# Глава 2

# Анализ алгоритмов и структуры данных

Итак, условимся, что вычисления в рамках всех методов, рассматриваемых в данном курсе, производятся с применением электронных вычислительных машин (ЭВМ). Хотя некоторые методы были предложены задолго до изобретения ЭВМ, их использование предоставляет ряд неоспоримых преимуществ, прежде всего возможность безошибочно обрабатывать большие массивы информации.

Поскольку нас интересует не только принципиальная возможность решения тех или иных типовых задач, возникающих при обработке данных физических экспериментов, но и практические способы доведения расчётов до конкретного числа[1] или чисел (например, интересующих нас физических параметров) за разумное время, то мы неизбежно должны принять во внимание факт ограниченности вычислительной мощности (в том или ином смысле) доступных нам ЭВМ. Изначально может показаться, что такая низменная техническая проблема не достойна внимания исследователя-физика, однако, к сожалению, это не так. Пожалуй, даже начинающий физик-экспериментатор знает о том, что всё оборудование в его лаборатории имеет ту или иную стоимость[2], и соображения финансового плана часто играют не последнюю роль в планировании экспериментов. Аналогичная ситуация складывается и в области обработки данных: вычислительная мощность доступных нам ЭВМ ограничена либо в строгом смысле, т.е. где-то существует самый быстрый компьютер, когда-либо построенный человечеством, либо, чаще всего, в слабом смысле, когда оказывается, что стоимость требуемых ЭВМ составляет существенную часть доступного бюджета исследования. Иными словами, мы *вынуждены* проводить рас-

[1] Под «конкретным числом» мы понимаем именно такое представление, которое сгодилось бы в качестве ответа на вопрос, например, «сколько топлива заливать в ракету?»

[2] И зачастую не маленькую стоимость, так как производство самых точных и хороших приборов требует самых совершенных из доступных человечеству технологий.

чёты эффективно.[3] Данная глава целиком посвящена основным вопросам эффективности вычислений в её современном понимании.

Чтобы количественно охарактеризовать эффективность вычислений, понадобится ввести несколько определений. Под *вычислительной задачей* будем понимать утверждение о том, *что* требуется вычислить. Например, задача сортировки состоит в том, чтобы найти такую перестановку заданного конечного набора чисел $x_1, x_2, ..., x_N$, чтобы в этой перестановке каждое следующее число оказалось не меньше предыдущего. Или, например, задача квадратичной оптимизации состоит в том, что необходимо вычислить вектор $\mathbf{x}$ такой, что квадратичная форма

$$\frac{1}{2}(\mathbf{x}, H\mathbf{x}) + (\mathbf{x}, \mathbf{c}) \tag{2.1}$$

принимает наименьшее возможное значение для известной матрицы $H$ и известного вектора $\mathbf{c}$.[4] Или, например, задача может состоять в вычислении определителя известной матрицы $\det A$.

Понятие *алгоритм* интуитивно воспринимается как последовательность действий для некоторой вычислительной машины. Подразумевается, что добросовестно выполнив эту последовательность электронная вычислительная машина предоставит нам финальный результат работы алгоритма — решение той или иной вычислительной задачи. Иными словами, алгоритм описывает, *как* нужно вычислять для достижения результата.

В случае задачи сортировки мы могли бы поступить разными способами. Во-первых, перебирать все перестановки и проверять каждую на выполнение условия. Во-вторых, использовать один из множества известных алгоритмов сортировки (Кнут 2019c).

Для задачи квадратичной оптимизации можно было бы вычислить требуемый вектор, используя аналитическую формулу

$$\mathbf{x} = -H^{-1}\mathbf{c}, \tag{2.2}$$

либо вычислять вектор $\mathbf{x}$ итеративно (см., например, раздел 5.2.2). Для достижения цели в первом случае нужно выбрать алгоритм обращения матрицы и алгоритм умножения матрицы на вектор.

Для вычисления определителя матрицы можно было бы воспользоваться определением:

$$\det A \equiv \sum_{\alpha_1,\alpha_2,\ldots,\alpha_n} (-1)^{N(\alpha_1,\alpha_2,\ldots,\alpha_n)} \cdot a_{1\alpha_1} a_{2\alpha_2} \ldots a_{n\alpha_n}, \tag{2.3}$$

либо применить алгоритм поиска собственных значений $\lambda_i$ матрицы $A$ и затем их перемножить:

$$\det A = \lambda_1 \cdot \lambda_2 \cdot \ldots \cdot \lambda_N \tag{2.4}$$

[3] Кроме того, проводить расчёты эффективно считается хорошим тоном в среде компьютерных наук.

[4] В данной главе мы не рассматриваем вопрос *зачем* могло бы потребоваться вычислять минимизирующий такую квадратичную форму вектор, но подобным вопросам посвящено большинство остальных глав пособия.

Для решения одной и той же задачи можно предложить несколько алгоритмов, позволяющих получить эквивалентные ответы (либо в точности одинаковые, либо близкие друг к другу в пределах требуемой точности вычислений), а значит, предстоит разобраться, как сравнивать их эффективность между собой.

*Реализация алгоритма*[5] — конкретный исходный код программы на некотором языке программирования, написанный определённым автором. В качестве примеров реализаций алгоритмов можно назвать популярные пакеты для языка программирования Python, такие как `numpy`, `scipy`, `sklearn` и другие. Эти пакеты содержат готовые функции для операций с матрицами и векторами, численной оптимизации, методов машинного обучения и т.д.

Другим примером реализаций алгоритмов служит, например, код домашних заданий, выполняемых студентами в рамках курса «Обработка и анализ данных физического эксперимента» на Факультете физики ВШЭ. Реализации одного и того же алгоритма могут существенно отличаться по скорости работы, как за счёт используемого языка программирования или оптимизирующего компилятора, так и за счёт квалификации программиста.

Пожалуй, самый наивный способ сравнения эффективности вычислений состоит в том, чтобы замерить время работы реализаций различных алгоритмов на конкретной ЭВМ, используя секундомер[6]. Хотя такой способ и применяется на практике, его одного недостаточно, так как он обладает рядом недостатков. Во-первых, сравниваются реализации, а значит, нужно потрудиться реализовать несколько алгоритмов, запустить реализации и несколько раз получить *эквивалентные* ответы. Ради того, чтобы просто получить ответ, было бы достаточно иметь одну реализацию, и время вычислений для неё было бы заведомо меньше, чем время вычислений для всего набора.

Во-вторых, невозможно заранее предсказать, как изменится время работы при изменении количества входных данных алгоритма или каких-то иных его параметров, например, точности вычисления результата. Потребуется запустить реализацию с новым набором данных, и даже если мы дождёмся завершения работы, то весь смысл в измерении эффективности опять теряется. Это наводит на мысль, что необходим способ оценки эффективности алгоритма до его реализации и запуска. И такой способ, к счастью, есть, но сначала необходимо уяснить отличия алгоритма от любой другой последовательности действий, например, от кулинарного рецепта.

## 2.1 Алгоритм как математический объект

Исторически первым, вероятно, формализовал понятие алгоритма британский математик Алан Тюринг в 1930-х годах (Turing 1937). Мы же обратим-

[5] implementation

[6] Часто такую меру называют *wall-clock time*, т.е. время, которое показывают обычные часы (например, настенные).

ся к определению, закрепленному американским математиком Дональдом Кнутом в своём фундаментальном многотомнике «Искусство программирования» (Кнут 2019а).

Рассмотрим следующую четвёрку $(Q, I \in Q, \Omega \in Q, f : Q \to Q)$, здесь $Q$, $I$, $\Omega$ — некоторые абстрактные множества.

- $Q$ — множество всех состояний алгоритма. Попросту говоря, если мы рассмотрим набор всех используемых алгоритмом «переменных», то все возможные комбинации их значений и определяет нам множество $Q$.
- $I$ — начальное подмножество, это состояния, из которых алгоритм может стартовать. Оно представляет собой способ задания каких-то входных данных или параметров.
- $\Omega$ — подмножество конечных состояний.
- Функция $f$ переводит состояние алгоритма само в себя до тех пор, пока алгоритм не завершится.

Мы считаем, что $f(q) = q$ для $\forall q \in \Omega$, а последовательность состояний $q_{k+1} = f(q_k)$ в конечном итоге сходится к некоторому $q \in \Omega$. В этот момент говорят, что исполнение алгоритма завершено.

Может показаться непонятным, почему бы не задать функцию $f$ таким образом, чтобы все последовательности сходились за один шаг. Подразумевается, что функция $f$ может выполнять только элементарные преобразования. Понятно, что теория должна адекватно описывать реальный моделируемый объект, поэтому в качестве таких элементарных преобразований выбираются те операции, которые центральный процессор умеет выполнять аппаратно, например:

- загрузка значения из памяти в регистр (чтение памяти),
- сравнение значений в регистрах,
- арифметические операции над регистрами (сложение, вычитание, умножение, деление),
- выгрузка значения из регистра в память (запись памяти).

Мы исходим из того, что выполнение элементарной операции каждого вида всегда занимает одинаковое время. Без замены оборудования такие операции сами по себе нельзя улучшить, т.е. сделать ещё более быстрыми.

Итак, алгоритм — это математический объект, способный решить для нас некоторую задачу по преобразованию данных. Фактически, алгоритм — это ещё один способ задать некоторое отображение (функцию) $F : \mathbf{X} \to \mathbf{Y}$, наряду с другими способами, такими как:

- записать функцию в виде аналитического выражения (например, $F(x) \equiv x + x^2$);

- записать таблицу, если множества $\mathbf{X}$ и $\mathbf{Y}$ конечны;

- обозначить бесконечный ряд, интеграл с параметром, решение дифференциального (или алгебраического) уравнения новой функцией.

Пожалуй, наиболее показательно алгоритмы в роли математических функций проявят себя в разделе 9.2.

Все эти способы не эквивалентны с точки зрения практического удобства доведения вычислений до числа, к счастью, в ряде случаев можно строго доказать эквивалентность определений, данных разными способами. Это позволяет в различных ситуациях подменять одно определение другим, исходя из практических требований. Примером является функция Бесселя $J_\nu(x)$, которую можно определить и как ряд, и как интеграл с параметром, и как одно из решений дифференциального уравнения, но все эти определения приведут к одной и той же функции. Аналогично этому, можно формально доказать корректность алгоритма, т.е. строго убедиться, что алгоритм действительно решает заявленную задачу.

Теперь алгоритм, как и любой другой математический объект, может быть подвергнут анализу и исследованию свойств. Такими свойствами могут быть, например, скорость работы (длина последовательности $q_k$, т.е. фактически число затраченных элементарных операций), количество требуемой оперативной памяти или точность вычисления результата.

## 2.2 $O$-нотация

На практике интересным вопросом является зависимость числа затраченных элементарных операций или количество требуемой дополнительной памяти от размера входных данных или необходимой точности вычислений. Зависимость объема работы от размера входных данных часто называют *вычислительной сложностью*[7].

Чаще всего входные данные представлены в виде набора элементов одинаковой природы. Например, набор чисел для сортировки или матрица для обращения. В таком случае под размером входных данных естественно понимать число таких элементов: количество чисел для сортировки, либо число строк квадратной матрицы, соответственно.

Оказывается, для многих алгоритмов можно подсчитать число операций аналитически, однако здесь есть две сложности. Во-первых, число операций может зависеть не только он размера входных данных, но и от их значения. Например, если мы хотим отсортировать уже упорядоченный набор чисел с помощью алгоритма сортировки вставками (Кнут 2019c), то можно показать, что такая процедура займёт меньше времени, чем в случае сортировки неупорядоченного набора. Должны ли мы изучать скорость работы алгоритма для каждого допустимого набора входных параметров? Такая

[7] computational complexity

детализация была бы излишней, поэтому на практике изучают число операций в *среднем случае*[8], когда оно усредняется по ансамблю всех возможных входных параметров, либо число операций в *худшем случае*[9], когда оно изучается для такого набора входных данных, с которым алгоритм будет работать дольше всего.

Во-вторых, точное аналитическое выражение, если его можно получить, как правило, выглядит громоздко и содержит в себе различные неэлементарные функции (например, гармоническое число $H(N) \equiv \sum_{j=1}^{N} 1/j$). К счастью, оказывается, что достаточно рассматривать лишь асимптотическое поведение зависимости требуемых элементарных операций (либо количества требуемой дополнительной памяти) от размера входных данных при стремлении этого размера к бесконечности.

Напомним, что из курса математического анализа известны следующие определения:

- $f(x) = O(g(x))$ значит, что

$$\exists x_0, C > 0 : \quad \forall x \geq x_0 \quad |f(x)| \leq C|g(x)|$$

- $f(x) = \Theta(g(x))$ значит, что

$$\exists x_0, C_1 > 0, C_2 > 0 : \quad \forall x \geq x_0 \quad C_1|g(x)| \leq |f(x)| \leq C_2|g(x)|$$

Т.е. если $f(x) = O(x^2)$, то из этого следует, что, например, $f(x) = O(x^4)$. Ирония состоит в том, что в мире компьютерных наук по сложившейся традиции используют обозначение $O(g(x))$ в смысле $\Theta(g(x))$. Вероятно, это обусловлено тем, что символ $\Theta$ был менее распространён на первых компьютерах. Но на практике никакой путаницы из-за этого не возникает. Например, говорят, что алгоритм сортировки слияниями (Кнут 2019c) имеет вычислительную сложность $O(N \log N)$ в худшем случае, а «пузырьковый» алгоритм сортировки имеет вычислительную сложность $O(N^2)$ в худшем случае. При этом подразумевается, что для алгоритмов сортировки подсчитываются операции попарного сравнения чисел. Или, например, умножение двух квадратных матриц размера $N$ с помощью наивного алгоритма потребует $O(N^3)$ операций умножения (и в среднем и в худшем случае, поэтому уточнение опускается). Обратим внимание, что основание логарифма не ставится, так как не имеет смысла под знаком асимптотики.

Аналогично этому, с помощью $O$-нотации можно оценить количество требуемой дополнительной памяти. Например, говорят, что операция поиска максимального значения в наборе чисел требует $O(1)$ дополнительной памяти, т.е. размер дополнительной памяти не зависит от размера входных данных. Однако, операций сравнения всё равно необходимо $O(N)$.

Итак, $O$-нотация — удобный формализм для изучения вычислительной сложности алгоритмов. Во-первых, асимптотическая вычислительная

[8] average-case
[9] worst-cast

сложность имеет простой вид: на практике большинство алгоритмов имеют один из следующих вариантов вычислительной сложности — $O(1)$, $O(\log N)$, $O(N)$, $O(N \log N)$, $O(N^p)$.

Во-вторых, асимптотика является хорошим начальным приближением, чтобы судить об эффективности алгоритма. Считается, что чем медленнее асимптотика, тем эффективнее алгоритм. Поэтому, например, говорят, что сортировка слияниями лучше, чем «пузырьковый» алгоритм сортировки, так как функция $N \log N$ растёт медленнее, чем $N^2$ при $N \to \infty$.

В-третьих, асимптотическая вычислительная сложность, дополненная методом прямого измерения вычислительного времени, позволяет хорошо экстраполировать, т.е. отвечать на вопросы вида «Что случится, когда данных станет в два раза больше?».

Впрочем, стоит помнить, что, согласно формальному определению $N \to \infty$, при этом какой именно $N$ можно считать достаточно большим, чтобы стать бесконечностью — вопрос сложный. Иначе говоря, $O$-нотация не учитывает мультипликативную константу. Таким образом лучшая асимптотика не всегда означает лучший алгоритм, классическим примером является задача перемножения двух квадратных матриц размера $N$.

Рассмотрим три следующих подхода. Во-первых, наивный подход, состоящий в прямом использовании определения матричного умножения

$$c_{ik} = \sum_j a_{ij} b_{jk}, \tag{2.5}$$

легко проверить, что в этом случае потребуется $O(N^3)$ операций умножения.

Во-вторых, алгоритм Штрассена (Strassen 1969), в котором задействуются $O(N^{\log_2 7}) = O(N^{2.8})$ операций умножения. Казалось бы, что такой алгоритм лучше, но на практике он превосходит наивный подход только при $N$ больше сотен или тысяч. Тут надо оговориться, что число это очень примерное и должно быть замерено для каждой конкретной реализации и модели процессора.

В-третьих, алгоритм Копперсмита-Винограда (Coppersmith и Winograd 1990), в котором требуется $O(N^{2.375})$ операций умножения, но практический смысл такого алгоритма сомнителен: пороговое значение $N$, при котором он опережает алгоритм Штрассена, оказывается очень велико. Судя по всему, идёт о матрицах с числом строк больше миллиона, а возможно, существенно больше.

Вообще говоря, существуют и обратные примеры. Так, например, для

решения задачи линейного программирования [10]:

$$\max \sum_{j=1}^{N} p_j x_j, \tag{2.6}$$

$$\sum_{j=1}^{N} a_{ij} x_j \le b_i, \tag{2.7}$$

где $x_j$ — неизвестные действительные числа, а $a_{ij}$, $b_i$, $p_j$ суть известные действительные числа, применяется симплекс-метод (предложенный Джорджем Данцигом в 1940-х), который имеет экспоненциальную сложность в худшем случае. Пример худшего случая известен под названием куба Клее-Минти. Однако, в среднем этот алгоритм требует линейного числа шагов и является одним из признанных эффективных способов решения указанной задачи.

И теперь про примеры из начала главы стало всё ясно. Для решения задачи сортировки следует предпочесть специализированный алгоритм, вычислительная сложность которого $O(N \log N)$, наивному перебору всех перестановок, сложность которого $O(N!)$. Оба предложенных метода решения задачи квадратичной оптимизации имеют сложность $O(N^3)$. Для вычисления определителя матрицы нужно воспользоваться разложением на собственные значения, со сложностью $O(N^3)$, а не наивным суммированием по всем перестановкам, количество которых опять $O(N!)$.

## 2.3 Теория алгоритмов

Мы видели, что для некоторых задач существует несколько алгоритмов решения, а исследование асимптотической вычислительной сложности даёт оценку их эффективности. Однако, может возникнуть вопрос, можно ли для некоторой задачи придумать новый алгоритм, который будет лучше, чем известные алгоритмы в смысле асимптотической вычислительной сложности? Оказывается, для некоторых задач можно сформулировать ответ на этот вопрос в виде соответствующих теорем.

Так, например, можно доказать теорему, что любой алгоритм сортировки, основанный на попарных сравнениях, требует не менее чем $\lceil \log N! \rceil$ операций сравнения. Иначе говоря, вычислительная сложность указанных алгоритмов сортировки — $\Omega(N \log N)$ (Кнут 2019c).

Запись $f(x) = \Omega(g(x))$ значит, что

$$\exists x_0, C > 0 : \quad \forall x \ge x_0 \quad |f(x)| \ge C|g(x)|.$$

Отметим, что по сложившейся традиции вместо обозначения $\Omega(g(x))$ часто используется обозначение $O(g(x))$, но, как и прежде, путаницы не возникает.

[10] Напомним, что термин «задача математического программирования» обозначает задачу оптимизации и в данном случае не имеет отношения к программированию ЭВМ.

Заметим, что теорема ничего не говорит о специализированных алгоритмах сортировки. Так, например, для целых положительных чисел существует алгоритм поразрядной сортировки, асимптотическая сложность которого $O(N\omega)$, где $\omega$ — размер числа в битах (Кнут 2019c).

Напротив, наилучшая сложность матричного умножения точно не известна, но предполагается, что она составляет $\Omega(N^{2+\epsilon})$, где $0 \leq \epsilon < 0.372$, причём верхняя граница уменьшается по мере изобретения новых алгоритмов.

Кроме того важно, что при сужении задачи, т.е. рассмотрения более частного случая, как правило, становится возможно предложить более эффективный алгоритм с точки зрения асимптотической вычислительной сложности. Например, рассмотрим обращение квадратной матрицы размером $N$. Хотя теоретическая сложность обращения произвольной матрицы такая же, как у умножения квадратных матриц (см., например, Кормен и др. 2019), сложность популярных алгоритмов обращения составляет $O(N^3)$. Но если мы рассматриваем верхнюю треугольную матрицу или нижнюю треугольную матрицу, то можно предложить алгоритм, имеющий вычислительную сложность $O(N^2)$. Если рассматривается трёхдиагональная матрица, то вычислительная сложность такой задачи уже $\Omega(N)$.

### 2.3.1 NP-задачи

Теперь мы поняли, что для некоторых задач мы можем теоретически оценить вычислительную сложность самого эффективного алгоритма. Иначе говоря, можно доказать, что получить более эффективный алгоритм без переформулировки задачи невозможно. Возникает вопрос, а бывают ли такие задачи, сложность которых настолько велика, что ставит под сомнение практическую ценность применения ЭВМ для их решения? Оказывается, что существует целый класс задач, которые невозможно без переформулировки эффективно решать на существующих вычислительных системах.

Примером является следующая задача дискретной оптимизации, известная как «задача о рюкзаке»:

$$\max \sum_{j=1}^{N} p_j x_j, \tag{2.8}$$

$$\sum_{j=1}^{N} \omega_j x_j \leq C, \tag{2.9}$$

где неизвестные переменные $x_j \in \{0, 1\}$, а параметры $\omega_j$, $C$ и $p_j$ суть известные действительные числа.

По легенде, злоумышленник проникает в музей и видит там $N$ экспонатов, каждый стоимостью $p_j$ и весом $\omega_j$. Задача грабителя — унести ценностей на наибольшую сумму, но грузоподъемность его рюкзака составляет $C$ и при её превышении рюкзак порвётся, поэтому для каждого экспоната нужно принять решение $x_j$ — оставить ли экспонат в музее или положить в

рюкзак и унести. Основная сложность этой задачи заключается в том, что каждый экспонат нужно либо взять целиком, либо целиком оставить.

Другим примером таких задач является задача целочисленного программирования:

$$\max \sum_{j=1}^{N} c_j x_j, \tag{2.10}$$

$$\sum_{j=1}^{N} a_{ij} x_j \leq b_i, \tag{2.11}$$

где $x_i \in \mathbf{Z}$, а $a_{ij}$, $b_i$ и $c_j$ суть известные целые числа.

Рассмотрим на примере задачи о рюкзаке связанную с проблемой терминологию. Обычно, помимо основной задачи оптимизации, рассматривают две сопряжённые с ней задачи. Во-первых, *задача разрешимости*: пусть задано $t$, существует ли набор $x_j$ такой, что $\sum_{j=1}^{N} p_j x_j \geq t$, при условии (2.9)? Понятно, что если для задачи разрешимости существует алгоритм решения, то ответ к изначальной задаче может быть найден с помощью деления отрезка пополам. Если существует алгоритм с асимптотической вычислительной сложностью $O(N^p)$, то говорят, что задача принадлежит классу $P$.

Во-вторых, *задача верификации*: пусть задано $t$ и набор $x_j$, нужно проверить, являются ли они решением задачи $\sum_{j=1}^{N} p_j x_j \geq t$, при условии (2.9). Если существует алгоритм, позволяющий выполнить проверку за $O(N^p)$, то говорят, что задача принадлежит классу $NP$. Для задачи о рюкзаке такой алгоритм, очевидно, существует.

$NP$-полными ($NP$-complete) называются такие задачи, к которым может быть сведена любая $NP$ задача за полиномиальное время. Класс $P$ содержится внутри $NP$, но совпадают ли они полностью — непонятно. Этот вопрос является одной из нескольких «математических задач тысячелетия» по версии института Клея.[11] Если бы классы полностью совпали ($P = NP$), тогда наличие полиномиального алгоритма для задачи верификации гарантировало бы нам, что где-то существует полиномиальный алгоритм для решения задачи разрешимости. А вот если классы не совпадают ($P \neq NP$), это значит, что существуют задачи, для которых не может существовать полиномиального алгоритма для задачи разрешимости.

На практике $NP$-полные задачи переформулируют в вид, допускающий существование алгоритмов с полиномиальной вычислительной сложностью. Например, вместо решения «задачи о рюкзаке» в исходном виде, можно найти такой набор $x_j \in \{0, 1\}$, что $\sum_{j=1}^{N} p_j x_j \geq \alpha \max \sum_{j=1}^{N} p_j x_j$, при условии (2.9). Т.е., например, при $\alpha = 0.95$ будет найдено решение, которое хуже оптимального не более чем на 5%.

[11] `https://www.claymath.org/millennium-problems`

## 2.4 Структуры данных

Итак, теперь нам известно, как измеряется эффективность различных алгоритмов и сложность вычислительных задач. В первом случае это позволяет нам сравнивать алгоритмы и выбирать наиболее подходящий, а во втором случае иметь ориентир, насколько выбранный алгоритм можно улучшить.

Пока в стороне оставался вопрос, откуда в принципе берутся новые алгоритмы. Понятно, что в конечном счёте это творческий процесс, однако, хорошо было бы иметь набор удачных универсальных заготовок.

При обработке данных часто бывает так, что сложности возникают уже на этапе их подготовки, до применения каких-то математизированных методов анализа. Дело может быть в том, что в рамках экспериментов данные записываются в неудобном порядке или для анализа требуется свести воедино измерения, выполненные в рамках разных экспериментов.

Например, рассмотрим два астрономических набора данных (или «каталога», как говорят астрономы). Набор данных космического эксперимента GAIA (Gaia Collaboration и др. 2023) содержит примерно $N_1 = 1.5 \cdot 10^9$ уникальных записей о точных положениях и расстояниях до звёзд Галактики. Набор данных наземного обзора ZTF (Bellm и др. 2018) содержит примерно $N_2 = 2.7 \cdot 10^9$ уникальных фотометрических кривых блеска (зависимостей яркости звезды от времени), но не содержит никакой информации о расстоянии, так как в рамках этого эксперимента его невозможно было определить. Сразу же возникает желание объединить эти два набора данных, чтобы в одном наборе анализировать как расстояние до звезды, так и её яркость. Трудность, с которой мы сталкиваемся, состоит в том, что положения звезд на небе измеряются с некоторой погрешностью как в первом, так и во втором случае, а это значит, что нам нужно для каждой записи из первого набора данных найти *ближайшего соседа* из второго набора данных. Строгое сравнение координат, записанных в виде пары чисел, здесь не сработает как раз из-за погрешности измерений.

Оценим, можно ли реализовать это объединение наивным способом: для каждой из записей первого набора данных рассчитаем расстояния до каждой из записей второго набора данных и попутно найдём минимальное, которое и будет указывать нам на запись, образующую нужную нам связь. Всего таких пар расстояний будет порядка $N_1 N_2 \approx 10^{18}$. Предположим, что координаты звёзд в каждом наборе данных представлены в виде пары чисел с плавающей точкой двойной точности, т.е. суммарно 16 байт. Координаты из двух наборов данных займут примерно $16\,(N_1 + N_2) = 16{\cdot}(1.5{+}2.7){\cdot}10^9 \approx$ 60GB памяти. В настоящее время компьютеры с 60GB и более оперативной памяти совсем не редки, но для интереса рассмотрим два случая: когда данные хранятся на хорошем быстром твердотельном жестком диске и когда данные предварительно загружены в более быструю оперативную память. Для наивного подхода, состоящего в подсчёте всех попарных расстояний, оценим суммарное количество информации, которое необходимо прочитать и пропустить через центральный процессор: результат будет $\approx 16 N_1 N_2 = 16 \cdot 1.5 \cdot 2.7 \cdot 10^{18} = 60 \cdot 10^9$GB.

Пусть чтение с очень хорошего твердотельного жесткого диска занимает около $2\,\mathrm{GB\,s^{-1}}$, тогда время, затраченное на чтение данных, займёт примерно 1000 лет. Если данные уместились в оперативную память, то можно обеспечить на порядок большую скорость, возьмём в качестве оценки $40\,\mathrm{GB\,s^{-1}}$ и получим 50 лет. Отметим, что обе оценки оптимистичные и не учитывают дополнительные накладные расходы в виде задержек с доступом к памяти. Понятно, что эти оценки скорости работы говорят о том, что наивный подход оказывается непрактичным и нужно искать альтернативный.

Из приведённого примера видно, что основная проблема возникает из-за того, что требуется $O(N_1 N_2)$ операций чтения данных. Оказывается, что это количество можно снизить, расположив данные в памяти удобным способом.

*Структуры данных* — способы расположения информации в адресном пространстве оперативной памяти или жёсткого диска, которые помогают нам составлять более эффективные алгоритмы. Примерами основных популярных структур данных являются, например: вектор (массив), список, деревья, хеш-таблицы, которые будут по очереди рассмотрены далее. Речь идёт о наборах (коллекциях) элементов одинаковой природы. В примере выше мы работали с двумя наборами, элементами первого были пара координат и расстояние, а элементами второго — пара координат и таблица зависимости яркости от времени на некоторой сетке.

Перед тем как познакомиться с какой-нибудь из структур данных, следует вспомнить, что из себя представляет операция чтения (и записи) памяти. На самом деле, с точки зрения стороннего наблюдателя, микросхема памяти работает предельно просто [12]: она меняет одно целое число, которое принято называть *адресом*, на другое число, подобно тому как в гардеробе меняют номерок на одежду. Не вдаваясь в детали, представим, что оба числа закодированы некоторым числом бит, например, 32. Тогда адрес подаётся на входные контакты микросхемы так, что каждому из битов соответствует один физический контакт микросхемы. Второе число — это и есть хранимые данные: как только адрес подан на вход микросхемы, на каждом из выходных контактов возникает логический сигнал, соответствующий значению бита. Будем считать, что скорость работы микросхемы не зависит от значения адреса, тогда говорят, что сложность произвольного доступа к памяти составляет $O(1)$ [13]. Запись производится аналогичным образом, только теперь все контакты играют роль входа.

Волшебство заключается в том, что, во-первых, адреса тоже можно хранить в этой же микросхеме в роли данных, а во-вторых, процессор умеет выполнять арифметические операции над адресами. С формальной точ-

[12] На самом-то деле, доступ к памяти в современном устройстве с многоядерным процессором работает предельно сложно. Стоит лишь упомянуть о многоуровневом кэше процессора, многоканальных контроллерах памяти и прямом доступе к памяти со стороны периферийного оборудования. К счастью, для понимания структур данных всё это не важно, и мы удовлетворимся простой моделью происходящего.

[13] Примером устройства, где время доступа к данным зависит от адреса, является накопитель с использованием магнитной ленты, но таковые ныне используются лишь для холодного хранения данных.

ки зрения, мы имеем дело с некоторым непрерывным *адресным пространством*, примерами которого являются оперативная память, жесткий диск или один файл в рамках диска. Структуры данных описывают правила размещения коллекции однотипных данных в таком адресном пространстве, и, следовательно, задают алгоритмы элементарных операций над этими коллекциями. Такими операциями, например, являются:

- поиск элемента по значению,
- добавление нового элемента в коллекцию,
- удаление старого элемента из коллекции,
- замена значения элемента.

Понятно, что для каждой структуры данных алгоритмы операций коллекции будут разными, значит знание сильных и слабых сторон каждой из структур данных позволяет эффективно комбинировать их в рамках более общих алгоритмов и программ. Например, если бы мы смогли разместить данные из каталога ZTF в такую структуру данных, которая предусматривала бы операцию поиска ближайшего соседа произвольной точки со сложностью $O(\log N_2)$, то мы смогли бы решить нашу исходную задачу за $O(N_1 \log N_2)$ чтений данных, т.е. потратили бы на неё несколько минут.

Рассмотрим основные структуры данных.

### 2.4.1 Массив

*Массив* (известен также под названием *вектор*, а в языке программирования Python — список или `numpy.array` из пакета numpy) — одна из самых простых структур данных, поэтому знакомство со структурами данных разумно начать с него. Пусть мы хотим разместить некоторый набор $X$ из $N$ объектов, требующих $s$ байт памяти для каждого. Например, рассмотрим пары чисел с плавающей точкой из предыдущего примера. Разместим их в памяти последовательно, без пропусков, в том порядке, в котором они нам предоставлены.

Нам достаточно запомнить адрес нулевого элемента (который мы назовём адресом массива) $a_0$, чтобы получить доступ к любому из элементов, зная индекс $i$ этого элемента. Адрес $a_i$ элемента номер $i$ можно легко вычислить следующим образом:

$$a_i = a_0 + i \cdot s. \tag{2.12}$$

Напомним, что произвольный доступ к любому адресу памяти имеет константную сложность, значит *чтение или запись $i$-го элемента массива* имеет сложность $O(1)$. Допустим, мы хотим проверить, содержится ли элемент со значением $x'$ в нашем массиве. В этом случае лучшим вариантом оказывается последовательное считывания всех элементов $x \in X$ и сопоставление с образцом $x'$, в худшем случае, когда элемент $x'$ отсутствует в массиве, это

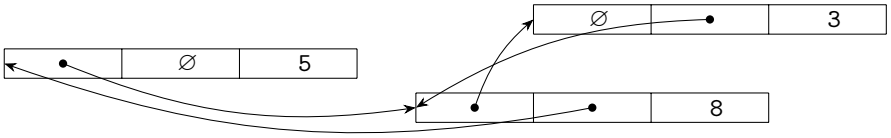


Рис. 2.1: Схематичное изображение двусвязного списка, хранящего последовательность $\{5, 8, 3\}$. Прямоугольники обозначают области адресного пространства. Стрелки показывают, что в заданном месте записан адрес соответствующего блока. Символ $\varnothing$ обозначает пустой адрес.

займёт $O(N)$ операций чтения и сравнения. Тогда говорят, что *поиск по значению в массиве* имеет сложность $O(N)$. Операции вставки или удаления элемента из позиции $i$ тоже имеют сложность $O(N)$, это связано с тем, что при добавлении нужно освободить $i$-ую позицию (т.е. переписать все значения на соседнее положение вправо), а при удалении нужно переместить на освободившуюся следующий элемент и т.д. Отметим, что добавление нового элемента перед нулевым элементом невозможно по соглашению, а добавление элемента после последнего тоже имеет сложность $O(N)$, но это уже связано с техническими ограничениями: адресное пространство (память) после последнего элемента может быть занято хранением не имеющих отношения к массиву данных. Это значит, что для добавления нового элемента в конец массива потребуется переместить весь массив в новое, более просторное место.

Заметим, что если мы отказываемся от необходимости хранить элементы в некотором заданном порядке, мы могли бы отсортировать такой массив, и тогда *поиск по значению в отсортированном массиве* можно осуществлять за $O(\log N)$ операций, используя метод *двоичного поиска*[14].

### 2.4.2 Список

*Список* (в языке программирования Python — `collections.deque`) может быть односвязным или двусвязным. Общий принцип состоит в том, что для каждого из $N$ объектов в адресном пространстве выбирается своё индивидуальное место, достаточное для хранения этого объекта и адреса, указывающего, где хранится следующий по порядку элемент списка (такой список называется *односвязным*), либо пары адресов, указывающих на предыдущий и следующий элементы (такой список называется *двусвязным*). Адрес $a_0$, указывающий на место хранения нулевого элемента, и есть адрес списка. Схематично список показан на рис. 2.1.

Отметим различия между списком и массивом. Во-первых, списку требуется больше памяти для хранения тех же самых элементов. Во-вторых, чтение или запись $i$-го элемента потребуют $i$ последовательных чтений, т.е. *чтение или запись $i$-го элемента списка* имеет сложность $O(N)$. Чтобы

[14] binary search

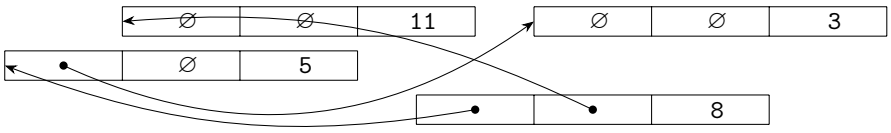


Рис. 2.2: Схематичное изображение двоичного дерева, корню которого приписано число 8. Прямоугольники обозначают области адресного пространства. Стрелки показывают, что в заданном месте записан адрес соответствующего блока. Символ ∅ обозначает пустой адрес.

узнать, где лежит элемент номер $i$, мы должны прочитать элемент номер $i-1$, где записан нужный нам адрес, и т.д. Заметим, что и *поиск по значению в списке* имеет сложность $O(N)$.

Однако, тут скрывается важное преимущество двусвязного списка: если мы нашли интересный нам элемент $x'$, который располагается по адресу $a'$, то удаление этого элемента, а так же вставка нового элемента до или после него, займёт $O(1)$ операций, так как потребует только исправления адресов в смежных элементах. Это же утверждение верно для вставки и удаления в конец и начало списка. Для односвязного списка сложность $O(1)$ имеют только некоторые из этих операций.

Списки используются, например, для представления в памяти разреженных матриц и векторов, т.е. таких, большинство элементов которых равно 0, или для организации очередей, когда необходимо часто удалять элементы из начала списка и добавлять новые элементы в конец.

### 2.4.3 Деревья поиска

*Дерево* — это математическая абстракция, частный случай графа. Деревом называется односвязный ациклический граф, одна из *вершин* (узлов) которого назначена *корнем* (зафиксирована). Как только мы фиксируем корень, некоторые вершины дерева становятся *листьями*, т.е. тупиковыми вершинами при обходе дерева, начиная от корня. Для каждого из листьев можно подсчитать число вершин на пути от корня до этого листа, назовём такое число расстоянием или *глубиной (высотой) листа*. Максимальную глубину листа среди всех листьев назовём *глубиной (высотой) дерева*.

Особый практический интерес в данном разделе для нас будут представлять *двоичные (бинарные) деревья*: у такого дерева каждый узел имеет не более двух потомков. Практическая польза связана с тем, что двоичные деревья достаточно просто представлять в линейном адресном пространстве: по аналогии со списком каждый узел будет соответствовать области памяти, где хранятся данные, приписанные к этому узлу, и дополнительно два адреса, указывающие на поддеревья (их для удобства называют левым и правым поддеревьями). Адресом дерева будем считать адрес его корня. Таким образом, зная адрес дерева, можно за конечное число операций получить доступ к данным, приписанным к любому из его узлов. Схематично

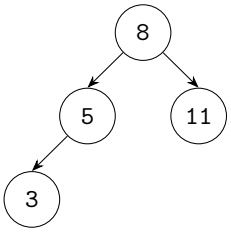


Рис. 2.3: Схематичное изображение двоичного дерева поиска. Возможное расположение элементов в адресном пространстве показано на рис. 2.2.

двоичное дерево показано на рис. 2.2.

Следует сразу сказать, что деревья оказались настолько эффективным инструментом, что применяются не только для организации коллекций, но и во многих алгоритмах машинного обучения (см., например, раздел 9.2). Одним из часто используемых деревьев — *двоичное дерево поиска*. Двоичное дерево поиска устроено следующий образом:

- каждый элемент $x$ коллекции $X$ приписывается своему узлу дерева;
- все элементы $x$ коллекции $X$ попарно сравнимы между собой, иначе говоря, для них определена операция «меньше» ($<$);
- для каждого промежуточного узла хранимое (приписанное) ему значение $x$ таково, что $x_L < x$, где $x_L$ — любое значение из левого поддерева (приписанное любому из узлов левого поддерева), и, одновременно, $x < x_R$, где $x_R$ — любое значение из правого поддерева.

Пример двоичного дерева поиска представлен на рис. 2.3.

Конечно, немного расстраивает необходимость уметь сравнивать элементы. Если для чисел операция сравнения вводится естественным образом, а для строк её очевидно можно определить, чтобы сохранялся лексографический порядок (т.е. алфавитный), то для векторов или комплексных чисел ситуация непонятная. Тем не менее, рассмотрим, какие операции и как можно выполнять над двоичным деревом поиска.

Из последнего свойства двоичного дерева поиска видно, что, в отличие от массива и списка, теперь каждый элемент не имеет своей произвольной позиции, а вернее, его положение полностью зависит от значения этого элемента. В массиве и списке мы могли бы произвольно менять элементы местами, т.е. сохранять нашу коллекцию в произвольном порядке. В дереве поиска любая коллекция хранится в определённом порядке, навязанном структурой данных. Это значит, что операция чтения или записи $i$-го элемента не существует для дерева поиска, так как не имеет смысла.

Зато это свойство позволяет сформулировать следующий алгоритм *поиска элемента $x'$ по значению в двоичном дереве поиска*. Начнём с корня, т.е. мы знаем значение, приписанное корню $x$, и адреса узлов корней левого и правого поддеревьев.

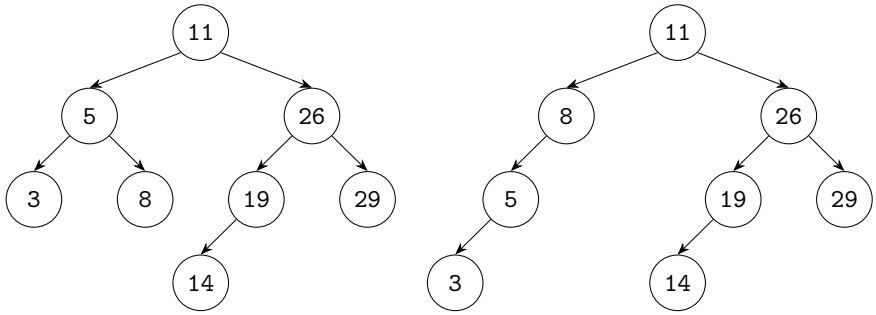


Рис. 2.4: Пример сбалансированного (слева) и несбалансированного (справа) двоичных деревьев поиска, хранящих одинаковую коллекцию.

- Если вдруг оказалось, что $x' = x$, то элемент найден.

- Если $x' < x$, тогда, согласно свойству дерева поиска, узла со значением $x'$ точно нет в правом поддереве. Если левое поддерево не пусто, то элемента в нем тоже может не быть, что мы проверим, применив этот же алгоритм поиска элемента $x'$ к левому поддереву, адрес корня которого нам уже известен.

- Если $x < x'$, тогда, согласно свойству дерева поиска, узла со значением $x'$ точно нет в левом поддереве, а вот в правом поддереве имеет смысл его поискать, применив этот же алгоритм.

Итак, либо узел, соответствующий элементу $x'$, будет обнаружен на каком-то этапе, либо будет обнаружен лист дерева. В последнем случае можно строго говорить об отсутствии элемента $x'$ в коллекции, так как по построению дерева поиска такой элемент обязан был бы оказаться в левом или правом поддереве листа. Понятно, что сложность такого алгоритма пропорциональна числу сравнений, т.е. глубине финального узла. В худшем случае, когда элемент отсутствует, сложность алгоритма поиска составит $O(h)$, где $h$ глубина дерева.

*Вставка нового элемента* осуществляется аналогичным образом: выполняется неудачный поиск и дописывание нового листа в качестве потомка финального листа. Сложность такого алгоритма опять пропорциональна числу сравнений, так как сама вставка нового листа потребует лишь конечного числа записей адресов в соответствующих узлах. Операция *удаления элемента* чуть более замысловата, но тоже начинается с поиска по значению элемента, который требуется удалить. Некоторую сложность представляет случай, когда у удаляемого узла два потомка, но остальные случаи тривиальны, и операция удаления опять имеет сложность $O(h)$.

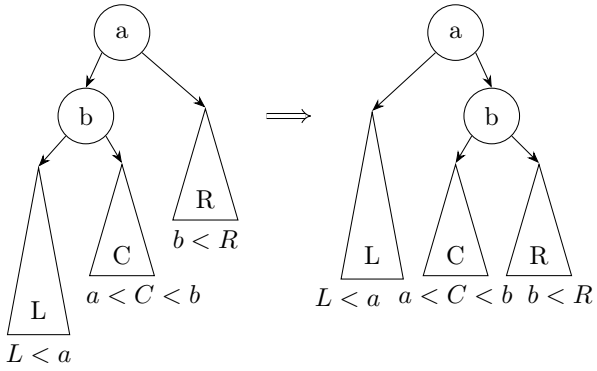


Рис. 2.5: Пример операции поворота «малое правое вращение». Треугольники обозначают поддеревья.

### Сбалансированные деревья

Было бы здорово, если бы оказалось, что глубина дерева $h$ пропорциональна $\log N$, где $N$ — число узлов дерева, и следовательно, элементов коллекции. В общем случае, так, конечно, сделать нельзя, однако, для *сбалансированного дерева* такое требование выполняется, а значит, позволяет нам осуществлять операцию поиска элемента за $O(\log N)$. Сбалансированным двоичным деревом называется такое двоичное дерево, у которого глубины левого и правого поддеревьев для каждого узла различаются не более чем на единицу. [15] На рис. 2.4 приведён пример сбалансированного и несбалансированного двоичных деревьев поиска. К сожалению, приведённые выше примеры операций вставки и удаления очевидным образом портят балансировку дерева. Поэтому приходится либо отказаться от модификации дерева, построенного сбалансированным один раз, либо изменить эти операции таким образом, чтобы гарантировать сбалансированность дерева.

Одним из примеров последнего подхода являются *АВЛ-деревья* (Адельсон-Вельский и Ландис 1962), названные так в честь авторов — советских математиков Адельсон-Вельского и Ландиса, которые предложили способы вставки и удаления элемента, не портящие балансировку дерева. Основные идеи в предложенном методе следующие. Во-первых, предлагается в каждый узел дополнительно записать *показатель сбалансированности* — разницу между высотами левого и правого поддеревьев. Согласно определению, этот показатель может принимать всего три различных значения $\{-1, 0, +1\}$. Во-вторых, после вставки или удаления элемента выполняется

[15] Иногда различают *сбалансированное дерево* и *идеально сбалансированное дерево*. Последнее требование более жёсткое и является частным случаем первого, оно состоит в том, что все уровни кроме последнего полностью заполнены, а значит глубины всех листьев различаются не более чем на 1. Однако, наша цель — получить асимптотическую сложность поиска в $O(\log N)$, и, как станет видно далее, для этой цели даже сбалансированность — избыточное требование.

обратный проход (от листа к корню, по ранее запомненному маршруту). В момент этого прохода для каждого узла удаётся последовательно определить как изменился его показатель сбалансированности (оказывается, что эту операцию можно выполнить за константное время), и, если показатель баланса стал $-2$ или $+2$, то выполнить одну из четырёх описанных операций балансировки, называемых вращениями, и требующих константного времени. Операции балансировки состоят в переподчинении поддеревьев другим узлам, пример одной из операций приведен на рис. 2.5. Таким образом, операции вставки и удаления в АВЛ-дереве, включая последующую перебалансировку дерева, требуют $O(h)$ операций.

Оказывается, что можно ослабить требования балансировки, сохранив асимптотическую сложность в $O(\log N)$ операций для удаления, вставки и поиска. Примерном являются *красно-черные деревья*, название которым дали американские исследователи Гибас и Седжвик (Guibas и Sedgewick 1978). Для красно-черных деревьев гарантируется, что минимальная высота листа и максимальная высота листа различаются не более чем в два раза, и удаётся показать, что асимптотическая сложность основных операций $O(\log N)$. Можно сказать, что красно-черные деревья аналогичны АВЛ-деревьям. Во-первых, предлагается хранить в каждом узле цвет (один бит информации — «красное» или «черное»). Цвета задаются не произвольно, красно-черное дерево подчиняется набору инвариантных требований, например, если узел красный, то оба потомка чёрные, и т.п. Во-вторых, при обратном проходе удаётся сформулировать набор правил перекрашивания (сравните с подсчётом показателя сбалансированности), и правил перебалансировки (которых в этот раз оказывается больше четырёх). Суть в том, что как и в случае с АВЛ-деревьями на каждый узел по пути обратного прохода потребуется константное время, а перечислять здесь полный набор условий и правил оказалось бы занудно, как и в случае с АВЛ-деревом.

Интересно, что для *случайного двоичного дерева поиска* средняя глубина (в отличии от максимальной глубины, которая рассматривалась в двух предыдущих случаях) тоже оказывается пропорциональна $\log N$. Случайным деревом называется дерево, в котором корень выбирается случайным образом, затем коллекция элементов разделяется на элементы, которые оказались меньше корня (левое поддерево) и больше корня (правое поддерево). Для левого и правого поддерева операция повторяется.

Деревья поиска наиболее часто используются в качестве *ассоциативного массива*. В таких случаях данные $x$ представляются в виде пары $(k, v)$, состоящей из *ключа* $k$ и *значения* $v$. Ключ и значение неравнозначны. Для ключей $k$ должна быть определена операция сравнения и они используются для построения дерева, а значения $v$ просто дополнительно приписываются узлам. Результатом является возможность эффективно отыскать $v'$, зная ключ $k'$, но не наоборот — зная $v'$, отыскать соответствующий ему ключ $k'$ можно только полным перебором дерева. Примером может служить телефонная книга. Когда на телефон поступает звонок с номера $k'$, мы заинтересованы в том, чтобы быстро найти в телефонной книге имя этого абонента $v'$ (ну или обнаружить, что номер неизвестный) и в таком виде

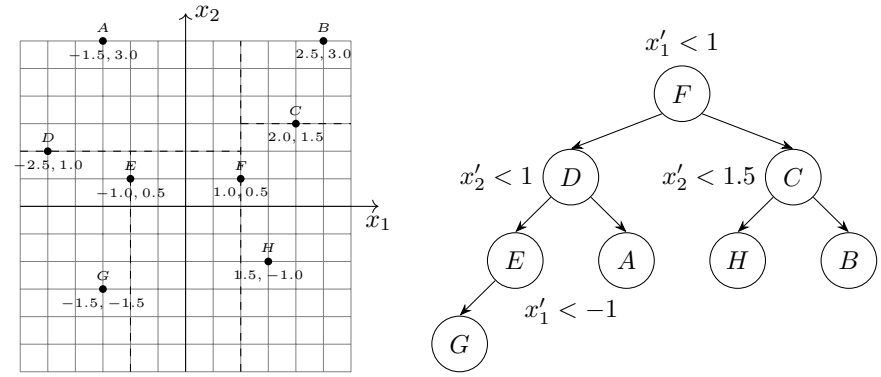


Рис. 2.6: Схематичное изображение $k$-мерного дерева для $\mathbf{R}^2$ (слева) и получаемого разбиения пространства (справа).

показать пользователю сообщение о поступающем вызове.

Кроме того, из-за упорядоченной структуры, деревья позволяют эффективно находить все элементы удовлетворяющие условиям $a \leq x$, $x \leq b$, $x \in [a, b]$, а так же выполнять поиск ближайшего снизу или ближайшего сверху к заданным элементам. Но для эффективного выполнения таких операций дерево должно быть дополнительно *прошито*, т.е. каждый лист должен хранить адрес узла, содержащего данных следующие за этим узлом. Если вспомнить пример про астрономические каталоги из начала этого раздела, то возможность эффективного поиска ближайших соседей начинает смотреться особенно ценно. Жаль только, что координаты представляют собой двумерный вектор, которые непонятно как упорядочивать, чтобы представить в виде дерева поиска.

### $k$-мерные деревья

Элементы многомерного пространства $\mathbf{R}^k$, $k > 1$ представляют особый интерес, но для хранения таких данных используются специализированные виды деревьев: *$k$-мерные деревъя*[16], деревья квадрантов, $R$-деревья и т.д.

Рассмотрим в качестве примера принцип организации многомерных данных в $k$-мерных деревьях (Bentley 1975). Этот подход пригодится нам в будущем в разделах про машинное обучение. Во-первых, $k$-мерное дерево — двоичное. Во-вторых, для заданной размерности пространства $k$ все существующие уровни двоичного дерева пронумерованы циклически от 1 до $k$, т.е. если в дереве больше $k$ уровней, то нумерация начинается сначала с 1. На практике номер уровня не нужно дополнительно хранить, так как он всегда может быть найден как побочный результат во время алгоритма поиска. В-третьих, на каждом уровне сравнивается только компонента векторов, соответствующая циклическому номеру уровня. Например, в случае

[16] k-d trees

плоскости в левом поддереве окажутся точки, которые находятся левее корневой точки (неважно, выше или ниже), а в правом поддереве — правее. На следующем уровне для каждого из двух поддеревьев точки будут делится по признаку выше или ниже точки корня поддерева. Пример $k$-мерного дерева приведён на рис. 2.6. В многомерном пространстве, $k$-мерные деревья позволяют эффективно выполнять операцию поиска элемента, и, кроме того, операцию поиска ближайших соседей ($O(\log N)$ в среднем).

Стоит сказать несколько слов о балансировке $k$-мерных деревьев. Из заданного набора точек достаточно просто построить сбалансированное дерево: для этого на каждом этапе достаточно выбирать медиану среди значений рассматриваемых координат векторов. Однако, в отличии от одномерного случая, операции удаления и записи представляют собой нетривиальную проблему, поэтому всегда хорошо, если модифицировать коллекцию не нужно. Но стоит сказать, что существуют и самобалансирующиеся варианты $k$-мерных деревьев, и стратегии с отложенной балансировкой, когда дерево перестраивается целиком или большими частями один раз в несколько вставок или удалений.

### 2.4.4 Хеш-таблицы

Зададимся вопросом можно ли осуществить поиск по значению (или по ключу), вставку и удаление элемента за константное время $O(1)$. Вспомним, что чтение и запись элемента массива по его номеру $i$ — единственная операция, имеющая константную сложность, из тех, что были описаны выше.

Рассмотрим вначале упрощённый, но мало практичный способ, который называется *таблица прямой адресации*. Предположим, что мы хотим хранить подмножество $X \subset \mathbf{Z}_{\geq \mathbf{0}}$ неотрицательных целых чисел, тогда мы могли бы организовать массив, значениями которого являются только 0 и 1, по следующему принципу. По индексу $i$ стоит 1 если $i \in X$ и 0 в противном случае. В таком случае, для поиска элемента по значению $x$ достаточно было бы обратиться к памяти по адресу $a_0 + i \cdot 1$, что занимает константное время. Аналогичным образом мог бы быть реализован ассоциативный массив. Вспомним пример про телефонную книгу из раздела 2.4.3, тогда в элементе номер $i$ таблицы прямой адресации будем либо адрес строки с именем абонента с номером телефона $i$, либо признак отсутствия номера в телефонной книге. Очевидно, что поиск, добавление и удаление записи с известным номером телефона займёт константное время.

Этот способ имеет два серьёзных недостатка, значительно сужающих область его практической применимости. Во-первых, не очевидно как этот подход адаптировать для хранения, например, строк. Что, если бы мы захотели построить обратную телефонную книгу и уметь эффективно находить номер, зная имя абонента? Во-вторых, способ может потребовать огромного количества памяти, порядка

$$\max_{i \in X}\{i\} - \min_{i \in X}\{i\},$$

большинство из которой, оказалось бы заполнено признаками отсутствия данных. Например, если телефонный номер записан в виде десятизначного десятичного числа, то потребуется $10^{10}$ ячеек таблицы прямой адресации. Пусть одна ячейка занимает 8 байт, тогда таблица потребовала бы около 75 GB памяти. Понятно, что в телефонной книге у одного человека может быть от силы тысяча контактов, т.е. эффективность использования памяти составляет порядка $10^{-7}$.

Оказывается, что эти недостатки можно преодолеть следующим образом. Назовём *хеш-фунцией*[17] такую детерминированную функцию $i = h(x)$, которая любому $x$ (любой требуемой природы: целому числу, вектору, действительному числу, или даже строке) сопоставляет целочисленный индекс $i$ в разумном интервале значений (например, $[0, 2^{32})$ или $[0, 2^{64})$), и при этом для любого распределения $x$ соответствующее распределение $i = h(x)$ должно быть как можно ближе к равномерному. Кроме того, желательно, чтобы хеш-функция вычислялась сравнительно просто и быстро.

Например, возможный вариант хеш-функции для натурального числа $x$:

$$h(k) = x \mod p, \tag{2.13}$$

где $p$ некоторое простое число, а операция $x \mod p$ означает взятие остатка от деления $x$ на $p$. Понятно, что хеш-функция не может быть определена единственным образом, поэтому возможен, например, такой альтернативный вариант:

$$h(k) = \lfloor m\{x \cdot A\} \rfloor, \tag{2.14}$$

где $A \in (0, 1)$ и $m \in \mathbf{Z}_+$ — некоторые параметры, $\{\cdot\}$ обозначает дробную часть числа, а $\lfloor \cdot \rfloor$ обозначает округление вниз до целого.

Пример хеш-функции для строки:

$$h(\mathbf{x}) = \left( \sum_{j=0}^{L-1} p^j x_j \right) \mod q, \tag{2.15}$$

где $p$ и $q$ некоторые простые числа, а $x_j$ - числовой код символа номер $j$ в строке из $L$ символов. Составление хорошей хеш-функции — процесс творческий, но, к счастью, на любой случай уже было придумано и реализовано достаточное количество эффективных вариантов.

Теперь следующее улучшение таблицы прямой адресации будем называть *хеш-таблицей*. Во-первых, зафиксируем некоторую хеш-функцию $h(x)$ для интересующего нас типа элементов. Во-вторых, зафиксируем заранее таблицу из $M$ одинаковых по размеру ячеек[18], в которых хранится либо некоторый элемент $x$, либо признак его отсутствия. В-третьих, договоримся, что будем хранить элемент со значением $x$ только в ячейке с номером $i' = h(x)$. На этот раз $x$ может иметь любой тип данных, так как преобразуется в индекс $i'$ с помощью хеш-функции. Если хеш-функция $h(x)$ принимает значения в интервале $[0, 2^\omega)$, в таком случае оказывается удобным

[17] hash function

[18] Часто ячейку называют *bucket* — корзинка.

выбирать размер таблицы как степень двойки $M = 2^m$, где $m \leq n$. Можно доказать, что если случайное число распределено равномерно в интервале $[0, 2^\omega)$, то его остаток от деления на $2^{\omega-1}$ будет равномерно распределён в интервале $[0, 2^{\omega-1})$, таким образом из одной хеш-функции порождается другая хеш-функция, адаптированная под требуемый размер таблицы. Возвращаясь к примеру с телефонной книгой, если мы используем вместо таблицы прямой адресации хеш-таблицу размером $M = 1024$, то эффективность использования памяти будет близка к 1.

На первый взгляд, сложность операций поиска, удаления и вставки элемента в хеш-таблицу по прежнему составляет $O(1)$, как это было в случае с таблицей прямой адресации. Однако, чаще всего хеш-функция это сюръективное отображение, т.е. найдутся $x_1$ и $x_2$ такие, что $h(x_1) = h(x_2)$. А значит, мы сможем записать в хеш-таблицу либо $x_1$, либо $x_2$, но никак не $x_1$ и $x_2$ одновременно. Эта ситуация называется *коллизией*.

Одним из способов разрешения коллизий является *разрешение коллизий медом цепочек*, т.е. с помощью списков. Допустим, что каждый элемент нашей таблицы из $M$ элементов является односвязным списком (возможно пустым), и этот список уже хранит реальные значения $x$ (либо пары ключ-значение $k$, $v$). Причём список номер $i$ хранит только элементы с одинаковой хеш-функцией. Теперь все интересующие нас операции выполняются в два этапа. Чтобы найти элемент $x$ (либо найти значение $v$ по ключу $k$), следует сначала за $O(1)$ операций обратится к ячейке $i' = h(x)$ (либо $i' = h(k)$), а затем пройти весь соответствующий список из $N_{i'}$ элементов и сопоставить каждый его элемент с искомым, что займёт $O(N_{i'})$ операций. Аналогично, чтобы удалить или добавить элемент $x$, нужно сначала обратиться к списку в ячейке $i' = h(x)$, затем найти в нём элемент $x$ и уже удалить или добавить новый элемент в этот список. В худшем случае, $N_{i'} = N$ и, следовательно, сложность рассматриваемых операций $O(N)$, однако, хеш-таблица всё ещё обеспечивает константную сложность операций *в среднем*.

Допустим, что мы выбрали количество ячеек таким образом, что $M \gg N$. В таком случае, вероятность коллизий будет убывать пропорционально $\frac{N}{M}$ и в конечном счёте окажется пренебрежимо мала. Понятно, что в этом случае хеш-таблица будет обеспечивать быстрый доступ, но с низкой эффективностью использования памяти, которая тоже будет убывать пропорционально $\frac{N}{M}$. В случае, если мы выбрали количество ячеек так, что $M \ll N$, то вероятность коллизий будет близка к 1, и каждая ячейка будет хранить очень длинный список из $\frac{N}{M}$ элементов. В таком случае эффективность использования памяти будет высока, а скорость доступа наоборот — линейная по числу элементов. На практике, в результате вставок и удалений элементов количество хранимых записей $N$ постоянно меняется, поэтому часто $M$ подбирается динамически, чтобы обеспечить оптимальный баланс между скоростью работы и использованием памяти: в моменты, когда заполненность хеш-таблицы сильно меняется в ту или иную сторону, происходит перебалансировка таблицы. Создаётся новая таблица размера $2\,M$ (или $\frac{1}{2}M$) с новой хеш-функцией $h'(x)$, куда перекладываются все элементы.

Стоит отметить интересный способ борьбы с коллизиями в случае, когда область допустимых значений $x$ (или ключей $k$) невелика. Например, нам попались названия химических элементов, закодированные в виде строк. В таком случае, во-первых, заранее можно перечислить все возможные варианты, и во-вторых, их число будет с запасом меньше, чем $2^8 = 256$.[19] Оказывается, существуют алгоритмы, позволяющие построить *идеальную хеш-функцию*[20], которая будет *взаимооднозначно* отображать допустимое множество ключей в множество индексов.

[19] На момент написания было известно 118 элементов.
[20] perfect hash function