2 Практическая теория

В этой главе

- ✓ Почему знания computer science важны для выживания
- ✓ Как заставить типы работать на вас
- ✓ Понимание особенностей алгоритмов
- ✓ Структуры данных и их странные свойства, о которых вам не рассказали родители

Вопреки широко распространенному мнению, программисты тоже люди. И они, так же как остальные люди, заблуждаются, думая о практике разработки. Они сильно переоценивают преимущества необязательного использования типов, не заботясь о правильных структурах данных или считая, что алгоритмы важны только для авторов библиотек.

Вы не исключение. От вас ожидают, что вы сделаете качественный продукт вовремя и с улыбкой. Как гласит поговорка, программист — это организм, который получает кофе на входе и создает программы на выходе. С таким же успехом вы можете использовать копирование и вставку, код, который вы нашли на Stack Overflow, простые текстовые файлы для хранения данных или заключить сделку с дьяволом,

если еще не продали душу, подписав NDA[1]. Только ваших коллег действительно волнует, как вы работаете, всем остальным нужен надежный готовый продукт.

Теория может быть избыточной и неуместной. Алгоритмы, структуры данных, теория типов, нотация «О-большое» и полиномиальная сложность могут показаться чересчур мудреными и неприменимыми для разработки. Существующие библиотеки и фреймворки и так уже справляются со всем этим, к тому же они оптимизированы и надежно протестированы. Вас всячески мотивируют не реализовывать алгоритмы с нуля, особенно ввиду вопросов информационной безопасности или сжатых сроков.

Тогда зачем нужна теория? Потому что знание теории computer science позволяет не только разрабатывать алгоритмы и структуры данных с нуля, но и правильно определять, когда их стоит использовать. Это помогает понять стоимость компромиссов и свойства масштабируемости кода, который вы пишете. Это заставляет смотреть вперед. Возможно, вы никогда не создадите структуру данных или алгоритм целиком, но знание того, как они работают, сделает вас эффективным разработчиком. Оно повысит ваши шансы выжить на улицах.

В этой книге будут рассмотрены только критически важные разделы теории, которые вы могли пропустить при ее изучении, — не самые известные свойства типов данных, оценка сложности алгоритмов и внутренних механизмов работы определенных структур данных. Если вы раньше не знали о типах, алгоритмах или структурах данных, эта глава подскажет, с чего начать подробное знакомство с ними.

2.1. КРАТКИЙ ОБЗОР АЛГОРИТМОВ

Алгоритм — это набор правил и шагов для решения задачи. Спасибо, что слушали меня на конференции TED[2]. Вы ожидали более сложного определения, не так ли? Примером простого алгоритма может служить перебор элементов массива, чтобы выяснить, содержит ли он заданное число:

```
public static bool Contains(int[] array, int lookFor) {
    for (int n = 0; n < array.Length; n++) {
        if (array[n] == lookFor) {
            return true;
```

[1] Non-disclosure agreement — Соглашение о неразглашении, запрещающее сотрудникам говорить о своей работе, если только они не начинают разговор со слов «Я вам этого не говорил, но...».

[2] TED Talks — научно-популярные конференции в разных сферах знания. — *Примеч. пер.*

```
        }
    }
    return false;
}
```

Его можно было бы назвать *алгоритмом Седата*, если бы он был изобретен мной, но скорее всего, это один из первых алгоритмов, придуманных человеком. Он не совершенен, но работает и имеет смысл. Это одно из важных и необходимых свойств алгоритма: он должен работать для ваших нужд и не обязательно творить чудеса. Когда вы ставите посуду в посудомоечную машину и нажимаете кнопку запуска, то следуете алгоритму. Алгоритм не обязательно должен быть сложным.

Тем не менее существуют и более интеллектуальные алгоритмы. В предыдущем примере кода, если известно, что список содержит только положительные целые числа, можно использовать специальную обработку неположительных чисел:

```
public static bool Contains(int[] array, int lookFor) {
    if (lookFor < 1) {
        return false;
    }
    for (int n = 0; n < array.Length; n++) {
        if (array[n] == lookFor) {
            return true;
        }
    }
    return false;
}
```

Так алгоритм будет работать намного быстрее при вызовах с отрицательными числами. В лучшем случае функция всегда будет вызываться с отрицательными числами или нулем и немедленно возвращаться, даже если массив содержит миллиарды целых чисел. В худшем — функция всегда будет вызываться с положительными числами и дополнительная проверка будет только снижать производительность. Здесь на помощь придет беззнаковый тип целых чисел в C# — `uint`. При его использовании компилятор выполнит проверку только для положительных чисел, что исключит лишние проверки:

```
public static bool Contains(uint[] array, uint lookFor) {
    for (int n = 0; n < array.Length; n++) {
        if (array[n] == lookFor) {
            return true;
        }
    }
    return false;
}
```

Мы обеспечили положительность проверяемого числа, не меняя алгоритм, за счет ограничения типа данных. Но можно ускорить работу алгоритма и путем изменения формы данных. Что мы знаем о данных? Массив отсортирован? Если да, можно ускорить поиск числа. Если сравнить число с любым элементом в отсортированном массиве, легко исключить огромное количество элементов (рис. 2.1).

Рис. 2.1. Одна операция сравнения позволяет исключить левую или правую часть отсортированного массива

Если искомое число — 3 и мы сравниваем его с элементом отсортированного по возрастанию массива, который оказался равным 5, можно быть уверенными, что наше число не может располагаться правее этого элемента. Это значит, что можно спокойно игнорировать все элементы справа от 5.

Таким образом, если мы выбираем элемент из середины массива, то после сравнения гарантированно можно исключить как минимум половину массива. Эту же логику можно применить к оставшейся половине, выбрать в ней среднее значение и продолжить. Таким образом, для отсортированного массива из 8 элементов потребуется максимум три сравнения, чтобы определить, имеется ли в нем искомое число. Что еще более важно, потребуется не более 10 операций для поиска числа в массиве из 1000 элементов.

Деление массива пополам — мощный инструмент. Его реализация представлена в листинге 2.1. Мы постоянно находим средний элемент и после сравнения исключаем половину элементов в зависимости от того, в какую из них попадает искомое значение. Для расчета среднего элемента используется формула, которую можно было бы упростить до `(start + end)v / 2`. Но мы этого упрощения не делаем, потому что `(start + end)` может привести к переполнению при больших значениях `start` и `end`, тогда среднее будет вычислено неверно. Формула в листинге ниже позволяет избежать такого переполнения.

Листинг 2.1. Двоичный поиск в отсортированном массиве

```
public static bool Contains(uint[] array, uint lookFor) {
  int start = 0;
  int end = array.Length - 1;
  while (start <= end) {
```

```
    int middle = start + ((end - start) / 2);
    uint value = array[middle];
    if (lookFor == value) {
      return true;
    }
    if (lookFor > value) {
      start = middle + 1;
    } else {
      end = middle - 1;
    }
  }
  return false;
}
```

Определение среднего элемента без риска переполнения

Исключение левой части массива

Исключение правой части массива

Здесь мы реализовали двоичный поиск — значительно более быстрый алгоритм, чем *алгоритм Седата*. Поскольку теперь понятно, почему двоичный поиск может быть быстрее, чем простой перебор, можно переходить к известной нотации «О-большое».

2.1.1. «О-большое» должно быть приемлемым

Понимание закономерностей — отличный навык для разработчика. Когда вы знаете, как быстро что-то увеличивается в размере или количестве, то можете предвидеть будущее и, следовательно, прогнозировать возможные проблемы, прежде чем потратите на них время. Это особенно полезно, когда свет в конце туннеля становится все ярче, даже если вы неподвижны.

Нотация «О-большое» — условное обозначение оценки сложности, но не все это понимают. Когда я впервые увидел O(N), то подумал, что это обычная функция, которая должна возвращать число. Это не так. С ее помощью математики выражают рост сложности расчетов. Она дает базовое представление о масштабируемости алгоритма. Последовательный обход каждого элемента (*алгоритм Седата*) подразумевает совершение количества операций, которое прямо пропорционально числу элементов в массиве. Запишем это утверждение в виде O(N), где N — количество элементов. O(N) не дает понять, сколько именно шагов потребуется алгоритму для выполнения, но показывает, что это число растет линейно. Благодаря этому можно оценить производительность алгоритма в зависимости от размера данных и предположить, в какой момент он станет неэффективным.

Реализованный нами двоичный поиск имеет сложность $O(\log 2^n)$. Логарифм — это функция, противоположная экспоненте, поэтому логарифмическую сложность можно считать замечательной, если только вопрос не касается денег. Если

бы алгоритм сортировки из нашего примера волшебным образом приобрел логарифмическую сложность, для сортировки массива из 500 000 элементов потребовалось бы всего 18 сравнений. Это великолепная эффективность.

«О-большое» используется для измерения роста не только количества вычислений, то есть *временно́й сложности*, но и объема используемой памяти, то есть *пространственной сложности*. Алгоритм может быть быстрым, но требовать полиномиального увеличения объема используемой памяти как в нашем примере с сортировкой. Необходимо понимать это.

> **ДЛЯ СПРАВКИ** Вопреки распространенному мнению, $O(N^x)$ не означает экспоненциальную сложность. Она обозначает полиномиальную сложность, которая хоть и плоха, но не настолько, как экспоненциальная $O(x^n)$. В массиве из 100 элементов $O(N^2)$ будет означать 10 000 итераций, а $O(2^n)$ — абсолютно умопомрачительное число из 30 цифр, которое даже выговорить сложно. Существует также факториальная сложность, которая еще хуже экспоненциальной. Мне почти не приходилось с ней сталкиваться, кроме алгоритмов вычисления перестановок и алгоритмов, где она сочетается с другими видами сложности — наверное, потому, что никто не хочет иметь с ней дела.

Поскольку нотация «О-большое» описывает рост, самое важное для нее — это сравнение функций роста. На практике O(N) эквивалентна O(4N), но не O(N.M) (точка — оператор умножения), если и N и M возрастают. Последняя может быть эквивалентной O (N^2). O(N.logN) несколько хуже, чем O(N), но не так плоха, как $O(N^2)$.

Функция O(1) удивительна. Она означает, что производительность алгоритма не зависит от количества элементов в имеющейся структуре данных. Поэтому его называют алгоритмом *постоянного времени*.

Представьте, что вы написали функцию поиска в базе данных, которая перебирает все записи. Это означает, что время выполнения алгоритма будет расти прямо пропорционально количеству элементов в базе данных. Предположим, что мы до сих пор пользуемся счетами и обработка каждой записи у нас занимает секунду. Это означает, что поиск в базе данных из 60 элементов займет до минуты. Это сложность O(N). Другие разработчики в команде могут предложить другие алгоритмы, как показано в табл. 2.1.

Чтобы принимать обоснованные решения при выборе алгоритма обработки данных, вы должны иметь представление о том, как нотация «О-большое» описывает рост сложности алгоритма и использование памяти. Прикиньте значение

«О-большого», даже если вам не нужно реализовывать алгоритм. Не пренебрегайте оценкой сложности.

Таблица 2.1. Влияние сложности на производительность

Алгоритм поиска	Оценка сложности	Время нахождения записи среди 60 строк
Самодельный квантовый компьютер из гаража	O(1)	1 секунда
Двоичный поиск	O(log N)	6 секунд
Линейный поиск (потому что ваш босс попросил вас написать код за час до презентации)	O(H)	60 секунд
Стажер по ошибке использовал вложенные циклы	$O(N^2)$	1 час
Код, найденный на Stack Overflow, параллельно ищет решение открытой математической проблемы	$O(2^N)$	36,5 миллиарда лет
Алгоритм пытается найти порядок даных, расшифровывающий запись, которую вы ищете. Хорошая новость в том, что разработчик этого здесь больше не работает	O(N!)	До конца вселенной, но все же раньше, чем обезьяна закончит печатать пьесу Шекспира[1]

2.2. СТРУКТУРЫ ДАННЫХ ИЗНУТРИ

> Вначале была пустота. Когда первые электрические импульсы достигли первого бита памяти, появились данные. Данные свободно плавали в виде байтов. Эти байты соединились и создали структуру.
>
> *Начало 0:1*

Структуры данных — это о том, как данные располагаются. Люди обнаружили, что данные более полезны, если они расположены определенным образом. Список покупок на листе бумаги легче читать, если каждый пункт начинается с новой строки. Таблица умножения более полезна, если она представлена в виде сетки. Чтобы стать хорошим программистом, важно понимать,

[1] Отсылка к теореме о бесконечных обезьянах, согласно которой абстрактная обезьяна, случайным образом ударяя по клавиатуре, рано или поздно напечатает пьесу Шекспира. — *Примеч. пер.*

как работает та или иная структура данных. Для этого надо изучить, как все устроено «под капотом».

Возьмем, к примеру, массивы. Массив в программировании — одна из простейших структур данных. В памяти он располагается в виде непрерывного набора элементов. Допустим, у вас есть такой массив:

```
var values = new int[] {1, 2, 3, 4, 5, 6, 7, 8};
```

Вы представляете, что в памяти он располагается так, как показано на рис. 2.2.

Рис. 2.2. Размещение массива в памяти — неверное представление

На самом деле это не так, потому что у каждого объекта в .NET имеется заголовок, указатель на таблицу виртуальных методов и информация о длине, как показано на рис. 2.3.

Рис. 2.3. Фактическое размещение массива в памяти

Структура данных становится еще понятнее, если представлять, как массив размещен в оперативной памяти, потому что последняя не состоит из целых чисел (рис. 2.4). Я рассказываю об этом, потому что хочу, чтобы вы не боялись низкоуровневых концепций. Их понимание помогает на всех уровнях программирования.

В реальности современные операционные системы выделяют для каждого процесса свой фрагмент оперативной памяти. И это придется принять как данность, если только вы не станете разрабатывать свою собственную операционную систему или собственные драйверы устройств.

Рис. 2.4. Процессы и массив в памяти

В целом способ организации данных может как ускорить операции и сделать работу эффективнее, так и наоборот. Поэтому необходимо знать некоторые основные структуры данных и принципы их работы.

2.2.1. Строки

Строки — самый понятный тип данных в программировании, поскольку они представляют собой текст. Не стоит использовать строки, если для ваших целей лучше подходит другой тип данных, но обойтись без них невозможно. Вдобавок они удобны. Однако не все даже основные свойства строк очевидны.

Строки в .NET неизменяемы, хотя и напоминают массивы по типу использования и структуре. Неизменяемость означает, что содержимое структуры данных не может быть изменено после инициализации. Предположим, что нам необходимо объединить имена и фамилии и получить единую строку, разделенную запятыми, и что мы отброшены на два десятка лет назад, поэтому нет лучшего способа сделать это, кроме следующего:

```
public static string JoinNames(string[] names) {
    string result = String.Empty;
    int lastIndex = names.Length - 1;
    for (int i = 0; i < lastIndex; i++) {
        result += names[i] + ", ";
    }
```

Неинициализированная строка имела бы по умолчанию нулевое значение, которое можно было бы обнаружить проверкой на допустимость такого значения

Индекс последнего элемента

```
    result += names[lastIndex];  ◄── Благодаря этому строка не будет оканчиваться запятой
    return result;
}
```

На первый взгляд может показаться, что мы модифицируем в цикле одну и ту же строку с именем `result`, но это не так. Каждый раз, когда мы присваиваем `result` новое значение, мы создаем новую строку в памяти. .NET необходимо определить длину новой строки, выделить для нее память, скопировать в эту память содержимое других строк и вернуть `result`. Это довольно затратная операция, и ее стоимость увеличивается по мере увеличения длины строки и сопутствующего мусора.

Во фреймворке есть бесплатные инструменты, позволяющие избежать этой проблемы. Благодаря им не нужно менять логику или из кожи вон лезть, чтобы повысить производительность. Один из таких инструментов — `StringBuilder`, с помощью которого за один раз можно создать финальную строку и извлечь ее с помощью вызова `ToString`:

```
public static string JoinNames(string[] names) {
    var builder = new StringBuilder();
    int lastIndex = names.Length - 1;
    for (int i = 0; i < lastIndex; i++) {
        builder.Append(names[i]);
        builder.Append(", ");
    }
    builder.Append(names[lastIndex]);
    return builder.ToString();
}
```

`StringBuilder` использует последовательные блоки памяти, вместо того чтобы перераспределять и копировать их каждый раз, когда нужно увеличить строку. Обычно это более эффективно, чем создавать строку с нуля.

Очевидно, что естественное и значительно более короткое решение доступно уже давно, хотя не всегда применимо на практике:

```
String.Join(", ", names);
```

Конкатенация обычно допустима при инициализации строки, потому что это требует только одного выделения буфера после вычисления требуемой общей длины. Например, если у вас есть функция, которая соединяет имя и фамилию с пробелом между ними с помощью оператора сложения, вы создаете одну новую строку за один шаг:

```
public string ConcatName(string firstName, string middleName,
  string lastName) {
  return firstName + " " + middleName + " " + lastName;
}
```

Это может показаться полной ерундой, если посчитать, что `firstName + " "` сначала создаст новую строку, затем — новую строку с `middleName` и так далее, но на самом деле компилятор превращает все это в один вызов функции `String.Concat( )`, которая выделяет новый буфер длиной, равной сумме длин всех строк и возвращает его за один раз. Поэтому все происходит быстро. Но объединение строк за несколько шагов с промежуточными условиями `if` или циклами компилятор уже не может оптимизировать. Нужно знать, когда можно объединять строки, а когда нет.

Тем не менее неизменяемость — это не нерушимый Святой Грааль. Существуют способы изменения строк и других неизменяемых структур. В основном они подразумевают использование небезопасного кода и вызов духов и обычно не рекомендуются, потому что строки дедуплицируются средой выполнения .NET и некоторые их свойства, такие как хеш-коды, кэшируются. Внутренняя реализация в значительной степени зависит от особенностей неизменяемой структуры.

Строковые функции по умолчанию работают с текущей системной культурой. Поэтому может оказаться неприятным сюрпризом то, что приложение перестанет работать в другой стране.

> **ПРИМЕЧАНИЕ** *Культура*, в некоторых языках программирования также называемая *локалью*, представляет собой набор правил для выполнения операций, зависящих от региона, таких как сортировка строк, формат даты и времени, раскладка столовых приборов и т. д. Текущая культура — это то, чем пользуется операционная система.

Понимание культурного контекста может повысить скорость и безопасность операций со строками. Например, рассмотрим код, определяющий, содержит ли имя файла расширение .gif:

```
public bool isGif(string fileName) {
    return fileName.ToLower().EndsWith(".gif");
}
```

Как видите, задав строке нижний регистр, мы предусмотрели случаи, когда расширение может быть набрано как .GIF, .Gif или другой комбинацией регистров. Но не во всех языках буква в нижнем регистре соответствует той же букве

в верхнем. Например, в турецком языке строчная буква для «I» — это не «i», а «ı», также известная как I без точки. Поэтому код в приведенном примере не сработает в Турции и, возможно, в Азербайджане и некоторых других странах. Переводя строку в нижний регистр, мы фактически создаем новую строку, что, как мы узнали, неэффективно.

.NET предоставляет не привязанные к культуре версии некоторых строковых методов, таких как `ToLowerInvariant`, а также возможности перегрузки метода при получении значения `StringComparison`, имеющего инвариантную и порядковую разновидности. Поэтому можно сделать метод более безопасным и быстрым:

```
public bool isGif(string fileName) {
    return fileName.EndsWith(".gif",
        StringComparison.OrdinalIgnoreCase);
}
```

Этот метод позволяет избежать создания новой строки, а сравнение строк происходит более безопасно и быстро без учета особенностей текущей культуры. Можно использовать и `StringComparison.InvariantCultureIgnoreCase`, но, в отличие от порядкового сравнения, в этом случае добавляются правила транслитерации, такие как замена немецких умляутов и специфических графем их латинскими аналогами (ß на ss), что может вызвать проблемы с именами файлов и другими идентификаторами ресурсов. При порядковом сравнении значения символов сравниваются напрямую, без транслитерации.

2.2.2. Массив

Мы уже видели, как массив размещается в памяти. Массивы удобны для хранения набора элементов, числовые значения которых не превышают размеров массива. Массивы — статические структуры. Поэтому если потребуется больший массив, придется создать новый и скопировать в него содержимое старого. Рассмотрим несколько особенностей массивов, о которых следует знать.

Массивы, в отличие от строк, изменяемы. Это их суть. Их содержимым можно свободно манипулировать. На самом деле очень сложно сделать массивы неизменяемыми, поэтому они не годятся для интерфейсов. Рассмотрим такое свойство:

```
public string[] Usernames { get; }
```

Хотя здесь нет сеттера, это массив, и он изменяем. Ничто не мешает сделать следующее:

```
Usernames[0] = "root";
```

Однако теперь все усложнилось, даже если этот класс используете только вы. Никогда не вносите изменения в состояние, если только в этом нет крайней необходимости. Корень всех зол — состояние, а не нулевое значение. Чем меньше состояний у приложения, тем меньше проблем.

Старайтесь придерживаться типов с минимальной необходимой функциональностью. Если вам нужно только просматривать элементы по порядку, используйте `IEnumerable<T>`. Если вдобавок нужно периодически вести подсчет, используйте `ICollection<T>`. Обратите внимание, что метод расширения LINQ `.Count()` содержит специальный код обработки для типов, поддерживающих `IReadOnlyCollection<T>`, поэтому даже если вы используете его для `IEnumerable`, есть вероятность, что он вернет кэшированное значение.

Массивы лучше всего подходят для использования внутри локальной области видимости функции. Для любых других целей в дополнение к `IEnumerable<T>` существует более подходящий тип или интерфейс, например `IReadOnlyCollection<T>`, `IReadOnlyList<T>` или `ISet<T>`.

2.2.3. Список

Список ведет себя как массив, который может постепенно увеличиваться, подобно тому как работает `StringBuilder`. Списки можно использовать вместо массивов практически везде, но это приведет к ненужному снижению производительности, поскольку индексированный доступ в списке представляет собой виртуальный вызов, в то время как массив использует прямой доступ.

Дело в том, что в объектно-ориентированном программировании есть такая замечательная вещь, как полиморфизм. Это означает, что объект может вести себя в соответствии с базовой реализацией, не меняя интерфейс. Если у вас есть, скажем, переменная `a` с типом интерфейса `IOpenable`, то `a.Open()` может открыть файл или установить сетевое подключение в зависимости от типа назначенного ей объекта. Это реализуется путем предоставления ссылки на таблицу, в которой вызываемые виртуальные функции сопоставлены с типом, указанным в начале объекта, — таблицу виртуальных методов ***`vtable`***. Таким образом, хотя `Open` сопоставляется с одной и той же записью в таблице для каждого объекта одного и того же типа, вы не будете знать, куда это приведет, пока не увидите фактическое значение в таблице.

Поскольку мы не знаем, что именно вызываем, такие вызовы называются виртуальными. Виртуальный вызов подразумевает дополнительный поиск в таблице виртуальных методов, поэтому он немного медленнее, чем обычные вызовы

функций. Это может не представлять проблемы, если вызовов несколько, но когда они осуществляются внутри сложного алгоритма, затраты могут расти полиномиально. Таким образом, если список не будет увеличиваться после инициализации, вместо него в локальной области видимости можно использовать массив.

Обычно о таких деталях думать не приходится. Но зная разницу, можно определить, когда массив предпочтительнее списка.

Списки похожи на `StringBuilder`. В обоих случаях структуры являются динамически растущими, но механизм роста в списках менее эффективен. Всякий раз, когда список должен увеличиться, он выделяет новый массив большего размера и копирует в него существующее содержимое. `StringBuilder`, напротив, сохраняет цепочку фрагментов памяти, что не требует копирования. Область буфера для списков увеличивается всякий раз при достижении предела, но размер нового буфера удваивается, поэтому потребность в увеличении со временем снижается. Тем не менее это пример того, как специальный класс может быть более эффективен, чем общий.

Улучшить производительность списка можно, указав емкость. Если ее не указать, список начнется с пустого массива. Затем он увеличит свою вместимость до нескольких элементов. После заполнения он удвоит вместимость. Задание емкости при создании списка позволит избежать ненужных операций увеличения и копирования. Используйте эту возможность, если заранее знаете максимальное количество элементов в списке.

Однако не стоит задавать емкость списка без необходимости. Это может вызвать ненужные затраты памяти, которые могут накапливаться. Возьмите за правило принимать продуманные решения.

2.2.4. Связанный список

Связанные списки — это списки, в которых элементы расположены в памяти не последовательно, но каждый элемент указывает на адрес следующего. Они полезны, поскольку производительность при их вставке и удалении равна O(1). Вы не сможете получить доступ к отдельным элементам по индексу, потому что они хранятся в разных местах памяти и вычислить их местоположение не получится. Но если вы обращаетесь к началу либо концу списка или если нужно только пересчитать элементы, производительность будет максимальной. Проверка же того, существует ли элемент в связанном списке, является операцией O(N), как в массивах и обычных списках. На рис. 2.5 показан пример представления связанного списка.