[h(k_1) = h(k_2)] \leq \frac{1}{m}.$$

Замечание 7.4.1. Отметим, что аналогичный факт верен в предположении гипотезы равномерного хеширования, но вероятность определяется по другому вероятностному пространству. Если в случае гипотезы равномерного хеширования значение хеш-функции на разных ключах выбираются случайно и независимо, то в случае универсального семейства хеш-функций случайным является только выбор хеш-функции, и она в свою очередь однозначно определяет хеш-коды для всех ключей.

При реализации хеш-таблицы, основанной на универсальном семействе хеш-функций, перед началом заполнения таблицы мы выбираем случайную функцию из универсального семейства и используем её для вычисления хеш-кодов.

За счёт этого обеспечивается хорошая в среднем производительность вне зависимости от распределения ключей, т.к. теперь усреднение происходит не только по распределению на ключах, но и по выбору хеш-функции из семейства. Другими словами, теперь противнику сложно подобрать распределение, на котором хеш-функция будет иметь много коллизий — он может это сделать для каждой хеш-функции из семейства, но он не знает заранее, какую функцию мы выберем.

Теорема 7.4.1. *При использовании универсального хеширования среднее время безуспешного поиска ключа равно* $\Theta(1+\alpha)$.

Доказательство. Пусть в хеш-таблице хранятся ключи $k_1, k_2, \cdots, k_n$. В процессе безуспешного поиска ключа k мы должны полностью просмотреть цепочку в ячейке $h(k)$. Для каждого $i \in [n]$ определим случайную величину X_i, равную 1, если случилась коллизия $h(k_i) = h(k)$, и равную 0 в противном случае. Из определения универсального семейства следует, что $\mathbb{E}[X_i] \leq \frac{1}{m}$. Тогда математическое ожидание длины цепочки в ячейке $h(k)$ равно

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] \leq \frac{n}{m} = \alpha.$$

Поэтому среднее время поиска равно $\Theta(1+\alpha)$ (единица возникает, т.к. мы всегда сделаем хотя бы одну операцию.) □

Теорема 7.4.2. *При использовании универсального хеширования среднее среднее время успешного поиска ключа равно* $\Theta(1+\alpha)$.

Доказательство. Доказательство полностью повторяет доказательство предыдущей теоремы за исключением того, что k теперь присутствует в таблице. Поэтому один из членов в получившейся сумме будет равен 1, и мы получим

$$\sum_{i=1}^{n} \mathbb{E}[X_i] \leq 1 + \frac{n-1}{m} \leq 1 + \alpha.$$

□

Получается, что используя универсальное семейство хеш-функций мы с хорошей вероятностью можем обеспечить эффективность хеш-таблицы. Однако пока не ясно, существуют ли такие семейства и как легко их построить. Оказывается, что такие семейства есть и построить их не сложно. Мы покажем, как построить такое семейство для целых чисел.

Теорема 7.4.3. *Пусть* $K = \{0, \ldots, n\}$. *Выберем простое* $p > n$. *Тогда семейство хеш-функций* $\mathcal{H}$, *состоящее из функций*

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

для всех $a \in \{1, \ldots, p-1\}$ *и* $b \in \{0, \ldots, p-1\}$, *является универсальным.*

Доказательство. Рассмотрим пару ключей $k_1 \neq k_2$ и посмотрим, как на них действует «внутреннее» линейное преобразование по модулю p. Положим

$$t_1 = (ak_1 + b) \bmod p, \quad \text{и} \quad t_2 = (ak_2 + b) \bmod p.$$

Т.к. ключи k_1 и k_2 не превосходят p, то $t_1 \neq t_2$. Действительно, если предположить, что $t_1 = t_2$, то из соотношения

$$ak_1 + b = ak_2 + b \pmod{p}$$

получается, что

$$a(k_1 - k_2) = 0 \pmod{p}.$$

Произведение двух чисел меньших p (тут важно, что p простое) делится на p тогда и только тогда, когда одно из них равно нулю. Мы выбираем $a \neq 0$, следовательно $k_1 - k_2 = 0$, а такого быть не может, т.к. мы рассматриваем различные k_1 и k_2.

Теперь покажем, что при фиксированных k_1 и k_2 различные пары (a, b) приводят к разным парам (t_1, t_2). Действительно, при заданных k_1 и k_2 по (a, b) можно вычислить (t_1, t_2), а при заданных (t_1, t_2) можно однозначно восстановить (a, b): из соотношений

$$\begin{cases} t_1 = (ak_1 + b) \bmod p, \\ t_2 = (ak_2 + b) \bmod p, \end{cases}$$

получаем $a = (t_1 - t_2) \cdot (k_1 - k_2)^{-1} \bmod p$ (тут обратное берётся по модулю p), а затем, уже зная a, вычисляем $b = (t_1 - ak_1) \bmod p$. Получается, что между парами (a, b) и (t_1, t_2), $t_1 \neq t_2$ есть взаимнооднозначное соответствие. При всех возможных $p(p-1)$ парах (a, b) каждая такая пара (t_1, t_2) встречается равно один раз. Т.е. если выбирать функцию $h_{a,b}$ случайно и равномерно из $\mathcal{H}$, то распределение соответствующих пар (t_1, t_2) тоже будет случайно и равномерно на всех парах $t_1, t_2 \in \{0, p-1\}$, $t_1 \neq t_2$.

Таким образом вероятность того, что $h_{a,b}(k_1) = h_{a,b}(k_2)$ равна вероятности того, что случайные различные $t_1, t_2 \in \{0, p-1\}$ окажутся равными по модулю m. Для фиксированного t_1 количество $t_2 \neq t_1$, которые дают тот же остаток по модулю m, не превосходит

$$\left\lceil \frac{p}{m} \right\rceil - 1 \leq \frac{p + m - 1}{m} - 1 = \frac{p-1}{m}.$$

Каждое t_2 встречается с вероятностью $1/p$, следовательно, вероятность получить коллизию равна

$$\frac{1}{p} \cdot \frac{p-1}{m} \leq \frac{1}{m},$$

что и требовалось доказать. □

Используя универсальное хеширование для целых чисел можно построить универсальное семейство хеш-функций для целочисленных последовательностей и строк, основанное на идее полиномиального хеширования. Пусть все элементы последовательности являются целыми числами из $\{0, \ldots, d-1\}$. Выберем простое $p \geq \max(d, m)$ и построим универсальное семейство хеш-функций вида $\{0, \ldots, p-1\} \to \{0, \ldots, m-1\}$, обозначив его $\mathcal{H}_p$. Тогда семейство хеш-функций, состоящее из всех функций вида:

$$h_x((a_0, a_1, \ldots, a_{n-1})) = h((a_0 x_0 + a_1 x_1 + \cdots + a_{n-1} x^{n-1}) \bmod p),$$

где $x \in \{1, \ldots, p-1\}$ и $h \in \mathcal{H}_p$, является универсальным.

7.5. Совершенное хеширование

В этом разделе мы покажем, что если в хеш-таблице хранится неизменяемое множество элементов (т.е. мы имеем дело со статической задачей поиска), то можно эффективным образом добиться отсутствия коллизий, и тогда сложность доступа будет $O(1)$ *в худшем*. Такая конструкция называется *совершенным хешированием*. Эту конструкцию можно применять даже в том случае, если множество ключей в таблице изменяется, но множество всех возможных ключей достаточно мало (тогда размер таблицы будет пропорционален мощности множества всех возможных ключей).

Пусть размер множества ключей K, которые мы хотим хранить в таблице, равен n. Структура совершенной хеш-таблицы состоит из двух уровней (см. рис. 7.5.1). Первый уровень образован хеш-таблицей H размера n с хеш-функцией h. В каждой ячейке этой таблицы хранятся ссылка на хеш-таблицу второго уровня и описание хеш-функции для неё, для ячейки с номером i будем обозначать их T_i и g_i соответственно. Таким образом, если $h(k) = i$, то ключ k хранится в хеш-таблице T_i в ячейке с номером $g_i(k)$. Осталось определить размер хеш-таблиц второго уровня. Для каждого $i \in \{0, \ldots, n-1\}$ через n_i обозначим количество ключей, для которых $h(k) = i$, тогда размер T_i равен n_i^2.

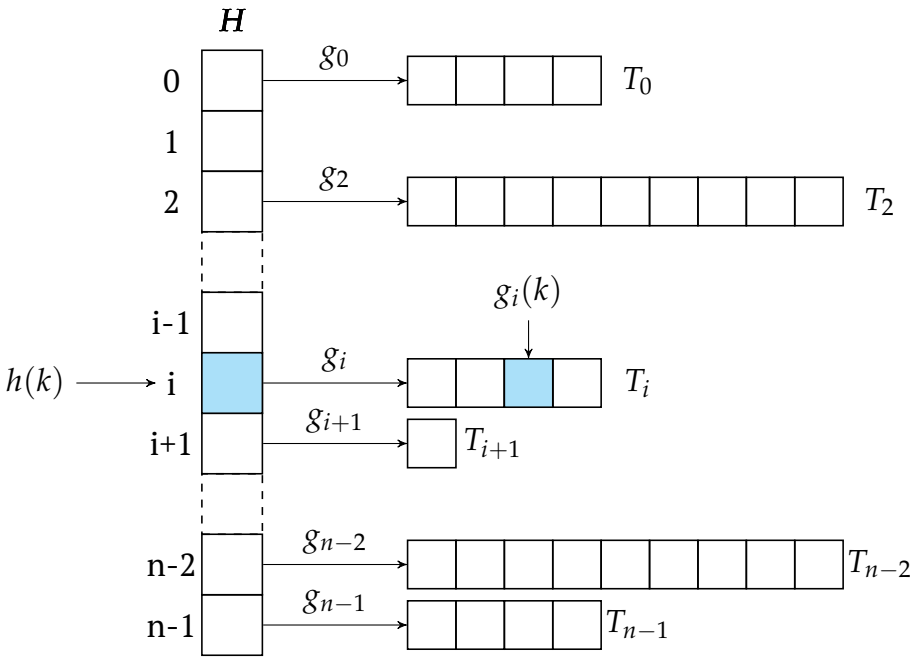


Рис. 7.5.1. Схема совершенного хеширование. Некоторые ячейки хеш-таблицы H могут оказаться пустыми.

Утверждается, что для фиксированного множества ключей при использовании универсального хеширования можно выбрать функции $h, g_0, g_1, \ldots, g_{n-1}$ таким образом, чтобы не было коллизий (т.е. в каждой ячейке хеш-таблиц второго уровня хранилось не более одного ключа) и при этом суммарный размер таблиц не превышал $O(n)$. Начнём с того, что покажем, как выбрать хеш-функции без коллизий хеш-таблиц второго уровня.

Теорема 7.5.1. *При использовании универсального хеширования для хеш-таблицы размера $m = n^2$ вероятность возникновения коллизий не превышает* $1/2$.

Доказательство. Если всего n ключей, то различных неупорядоченных пар $\binom{n}{2}$. Вероятность коллизии для каждой пары по определению универсального семейства хеш-функций не превосходит $1/m$. Пусть случайная величина X равна количеству коллизий. Тогда её математическое ожидание

$$\mathbb{E}[X] \leq \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{m} = \frac{n^2 - n}{2} \cdot \frac{1}{n^2} < \frac{1}{2}.$$

Теперь нужно воспользоваться неравенством Маркова (теорема **??**):

$$\Pr[X \geq 1] \leq \frac{\mathbb{E}[X]}{1} < \frac{1}{2}.$$

□

Эта теорема показывает, что как минимум половина хеш-функций из универсального семейства не будут иметь коллизий для заданного множества ключей, если размер таблицы равен квадрату количества элементов. Поэтому в среднем для каждой таблицы второго уровня нам нужно проверить две случайные хеш-функции из универсального семейства, чтобы найти такую, которая не имеет коллизий. Остаётся показать, что можно выбрать такую h, что суммарный размер таблиц второго уровня будет ограничен $O(n)$. Вспомним, что n_i — это количество ключей, которые попадают в ячейку i таблицы H. Тогда суммарный размер хеш-таблиц второго уровня равен $\sum_{i=0}^{n-1} n_i^2$.

Теорема 7.5.2. *При использовании универсального хеширования для хеш-таблицы размера $m = n$*

$$\Pr\left[\sum_{i=0}^{n-1} n_i^2 \geq 4n\right] \leq 1/2.$$

Доказательство. Сначала покажем, что

$$\mathbb{E}\left[\sum_{i=0}^{n-1} n_i^2\right] < 2n.$$

Для этого воспользуемся соотношением $n_i^2 = n_i + 2\binom{n_i}{2}$:

$$\mathbb{E}\left[\sum_{i=0}^{n-1} n_i^2\right] = \mathbb{E}\left[\sum_{i=0}^{n-1} n_i + 2\sum_{i=0}^{n-1} \binom{n_i}{2}\right] = n + 2\,\mathbb{E}\left[\sum_{i=0}^{n-1} \binom{n_i}{2}\right].$$

Заметим, что $\sum_{i=0}^{n-1} \binom{n_i}{2}$ — это общее число пар ключей, которые дают коллизию. Поэтому математическое ожидание этой величины можно оценить аналогично предыдущей теореме: для каждой пары ключей вероятность коллизии не превосходит $1/m = 1/n$, следовательно математическое ожидание общего числа коллизий не превосходит $\binom{n}{2} \cdot \frac{1}{n}$. Таким образом получаем, что

$$\mathbb{E}\left[\sum_{i=0}^{n-1} n_i^2\right] \leq n + 2\binom{n}{2}\frac{1}{n} = n + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n} = n + n - 1 < 2n.$$

Остаётся применить неравенство Маркова (теорема **??**):

$$\Pr\left[\sum_{i=0}^{n-1} n_i^2 \geq 4n\right] \leq \frac{\mathbb{E}\left[\sum_{i=0}^{n-1} n_i^2\right]}{4n} < \frac{2n}{4n} = \frac{1}{2}.$$

□

Теперь можно описать (вероятностный) алгоритм построения таблицы для совершенного хеширования. Пусть нам дано множество n ключей. Сначала построим хеш-таблицу H и выберем хеш-функцию $h : K \to \{0, \ldots, n-1\}$ так, чтобы сумма квадратов количества элементов в ячейках H не превосходила $4n$. По теореме 7.5.2 для этого нам в среднем потребуется проверить две случайные хеш-функции из универсального семейства. Когда такая h найдена, то для каждой ячейки i хеш-таблицы H мы заводим хеш-таблицу второго уровня T_i размера n_i^2 и выбираем для неё хеш-функцию $g_i : K \to \{0, \ldots, n_i^2 - 1\}$ так, чтобы она не имела коллизий на множестве ключей, для которых $h(k) = i$. По теореме 7.5.1 для этого нам тоже в среднем потребуется проверить две случайные хеш-функции из универсального семейства. В результате, в среднем предложенный алгоритм построения работает за $O(n)$.

Глава 8

Числовые алгоритмы

8.1. Арифметические операции с n-битовыми числами

В данном разделе мы обсудим алгоритмы для выполнения арифметических операций с целыми числами произвольной длины. Числа будут задаваться их представлением в двоичной системе счисления, т.е. можно считать, что число задаётся битовым массивом его двоичных цифр. Числа в таком представлении легко умножать и делить на два, для этого нужно, соответственно, добавить ноль в конец массива или удалить последний элемент массива. Ещё проще вычислить остаток при делении на два — остаток записан в последнем элементе массива. Для сравнения двух таких чисел нужно сначала сравнить их длины (игнорируя ведущие нули, если по какой-то причине они присутствуют в массиве), а если длины оказались равны, то просто сравнить эти два массива как строки.

Сложение и вычитание n-битовых целых легко реализовать за $O(n)$ используя школьный метод сложения в столбик. Умножение мы тоже можем реализовать в столбик за $O(n^2)$, но значительно для битового представления проще реализовать умножение в виде следующего рекурсивного алгоритма. В качестве идеи алгоритма возьмём следующее соотношение:

$$a \cdot b = \begin{cases} 2(a \cdot \lfloor b/2 \rfloor), & b \text{ — чётное,} \\ 2(a \cdot \lfloor b/2 \rfloor) + a, & b \text{ — нечётное.} \end{cases}$$

```
# Умножение двух битовых чисел
def multiply(a, b):
    if b == 0:
        return 0

    # рекурсивно вычисляем 2(a · ⌊b/2⌋)
    c = 2 * multiply(a, b // 2)

    # обработка нечётного b
    if b % 2 == 1:
        c = c + a

    return c
```

Проанализируем сложность этого алгоритма, если длина a составляет n битов, длина b — m битов, и $n \geq m$. В этом коде проверка на равенство нулю, умножение и деление на два, а также взятие остатка по модуля два, можно реализовать за $O(1)$, пользуясь тем, что числа заданы битовыми массивами. Остаётся

только сложение c и a, которое занимает время пропорциональное сумму длин $|c| + |a| \leq 3n$. Таким образом каждый вызов `multiply` без учёта вложенных рекурсивных вызовов выполняется за $O(n)$. Глубина рекурсии равна количеству бит в записи b, поэтому в сумме получается $O(n \cdot m)$.

Теперь поговорим о делении. Запрограммировать «школьный» алгоритм деления в столбик довольно сложно (попробуйте сами!). Значительно проще реализовать следующий рекурсивный для деления битовых чисел с остатком (идея алгоритма аналогична идеей алгоритма умножения, рассмотренного выше).

```
# Деление двух битовых чисел с остатком
# Возвращает пару (частное, остаток)
def divide(a, b):
    # база рекурсивного алгоритма
    if a < b:
        return (0, a)

    # рекурсивное вычисляем ⌊⌊a/2⌋/b⌋ и ⌊a/2⌋ mod 2
    (q, r) = divide(a // 2, b)

    # умножаем результаты на 2
    q = q * 2
    r = r * 2

    # если a был нечётным, то к остатку ещё нужно добавить 1
    if a % 2 == 1:
        r = r + 1

    # исправляем результат, если остаток стал больше делителя
    if r > b:
        q = q + 1
        r = r - b

    return (q, r)
```

Оценим сложность этого алгоритма для n-битового a и m-битового b, где $n \geq m$. Умножение на два, деление на два, а также взятие остатка по модуля два, реализуются за $O(1)$. Сравнение a и b, сложение и вычитание можно реализовать за $O(n+m) = O(n)$, т.е. каждый вызов `divide` без учёта рекурсивных вызовов выполняется за $O(n)$. Общее количество рекурсивных не более в $n - m$, в результате получается $O(n \cdot (n - m + 1))$ (если $n = m$, то алгоритм выполнится за $O(n)$).

Для наших целей будет полезно научиться также вычислять *наибольший общий делитель (НОД, greatest common divisor, GCD)* двух чисел. Для этой задачи существует широко известный алгоритм Евклида, который появился задолго по появления вычислительных машин. Идея алгоритма Евклида содержится в следующем соотношении:

$$\gcd(a, b) = \gcd(b, a \bmod b).$$

Действительно, пусть $a = bq + r$, где $r = a \bmod b$. Если какое-то число делит b и r, то оно обязательно делит a. Если какое-то число делит a и b, то оно также делит и $r = a - bq$. Таким образом множество общих делителей пары (a, b) совпадает с множеством общих делителей (b, r), а следовательно наибольший делитель у этих

пар тоже общий. Таким образом можно использовать следующий алгоритм для вычисления наибольшего общего делителя двух чисел:

```
# алгоритм Евклида
# вычисление наибольшего общего делителя
def gcd(a,b):
    if b == 0:
        return a

    return gcd(b, a % b)
```

Однако для дальнейших целей на будет полезно научиться также находить так называемые *коэффициенты Безу*.

Теорема 8.1.1 (Теорема Безу). *Для любых целых a и b из того, что $\gcd(a,b) = d$, следует, что существуют такие целые x и y, что*

$$ax + by = d.$$

Такие коэффициенты x и y называются *коэффициентами Безу*. Для нахождения коэффициентов Безу используется *расширенный алгоритм Евклида*.

Замечание 8.1.1. Уравнения в целых числах называются *Диофантовыми уравнениями*. В теореме Безу мы имеем дело с линейным Диофантовым уравнением от двух переменных $ax + by = d$. Можно показать, что у такого уравнения будет бесконечное количество решений (тут важно, что $d = \gcd(a,b)$). Алгоритм Евклида находит в некотором смысле наименьшее решение, а именно такое, что $|x| \leq b$ и $|y| \leq a$.

Пусть пусть $a = bq + r$, где $r = a \bmod b$, и $\gcd(a,b) = d$. Предположим, что мы нашли некоторые x' и y' такие, что

$$bx' + ry' = d.$$

Покажем, как вычислить x и y. Подставим $r = a - bq$.

$$bx' + (a - bq)y' = d.$$

Перегруппируем члены этого выражения:

$$ay' + b(x' - qy') = d.$$

Получается, что $x = y'$, а $y = x' - qy'$. Что приводит нас к следующему алгоритму.

```
# расширенный алгоритм Евклида
# вычисление наибольшего общего делителя и коэффициентов Безу
# возвращает тройку из НОД и коэффициентов Безу
def egcd(a, b):
    if b == 0:
        return (a, 1, 0)

    # вычисляем q и r
    (q, r) = divide(a, b)

    # рекурсивно вычисляем d, x' и y'
```

```
    (d, x_, y_) = egcd(b, r)

    # вычисляем x и y
    x = y_
    y = x_ - q * y_

    return (d, x, y)
```

Для оценки сложности этого алгоритма заметим, если вызвать алгоритм для $a < b$, то первый же рекурсивный вызов «переставит» аргументы местами, и во всех следующих вызовах $a \geq b$. Если $a \geq b$, то $r < a/2$, т.е один из аргументов на каждой итерации уменьшается как минимум вдвое. Действительно, если $a \geq 2b$, то $r < b \leq a/2$, если же $a < 2b$, то $r = a - b < a/2$. Поэтому можно оценить время работы этого алгоритма для двух для n-битовых как $O(n^3)$, т.к. глубина рекурсии ограничена $O(n)$ и на каждой итерации требуется вычислить `divide(a,b)`, который мы умеем вычислять за $O(n^2)$.

Более аккуратный анализ позволяет получить оценку $O(n^2)$. Для этого сначала докажем, следующую лемму.

Лемма 8.1.1. *Для любых целых a и b, $a \geq b > 0$, функция* `egcd(a,b)` *вернёт тройку (d, x, y), где $|x| \leq b$ и $|y| \leq a$.*

Доказательство. Будем доказывать по индукции. В качестве базы рассмотрим случай, когда a делится на b. Тогда `r` будет равен нулю, а следовательно, рекурсивный вызов `egcd(b,r)` вернёт $(b, 1, 0)$. Получаем, что $|x| = 0 \leq b$ и $|y| = 1 \leq a$, т.е. для этого базового случая лемма верна.

Теперь рассмотрим случай, когда $a > b > 0$, и a не делится на b. Предположим, что для всех $a' < a$ и $b' < b$ лемма верна. Положим, как в алгоритме, $q = \lfloor a/b \rfloor$ и $r = a \bmod b$. Тогда вложенный в `egcd(a,b)` вызов `egcd(b,r)` вернёт тройку (d, x', y'), для которой по предположению индукции выполняется $|x'| \leq r$ и $|y'| \leq b$. Тогда

$$
\begin{aligned}
&|x| = |y'| \leq r = a \bmod b < b, \\
&|y| = |x' - qy'| \leq |x'| + |qy'| \leq r + qb = a. \quad \square
\end{aligned}
$$

Теперь используя эту лемму покажем, что n-битового a и m-битового b, где $n \geq m$, алгоритм выполняется за $O(n^2)$ операций. Будем тоже доказывать это по индукции. База индукции: для однобитных чисел достаточно константного количества операций. Предположим, что для любых пар чисел, длины которых не превосходят m, соответственно, алгоритм выполняется за $O(m^2)$. Тогда сложность вычисления `egcd(a,b)` складывается из сложности вызова `divide`, рекурсивного вызова `egcd` и сложности вычисления y, т.е. $O(n(n-m+1))$, $O(m^2)$ и $O(|x'| + |q| \cdot |y'|)$. Последнее слагаемое по лемме 8.1.1 можно ограничить $O(n + (n-m) \cdot m)$. Все три слагаемые не превосходят $O(n^2)$, так мы и получаем желаемую оценку.

Замечание 8.1.2. В данном алгоритме нам потребуется работа с отрицательными числами. Для представления отрицательных чисел потребуется ещё один дополнительный бит, отвечающий за знак числа, т.е. для хранения n-битового числа нужен $n+1$ бит. Кроме того, в реализацию арифметических операций нужно будет добавить код с обработкой знака.

Ещё один полезный алгоритм — это алгоритм для возведения в степень. Наивный алгоритм, который для вычисления числа a^b в цикле b умножает на a (и,

следовательно, требует b умножений), нельзя назвать эффективным. Дело в том, что значение n-битового числа может достигать 2^n, а значит время предложенного алгоритма *экспоненциально* от длины числа b. Более эффективный алгоритм можно построить, если действовать аналогично алгоритму умножения, рассмотренному выше.

```
# Быстрое возведение в степень
def power(a, b):
    if b == 0:
        return 1

    # рекурсивно вычисляем a^⌊b/2⌋
    res = power(a, b // 2)

    # возводим результат в квадрат
    res = res * res

    # обработка нечётного b
    if b % 2 == 1:
        res = res * a

    return res
```

Давайте проанализируем время работы этого алгоритма для пары n-битовых чисел a и b. Глубина рекурсии не превосходит n, а на каждом уровне рекурсии вычисляется три умножения. В результате получаем оценку $O(n^3)$, что значительно лучше экспоненциальной оценки наивного алгоритма.

К сожалению, в предыдущем рассуждении мы совершили очень грубую ошибку. Мы предположили, что все умножения будут стоить не более $O(n^2)$, но эта оценка верна только для n-битовых чисел. Давайте проанализируем, какой длины будут числа, которые получаются в данном алгоритме. Для простоты будем считать, что $a = b = 2^n$. Сколько бит в числе a^b? Количество битов в двоичной записи числа с точностью до константы равно двоичному логарифму этого числа. В нашем случае $\log_2 a^b = \log_2 2^{n2^n} = n2^n$. Таким образом на верхнем уровне рекурсии будут умножаться два $n2^{n-1}$-битных числа. Соответственно, сложность этого алгоритма уж точно не меньше $\Omega(n^2 2^{2n})$, т.е. алгоритм имеет экспоненциальную сложность. Эта ошибка показывает, насколько внимательным нужно быть при оценке сложности алгоритмов.

8.2. Модульная арифметика

Модульная арифметика — это работа с целыми числами в кольце остатков по некоторому модулю m, т.е. все числа и все результаты операций берутся по модулю m. Реализовать сложение двух чисел по модулю очень просто — нужно вычислить результат как с обычными целыми числами, а от результата взять остаток по модулю m.

```
# Сложение двух чисел по модулю
def plus_mod(a, b, m):
    return (a + b) % m
```

```
# Умножение двух чисел по модулю
def multiply_mod(a, b, m):
    return (a * b) % m
```

Пусть m — n-битное число. Тогда оба алгоритма имеют сложность $O(n^2)$.

Осталось разобраться с делением по модулю. Деление по модулю — это деление в кольце остатков, оно работает не так, как деление обычных целых чисел. Для того, чтобы поделить a на b по модулю m, нужно найти такое число $q \in [m-1]$, что

$$bq = 1 \mod m.$$

Такое число q называется *обратным* к b и обозначается b^{-1}. Теперь можно определить деление на b как умножение на b^{-1}:

$$a/b = ab^{-1} \mod m.$$

Нетрудно проверить, что обратные могут быть только у чисел взаимно простых с m. Пусть $\gcd(b,m) = d > 1$. Тогда bq делится на d для любого q, а следовательно остаток от bq при делении на m тоже делится на d, т.е. $(bq \bmod m) \neq 1$.

Пусть b и m взаимно просты. Как найти обратное к b? Тут нам поможет расширенный алгоритм Евклида. Давайте найдём коэффициенты Безу для b и m. Это будет пара x и y удовлетворяющие следующему соотношению $bx + my = 1$. Рассмотрим это соотношение по модулю m. Тогда в левой части второе слагаемое обнуляется, и мы получаем

$$bx = 1 \mod m,$$

т.е. x — обратное к b. Осталось заметить, что расширенный алгоритм Евклида гарантирует, что $|x| \leq m$. Это даёт следующий алгоритм деления.

```
# Деление на a на b по модулю m
# предполагается, что gcd(b, m) = 1
def divide_mod(a, b, m):
    # вычисляем коэффициенты Безу
    (d,x,y) = egcd(b, m)

    # обработка отрицательного x
    if x < 0:
        x = x + m

    return (a * x) % m
```

Сложность полученного алгоритма $O(n^2)$.

Давайте также проанализируем следующий код, вычисляющий $a^b \bmod m$.

```
# Быстрое возведение в степень по модулю
def power_mod(a, b, m):
    if b == 0:
        return 1

    # рекурсивно вычисляем a^⌊b/2⌋ mod m
    res = power_mod(a, b // 2, m)

```

```
2884     # возводим результат в квадрат
2885     res = res * res
2886
2887     # обработка нечётного b
2888     if b % 2 == 1:
2889         res = res * a
2890
2891     return res % m
```

2892 Поскольку все числа в этом алгоритме не превосходят m^3, то алгоритм имеет слож-
2893 ность $O(n^3)$ (теперь без подвоха).

Глава 9

Альтернативные модели вычисления

До этого момента мы всегда работали с моделью памяти с произвольным доступом к памяти. Такая модель позволяет формализовать алгоритмы, которые выполняются на одном процессоре, а все данные хранятся в оперативной памяти. Однако, современные вычислительные системы могут быть значительно сложнее. Например, может оказаться, что данных так много, что не влезают в оперативную память одного компьютера. Такие данные приходится хранить на диске или даже не нескольких дисках, доступ к которым значительно более медленный, и это нужно учитывать при разработке алгоритмов. Кроме того, вычисления могут производиться не на одном процессоре, а сразу на нескольких машинах, которые взаимодействуют между собой. В этой главе мы рассмотрим несколько моделей, которые позволяют описать алгоритмы для работы с большими данными и распределённые вычисления. В заключение, поговорим о других моделях, которые не вошли в этот обзор.

9.1. Работа с большими данными

В этом разделе мы поговорим про модель внешней памяти, которая позволяет формализовать взаимодействие в внешними запоминающими устройствами, и подходит для описания алгоритмов над данными, которые не влезают в оперативную память. Кроме этого мы обсудим важное уточнение этой модели — модель cache-oblivious.

9.1.1. Модель внешней памяти

Модель внешней памяти (external memory model) описывает вычислительное устройство, состоящее из процессора с внутренней памятью размера M (соответствует оперативной памяти), также называемой *кешем*, и подключенной к нему неограниченной *внешней памяти* (соответствует внешнему запоминающему устройству или взаимодействию по сети). Внутренняя и внешняя память разбиты на блоки размера B. Все операции с внешней памятью оперируют блоками: за одну операцию можно прочитать или записать один блок размера B. Сложность алгоритма в этой модели определяется, как количество таких операций со внешней памятью.

Может показаться странным, что сложность алгоритма никак не учитывает количество операций процессора. Дело в том, что в данной модели предполагается, что операции с данными во внутренней памяти на несколько порядков быстрее, чем чтение и запись данных из внешней памяти. Поэтому вклад процессорного времени в сложность алгоритма незначителен по сравнению вкладом операций ввода-вывода.

Какой сложности алгоритмов мы хотели бы добиться в этой модели? В идеале, мы хотели бы «перенести» сложность алгоритмов из модели произвольной памяти, заменив оценках n на n/B. В этом случае мы будем говорить, что полученный алгоритм является эффективным. Например, если в RAM-модели некоторая задача решается за $O(n)$, то в модели внешней памяти мы хотим построить алгоритм, работающий за $O(n/B)$. Далее, на примере нескольких задач мы увидим, когда это удаётся сделать, а когда — нет.

Поиск элемента в массиве

В RAM-модели задача поиска элемента в массиве решается за $O(n)$. Например, можно последовательно перебрать все элементы массива и сравнить их с искомым. В модели внешней памяти эта задача решается за $O(n/B)$ совершенно аналогичным образом: массив последовательно считывается во внутреннюю память по блокам, и все элементы текущего блока сравниваются с искомым элементом. В худшем случае такой алгоритм считает не более $\lceil n/B \rceil + 1 = O(n/B)$ блоков.

Двоичный поиск в упорядоченном массиве

В RAM-модели задача поиска элемента в упорядоченном массиве решается за $O(\log n)$. Попробуем реализовать аналогичный алгоритм в модели внешней памяти. Несложно заметить, что такой алгоритм всё так же будет работать за $\Theta(\log n)$. Это происходит из-за того, что при двоичном поиске нам нужно проверять элементы массива, которые далеко находятся друг от друга, следовательно на каждый элемент потребуется отдельная операция чтения. Так как чтение происходит по блокам, то на каждый интересующий нас элемент массива мы невольно дополнительно считываем $B - 1$ элемент, которые никак не используются. Это не эффективно, но, к сожалению, при двоичном поиске это неизбежно.

Тем не менее, есть способ организовать данные во внешней памяти так, чтоб поиск можно было реализовать эффективно. Для этого нужно хранить данные в сбалансированном B-дереве поиска, размер вершины которого, совпадает с размером блока. В таком B-дереве можно искать за $O(\log_B n)$.

Сортировка слиянием

Сортировка слиянием является хорошим кандидатом на сортировку, которая будет хорошо себя вести в модели со внешней памятью. Действительно, в этом алгоритме данные в массивах читаются последовательно, что позволяет легко модифицировать его для чтения по блокам. Для того, чтобы не задумываться о том, достаточно ли внутренней памяти для хранения дерева рекурсии, мы будем использовать нерекурсивную версию этого алгоритма. Вся работа с очередь массивов, которая нужна для нерекурсивной реализации этого алгоритма, эффективно реализуется во внешней памяти, т.к. по своей сути является работой с последовательными данными.

Осталось разобраться с реализацией слияния. Для слияния двух массивов будем поддерживать во внутренней памяти три блока: текущий блок с данными первого массива, текущий блок с данными второго массива и блок с результатами слияния. Когда элементы какого-то из первых двух блоков полностью обработаны, этот блок заменяется следующим блоком соответствующего массива. Когда

блок с результатами заполняется, то его содержимое записывается во внешнюю память. В результате мы получаем алгоритм, который имеет сложность $O(\frac{n}{B}\log\frac{n}{B})$.

Полученный алгоритм можно сделать ещё более эффективным. Недостаток предложенной реализации заключается в том, что внутренняя память используется неэффективно — по сути используются только три блока. Это можно исправить, если реализовать сортировку слиянием, которая разбивает массив не на две части, а на k частей, и, соответственно, сливает массивы сразу k упорядоченных массивов (k-ways mergesort). В RAM-машине такой алгоритм будет иметь сложность $\Theta(nk\log_k n)$ — множитель k в оценке возникает из-за того, что при слиянии k массивов на каждой итерации требуется выбрать минимум из k элементов. Однако в модели внешней памяти операции процессора не учитываются. Поэтому реализация такого алгоритма в этой модели будет иметь сложность $O(\frac{n}{B}\log_k\frac{n}{B})$ при условии, что $k+1$ блоков влезет во внутреннюю память (дополнительный блок нужен для хранения результата слияния). Поэтому возьмём $k = M/B - 1$. Итоговая сложность в модели внешней памяти $O(\frac{n}{B}\log_{M/B}\frac{n}{B})$.

9.1.2. Модель cache-oblivious

Модель *cache-oblivious* (можно примерно перевести как модель для алгоритмов не обращающие внимание на кеш) — это модификация модели внешней памяти, в которой алгоритму неизвестны значения M и B (и у алгоритма нет возможности их узнать). Другими словами, в этой модели такое же вычислительное устройство, как и в модели внешней памяти, но всё кеширование и чтение по блокам реализовано на более низком уровне, к которому у алгоритма нет доступа. Т.е. алгоритму известно про кеширование, но не известно какие именно параметры кеширования используются. Такое допущение сильно всё усложняет. Например, в таких условиях у нас уже не получится применить алгоритм сортировки, рассмотренный выше — в нём мы опирались на знание M и B. С другой стороны, обычный алгоритм линейного поиска элемента в массиве будет работать вне зависимости от значений M и B: при последовательном чтении элементов из внешней памяти они будут загружаться в память блоками (неизвестного алгоритму размера) и таким образом каждый элемент будет прочитан только один раз. Следовательно, такой простой алгоритм будет иметь оптимальную сложность $O(n/B)$. Про такие алгоритмы принято говорить, что они являются *cache-friendly*.

Удивительным образом, даже в отсутствии информации о значении M и B, удаётся построить оптимальные алгоритмы для множества задач. В том числе можно построить алгоритм сортировки за время $O(\frac{n}{B}\cdot\log_{M/B}\frac{n}{B})$. Однако алгоритмы получаются значительно сложнее, чем сортировка в модели внешней памяти (например, там могут потребоваться специальные довольно специфические структуры данных, которые позволяют эффективно сливать упорядоченные массивы, не зная про параметры кеширования). Изучение этих алгоритмов выходит за пределы этого курса.

Нельзя не упомянуть, почему изучение этой модели так важно. Дело в том, что имеет место следующий факт (тут он изложен неформально, но его можно сформулировать строго и доказать).

Утверждение 9.1.1. *Если алгоритм эффективен в модели* cache-oblivious*, то он эффективно использует кеширование в системах с иерархическим кешированием.*

Это особенно интересно, т.к. современные вычислительные устройства широко используют иерархическое кеширование: у процессора есть несколько уров-

ней кеширования для работы в оперативной памятью, далее идут кеши внешних запоминающих устройств, которые так же могут быть иерархическими, взаимодействие по сети тоже кешируется, и т.д. Получается, что эффективные алгоритмы для этой модели очень полезны на практике. К сожалению, зачастую сложность таких алгоритмов нивелирует все их преимущества.

9.2. Параллельные и распределённые вычисления

В этом разделе мы рассмотрим две модели для параллельных и распределённых вычислений.

9.2.1. Модель PRAM

Параллельная модель RAM (parallel RAM, PRAM) описывает вычислительное устройство с P процессорами и единой общей разделяемой между всеми процессорами памятью. При этом считается, что для работы с памятью уже реализованы необходимые механизмы синхронизации. В зависимости от типа механизмов синхронизации выделяют три разновидности PRAM:

- с эксклюзивным чтением и записью (EREW, exclusive read — exclusive write),
- с одновременным чтением и эксклюзивной записью (CREW, concurrent read — exclusive write),
- с одновременным чтением и записью (CRCW, concurrent read — concurrent write).

Сложность вычисления определяется максимальное среди всех процессоров количество совершённых операций.

Данная модель позволяет описывать алгоритмы для параллельных вычислений, но на практике возможность применения таких алгоритмов ограниченна. Дело в том, что количество процессоров в пределах одного компьютера ограниченно в т.ч. по инженерным соображения (на практике P обычно исчисляется десятками или сотнями, в редких случаях несколькими тысячами). При необходимости увеличения числа процессоров на практике используется кластеры, состоящие из множества отдельных компьютеров, соединённых по сети, каждый из которых обладает своей собственной памятью. Таким образом, эта модель имеет практическое применение только для $P = O(1)$, что мало интересно с точки зрения сложности алгоритмов, т.к. такое допущение не позволяет ускорить вычисления более чем в константу раз.

Если же разрешить количеству процессоров P быть неконстантным, то в этой модели возникают алгоритмы с неожиданными оценками сложности, которые бесполезны для применения на практике. Для иллюстрации давайте посмотрим на алгоритм поиска максимума для модели CRCW PRAM, который находит максимум в массиве длины массиве из n за $O(1)$ для $P = n^2$.

Поиск максимума в модели CRCW PRAM

В CRCW PRAM нескольким процессорам разрешено одновременно обращаться к одной и той же ячейке памяти, как на чтение, так и на запись. При этом гарантируется, что при одновременном чтении все прочитанные данные будут

корректны (соответствовать значениям, которые были записаны в ячейку), а при одновременной записи разными процессорами в одну ячейку новое значение ячейки будет соответствовать одному из записываемых значений. Алгоритм будет состоять из трёх фаз.

1. На первой фазе n процессоров одновременно заполняют массив `maximum` размера n значениями `True`. Таким образом, изначально каждый элемент массива является кандидатом на то, чтобы быть максимумом. При этом каждый процессор выполняет $O(1)$ операций.

```
# цикл выполняется параллельно на n процессорах
for i in range(n):
    maximum[i] = True
```

2. На второй фазе нам потребуются все n^2 процессоров. Каждый процессор сравнивает два элемента массива и, если оказывается, что первый элемент меньше второго, то отмечает, что первый элемент не может быть максимумом. При этом каждый процессор выполняет $O(1)$ операций.

```
# цикл выполняется параллельно на n^2 процессорах
for i in range(n):
    for j in range(n):
        if a[i] < a[j]:
            maximum[i] = False
```

3. К третьей фазе в массиве `maximum` значения `True` могут остаться только в тех ячейках, которые соответствуют элементам массива, которые не меньше всех оставшихся. Другими словами, `True` записано в только ячейках, которые соответствую максимумам. Осталось найти такую ячейку и записать её номер в переменную `result`. На этой фазе нам потребуется n процессоров, каждый из которых опять выполнит только $O(1)$ операций.

```
# n процессоров вычисляют ответ
for i in range(n):
    if maximum[i]:
        result = i
```

Полученный алгоритм действительно имеет сложность $O(1)$, но совершенно бесполезен на практике, ведь для того, чтобы найти максимум среди тысячи элементов, требуется компьютер с миллионом процессоров.

Упражнение 9.2.1. Укажите, где в этом алгоритме используются свойства CRCW PRAM? Будет ли алгоритм работать в других вариантах PRAM?

9.2.2. Модель BSP

Модель *частично-синхронного параллелизма (Bulk-Synchronous Parallel, BSP)* состоит из

- P процессоров, каждый со своей собственной локальной памятью (1 единица времени на операцию),

- *коммуникационной среды*: сети для обмена сообщениями и общей внешней памяти (G единиц времени на пересылку одного значения)[1],
- *барьерный механизм синхронизации* (L единиц времени на синхронизацию).

Вычисления в такой модели устроены следующим образом. Вход алгоритма записывается во внешнюю память. Процесс вычисления разбит на фазы — *супершаги*. Внутри супершага процессоры взаимодействуют с внешней памятью, производят вычисления, используя данные во внутренней памяти, и посылают сообщения другим процессорам. Все эти действия происходят асинхронно, т.е. один процессор ничего не знает про состояние другого процессора. Внутри супершага процессоры не могут как-то взаимодействовать между собой (один процессор может послать сообщение другому процессору, но доставка может случиться только в самом конце супершага). Супершаг заканчивается, когда происходит барьерная синхронизация. Гарантируется, что после завершения синхронизации все процессоры закончат выполнение своих текущих заданий, а все посланные сообщения будут доставлены. После завершения синхронизации начинается следующий супершаг.

Такая общая модель позволяет описать почти любую распределённую вычислительную систему. Тут стоит отметить, что внешняя память и механизм синхронизации могут быть реализованы на основе сети для обмена сообщениями. В то же время, сама сеть может быть реализована на основе доступа к общей внешней памяти. Таким образом, некоторые составляющие части модели BSP, которые могут отсутствовать в какой-то конкретной вычислительной системе, могут быть реализованы за счёт других составляющих.

Для каждой конкретной распределённой вычислительной системы можно подобрать значения P, G и L, которые её описывают. Параметр G соответствует *отставанию сети (обратной пропускной способности)*, т.е. показывает во сколько раз время пересылки одной ячейки памяти превышает время доступа к ячейке памяти во внутренней памяти процессора, которое принято за единицу. Параметр L соответствует *задержке* сети, т.е. ограничивает время, которое в худшем случае нужно на доставку сообщения. Например, для суперкомпьютера Cray T3E значения параметров будут такими: $P = 64$, $G \approx 78$, $L \approx 1825$.

Сложность вычисления естественным образом соответствует количество потраченных единиц времени (по глобальному времени). Введём следующие обозначения. Для конкретного процессора обозначим

- $\mathrm{comp}(s, p)$ — число операций совершенных процессором p на супершаге s,
- $\mathrm{comm}(s, p)$ — число ячеек переданных и полученных процессором p на супершаге s.

Теперь для всего супершага s можно обозначить

- $\mathrm{comp}(s) = \max_{p \in [P]} \mathrm{comp}(s, p)$ — общее число операций на супершаге s,
- $\mathrm{comm}(s) = \max_{p \in [P]} \mathrm{comm}(s, p)$ — общая коммуникация на супершаге s.

Тогда сложность шага s можно определить так:

$$\mathrm{cost}(s) = \mathrm{comp}(s) + G \cdot \mathrm{comm}(s) + L.$$

[1] В рамках модели BSP предполагается, что посылка сообщений и взаимодействие с внешней памятью имеют общую природу и стоят одинаково.

Если вычисление содержит sync синхронизаций (и, следовательно, sync супершагов), то общая сложность вычисления определяется так:

$$\text{cost} = \sum_{s \in [\text{sync}]} \text{cost}(s) = \text{comp} + G \cdot \text{comm} + L \cdot \text{sync},$$

где comp и comm — это общее количество вычислений и общее количество коммуникаций, соответственно. Так как мы не знаем конкретных значений P, G и L, то мы будем оценивать составляющие comm, comp и sync по-отдельности.

При построении алгоритмов в этой модели мы будем предполагать, что размер входных данных всегда значительно превышает количество процессоров, т.е. $n \gg P$. Какие алгоритмы в этом модели мы могли бы назвать эффективными? В идеале мы ожидаем, что ускорение от использования распределённых вычислений будет пропорционально количеству процессоров. Поэтому для задач, которые решаются на RAM машине за $O(f(n))$ с использованием $O(g(n))$ операций ввода-вывода, мы будем стремиться получить алгоритм, для которого

$$\text{comp} = O(f(n)/P), \quad \text{comm} = O(g(n)/P).$$

Более того, мы хотели бы минимизировать количество синхронизаций. Мы будем стремиться разрабатывать алгоритмы, в которых количество синхронизаций не зависит от размера входа, т.е. sync может быть функцией P, но не n, а ещё лучше — константой. Давайте рассмотрим пару примеров таких алгоритмов.

Поиск элемента в массиве

Для определённости будем искать первое вхождение элемента в массив в предположении $n = \Omega(P^2)$. Алгоритм будет состоять из двух супершагов. На первом супершаге каждый процессор считает из внешней памяти и проверит свой подмассив размера n/P. После этого каждый процессор, который нашёл искомый элемент в своём подмассиве, отправит индекс первого вхождения некоторому выделенному процессору, например, процессору с номером 1. Таким образом, для первого супершага мы получаем

$$\text{comp}(1) = O(n/P), \quad \text{comm}(1) = O(n/P).$$

На втором супершаге этот выделенный процессор выберет из присланных индексов наименьший и запишет его в переменную для ответа. Соответственно,

$$\text{comp}(2) = O(P), \quad \text{comm}(2) = O(1).$$

Таким образом, при $n = \Omega(P^2)$ данный алгоритм имеет следующие оценки:

$$\text{comp} = O(n/P), \quad \text{comm} = O(n/P), \quad \text{sync} = O(1).$$

Сортировка методом регулярного сэмплирования

Будем предполагать, что $n = \Omega(P^3)$, и что все элементы массива различны. Алгоритм будет состоять из четырёх супершагов.

1. Каждый процессор считывает из внешней памяти и сортирует свой подмассив размера n/P. Из полученного упорядоченного массива выбирается набор из P порядковых статистик с шагом n/P^2 начиная с индекса 0, будем

называть их *пробами*. Таким образом массив разбивается на P блоков одинакового размера, причём каждый блок начинается с элемента-пробы. Все процессоры пересылают свои наборы проб некоторому выделенному процессору.

$$\mathrm{comp}(1) = O\big((n \log n)/P\big), \quad \mathrm{comm}(1) = O(n/P).$$

2. Выделенный процессор получает P^2 проб, сортирует их и выбирает из них P порядковых статистик с шагом P начиная с индекса 0, будем называть их *разделителями* и обозначать $s_1, \ldots, s_P$. Полученный набор разделителей пересылается всем процессорам.

$$\mathrm{comp}(2) = O(P^2 \cdot \log P), \quad \mathrm{comm}(2) = O(P^2).$$

3. Разделители задают разбиение элементов исходного массива на P (не обязательно равных) частей. Теперь каждому процессору с номером i будут соответствовать полуинтервал элементов от $[s_i, s_{i+1})$ (последнему процессору соответствует полуинтервал $[s_P, \infty)$). На третьем супершаге каждый процессор просматривает элементы своего подмассива (полученного на первом супершаге и уже к этому моменту упорядоченного) и отсылает каждый элемент тому процессору, которой отвечает за соответствующий полуинтервал. Для того, чтобы оценить число сообщений, которое в конце супершага получит каждый процессор, нам придётся доказать следующее утверждение.

Утверждение 9.2.1. *Каждый полуинтервал содержит не более $2n/P$ элементов исходного массива.*

Доказательство. Для каждого полуинтервала T мы будем называть блок B *левым*, если его левая граница не больше левой границы T. Все остальные блоки мы будем называть *правыми*. Верны следующие наблюдения.

- У каждого процессора есть не более одного левого блока, который бы пересекался с T. Действительно, если какой-то левый блок некоторого процессора пересекает T, то все блоки до него не пересекают T, а все блоки после него — правые.
- Среди всех блоков всех процессоров есть не более P правых блоков, которые пересекаются с T. Каждый правый блок пересекающийся с T начинается с некоторой пробы, причём эта проба обязательно принадлежит T. Так как каждый полуинтервал содержит не более P проб, то таких блоков не больше P.

Следовательно, каждый интервал содержит элементы из не более, чем $2P$ блоков, т.е. в сумме не более $2P \cdot n/P^2 = 2n/P$ элементов. □

Это позволяет оценить сложность третьего супершага:

$$\mathrm{comp}(3) = O(n/P), \quad \mathrm{comm}(3) = O(n/P).$$

Оценка на коммуникацию $O(n/P)$ учитывает и n/P элементов, которые послал каждый процессор, и не более $2n/P$ элементов, которые каждый процессор получил.

4. Каждый процессор сортирует элементы своего интервала и записывает их во внешнюю память.

$$\mathrm{comp}(4) = O((n \log n)/P), \quad \mathrm{comm}(4) = O(n/P).$$

В сумме при условии $n = \Omega(p^3)$ получаем желаемые оценки:

$$\mathrm{comp} = O\big((n \log n)/P\big), \quad \mathrm{comm} = O(n/P), \quad \mathrm{sync} = O(1).$$

Можно показать, что для данной эти оценки являются оптимальными для сортировок, основанных на сравнениях.

9.3. Другие модели

Есть множество других моделей, которые оказались за рамками этого экскурса. Например, есть гибридная модель, объединяющая модель внешней памяти и PRAM — *параллельная модель внешней памяти (parallel external memory)*. Можно описать модель вычисления для формализации алгоритмов на GPU — модель описывающая принцип *«одиночный поток команд, множественный поток данных», ОКМД (SIMD, single instruction multiple data)*. Есть и другие модели, непохожие на то, что мы уже видели. К ним, например, относится модель вычислений для *потоковых алгоритмов*, в которой вычисляющее устройство с ограниченной памятью может читать входные данные исключительно последовательно и только один раз. Представьте себе, например, сетевой маршрутизатор, обрабатывающий сетевой трафик в реальном времени, пытающийся выделить наиболее популярные IP адреса. Память маршрутизатора несравнимо меньше, чем объём трафика, который через него проходит. Однако, некоторые характеристики можно вычислить и в такой ограниченной модели, чему посвящено большое число статей и даже книг.

Кроме того, стоит упомянуть ещё модели, которые позволяют формализовать понятие алгоритма для квантовых компьютеров. Это отдельная интересная тема, которая на данный момент всё ещё является исключительно теоретической (существующие квантовые компьютеры имеют очень ограниченные возможности).

Приложение А

Математические факты

А.1. Суммирование последовательностей

Лемма А.1.1. *Сумма первых n членов* арифметической прогрессии

$$a_1 = a, \quad a_i = a_{i-1} + d = a + (i-1)d,$$

вычисляется по формуле

$$S_n = \frac{a_1 + a_n}{2} \cdot n = \frac{2a + (n-1)d}{2} \cdot n.$$

Доказательство.

$$2S_n = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} a_{n-i+1} = \sum_{i=1}^{n} (a_i + a_{n-i+1}) = \sum_{i=1}^{n} (2a + (n-1)d).$$

Отсюда получаем, что

$$S_n = \frac{1}{2} \sum_{i=1}^{n} (2a + (n-1)d) = \frac{n}{2} \cdot (2a + (n-1)d).$$

□

Лемма А.1.2. *Сумма первых n членов* геометрической прогрессии

$$a_1 = a, \quad a_i = a_{i-1} \cdot q = a \cdot q^{i-1},$$

вычисляется по формуле

$$S_n = \frac{a_{n+1} - a_1}{q - 1} = a \cdot \frac{q^n - 1}{q - 1}.$$

Доказательство.

$$S_n = \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} a \cdot q^{i-1}.$$

$$qS_n = q \sum_{i=1}^{n} a \cdot q^{i-1} = \sum_{i=2}^{n+1} a \cdot q^{i-1}.$$

$$qS_n - S_n = \sum_{i=2}^{n+1} a \cdot q^{i-1} - \sum_{i=1}^{n} a \cdot q^{i-1} = a \cdot q^n - a.$$

Отсюда получаем, что

$$S_n = \frac{a \cdot q^n - a}{q - 1} = a \cdot \frac{q^n - 1}{q - 1}.$$

□

Лемма А.1.3. *Сумма бесконечной убывающей геометрической прогрессии с* $q \in (0,1)$ *вычисляется по формуле* $S = \frac{1}{1-q}$.

Доказательство.

$$S = \lim_{n\to\infty} a \cdot \frac{q^n - 1}{q-1} = a \cdot \frac{-1}{q-1} = \frac{a}{1-q}.$$

□

Лемма А.1.4. *Сумма первых* n *членов бесконечной убывающей последовательности* $a_i = i \cdot q^i$ *при* $q \in (0,1)$ *вычисляется по формуле* $S_n = \frac{q}{(1-q)^2}$

Доказательство. Рассмотрим функцию $f(x) = \sum_{i=1}^{\infty} x^{i-1}$ и вычислим её вычислим производную.

$$f'(x) = \sum_{i=1}^{\infty} (i-1)x^{i-2} = \sum_{i=1}^{\infty} i \cdot x^{i-1}.$$

С другой стороны по лемме А.1.3 $f(x) = \frac{1}{1-x}$ при $x \in (0,1)$.

$$f'(x) = \left(\frac{1}{1-x}\right)' = \frac{1}{(1-x)^2}.$$

Осталось заметить, что

$$S_n = \sum_{i=1}^{\infty} i \cdot q^i = q \cdot f'(q),$$

а следовательно $S_n = \frac{q}{(1-q)^2}$. □

Литература

[1] Бабенко М.А., Левин М.В. Введение в теорию алгоритмов и структур данных. — Москва : МЦНМО, 2016. — ISBN: 9785443923963.

[2] Алгоритмы. Построение и анализ [пер. с англ.] / Кормен Т., Лейзерсон Ч., Ривест Р., and Штайн К. — Вильямс, 2009. — ISBN: 9785845908575. — Access mode: https://books.google.ru/books?id=UVg1LPacgRcC.

[3] Шень А. Программирование: теоремы и задачи. — Москва : МЦНМО, 2017. — ISBN: 978-5-4439-0685-0. — Access mode: https://hal.archives-ouvertes.fr/hal-01480636/document.

[4] Sleator Daniel Dominic, Tarjan Robert Endre. Self-adjusting Binary Search Trees // J. ACM. — 1985. — July. — Vol. 32, no. 3. — P. 652–686. — Access mode: http://doi.acm.org/10.1145/3828.3835.

[5] Vuillemin Jean. A Unifying Look at Data Structures // Commun. ACM. — 1980. — Apr. — Vol. 23, no. 4. — P. 229–239. — Access mode: http://doi.acm.org/10.1145/358841.358852.

[6] Seidel Raimund, Aragon Cecilia R. Randomized search trees // Algorithmica. — 1996. — Vol. 16. — P. 464–497.