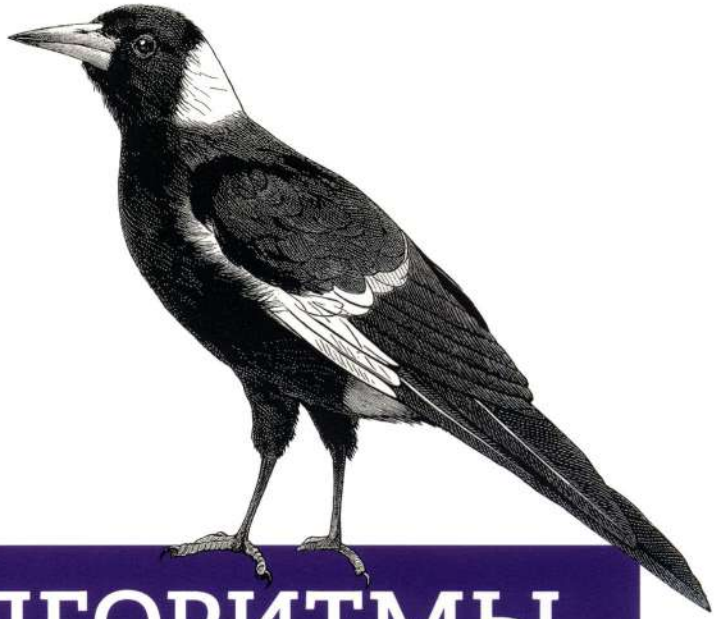
O'REILLY®



АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Извлечение информации на языке JAVA

ПИТЕР®

Аллен Б. Доуни

Think Data Structures

Algorithms and Information Retrieval in Java

Allen B. Downey

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY

Аллен Б. Доуни

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Извлечение информации на языке JAVA

ПИТЕР Санкт-Петербург · Москва · Екатеринбург · Воронеж
Нижний Новгород · Ростов-на-Дону · Самара · Минск

2018

ББК 32.972.233.02
УДК 004.62
Д71

Доуни Аллен Б.

Д71 Алгоритмы и структуры данных. Извлечение информации на языке Java. — СПб.: Питер, 2018. — 240 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-0572-4

Изучите, как следует реализовывать эффективные алгоритмы на основе важнейших структур данных на языке Java, а также как измерять производительность этих алгоритмов. Каждая глава сопровождается упражнениями, помогающими закрепить материал.

- Научитесь работать со структурами данных, например, со списками и словарями, разберитесь, как они работают.
- Напишите приложение, которое читает страницы Википедии, выполняет синтаксический разбор и обеспечивает навигацию по полученному дереву данных.
- Анализируйте код и учитесь прогнозировать, как быстро он будет работать и сколько памяти при этом потреблять.
- Пишите классы, реализующие интерфейс Map, пользуйтесь при этом хеш-таблицей и двоичным деревом поиска.
- Создайте простой веб-поисковик с собственным поисковым роботом: он будет индексировать веб-страницы, сохранять их содержимое и возвращать нужные результаты.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.972.233.02
УДК 004.62

ISBN 978-1491972397 англ.

ISBN 978-5-4461-0572-4

Краткое содержание

Оглавление

Предисловие

Философия книги

Структуры данных и алгоритмы — одни из самых важных изобретений последних 50 лет. Это фундаментальные инструменты, которыми должны владеть все разработчики программного обеспечения. Однако, на мой взгляд, большинство книг по этим темам слишком теоретические, слишком объемные и слишком «снизу вверх».

- *Слишком теоретические.* Математический анализ алгоритмов основан на упрощающих допущениях, которые ограничивают его полезность на практике. Большинство пособий по теме игнорируют упрощения и фокусируются на математике. В этом издании я представляю наиболее практичную выборку по данной теме и опускаю или отодвигаю на второй план остальное.

- *Слишком объемные.* Чаще всего книги об алгоритмах и структурах данных содержат не менее 500 страниц, а некоторые — больше 1000. Сфокусировав внимание на темах, как мне кажется, наиболее полезных для программистов, я постарался сохранить небольшой объем этой книги.

- *Слишком «снизу вверх».* Большинство книг по структурам данных сосредоточены на том, как работают эти структуры (реализации), и слабо освещают их использование (интерфейсы). В этом издании я иду «сверху вниз», начиная с интерфейсов. Читатели учатся применять структуры в Java Collections Framework, прежде чем вникать в детали того, как они работают.

Наконец, некоторые книги представляют материал вне контекста и без мотивации: просто одна проклятая структура данных за другой! Я пытаюсь оживить процесс чтения, организовав темы на примере приложения для веб-поиска, которое широко использует структуры данных и само по себе интересно и важно.

С помощью этого приложения я буду объяснять отдельные темы, обычно не рассматриваемые во вводном курсе по структурам данных, включая неизменяемые структуры данных в Redis.

Я принял непростое решение о том, какие материалы исключить из книги, но пошел на компромиссы. В частности, добавил несколько тем, бесполезных для большинства читателей, но знание которых пригодится, например, в техническом интервью. По этим темам я излагаю как общепринятое мнение, так и свое скептическое отношение.

Кроме того, в этой книге раскрываются основные практические аспекты разработки программного обеспечения, в том числе контроль версий и модульное тестирование. Большинство глав включают упражнение, позволяющее читателям применить полученные знания на практике. Каждое из них предлагает автоматизированные тесты, которые проверяют решение. И для большинства упражнений я привожу свое решение в начале следующей главы.

Обязательные условия

Эта книга адресована студентам, изучающим информатику и смежные с ней области, а также инженерам-программистам, людям, обучающимся разработке программного обеспечения, и тем, кто готовится к техническим интервью.

Для того чтобы извлечь из предложенного материала максимальную пользу, вам следует хорошо знать Java; в частности, вы должны знать, как определить новый класс, который расширяет существующий класс или реализует интерфейс. Улучшить знания Java помогут две эти книги:

- *Think Java* Аллена Б. Доуни (Allen B. Downey) и Криса Мэйфилда (Chris Mayfield) (O'Reilly Media, 2016) — для тех, кто никогда не программировал;
- *Head First Java* Кэти Сиеры (Kathy Sierra) и Берта Бейтса (Bert Bates) (O'Reilly Media, 2005) — для тех, кто уже знаком с другим языком программирования.

Если вы не знакомы с интерфейсами в Java, то, возможно, понадобится проработать урок под названием «Что такое интерфейс?» на сайте http://thinkdast.com/interface.

Слово «интерфейс» может ввести в заблуждение. В контексте программного интерфейса приложения (application programming interface, API) речь идет о наборе классов и методов, предоставляющих определенную функциональность.

В контексте Java слово «интерфейс» также относится к функции языка, подобной классу, которая задает набор методов.

Кроме того, вам следует знать параметры типа и обобщенные (generic) типы. Например, вы должны уметь создать объект с параметром типа, скажем, `ArrayList<Integer>`. Прочитать о параметрах типа можно на сайте http://thinkdast.com/types.

Вы должны иметь опыт работы в Java Collections Framework (JCF) (о котором можете прочитать на сайте http://thinkdast.com/collections), в частности знать интерфейс `List`, классы `ArrayList` и `LinkedList`.

В идеале нелишним будет иметь представление об утилите Apache Ant — встроенной системе сборки кода для Java. Узнать о ней больше можно по адресу http://thinkdast.com/anttut.

Вы также должны хорошо разбираться в `JUnit`, которая представляет собой библиотеку модульного тестирования для Java. Больше информации о ней см. по адресу http://thinkdast.com/junit.

Работа с кодом

Код для этой книги находится в репозитории Git по адресу http://thinkdast.com/repo.

Git — *система контроля версий*, которая позволяет отслеживать файлы, составляющие проект. Коллекция файлов под управлением Git называется *репозиторием*.

GitHub — сервис хостинга, который предоставляет хранение репозиториев Git и удобный веб-интерфейс. Работать с кодом в данном сервисе можно несколькими способами.

- Вы можете создать копию репозитория на GitHub, нажав кнопку Fork (Ветвление). При отсутствии учетной записи GitHub необходимо ее создать. После создания ответвления

(forking) у вас будет собственный репозиторий на GitHub, который пригоден для отслеживания кода. Затем вы можете *клонировать* (clone) репозиторий — эта процедура загрузит копию файлов на ваш компьютер.

- В качестве альтернативы клонировать репозиторий можно и без ветвления. При выборе этого варианта не понадобится учетная запись GitHub, но не получится сохранить свои изменения в GitHub.
- Если вы вообще не хотите использовать Git, то можете скачать код в ZIP-архиве, нажав кнопку Download (Загрузить) на странице GitHub или перейдя по ссылке http://thinkdast.com/zip.

После того как вы клонируете репозиторий или распакуете ZIP-файл, у вас должен появиться каталог ThinkDataStructures с подкаталогом code.

Примеры в этой книге были разработаны и протестированы с помощью Java SE Development Kit 7. Если у вас установлена более старая версия, то некоторые примеры не будут работать, чего не случится, если вы используете более новую.

Условные обозначения

В книге используются следующие обозначения.

Курсив

Курсивом выделены новые термины.

`Моноширинный шрифт`

Применяется для листингов программ, а также внутри абзацев, чтобы обратиться к элементам программы вроде переменных,

функций, баз данных, типов данных, переменных среды, инструкций и ключевых слов, имен файлов и расширений этих имен.

Моноширинный полужирный шрифт

Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок, каталогов.

Этот элемент содержит совет или предложение.

Такой элемент указывает на примечание общего характера.

Соавторы

Данная книга — адаптированная версия учебного курса, который я написал для школы Flatiron в Нью-Йорке, предлагающей множество онлайн-уроков, связанных с программированием и веб-разработкой и в том числе основанных на этом материале. Школа предоставляет интерактивную среду разработки, помощь преподавателей и других студентов, а также выдает сертификат о прохождении курса обучения. Дополнительную информацию можно найти на сайте http://flatironschool.com.

- В школе Flatiron Джо Берджесс (Joe Burgess), Энн Джон (Ann John) и Чарльз Плетчер (Charles Pletcher) вносили

предложения и исправления на всех этапах: от начальной спецификации до внедрения и тестирования. Спасибо вам всем!

- Я очень благодарен моим техническим рецензентам Барри Уитмену (Barry Whitman), Патрику Уайту (Patrick White) и Крису Мэйфилду (Chris Mayfield), которые внесли ряд полезных предложений и нашли много ошибок. Конечно, любые оставшиеся ошибки — это моя вина, а не их!

- Спасибо преподавателям и студентам курса Data Structures and Algorithms («Структуры данных и алгоритмы») в Olin College, которые прочитали эту книгу и внесли полезные замечания и предложения.

- Чарльз Румелиотис (Charles Roumeliotis) редактировал книгу для O'Reilly Media и внес много исправлений и улучшений.

Если у вас есть комментарии или идеи по поводу текста, отправляйте их по адресу feedback@greentea-press.com.

1 Интерфейсы

В этой книге представлены три темы.

- *Структуры данных.* Начав со структур в Java Collections Framework (JCF), вы узнаете о способах применения таких структур данных, как списки (lists) и карты (maps), и изучите принципы их работы.
- *Анализ алгоритмов.* Я предложу методы для анализа кода и прогнозирования того, как быстро он будет работать и сколько пространства (памяти) потребует.
- *Поиск информации.* Объяснить первые две темы и сделать упражнения более интересными помогут структуры данных и алгоритмы, которые мы используем для создания простой поисковой системы.

Темы будут излагаться по следующему плану.

1. Начнем с интерфейса `List`; вы будете писать классы, реализующие этот интерфейс, двумя разными способами. Далее

сравним ваши реализации с классами `ArrayList` и `LinkedList` из Java.

2. Затем я представлю древовидные структуры данных, а вы будете работать над первым приложением — программой, которая читает страницы из «Википедии», анализирует содержимое и перемещается по полученному дереву, чтобы выявить ссылки, а также выполняет другие функции. Мы используем эти инструменты для проверки гипотезы Getting to Philosophy (с ней можно ознакомиться, перейдя по ссылке http://thinkdast.com/getphil).

3. Вы узнаете об интерфейсе `Map` и реализации `HashMap` в Java. Затем напишете классы, реализующие этот интерфейс, используя хеш-таблицу и бинарное дерево поиска.

4. Наконец, вы будете применять эти классы (а также несколько других — о них я расскажу параллельно) для реализации поискового робота (crawler), который находит и читает страницы, индексатора (indexer), который хранит содержимое веб-страниц в форме, позволяющей эффективно выполнять поиск, и поисковика (retriever), принимающего запросы от пользователя и возвращающего соответствующие результаты.

Начнем.

Почему существует два типа List

Когда люди начинают работать с Java Collections Framework, они иногда путают `ArrayList` и `LinkedList`. Почему в Java представлены две реализации интерфейса `List`? И как выбрать, ка-

кой из них использовать? Я отвечу на эти вопросы в следующих нескольких главах.

Я начну с рассмотрения интерфейсов и классов, которые их реализуют, а также представлю идею «программирования в соответствии с интерфейсом».

В первых упражнениях вы реализуете классы, похожие на `ArrayList` и `LinkedList`, поэтому будете знать, как они работают, их плюсы и минусы. Одни операции выполняются быстрее или используют меньше памяти по сравнению с `ArrayList`; другие быстрее или экономнее по сравнению с `LinkedList`. Выбор класса для конкретного приложения зависит от того, какие операции он выполняет чаще всего.

Интерфейсы в Java

Интерфейс в Java определяет набор методов; любой класс, реализующий этот интерфейс, должен предоставлять конкретные методы. Например, вот исходный код для `Comparable`, который является интерфейсом и определен в пакете `java.lang`:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

В этом определении интерфейса используется параметр типа `T`, который делает `Comparable` *обобщенным типом*. Чтобы реализовать интерфейс, класс должен:

- указывать тип `T`, к которому относится;
- предоставлять метод с именем `compareTo`, который принимает объект как параметр и возвращает `int`.

Пример исходного кода для `java.lang.Integer`:

```
public final class Integer extends Number implements
Comparable<Integer> {

    public int compareTo(Integer anotherInteger) {
        int thisVal + this.value;
        int anotherVal + anotherInteger.value;
        return (thisVal<anotherVal ? -1 :
        (thisVal==anotherVal ? 0 : 1));
    }

    // другие методы опущены
}
```

Этот класс расширяет `Number`; таким образом он наследует методы и переменные экземпляра `Number` и реализует `Comparable<Integer>`, поэтому предоставляет метод с именем `compareTo`, который принимает параметр типа `Integer` и возвращает целочисленное значение.

Когда класс объявляет, что реализует интерфейс, компилятор проверяет, предоставляет ли он все методы, определенные данным интерфейсом.

Кстати, эта реализация `compareTo` использует «тернарный оператор», который иногда записывается так: `?:`. Если вы с ним не знакомы, то можете прочитать о нем на сайте http://thinkdast.com/ternary.

Интерфейс List

Java Collections Framework (JCF) определяет интерфейс под названием `List` и предлагает две реализации: `ArrayList` и `LinkedList`.

Интерфейс определяет, что это должен быть `List`; любой класс, реализующий данный интерфейс, должен обеспечить конкретный набор методов, включая `add`, `get`, `remove` и еще около 20.

Реализации `ArrayList` и `LinkedList` предоставляют эти методы, поэтому их можно использовать как взаимозаменяемые. Метод, написанный для работы с `List`, будет работать с `ArrayList`, `LinkedList` или любым другим объектом, который реализует `List`.

Ниже представлен специально придуманный пример, демонстрирующий данную особенность:

```
public class ListClientExample {
    private List list;

    public ListClientExample() {
        list + new LinkedList();
    }

    private List getList() {
        return list;
    }

    public static void main(String[] args) {
        ListClientExample lce + new ListClientExample();
        List list + lce.getList();
        System.out.println(list);
    }
}
```

`ListClientExample` не делает ничего полезного, но предоставляет важные элементы класса, *инкапсулирующего* `List`, то есть содержит `List` как переменную экземпляра. Сейчас я использую этот класс, чтобы донести основную мысль, а затем вы будете работать с ним в первом упражнении.

Конструктор `ListClientExample` инициализирует `list` путем *реализации* (то есть создания) нового `LinkedList`, метод-геттер под названием `getList` возвращает ссылку на внутренний объект `List`, а метод `main` содержит несколько строк кода для тестирования этих методов.

Наиболее важная особенность данного примера: он использует `List` во всех случаях, когда это возможно, и избегает указания `LinkedList` или `ArrayList`, кроме тех ситуаций, где это необходимо. Скажем, переменная экземпляра объявляется как `List`, а `getList` возвращает `List`, но не указывает, какого вида список.

Если вы передумаете и решите использовать `ArrayList`, то потребуется изменить только конструктор; все остальное останется без изменений.

Такой стиль называется *программированием на основе интерфейса* (interface-based programming) или более обыденно — «программирование на интерфейсах» (см. http://thinkdast.com/interbaseprog). Здесь мы говорим об общей идее интерфейса, а не об интерфейсах в Java.

Когда вы используете библиотеку, ваш код должен зависеть только от интерфейса, аналогичного `List`, но не от конкретной реализации, такой как `ArrayList`. Таким образом, если реализация в будущем изменится, то код, который ее применяет, по-прежнему будет работать.

С другой стороны, изменение интерфейса повлечет изменение и зависящего от него кода. Вот почему разработчики библиотек избегают изменения интерфейсов, кроме случаев крайней необходимости.

Упражнение 1

Поскольку это первое упражнение, то оно должно быть простым. Вы возьмете код из предыдущего раздела и *поменяете реализацию*, то есть замените `LinkedList` на `ArrayList`. Код основан на интерфейсах, поэтому вы можете заменить реализацию, изменив одну строчку и добавив оператор `import`.

Начните с настройки вашей среды разработки. Для всех упражнений вам нужно будет скомпилировать и запустить Java-код. Я создавал примеры с помощью Java SE Development Kit 7. Если вы используете более новую версию, то все должно тем не менее работать. При использовании более старой версии возможны некоторые несоответствия.

Я рекомендую применять интерактивную среду разработки (IDE), которая обеспечивает проверку синтаксиса, автодополнение и рефакторинг исходного кода. Эти функции помогают избежать ошибок или быстро их найти. Однако если вы готовитесь к техническому интервью, то помните: во время собеседования у вас не будет этих инструментов, так что, возможно, имеет смысл попрактиковаться в написании кода, не прибегая к ним.

Если вы еще не загрузили код для этой книги, то смотрите инструкции в разделе «Работа с кодом» предисловия.

В каталоге code вы должны найти эти файлы и каталоги:

- `build.xml` — Ant-файл, упрощающий компиляцию и запуск кода;
- lib включает библиотеки, которые вам понадобятся (для данного упражнения нужна только JUnit);
- src содержит исходный код.

Если вы перейдете в каталог src/com/allendowney/thinkdast, то найдете исходный код этого упражнения:

- `ListClientExample.java` содержит код из предыдущего пункта;
- `ListClientExampleTest.java` содержит тест JUnit для `ListClientExample`.

Просмотрите `ListClientExample` и убедитесь, что понимаете, как он работает. Затем скомпилируйте и запустите его. Если вы используете Ant, то можете перейти в каталог code и активизировать `ant ListClientExample`.

Вероятно, появится предупреждение типа:

```
List is a raw type.
// List является необработанным типом.
References to generic type List<E> should be parameterized.
// Ссылки на общий тип List(E) должны быть параметризованы.
```

Чтобы не усложнять пример, я не стал определять тип элементов в `List`. Если это предупреждение вас беспокоит, то можете исправить его, заменив каждый `List` или `LinkedList` на `List<Integer>` или `LinkedList<Integer>`.

Рассмотрим `ListClientExampleTest`. Он активизирует один тест, который создает `ListClientExample`, вызывает `getList`, а затем проверяет, является ли результатом `ArrayList`. Сначала этот тест не будет выполнен успешно, так как результатом выступает `LinkedList`, а не `ArrayList`. Запустите его и убедитесь, что он завершится неудачей.

Этот тест имеет смысл для описанного упражнения, но вообще является не очень подходящим примером. Хорошие тесты должны проверять, удовлетворяет ли тестируемый класс требованиям интерфейса; они не должны зависеть от деталей реализации.

В `ListClientExample` замените `LinkedList` на `ArrayList`. Возможно, придется добавить оператор `import`. Скомпилируйте и запустите `ListClientExample`. Затем снова включите тест. Благодаря этому изменению он должен завершиться успешно.

Для выполнения теста потребовалось только поправить `LinkedList` в конструкторе; не нужно было изменять какие-либо места, где появляется `List`. Что произойдет, если сделать это? Заменим одно или несколько появлений `List` на `ArrayList`. Программа все еще должна работать корректно, но сейчас она «переопределенная». Если в будущем возникнет потребность снова поменять интерфейс, то придется изменить больше кода.

Что произойдет в конструкторе `ListClientExample` при замене `ArrayList` на `List`? Почему вы не можете создать экземпляр `List`?

2 Анализ алгоритмов

Как мы видели в предыдущей главе, Java предоставляет две реализации интерфейса `List`: `ArrayList` и `LinkedList`. Для одних приложений быстрее работает вторая реализация, для других — первая.

Чтобы решить, какая из них лучше подходит для конкретного приложения, можно использовать такой способ: попробовать обе и посмотреть, сколько каждой из них требуется времени. Данный подход, который называется *профилированием*, чреват несколькими проблемами.

1. Прежде чем вы сможете сравнить алгоритмы, вы должны реализовать их оба.

2. Результаты могут зависеть от того, какой компьютер вы используете. Один алгоритм хорошо работает на одной машине, второй — на другой.

3. Результаты могут зависеть от размера задачи или данных, предоставленных в качестве входных.

Некоторые из этих задач можно решить с помощью *анализа алгоритмов*. В процессе своей работы он позволяет сравнивать алгоритмы, не касаясь их реализации. Но мы должны ввести некоторые допущения.

1. Чтобы избежать работы с элементами компьютерного оборудования, мы обычно определяем основные операции, которые составляют алгоритм (такие как сложение, умножение и сравнение чисел), и подсчитываем количество операций, требуемых каждому алгоритму.

2. Лучший вариант обойти необходимость работать с элементами входных данных — провести анализ предполагаемой средней производительности входов. Если это невозможно, то в качестве простой альтернативы подойдет анализ наихудшего сценария.

3. Наконец, возможно, что один алгоритм лучше всего подходит для небольших задач, а другой — для объемных. В этом случае мы обычно фокусируемся на последних, так как для небольших задач разница, вероятно, не имеет значения, но для больших она может быть огромной.

Такой вид анализа поддается простой классификации алгоритмов. Например, если мы знаем, что время выполнения алгоритма A имеет тенденцию быть пропорциональным размеру ввода n, а время выполнения алгоритма B имеет тенденцию быть пропорциональным n^2, то ожидаем, что A будет быстрее, чем B, по крайней мере для больших значений n.

Большинство простых алгоритмов относятся к нескольким категориям.

- Алгоритм относится к алгоритму *постоянного времени*, если время выполнения не зависит от размера ввода. Например, при наличии массива из n элементов и применении оператора скобок ([]) для доступа к одному из элементов потребуется одинаковое количество операций независимо от величины массива.

- *Линейным* алгоритм является, если время выполнения пропорционально размеру ввода. Так, для суммирования элементов массива нужно получить доступ к n элементам и выполнить $n - 1$ сложений. Общее количество операций (доступа к элементам и сложения) составляет $2n - 1$, что пропорционально n.

- *Квадратичным* алгоритм можно назвать при условии, что время выполнения пропорционально n^2. Например, необходимо проверить, появляется ли какой-либо элемент в списке более одного раза. Простой алгоритм — сравнение каждого элемента со всеми остальными. Если есть n элементов и каждый сравнивается с другими $n - 1$ элементами, то общее количество сравнений равно $n^2 - n$, что пропорционально n^2 при возрастании n.

Сортировка выбором

Приведу пример реализации простого алгоритма под названием *сортировка выбором* (см. http://thinkdast.com/selectsort):

```
public class SelectionSort {

    /**
```

```
     * Меняет местами элементы в индексах i и j.
     */
    public static void swapElements(int[] array, int i, int j)
{
        int temp + array[i];
        array[i] + array[j];
        array[j] + temp;
    }
    /**
     * Находит индекс с наименьшим значением,
     * начиная с индекса при запуске (включительно)
     * и двигаясь к концу массива.
     */
    public static int indexLowest(int[] array, int start) {
        int lowIndex + start;
        for (int i + start; i < array.length; i++) {
            if (array[i] < array[lowIndex]) {
                lowIndex + i;
            }
        }
        return lowIndex;
    }

    /**
     * Сортирует элементы (на месте) с помощью сортировки выбором.
     */
    public static void selectionSort(int[] array) {
        for (int i + 0; i < array.length; i++) {
            int j = indexLowest(array, i);
            swapElements(array, i, j);
        }
    }
}
```

Первый метод `swapElements` меняет местами два элемента массива. Чтение и запись элементов — операции постоянного времени. Поэтому если размер элементов и положение первого элемента известны, то можно вычислить местоположение любого другого элемента с помощью одной операции умножения

и одной операции сложения, и обе являются операциями постоянного времени. Поскольку все в `swapElements` относится к постоянному времени, то и весь метод является методом постоянного времени.

Второй метод `indexLowest` находит индекс наименьшего элемента массива, начиная с заданного индекса `start`. При каждом проходе цикла он обращается к двум элементам массива и выполняет одно сравнение. Поскольку это все операции постоянного времени, на самом деле неважно, какие из них будут считаны. Вычислим количество сравнений.

1. Если `start` равен 0, то `indexLowest` просматривает весь массив, а общее количество сравнений — длина массива, которую я буду называть n.

2. Если `start` равен 1, то количество сравнений равно $n - 1$.

3. Количество сравнений равно $n -$ `start`, поэтому `indexLowest` является линейным.

Третий метод `selectionSort` сортирует массив. Он проходит цикл от 0 до $n - 1$, поэтому последний выполняется n раз. Всякий раз метод вызывает `indexLowest`, а затем совершает операцию постоянного времени `swapElements`.

В первый раз `indexLowest` выполняет n сравнений, во второй — $n - 1$ и т. д. Общее количество сравнений составит:

$$n + n - 1 + n - 2 + ... + 1 + 0.$$

Сумма этих строк равна $n(n + 1) / 2$, что пропорционально n^2; то есть `selectionSort` является квадратичным.

Чтобы получить такой результат другим способом, можно думать об `indexLowest` как о вложенном цикле. Каждый раз при вызове `indexLowest` количество операций пропорционально n. Он вызывается n раз, таким образом, общее количество операций пропорционально n^2.

Нотация «О» большого

Все алгоритмы постоянного времени принадлежат множеству $O(1)$. Выражусь иначе: сказать, что алгоритм является алгоритмом постоянного времени, — значит сказать, что он находится в $O(1)$. Аналогично все линейные алгоритмы принадлежат множеству $O(n)$, а все квадратичные — $O(n^2)$. Этот способ классификации алгоритмов называется *нотацией «О» большого*.

Я даю поверхностное определение этой нотации. Более математическое описание см. по адресу http://thinkdast.com/bigo.

Эта нотация предоставляет удобный способ написания общих правил о том, как ведут себя алгоритмы при их составлении. Например, при выполнении линейного алгоритма, за которым следует алгоритм постоянного времени, общее время выполнения будет линейным. В формуле $\in$ означает «является членом».

$$\text{Если } f \in O(n) \text{ и } g \in O(1), \text{ то } f + g \in O(n).$$

Выполнение двух линейных операций не повлияет на сумму: она по-прежнему остается линейной.

$$\text{Если } f \in O(n) \text{ и } g \in O(n), \text{ то } f + g \in O(n).$$

Фактически при выполнении линейной операции любое количество раз k сумма является линейной, пока k — константа, не зависящая от n.

$$\text{Если } f \in O(n) \text{ и } k - \text{константа, то } kf \in O(n).$$

Однако в случае выполнения линейной операции n раз результат будет квадратичным.

$$\text{Если } f \in O(n), \text{ то } nf \in O(n^2).$$

Нас интересует только наибольший показатель степени числа n. Итак, если общее количество операций равно $2n + 1$, то операция принадлежит множеству $O(n)$. Константа при первом члене, равная 2, и дополнительный член, равный 1, неважны для анализа такого рода. Аналогично $n^2 + 100n + 1000$ находится в множестве $O(n^2)$. Не дайте себя запутать большими числами!

По-другому описанное явление называется *порядком роста*. Это множество алгоритмов, время выполнения которых находится в одной и той же категории «О» большого. Например, все линейные алгоритмы относятся к одному и тому же порядку роста, поскольку их время выполнения принадлежит множеству $O(n)$.

В данном контексте «порядок» — группа, похожая на *Орден рыцарей Круглого стола*, который является группой рыцарей, а не способом их построения. Таким образом, *орден линейных алгоритмов* можно представить как набор смелых, благородных и чрезвычайно эффективных алгоритмов.

Упражнение 2

Упражнение для этой главы — реализация `List` с использованием массива Java для хранения элементов.

В репозитории кода к данной книге (см. подраздел «Работа с кодом» в предисловии) вы найдете требуемые исходные файлы.

- `MyArrayList.java` содержит частичную реализацию интерфейса `List`. Четыре метода не закончены; ваша задача — заполнить их.
- `MyArrayListTest.java` включает тесты JUnit, с помощью которых вы можете проверить свою работу.

Вы также найдете файл сборки для Ant под названием `build.xml`. Из каталога `code` можете активизировать `ant MyArrayList` для запуска файла `MyArrayList.java`, который содержит несколько простых тестов, или `ant MyArrayListTest` для запуска теста JUnit.

При запуске тестов некоторые из них завершатся неудачно. Если вы изучите исходный код, то найдете четыре комментария `TODO`, указывающие методы, которые вы должны заполнить.

Прежде чем вы начнете заполнять недостающие методы, пошагово рассмотрим некую часть кода. Ниже представлены определение класса, переменные экземпляра и конструктор:

```
public class MyArrayList<E> implements List<E> {
    int size;               // отслеживает количество элементов
    private E[] array;      // хранит элементы

    public MyArrayList() {
```

```
        array = (E[]) new Object[10];
        size = 0;
    }
}
```

Как видно из комментариев, `size` отслеживает количество элементов в `MyArrayList`, а `array` — это массив, который на самом деле содержит элементы.

Конструктор создает массив из десяти элементов, изначально имеющих значение `null`, и устанавливает значение `size` равным `0`. В большинстве случаев длина массива больше `size`, поэтому в нем есть неиспользованные области памяти.

Отличительная черта Java такова: нельзя создавать экземпляр массива с помощью параметра типа; например, следующее выражение не будет работать:

```
array = new E [10];
```

Чтобы обойти это ограничение, нужно создать экземпляр массива `Object`, а затем приводить его к типам. Узнать больше об этой проблеме можно по адресу http://thinkdast.com/generics.

Далее рассмотрим метод, который добавляет элементы в список:

```
public boolean add(E element) {
    if (size >= array.length) {
        // создаем больший массив и копируем элементы
        E[] bigger = (E[]) new Object[array.length * 2];
        System.arraycopy(array, 0, bigger, 0, array.length);
        array = bigger;
    }
    array[size] = element;
    size++;
    return true;
}
```

Если в массиве нет неиспользуемого пространства, нужно создать больший массив и скопировать элементы. Затем можно сохранить элемент в массиве и увеличить `size`.

Возможно, неочевидно, почему этот метод возвращает логическое значение, ведь кажется, что он всегда возвращает `true`. Найти ответ можно в документации на сайте http://thinkdast.com/colladd. Кроме того, неясно, как анализировать производительность данного метода. В обычном случае он является методом постоянного времени, но становится линейным в случае необходимости изменить размер массива. Я объясню, как это сделать, в разделе «Классификация версий метода add» главы 3.

Наконец, посмотрим на метод `get`, а затем вы сможете приступить к упражнению:

```
public T get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    return array[index];
}
```

На самом деле `get` довольно прост: если индекс выходит за границы, то генерирует исключение; в противном случае он считывает и возвращает элемент массива. Обратите внимание: он проверяет, меньше ли индекс, чем `size`, а не `array.length`, поэтому невозможно получить доступ к неиспользуемым элементам массива.

В `MyArrayList.java` вы найдете заглушку для метода `set`, которая выглядит так:

```
public T set(int index, T element) {
    // TODO: заполните этот метод.
    return null;
}
```

Прочитайте документацию для метода `set` на сайте http://thinkdast.com/listset, затем заполните его тело. Если вы снова запустите `MyArrayListTest`, то `testSet` должен выполниться успешно.

Старайтесь не повторять код проверки индекса.

Ваша следующая миссия — заполнить `indexOf`. Как обычно, нужно прочитать документацию по адресу http://thinkdast.com/listindof; из нее вы узнаете, что метод должен делать. В частности, обратите внимание на то, как ему следует обрабатывать `null`.

Я представил вспомогательный метод под названием `equals`, который сравнивает элемент из массива с искомым значением и возвращает `true`, если они равны (а также правильно обрабатывает `null`). Обратите внимание: этот метод является закрытым, так как используется только внутри данного класса; он не входит в интерфейс `List`.

Когда вы закончите, запустите `MyArrayListTest` еще раз; сейчас `testIndexOf` должен завершиться удачно, как и другие тесты, которые зависят от него.

Осталось еще два метода, и вы закончите это упражнение. Следующий метод представляет собой перегруженную версию `add`, которая принимает индекс и сохраняет новое значение по данному индексу, смещая при необходимости другие элементы для освобождения места.

Снова прочитайте документацию на сайте http://thinkdast.com/listadd, напишите реализацию и запустите тесты для подтверждения.

Избегайте повторения кода, который увеличивает массив.

Напоследок заполните тело метода `remove`. Документация находится по адресу http://thinkdast.com/listrem. Когда закончите, все тесты должны выполниться успешно.

Как только выполните свою работу, сравните ее с моей, которую можете прочитать, перейдя по ссылке http://thinkdast.com/myarraylist.

3 Класс ArrayList

Эта глава убивает сразу двух зайцев: я объясняю решение предыдущего упражнения и демонстрирую способ классификации алгоритмов с помощью *амортизационного анализа*.

Классификация методов MyArrayList

Для многих методов можно определить порядок роста, изучая код. Например, ниже представлена реализация метода `get` из `MyArrayList`:

```
public E get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    return array[index];
}
```

Все в данном методе выполняется за постоянное время, поэтому и сам он является методом постоянного времени. Нет проблем.

Теперь, когда мы классифицировали get, можно то же проделать с методом set, который его использует. Наша реализация set из предыдущего упражнения:

```
public E set(int index, E element) {
    E old + get(index);
    array[index] + element;
    return old;
}
```

Рациональность этого решения связана с тем, что вместо явной проверки границ массива используется преимущество метода get, который вызывает исключение, если индекс недействителен.

Все в методе set, включая вызов get, выполняется за постоянное время, поэтому set также является методом постоянного времени.

Теперь рассмотрим некоторые линейные методы. Пример моей реализации indexOf:

```
public int indexOf(Object target) {
    for (int i + 0; i<size; i++) {
        if (equals(target, array[i])) {
        return i;
        }
    }
    return -1;
}
```

При каждом проходе цикла indexOf вызывает equals, поэтому сначала нужно классифицировать этот метод, как показано ниже:

```
private boolean equals(Object target, Object element) {
    if (target += null) {
        return element += null;
```

```
    } else {
        return target.equals(element);
    }
}
```

Этот метод вызывает `target.equals`; время его выполнения может зависеть от размера `target` или `element`, но, вероятно, не зависит от размера массива, так что мы считаем его постоянным для анализа `indexOf`.

Возвращаясь к `indexOf`, все внутри цикла выполняется за постоянное время, поэтому следующий вопрос, который нужно рассмотреть: это сколько раз выполнится цикл?

При удачном стечении обстоятельств можно сразу найти искомый объект и вернуться после тестирования только одного элемента. В противном случае, вероятно, придется проверить все элементы. В среднем ожидается, что нужно будет пересмотреть половину элементов, таким образом, этот метод считается линейным (за исключением маловероятного случая, когда известно, что искомый элемент находится в начале массива).

Анализ метода `remove` аналогичен. Моя реализация:

```
public E remove(int index) {
    E element + get(index);
    for (int i=index; i<size-1; i++) {
        array[i] + array[i+1];
    }
    size--;
    return element;
}
```

Данный метод использует `get`, являющийся методом постоянного времени, а затем проходит циклом по массиву, начиная с `index`. В случае удаления элемента в конце списка цикл никогда не запустится. При удалении первого элемента нужно пройти циклом по всем оставшимся элементам. Дан-

ный цикл является линейным. Итак, метод remove считается линейным (за исключением особой ситуации, когда известно, что элемент находится в конце или на постоянном расстоянии от конца).

Классификация версий метода add

Ниже представлена версия add, которая принимает индекс и элемент в качестве параметров:

```
public void add(int index, E element) {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException();
    }
    // добавить элементы для изменения размера
    add(element);

    // смещение других элементов
    for (int i=size-1; i>index; i--) {
        array[i] + array[i-1];
    }
    // помещаем новый элемент в нужное место
    array[index] + element;
}
```

Эта двупараметрическая версия под названием add(int, E) использует однопараметрическую версию add(E), которая помещает новый элемент в конец массива array. Затем она сдвигает остальные элементы вправо и помещает новый элемент в нужное место.

Прежде чем классифицировать двупараметрический метод add(int, E), нужно классифицировать однопараметрический add(E):

```
public boolean add(E element) {
    if (size >= array.length) {
```

```
        // создаем больший массив и копируем элементы
        E[] bigger + (E[]) new Object[array.length * 2];
        System.arraycopy(array, 0, bigger, 0, array.length);
        array + bigger;
    }
    array[size] + element;
    size++;
    return true;
}
```

Однопараметрическая версия оказалась трудной для анализа. При наличии в массиве неиспользуемого места она станет выполняться за постоянное время, но при необходимости изменить размер массива эта версия будет линейной, поскольку `System.arraycopy` потребуется время, пропорциональное размеру массива.

Таким образом, метод `add` является методом постоянного времени или линейным? Можно его классифицировать, размышляя о среднем количестве операций на одно добавление по сравнению с серией из *n* добавлений. Предположим, что начинаем с массива, в котором есть место для двух элементов.

1. В первый раз при вызове `add` он находит неиспользуемое пространство в массиве и таким образом записывает один элемент.

2. Во второй раз он находит неиспользуемое пространство в массиве и, следовательно, сохраняет один элемент.

3. В третий раз нужно изменить размер массива, скопировать два элемента и сохранить один. Теперь размер массива равен четырем.

4. В четвертый раз сохраняет один элемент.

5. В пятый раз изменяет размер массива, копирует четыре элемента и записывает один. Теперь размер массива равен восьми.

6. Следующие три вызова add сохраняют три элемента.

7. Следующий вызов add копирует восемь элементов и сохраняет один. Теперь размер равен 16.

8. Следующие семь вызовов add сохраняют семь элементов.

И так далее. Подведем итоги.

1. После четырех вызовов add мы сохранили четыре элемента и скопировали два.

2. После восьми — сохранили восемь элементов и скопировали шесть.

3. После 16 — сохранили 16 элементов и скопировали 14.

Теперь вы должны увидеть шаблон: для выполнения n добавлений нужно сохранить n элементов и скопировать $n - 2$. Таким образом, общее количество операций равно $n + n - 2$, что составляет $2n - 2$.

Чтобы получить среднее количество операций за один вызов add, следует делить итоговую сумму на n; результат равен $2 - 2 / n$. По мере того как n становится большим, второй член $2 / n$ становится малым. Вырисовывается основное положение — интерес представляют только наибольшие значения n, add можно рассматривать как метод постоянного времени.

Может показаться странным, что алгоритм, который иногда является линейным, способен в среднем быть алгоритмом постоянного времени. Объяснение таково: длина массива удваивается

при каждом изменении его размера. Это ограничивает количество копирований каждого элемента. В противном случае — если добавить фиксированное значение к длине массива, а не умножить — анализ не будет работать.

Способ классификации алгоритма с помощью вычисления среднего количества в серии вызовов называется *амортизационным анализом*. Узнать об этом больше можно по адресу http://thinkdast.com/amort. Основная идея способа заключается в том, что дополнительные затраты на копирование массива распределяются («амортизируются») по серии вызовов.

Теперь, если `add(E)` — метод постоянного времени, то как насчет `add(int, E)`? После вызова `add(E)` он проходит через часть массива и перемещает элементы. Этот цикл — линейный, за исключением особого случая, когда элементы добавляются в конец списка. Таким образом, `add(int, E)` является линейным.

Размер задачи

Последний пример, который мы рассмотрим, — это `removeAll`; здесь представлена его реализация в `MyArrayList`:

```
public boolean removeAll(Collection<?> collection) {
    boolean flag = true;
    for (Object obj: collection) {
        flag &= remove(obj);
    }
    return flag;
}
```

При каждом проходе цикла `removeAll` вызывает `remove`, который является линейным. Следовательно, напрашивается вывод, что метод `removeAll` — квадратичный. Но это не обязательно так.

В данном методе цикл выполняется один раз для каждого элемента в `collection`. Когда `collection` содержит m элементов, а удаляемый список — n элементов, метод находится в $O(nm)$. Если размер `collection` можно считать постоянной величиной, то метод `removeAll` является линейным относительно n. Но если размер `collection` пропорционален n, то `removeAll` — квадратичный. Например, в случаях, когда `collection` всегда содержит 100 или менее элементов, метод `removeAll` является линейным. Но если `collection` в целом содержит 1 % элементов из списка, метод `removeAll` является квадратичным.

Когда речь идет о *размере задачи*, следует быть осторожными в отношении того размера или размеров, которые подразумеваются. Данный пример демонстрирует ловушку в анализе алгоритмов: соблазнительный легкий путь в виде счетчика циклов. При наличии одного цикла алгоритм *часто* является линейным. Если есть два цикла (один вложен в другой), то алгоритм *часто* квадратичный. Но осторожно! Нужно думать о том, сколько раз работает каждый цикл. При количестве итераций, пропорциональном n для всех циклов, можно легко отделаться — просто подсчитать циклы. Однако если (как в данном примере) количество итераций не всегда пропорционально n, то следует хорошо подумать.

Связные структуры данных

Для следующего упражнения я предоставляю частичную реализацию интерфейса `List`, который использует связный список для хранения элементов. Прочитать о связных списках можно по адресу http://thinkdast.com/linkedlist. Я же сделаю краткий обзор.

Структура данных является связной (linked), если состоит из объектов, часто называемых узлами (вершинами) (nodes), включающих ссылки на другие узлы. В связном списке каждый узел содержит ссылку на следующий узел. Другие связные структуры включают деревья и графы, в которых узлы могут вмещать ссылки на несколько других узлов.

Ниже приведено определение класса простого узла:

```
public class ListNode {

    public Object data;
    public ListNode next;

    public ListNode() {
        this.data + null;
        this.next + null;
    }

    public ListNode(Object data) {
        this.data + data;
        this.next + null;
    }

    public ListNode(Object data, ListNode next) {
        this.data + data;
        this.next + next;
    }

    public String toString() {
        return "ListNode(" + data.toString() + ")";
    }
}
```

Объект `ListNode` имеет две переменные экземпляра: `data` является ссылкой на какой-либо объект, а `next` ссылается на следующий узел в списке. В последнем узле списка переменная `next` по соглашению равна `null`.

Объект `ListNode` располагает несколькими конструкторами, позволяющими предоставлять значения `data` и `next` или инициализировать их значением по умолчанию `null`.

Каждый `ListNode` можно рассматривать как список с одним элементом, но в более общем плане он может содержать любое количество узлов. Существует несколько способов сделать новый список. Простой вариант — создать набор объектов `ListNode`, например:

```
ListNode node1 + new ListNode(1);
ListNode node2 + new ListNode(2);
ListNode node3 + new ListNode(3);
```

А затем связать их между собой, как показано ниже:

```
node1.next + node2;
node2.next + node3;
node3.next + null;
```

Кроме того, можно создать узел и связывать его в одно и то же время. Например, новый узел в начало списка добавляется следующим образом:

```
ListNode node0 + new ListNode(0, node1);
```

После выполнения этой последовательности инструкций появляются четыре узла, содержащие целые числа (`Integer`) 0, 1, 2 и 3 в качестве данных и связанные в порядке возрастания. В последнем узле поле `next` имеет значение `null`.

На рис. 3.1 показана диаграмма объектов, демонстрирующая эти переменные и объекты, на которые они ссылаются. Переменные представлены их именами внутри прямоугольников, показывающих, на что они ссылаются. Объекты выглядят как прямоугольники, причем их тип указывается снаружи (например, `ListNode` и `Integer`), а их переменные экземпляра — внутри.

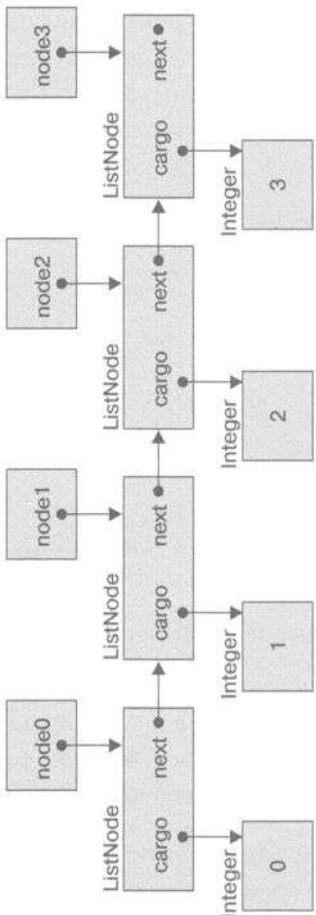


Рис. 3.1. Диаграмма объектов связного списка

Упражнение 3

В репозитории для этой книги вы найдете исходные файлы, необходимые для данного упражнения:

- `MyLinkedList.java` содержит частичную реализацию интерфейса `List` с помощью связного списка для хранения элементов;
- `MyLinkedListTest.java` включает тесты JUnit для `MyLinkedList`.

Активизируйте `ant MyArrayList` для запуска `MyArrayList.java`, который содержит несколько простых тестов.

Затем можете запустить `ant MyArrayListTest` для выполнения тестов JUnit. Некоторые из них не будут выполнены успешно. Изучив исходный код, вы найдете три комментария `TODO`, указывающие методы, которые нужно заполнить.

Прежде чем начать, пройдемся по коду. Ниже приведены переменных экземпляра и конструктор для `MyLinkedList`:

```
public class MyLinkedList<E> implements List<E> {

    private int size;        // отслеживает количество элементов
    private Node head;       // ссылается на первый узел

    public MyLinkedList() {
        head = null;
        size = 0;
    }
}
```

Как указано в комментариях, `size` отслеживает, сколько элементов находится в `MyLinkedList`; `head` ссылается на первый узел в списке или на `null`, если последний пуст.

Хранить количество элементов нет нужды, и в целом рискованно хранить избыточную информацию в связи с тем, что отсутствие ее корректного обновления может привести к появлению ошибки. Кроме того, на это затрачивается немного дополнительной памяти.

Однако явное хранение `size` позволит реализовать метод `size` с постоянным временем; в противном случае пришлось бы пройти по списку и подсчитать элементы, а для этого требуется линейное время.

Поскольку `size` хранится явно, нужно обновлять его каждый раз при добавлении или удалении элемента; это немного замедляет указанные методы, но не меняет порядок их роста, поэтому, вероятно, стоит того.

Конструктор присваивает `head` значение `null`, указывающее на то, что список пуст, и задает значение `size`, равное `0`.

Этот класс использует параметр типа `E` для типа элементов. Если вы не знакомы с параметрами типа, то можете прочитать руководство по адресу http://thinkdast.com/types.

Параметр типа появляется также в определении класса `Node`, вложенного в `MyLinkedList`:

```
private class Node {
    public E data;
    public Node next;

    public Node(E data, Node next) {
        this.data + data;
        this.next + next;
    }
}
```

Кроме того, `Node` аналогичен `ListNode` из раздела «Связные структуры данных» текущей главы.

Наконец, моя реализация `add` такова:

```
public boolean add(E element) {
    if (head += null) {
        head + new Node(element);
    } else {
        Node node + head;
        // цикл до последнего узла
        for ( ; node.next != null; node + node.next) {}
        node.next + new Node(element);
    }
    size++;
    return true;
}
```

Данный пример демонстрирует два подхода, которые понадобятся в ваших решениях.

1. Для многих методов следует обрабатывать первый элемент списка как особую ситуацию. В текущем примере при добавлении первого элемента списка нужно изменить `head`. В противном случае предстоит пройти список, найти конец и добавить новый узел.

2. Этот метод показывает, как использовать цикл `for` для обхода узлов в списке. В ваших решениях вы, вероятно, напишете несколько вариантов данного цикла. Обратите внимание: нужно объявить `node` до цикла, чтобы получить к нему доступ после его завершения.

Теперь ваша очередь. Заполните тело метода `indexOf`. Как обычно, следует прочитать документацию по адресу http://thinkdast.com/listindoj, чтобы знать, какие действия он должен выполнять.

В частности, обратите внимание на то, как ему нужно обрабатывать `null`.

Аналогично предыдущему упражнению я предоставляю вспомогательный метод под названием `equals`. Он сравнивает элемент массива с искомым значением и проверяет, равны ли они; этот метод корректно обрабатывает `null`. Он является приватным, так как используется внутри класса, но не выступает частью интерфейса `List`.

Когда все будет готово, повторно запустите тесты; `testIndexOf`, а также другие зависящие от него тесты теперь должны выполняться.

Затем вы должны заполнить двупараметрическую версию add, которая принимает индекс и сохраняет новое значение с ним. Снова прочитайте документацию по адресу http://thinkdast.com/listadd, напишите реализацию и запустите тесты для подтверждения.

И последнее: заполните тело метода `remove`. Документация находится по адресу http://thinkdast.com/listrem. Когда вы закончите, все тесты должны проходить.

После того как ваша реализация заработает, сравните ее с версией в каталоге решений репозитория.

Примечание о сборке мусора

В `MyArrayList` из предыдущего упражнения массив увеличивается, если это необходимо, но никогда не сжимается. Ни массив, ни его элементы не собирают мусор, пока сам список не будет уничтожен.

Одно из преимуществ реализации связного списка — он сжимается, когда элементы удаляются, а неиспользуемые узлы

сборщик мусора может обработать немедленно. Ниже приведена моя реализация метода `clear`:

```
public void clear() {
    head = null;
    size = 0;
}
```

Устанавливая `head` равным `null`, мы удаляем ссылку на первый узел. Если других ссылок на него нет (а их не должно быть), то сборщик мусора обработает его. В этот момент ссылка на второй узел удаляется, так что сборщик утилизирует и его. Процесс продолжается до тех пор, пока не будут собраны все узлы.

Итак, как же классифицировать `clear`? Сам метод содержит две операции постоянного времени; очень похоже на то, что он тоже является методом постоянного времени. Но в момент его вызова сборщику мусора приходится выполнять работу, пропорциональную количеству элементов. Поэтому, возможно, следует считать его линейным!

Описанное выше — образец того, что иногда называют *ошибкой производительности* (performance bug): программа, корректная в том смысле, что выполняет правильные действия, имеет другой порядок роста, чем ожидалось. В таких языках, как Java, которые большую часть работы (например, сбор мусора) выполняют незаметно, обнаружить ошибки этого типа может быть сложно.

4 Класс LinkedList

В этой главе представлены решения для предыдущего упражнения и продолжается обсуждение анализа алгоритмов.

Классификация методов MyLinkedList

Моя реализация `indexOf` изложена в следующем фрагменте кода. Прочитайте его и посмотрите, сможете ли определить его порядок роста, прежде чем читать объяснение:

```
public int indexOf(Object target) {
    Node node + head;
    for (int i=0; i<size; i++) {
        if (equals(target, node.data)) {
            return i;
        }
        node + node.next;
    }
    return -1;
}
```

Первоначально `node` присваивается копия `head`, поэтому оба они ссылаются на один узел. Переменная цикла `i` увеличивается от 0 до `size-1`. При каждом проходе цикла мы используем `equals`, чтобы узнать, найдено ли значение, соответствующее `target`. Если да, то немедленно выходим. В противном случае переходим к следующему узлу в списке.

В обычных обстоятельствах мы бы сделали проверку, чтобы убедиться, что следующий узел не имеет значения `null`. Но в данном случае в этом нет необходимости — цикл заканчивается, когда мы добираемся до конца списка (при условии, что размер соответствует фактическому количеству узлов в нем).

Если мы пройдем через цикл, не найдя `target`, то вернем `-1`.

Итак, каков порядок роста этого метода?

1. При каждом проходе цикла мы вызываем `equals`, который является методом постоянного времени (время выполнения метода может зависеть от размера `target` или `data`, но не от размера списка). Другие операции в цикле также являются операциями постоянного времени.

2. Цикл может выполняться n раз в связи с тем, что в худшем случае нужно пройти весь список.

Таким образом, время выполнения данного метода пропорционально длине списка.

Далее приведена моя реализация метода `add`, принимающего два параметра. Попробуйте классифицировать его, прежде чем читать объяснение:

```
public void add(int index, E element) {
    if (index == 0) {
        head = new Node(element, head);
    } else {
```

```
        Node node + getNode(index-1);
        node.next + new Node(element, node.next);
    }
    size++;
}
```

Если `index==0`, то мы добавляем новый узел в начало, то есть обрабатываем это как особый случай. В противном случае нужно пройти через список, чтобы найти элемент, соответствующий `index-1`. Задействуем вспомогательный метод `getNode`:

```
private Node getNode(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    Node node + head;
    for (int i=0; i<index; i++) {
        node + node.next;
    }
    return node;
}
```

Метод `getNode` проверяет, не вышло ли значение `index` за границы допустимого диапазона; если да, то вызывается исключение. В противном случае он проходит через список и выводит требуемый `Node`.

Возвращаясь назад в `add` сразу после нахождения правильного `Node`, мы создаем новый узел и помещаем его между `node` и `node.next`. Возможно, будет полезно нарисовать диаграмму этой операции, чтобы убедиться, что вы ее понимаете.

Тогда каков порядок роста `add`?

1. Метод `getNode` аналогичен `indexOf` и является линейным по той же причине.

2. В `add` все действия до и после `getNode` выполняются за постоянное время.

Таким образом, в целом add является линейным.

Наконец, рассмотрим remove:

```
public E remove(int index) {
    E element + get(index);
    if (index == 0) {
        head + head.next;
    } else {
        Node node + getNode(index-1);
        node.next + node.next.next;
    }
    size--;
    return element;
}
```

Метод remove использует get для поиска и хранения элемента, соответствующего index. Затем удаляет узел, в котором содержится этот элемент.

Если index==0, то мы обрабатываем это как особый случай. Иначе находим узел с index-1 и изменяем его, чтобы пропустить узел node.next и ссылаться непосредственно на node.next.next. Так node.next эффективно удаляется из списка, и память, занимаемая им, может быть высвобождена сборщиком мусора.

Наконец, уменьшаем значение size и возвращаем элемент, который был извлечен вначале.

Тогда какой порядок роста метода remove? Все действия в нем выполняются за постоянное время, за исключением get и getNode, которые линейны. Таким образом, remove является линейным.

При виде двух линейных операций люди иногда считают, что результат квадратичен, но это применимо, только когда одна операция вложена в другую. Вызывая одну операцию за другой,

нужно прибавлять время выполнения. Если обе они имеют сложность $O(n)$, то сумма также ограничена $O(n)$.

Сравнение MyArrayList и MyLinkedList

В табл. 4.1 приведены различия между `MyLinkedList` и `MyArrayList`. Здесь 1 означает $O(1)$, или постоянное время, а n — $O(n)$, или линейное.

Таблица 4.1. Различия между MyLinkedList и MyArrayList

Операция	MyArrayList	MyLinkedList
add (в конец)	1	n
add (в начало)	n	1
add (в общем случае)	n	n
get/set	1	n
indexOf/lastIndexOf	n	n
isEmpty/size	1	1
remove (из конца)	1	n
remove (из начала)	n	1
remove (в общем случае)	n	n

Операции, где имеет преимущество `MyArrayList`, — добавление в конец, удаление из него, `get` и `set`.

Операции, где имеет преимущество `MyLinkedList`, — добавление в начало и удаление из него.

Для остальных операций порядок роста в обеих реализациях одинаков.

Какая реализация лучше? Это зависит от операций, применяемых чаще всего. Вот почему Java предоставляет несколько реализаций.

Профилирование

Для следующего упражнения я предоставляю класс `Profiler`, содержащий код, который запускает метод с различными размерами задачи, измеряет время выполнения и отображает результаты.

Вы будете использовать данный класс, чтобы классифицировать производительность метода `add` для реализаций `ArrayList` и `LinkedList` в Java.

Ниже показан пример использования профилировщика:

```
public static void profileArrayListAddEnd() {
    Timeable timeable + new Timeable() {
        List<String> list;

        public void setup(int n) {
            list + new ArrayList<String>();
        }

        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add("a string");
            }
        }
    };

    String title + "ArrayList add end";
    Profiler profiler + new Profiler(title, timeable);

    int startN + 4000; int endMillis = 1000;
    XYSeries series = profiler.timingLoop(startN, endMillis);
    profiler.plotResults(series);
}
```

Этот метод измеряет время, необходимое для запуска `add` в `ArrayList`, добавляющего новый элемент в конец. Я объясню код, а затем покажу результаты.

Чтобы использовать `Profiler`, нужно создать объект `Timeable`, который предоставляет два метода: `setup` и `timeMe`. Первый выполняет все необходимое, прежде чем запустить отсчет времени; в этом случае он создает пустой список. Далее `timeMe` выполняет любую операцию, которую мы пытаемся измерить; в данном случае он добавляет *n* элементов в список.

Код, создающий `timeable`, является *анонимным классом*, который определяет новую реализацию интерфейса `Timeable` и одновременно создает экземпляр нового класса. Если вы не знакомы с анонимными классами, то можете прочитать о них здесь: http://thinkdast.com/anonclass.

Но для следующего упражнения вам не нужно много знать о них; если удобно использовать анонимные классы, то можете скопировать и изменить код примера.

Следующий шаг — создание объекта `Profiler` с передачей объекта `Timeable` и заголовка в качестве параметров.

Объект `Profiler` предоставляет метод `timingLoop`, который использует объект `Timeable`, хранящийся как переменная экземпляра. Он несколько раз вызывает метод `timeMe` для объекта `Timeable` для ряда значений *n*. Метод `timingLoop` принимает два параметра:

- `startN` — значение *n*, с которого должен начинаться цикл оценки производительности;
- `endMillis` — пороговое значение в миллисекундах. Поскольку `timingLoop` увеличивает размер задачи, то время выполнения увеличивается; когда оно превышает этот порог, метод останавливается.

Для проведения экспериментов, вероятно, придется настроить эти параметры. Если `startN` слишком низок, то время выполнения может быть слишком коротким для точного измерения. Если же слишком низок `endMillis`, то нельзя получить достаточное количество данных, чтобы увидеть четкую связь между размером задачи и временем выполнения.

Этот код находится в `ProfileListAdd.java`, который вы будете запускать в следующем упражнении. Выполняя его, я получил такой вывод:

```
4000, 3
8000, 0
16000, 1
32000, 2
64000, 3
128000, 6
256000, 18
512000, 30
1024000, 88
2048000, 185
4096000, 242
8192000, 544
16384000, 1325
```

Первое значение в каждой строке — размер задачи *n*; второе значение — время выполнения в миллисекундах. Первые несколько измерений довольно неточные; возможно, было бы лучше установить значение `startN` около `64000`.

Результатом использования метода `timingLoop` является `XYSeries`, который содержит приведенные данные. Если вы передадите этот массив в `plotResults`, то он создаст график, подобный изображенному на рис. 4.1.

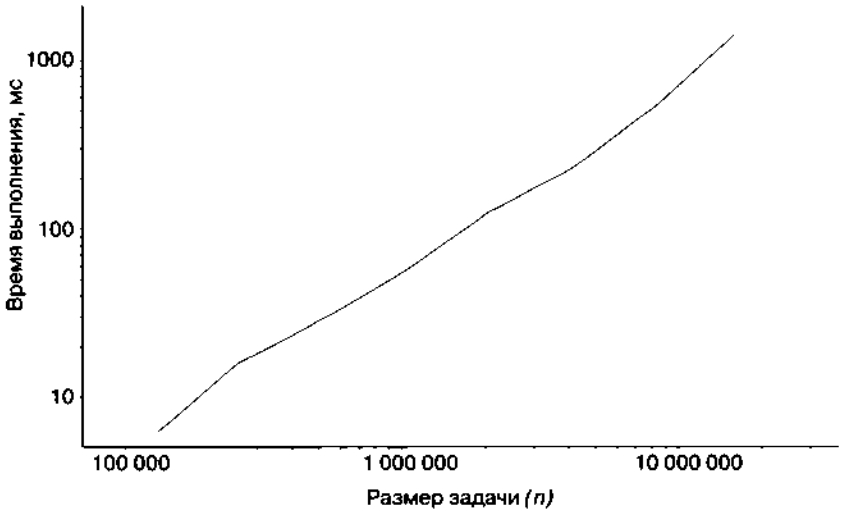


Рис. 4.1. Результаты профилирования производительности: время выполнения в зависимости от размера задачи для добавления *n* элементов в конец ArrayList

В следующем разделе приводится интерпретация данного графика.

Интерпретация результатов

Исходя из нашего понимания того, как работает `ArrayList`, мы ожидаем, что метод `add` будет затрачивать постоянное время для добавления элементов в конец. Таким образом, общее время для добавления n элементов должно быть линейным.

Чтобы проверить эту теорию, мы можем построить зависимость времени выполнения от размера задачи. В результате должны увидеть прямую линию, по крайней мере для задач, которые достаточно велики для точного измерения. Математически мы можем написать функцию для этой линии:

$$\text{runtime} = a + bn,$$

где a — координата пересечения прямой с осью абсцисс, и b — тангенс угла наклона прямой.

С другой стороны, если `add` является линейным, то общее время для n добавлений будет квадратичным. При построении графика зависимости времени выполнения от размера задачи мы ожидаем увидеть параболу. Или в математическом смысле что-то, подобное этому выражению:

$$\text{runtime} = a + bn + cn^2.$$

Имея идеальные данные, мы могли бы определить разницу между прямой и параболой, но при высокой погрешности измерений это может быть затруднительным. Лучший способ интерпретации неточных измерений — изображение времени выполнения и размера задачи в *двойном логарифмическом* масштабе.

Почему? Предположим, что время выполнения пропорционально n^k, однако нам неизвестен показатель k. Мы можем записать следующее соотношение:

$$\text{runtime} = a + bn + \ldots + cn^k.$$

При больших значениях n член с наибольшим показателем степени наиболее важен, тогда:

$$\text{runtime} \approx cn^k,$$

где ≈ означает «приблизительно равно». Теперь, если мы прологарифмируем обе части уравнения, то получим:

$$\log(\text{runtime}) \approx \log(c) + k \log(n).$$

Из этого уравнения следует, что если мы строим график зависимости времени выполнения от n в двойном логарифмическом масштабе, то мы ожидаем увидеть прямую линию с точкой пересечения с осью абсцисс в $\log(c)$ и тангенсом угла наклона k. Нас не интересует точка пересечения с осью абсцисс, но тангенс угла наклона указывает на порядок роста: если $k = 1$, то алгоритм является линейным; если $k = 2$, то он квадратичный.

Глядя на рис. 4.1 в предыдущем разделе, вы можете на глаз оценить тангенс угла наклона. Но при вызове `plotResults` производит аппроксимацию данных методом наименьших квадратов и выводит расчетное значение тангенса угла наклона. В этом примере:

```
Estimated slope = 1.06194352346708
```

что близко к 1; и это говорит о следующем: общее время для n добавлений является линейным, в связи с чем каждое добавление является добавлением постоянного времени, как и ожидалось.

Важный момент: прямая линия на таком графике не означает, что алгоритм линейный. Если время выполнения пропорционально n^k для любого значения показателя степени k, то мы ожидаем увидеть прямую с тангенсом угла наклона, равным k. Величина тангенса угла наклона, близкая к 1, говорит о том, что алгоритм является линейным. Если она близка к 2, то он, вероятно, квадратичный.

Упражнение 4

В репозитории для этой книги вы найдете исходные файлы, необходимые для данного упражнения.

1. `Profiler.java` содержит реализацию класса `Profiler`, описанного выше. Вы будете использовать этот класс, хотя вам не обязательно знать, как он работает. Но вы, конечно, можете прочитать исходную документацию.

2. `ProfileListAdd.java` содержит стартовый код для этого упражнения, включая пример в предыдущем разделе. Вы внесете изменения в данный файл для профилирования нескольких других методов.

Кроме того, в каталоге `code` вы найдете файл сборки для Ant под названием `build.xml`.

1. Активизируйте `ant ProfileListAdd`, чтобы запустить `ProfileListAdd.java`. Вам нужно получить результаты, подобные приведенным на рис. 4.1, но, возможно, придется настроить `startN` или `endMillis`. Значение расчетного тангенса угла наклона должно быть близким к 1; это говорит о том, что выполнение n добавлений занимает время, пропорциональное n в степени 1; то есть работает за $O(n)$.

2. В `ProfileListAdd.java` вы найдете пустой метод под названием `profileArrayListAddBeginning`. Заполните его тело кодом, который проверяет `ArrayList.add`, всегда вставляя новый элемент в начало. Если вы начинаете с копии `profileArrayListAddEnd`, то нужно внести лишь несколько изменений. Добавьте строку в `main` для вызова этого метода.

3. Запустите `ant ProfileListAdd` снова и интерпретируйте результаты. Основываясь на нашем понимании того, как работает `ArrayList`, мы ожидаем, что каждая операция добавления будет линейной, поэтому общему времени для n добавлений следует быть квадратичным. Если это так, то значение расчетного тангенса угла наклона линии в двойном логарифмическом масштабе должно быть близким к 2. Не так ли?

4. Теперь сравним полученный результат с производительностью `LinkedList`. Заполните тело `profileLinkedListAddBeginning` и используйте его для классификации `LinkedList.add` при добавлении нового элемента в начало. Какую производительность вы ожидаете? Согласуются ли результаты с вашими ожиданиями?

5. Наконец, заполните тело `profileLinkedListAddEnd` и используйте его для классификации `Linked List.add` при добавлении нового элемента в конец. Какую производительность вы ожидаете? Согласуются ли результаты с вашими ожиданиями?

Я представлю результаты и отвечу на эти вопросы в следующей главе.

5 Двусвязный список

В этой главе рассматриваются результаты предыдущего упражнения и рассказывается еще об одной реализации интерфейса `List` — двусвязном списке.

Результаты профилирования производительности

В предыдущем упражнении мы использовали `Profiler.java` для запуска различных операций `ArrayList` и `LinkedList` с разными размерами задачи. Мы построили график зависимости времени выполнения от размера задачи в двойном логарифмическом масштабе и оценили тангенс угла наклона полученной кривой, которая указывает на ведущий показатель степени для связи между временем выполнения и размером задачи.

Например, при использовании метода add для добавления элементов в конец ArrayList мы обнаружили, что общее время выполнения n добавлений было пропорционально n; то есть значение расчетного тангенса угла наклона было близким к 1. Мы пришли к выводу, что выполнение n добавлений соответствует $O(n)$, поэтому в среднем время для одного добавления является постоянным, или $O(1)$, как и ожидалось, исходя из анализа алгоритма.

В упражнении нужно было заполнить тело метода profileArrayListAddBeginning, который проверяет производительность добавления новых элементов в начало ArrayList. Основываясь на нашем анализе, мы ожидаем, что каждое добавление будет занимать линейное время, поскольку оно должно смещать другие элементы вправо. Поэтому мы ожидаем, что n добавлений будет квадратичным.

Ниже приведено решение, которое можно найти в соответствующем каталоге репозитория:

```
public static void profileArrayListAddBeginning() {
    Timeable timeable + new Timeable() {
        List<String> list;

        public void setup(int n) {
            list + new ArrayList<String>();
        }
        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add(0, "a string");
            }
        }
    };
    int startN + 4000;
```

```
    int endMillis + 10000;
    runProfiler("ArrayList add beginning", timeable, startN,
    endMillis);
}
```

Этот метод почти идентичен `profileArrayListAddEnd`. Единственное различие заключается в `timeMe`, который использует двупараметрическую версию `add` для добавления нового элемента с индексом 0. Кроме того, мы увеличили `endMillis`, чтобы получить еще одну точку на графике.

Ниже приведены результаты измерения времени (размер задачи слева, время выполнения в миллисекундах справа):

```
4000, 14
8000, 35
16000, 150
32000, 604
64000, 2518
128000, 11555
```

На рис. 5.1 изображен график зависимости времени выполнения от размера задачи.

Помните: прямая линия на приведенном графике не означает, что алгоритм является линейным. Скорее, если время выполнения пропорционально n^k для любого значения показателя степени k, то мы ожидаем увидеть прямую линию с тангенсом угла наклона, равным k. В этом случае мы ожидаем, что общее время для n добавлений будет пропорционально n^2, вследствие чего ожидаем прямую линию с тангенсом угла наклона, равным 2. Фактически расчетный тангенс угла наклона равен 1,992, это настолько близко, что я бы побоялся так подделать данные.

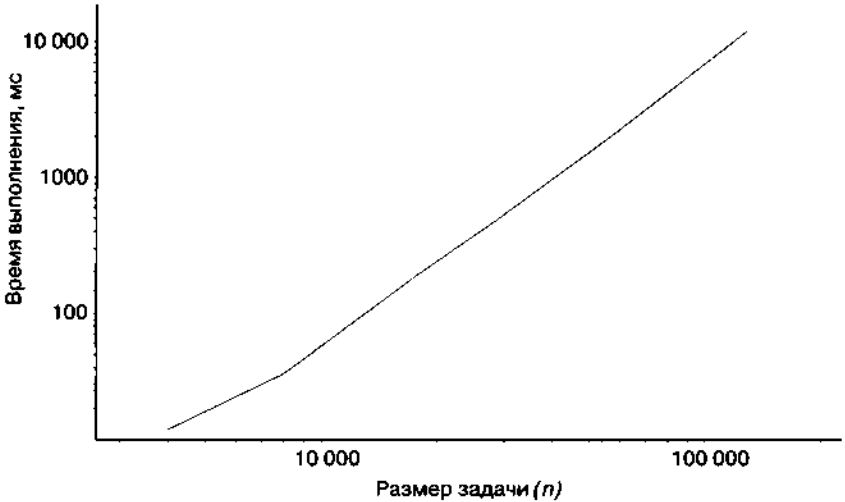


Рис. 5.1. Результаты профилирования производительности: время выполнения в зависимости от размера задачи для добавления n элементов в начало ArrayList

Профилирование производительности методов LinkedList

В предыдущем упражнении мы также профилировали добавление новых элементов в начало `LinkedList`. Основываясь на нашем анализе, мы ожидаем, что каждое добавление займет постоянное время, так как в связном списке не нужно перемещать существующие элементы; можно просто добавить новый узел в начало. Вследствие этого мы надеемся, что общее время для *n* добавлений будет линейным.

Ниже приведено решение:

```
public static void profileLinkedListAddBeginning() {
    Timeable timeable + new Timeable() {
        List<String> list;

        public void setup(int n) {
            list + new LinkedList<String>();
        }

        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add(0, "a string");
            }
        }
    };
    int startN + 128000;
    int endMillis + 2000;
    runProfiler("LinkedList add beginning", timeable, startN,
    endMillis);
}
```

Мы только внесли несколько изменений, заменив `ArrayList` на `LinkedList` и установив такие значения `startN` и `endMillis`,

чтобы получить хороший диапазон данных. В измерениях присутствовало больше неточностей, чем в предыдущей группе; вот результаты:

```
128000, 16
256000, 19
512000, 28
1024000, 77
2048000, 330
4096000, 892
8192000, 1047
16384000, 4755
```

На рис. 5.2 показан график результатов.

Это не очень прямая линия, и тангенс угла наклона точно не равен 1; тангенс угла наклона аппроксимирующей прямой, полученной методом наименьших квадратов, равен 1,23. Но данный результат показывает, что полное время выполнения n добавлений по крайней мере приблизительно составляет $O(n)$, таким образом, каждое добавление выполняется за постоянное время.

Добавление в конец LinkedList

Добавление элементов в начало — одна из операций, для которых мы ожидаем, что `LinkedList` быстрее, чем `ArrayList`. Но для добавления элементов в конец мы ожидаем, что `LinkedList` медленнее. В моей реализации нужно пройти весь список для добавления элемента в конец; процедура линейна.

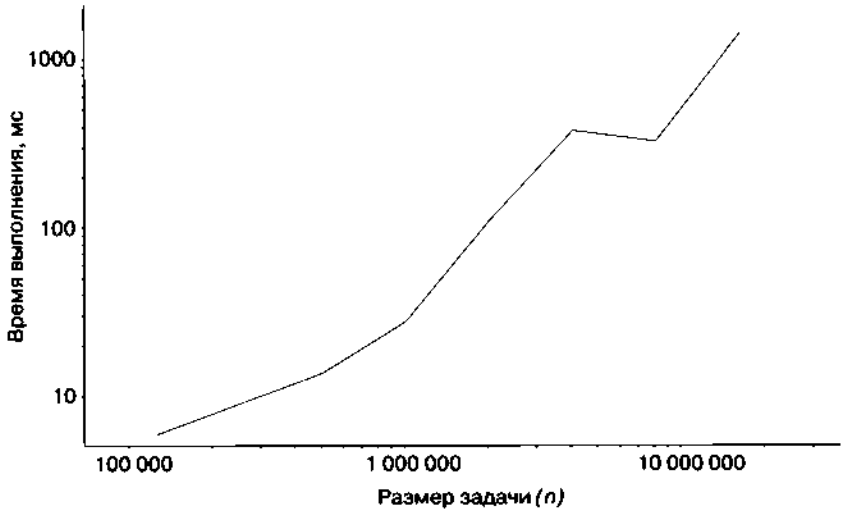


Рис. 5.2. Результаты профилирования производительности: время выполнения в зависимости от размера задачи для добавления n элементов в начало LinkedList

Итак, мы ожидаем, что общее время для *n* добавлений будет квадратичным.

Но это не так. Вот код:

```
public static void profileLinkedListAddEnd() {
    Timeable timeable + new Timeable() {

        List<String> list;

        public void setup(int n) {
            list + new LinkedList<String>();
        }

        public void timeMe(int n) {
            for (int i=0; i<n; i++) {
                list.add("a string");
            }
        }
    };
    int startN + 64000;
    int endMillis + 1000;
    runProfiler("LinkedList add end", timeable, startN,
    endMillis);
}
```

Результаты таковы:

```
64000, 9
128000, 9
256000, 21
512000, 24
1024000, 78
2048000, 235
4096000, 851
8192000, 950
16384000, 6160
```

На рис. 5.3 показан график результатов.

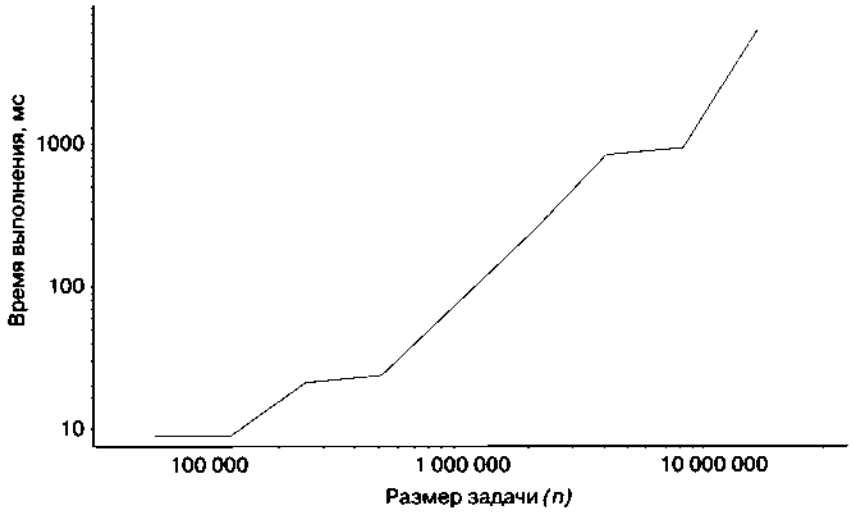


Рис. 5.3. Результаты профилирования производительности: время выполнения в зависимости от размера задачи для добавления n элементов в конец LinkedList

И снова в измерениях много погрешностей, а линия не совсем прямая, но расчетный тангенс угла наклона равен 1,19. Это близко к тому, что мы получили в начале, и не очень близко к 2, что мы ожидали получить, основываясь на нашем анализе. Фактически тангенс ближе к 1; данный факт говорит о том, что добавление элементов в конец выполняется за постоянное время. Что происходит?

Двусвязный список

Моя реализация связного списка `MyLinkedList` использует односвязный список; то есть каждый элемент содержит ссылку на следующий, а сам объект `MyArrayList` имеет ссылку на первый узел.

Однако в документации для `LinkedList` (http://thinkdast.com/linked) говорится:

> «Основанная на двусвязном списке реализация интерфейсов List и Deque... Все операции выполняются так же, как можно было бы ожидать для двусвязного списка. Операции, которые индексируются внутри списка, будут перемещаться по списку с начала или с конца, в зависимости от того, что ближе к указанному индексу».

Если вы не знакомы с двусвязными списками, то можете узнать о них больше на сайте http://thinkdast.com/doublelist, но краткая версия такова:

- каждый узел содержит ссылки на следующий и предыдущий узлы;
- объект `LinkedList` включает ссылки на первый и последний элементы списка.

Таким образом, можно начать с обоих концов списка и обойти его в любом направлении. В результате можно добавлять и удалять элементы из начала и конца списка за постоянное время!

В табл. 5.1 приведены результаты, которые мы ожидаем от `ArrayList`, `MyLinkedList` (односвязный) и `LinkedList` (двусвязный).

Таблица 5.1. Результаты, ожидаемые от ArrayList, MyLinkedList и LinkedList

Операция	MyArrayList	MyLinkedList	LinkedList
add (в конец)	1	n	1
add (в начало)	n	1	1
add (в общем случае)	n	n	n
get/set	1	n	n
indexOf/lastIndexOf	n	n	n
isEmpty/size	1	1	1
remove (из конца)	1	n	1
remove (из начала)	n	1	1
remove (в общем случае)	n	n	n

Выбор структуры

Двусвязная реализация лучше, чем `ArrayList` для добавления и удаления из начала, и так же хороша, как `ArrayList` для добавления и удаления из конца. Таким образом, единственное преимущество `ArrayList` — методы `get` и `set`, которые требуют линейного времени в связном списке, даже если он двусвязный.

Если известно, что время выполнения приложения зависит от времени, необходимого для получения и установки значений

элементов, то лучшим вариантом может быть `ArrayList`. Когда же время выполнения зависит от добавления и удаления элементов из начала или конца, более подходящим будет `LinkedList`.

Но помните, что эти рекомендации основаны на порядке роста для больших задач. Есть и другие факторы, которые следует учитывать.

- Если эти операции не занимают значительную часть времени выполнения для приложения (то есть приложения тратят большую часть времени на другие действия), то выбор реализации `List` не будет иметь большого значения.

- Не очень большие рабочие списки не всегда дают ожидаемую производительность. Для небольших задач квадратичный алгоритм может быть быстрее, чем линейный, или линейный — быстрее, чем алгоритм постоянного времени. И для небольших задач разница, разумеется, несущественна.

- Не стоит забывать и о пространстве. До сих пор мы ориентировались на время выполнения, но разные реализации требуют разного объема пространства. В `ArrayList` элементы хранятся бок о бок в одном блоке памяти, поэтому пространство теряется в незначительных количествах, а компьютерное оборудование часто быстрее работает с непрерывными массивами данных. В связанном списке каждый элемент требует узла с одной или двумя ссылками. Ссылки занимают пространство (иногда больше, чем данные!). И при работе с узлами, разбросанными по памяти, аппаратное обеспечение может быть менее эффективным.

В целом анализ алгоритмов может служить основанием для выбора структур данных, но только если:

1) время выполнения приложения важно;

2) время выполнения приложения зависит от выбора структуры данных;

3) размер задачи достаточно велик, чтобы порядок роста действительно предсказывал, какая структура данных лучше.

Можно иметь большой опыт в качестве разработчика программного обеспечения и так ни разу и не встретить подобную ситуацию.

6 Обход дерева

В этой главе мы рассмотрим приложение для поисковой системы, которое будем разрабатывать на протяжении оставшейся части книги. Я описываю элементы поисковой системы и представляю первое приложение, поискового робота, который загружает и анализирует страницы из «Википедии». В данной главе также представлена рекурсивная реализация поиска в глубину и итеративная реализация, использующая `Deque` из Java для реализации стека типа «последним вошел, первым вышел».

Поисковые системы

Поисковая система, такая как Google Search или Bing, принимает набор поисковых терминов и возвращает список веб-страниц, которые релевантны этим терминам. На сайте http://thinkdast.com/searcheng можно прочитать больше, но я объясню, что нужно, по мере продвижения.

Рассмотрим основные компоненты поисковой системы.

- *Сбор данных* (crawling). Понадобится программа, способная загружать веб-страницу, анализировать ее и извлекать текст и любые ссылки на другие страницы.
- *Индексирование* (indexing). Нужен индекс, который позволит найти поисковый запрос и страницы, его содержащие.
- *Поиск* (retrieval). Необходим способ сбора результатов из индекса и определения страниц, наиболее релевантных поисковым терминам.

Начнем с поискового робота. Его цель — обнаружение и загрузка набора веб-страниц. Для поисковых систем, таких как Google и Bing, задача состоит в том, чтобы найти *все* веб-страницы, но часто эти роботы ограничиваются меньшим доменом. В нашем случае мы будем читать только страницы из «Википедии».

В качестве первого шага мы создадим поискового робота, который читает страницу «Википедии», находит первую ссылку, переходит по ней на другую страницу и повторяет предыдущие действия. Мы будем использовать этот поисковик, чтобы проверить гипотезу Getting to Philosophy («Путь к философии»). В ней говорится:

> «Щелкнув на первой ссылке, написанной строчными буквами, в основном тексте статьи в "Википедии" и повторив затем это действие для последующих статей, вы, скорее всего, попадете на страницу со статьей о философии».

Ознакомиться с этой гипотезой и ее историей можно по адресу http://thinkdast.com/getphil.

Проверка гипотезы позволит создавать основные части поискового робота, не нуждаясь в обходе всего Интернета или даже

всей «Википедии». И я думаю, что это упражнение довольно интересное!

В нескольких главах мы будем работать над индексатором, а затем перейдем к поисковику.

Парсинг HTML

Когда вы загружаете веб-страницу, ее содержимое написано на языке гипертекстовой разметки (HyperText Markup Language, HTML). Например, ниже представлен простейший HTML-документ:

```
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

Фразы `This is a title` («Это название») и `Hello world!` («Привет, мир!») — текст, который действительно отображается на странице; другие элементы — *теги*, указывающие, как должен отображаться текст.

Нашему роботу при загрузке страницы нужно проанализировать код HTML, чтобы извлечь текст и найти ссылки. Для этого мы будем использовать *jsoup* — библиотеку Java с открытым исходным кодом, предназначенную для загрузки и парсинга (синтаксического анализа) HTML.

Результат парсинга HTML — *дерево DOM* (Document Object Model), содержащее элементы документа, включая текст и теги.

Дерево — связанная структура данных, состоящая из вершин, которые представляют текст, теги и другие элементы документа.

Связь между вершинами определяется структурой документа. В предыдущем примере первым узлом, называемым *корнем*, является тег `<html>`, который включает ссылки на две содержащиеся в нем вершины `<head>` и `<body>`; эти узлы — *дочерние* элементы корневого узла.

Узел `<head>` имеет одну дочернюю вершину `<title>`, а узел `<body>` — одну дочернюю вершину `<p>` (абзац, от англ. paragraph). На рис. 6.1 представлено графическое изображение данного дерева.

Рис. 6.1. Дерево DOM для простой HTML-страницы

Каждая вершина включает ссылки на свои дочерние узлы; кроме того, каждый узел содержит ссылку на своего *родителя*, поэтому вверх и вниз по дереву можно перемещаться из любого узла. Дерево DOM для реальных страниц обычно сложнее, чем описанный пример.

В большинстве браузеров есть инструменты для проверки DOM просматриваемой страницы. В Chrome можно щелкнуть правой

кнопкой мыши на любой части веб-страницы и в появившемся меню выбрать пункт Просмотреть код. В Firefox можно щелкнуть правой кнопкой мыши и выбрать в контекстном меню пункт Исследовать элемент. Safari предоставляет инструмент Web Inspector, информация о котором находится на сайте http://thinkdast.com/safari. Инструкции для Internet Explorer можно прочитать, перейдя по ссылке: http://thinkdast.com/explorer.

На рис. 6.2 показан скриншот с изображением дерева DOM для страницы «Википедии» о Java (http://thinkdast.com/java). Выделенный элемент — первый абзац основного текста статьи, который содержится в элементе `<div>` с `id="mw-content-text"`. Мы будем использовать этот идентификатор элемента, чтобы определить основной текст каждой загружаемой нами статьи.

Применение jsoup

Библиотека jsoup позволяет легко загружать и анализировать веб-страницы и перемещаться по дереву DOM. Например:

```
String url = "http://en.wikipedia.org/wiki/Java_(programming_
language)";

// загрузка и парсинг документа
Connection conn = Jsoup.connect(url);
Document doc = conn.get();

// выбирает текст контента и разделяет его по параграфам
Element content = doc.getElementById("mw-content-text");
Elements paragraphs = content.select("p");
```

Элемент `Jsoup.connect` принимает URL в виде строки и устанавливает соединение с веб-сервером; метод `get` загружает код HTML, анализирует его и возвращает объект `Document`, который представляет собой DOM.

Рис. 6.2. DOM Inspector в Chrome

Этот объект включает методы для навигации по дереву и выбора узлов. На самом деле он предоставляет так много методов, что это может сбить с толку. Следующий пример демонстрирует два способа выбора узлов.

- `getElementById` принимает параметр строкового типа и ищет дерево для элемента с соответствующим полем `id`. Найдя

его, он выбирает узел `<div id="mw-content-text" lang="en" dir="ltr" class="mw-content-ltr">`, который появляется на каждой странице «Википедии», чтобы идентифицировать элемент `<div>`, содержащий основной текст страницы, в отличие от боковой панели навигации и других элементов.

Возвращаемое значение `getElementById` — это объект класса `Element`, который представляет этот `<div>` и содержит вложенные в `<div>` дочерние элементы, такие как дети, дети детей и т. д.

- `select` принимает `String`, обходит дерево и возвращает все элементы с тегами, соответствующими `String`. В данном примере он возвращает все теги абзацев, которые появляются в `content`. Возвращаемое значение — объект типа `Elements`.

Прежде чем продолжить, вы должны просмотреть документацию этих классов, чтобы знать, какие действия они могут выполнять. Наиболее важные классы — `Element`, `Elements` и `Node`, прочитать о которых можно, перейдя по ссылкам http://thinkdast.com/jsoupelt, http://thinkdast.com/jsoupelts и http://thinkdast.com/jsoupnode.

Класс `Node` представляет собой вершину в дереве DOM. Существует несколько субклассов, расширяющих `Node`, включая `Element`, `TextNode`, `DataNode` и `Comment`. Класс `Elements` — коллекция объектов типа `Element`.

На рис. 6.3 представлена диаграмма классов UML, показывающая отношения между ними. Линия с пустой стрелкой говорит о расширении одного класса другим. Например, эта диаграмма указывает на то, что `Elements` расширяет `ArrayList`. Мы вернемся к диаграммам классов UML в одноименном разделе главы 11.

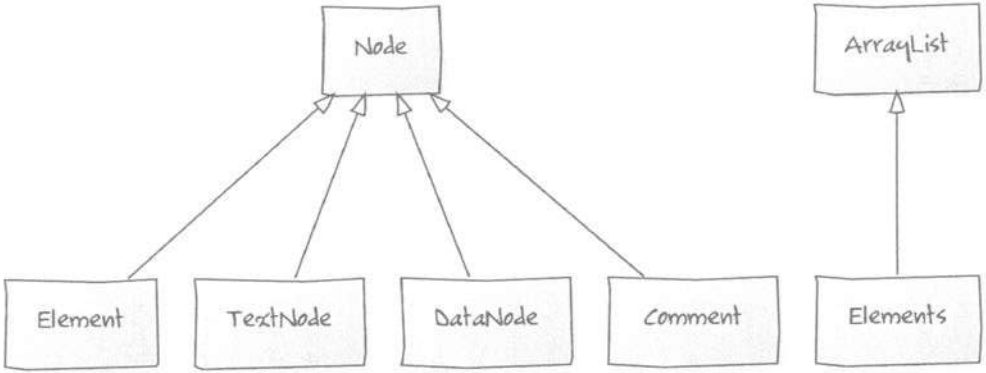


Рис. 6.3. Диаграмма UML для отдельных классов, предоставляемых jsoup

Итерация по дереву DOM

Для того чтобы облегчить вам жизнь, я предоставляю класс `WikiNodeIterable`, позволяющий проходить по дереву DOM. Ниже приведен пример, который показывает, как использовать этот класс:

```
Elements paragraphs + content.select("p");
Element firstPara + paragraphs.get(0);

Iterable<Node> iter + new WikiNodeIterable(firstPara);
for (Node node: iter) {
    if (node instanceof TextNode) {
        System.out.print(node);
    }
}
```

Этот пример начинается с того момента, на котором остановился предыдущий. Он выбирает первый абзац в `paragraphs` и затем создает класс `WikiNodeIterable`, который реализует интерфейс `Iterable<Node>`. Данный класс выполняет *поиск в глубину*, создавая узлы в том порядке, в котором они будут показываться на странице.

В текущем примере мы отображаем `Node`, только если он является `TextNode`, и игнорируем другие его типы, в частности объекты типа `Element`, представляющие теги. Результат — простой текст абзаца HTML без какой-либо разметки. Его вывод:

> Java is a general-purpose computer programming language that is concurrent, class- based, object-oriented, [13] and specifically designed...[1]

[1] Java — универсальный компьютерный язык программирования, который является объектно-ориентированным языком, основанным на классах, с возможностью параллельного программирования [13] и специально разработан...

Поиск в глубину

Существует несколько способов разумного обхода дерева. Мы начнем с *поиска в глубину* (depth-first search, DFS). Поиск начинается с корня дерева и выбора первого дочернего узла. Если у последнего есть дети, то снова выбирается первый дочерний узел. Когда попадается вершина без детей, нужно вернуться, перемещаясь вверх по дереву к родительскому узлу, где выбирается следующий ребенок, если он есть. В противном случае нужно снова вернуться. Когда исследован последний дочерний узел корня, обход завершается.

Есть два общепринятых способа реализации поиска в глубину: рекурсивный и итеративный. Рекурсивная реализация проста и элегантна:

```
private static void recursiveDFS(Node node) {
  if (node instanceof TextNode) {
      System.out.print(node);
  }
  for (Node child: node.childNodes()) {
      recursiveDFS(child);
  }
}
```

Этот метод вызывается для каждого `Node` в дереве, начиная с корня. Если `Node` является `TextNode`, то печатается его содержимое. При наличии у `Node` детей происходит вызов `recursiveDFS` для каждого из них по порядку.

В текущем примере мы распечатываем содержимое каждого `TextNode` до посещения его дочерних узлов, то есть это пример прямого (pre-order) обхода. О прямом, обратном (post-order) и симметричном (in-order) обходе можно прочитать, перейдя по ссылке http://thinkdast.com/treetrav. Для данного приложения порядок обхода не имеет значения.

Выполняя рекурсивные вызовы, `recursiveDFS` использует стек вызовов (см. http://thinkdast.com/callstack) с целью отслеживать дочерние вершины и обрабатывать их в правильном порядке. В качестве альтернативы можно использовать структуру данных стека, чтобы отслеживать узлы самостоятельно; это позволит избежать рекурсии и обойти дерево итеративно.

Стеки в Java

Прежде чем объяснять итеративную версию поиска в глубину, я рассмотрю структуру данных стека. Мы начнем с общей концепции стека, а затем поговорим о двух Java-`interfaces`, которые определяют методы стека: `Stack` и `Deque`.

Стек представляет собой структуру данных, похожую на список: это коллекция, которая поддерживает порядок элементов. Основное различие между стеком и списком состоит в том, что первый включает меньше методов. По общепринятому соглашению стек предоставляет методы:

- `push`, добавляющий элемент в вершину стека;
- `pop`, который производит удаление и возвращает значение самого верхнего элемента стека;
- `peek`, возвращающий самый верхний элемент стека без изменения самого стека;
- `isEmpty`, который показывает, является ли стек пустым.

Поскольку `pop` всегда возвращает самый верхний элемент, то стек еще называется *LIFO*, что значит «последним вошел, первым вышел» (last in, first out). Альтернатива стеку — *очередь*,

возвращающая элементы в том же порядке, в котором они были добавлены, то есть «первым вошел, первым вышел» (first in, first out), или FIFO.

На первый взгляд непонятно, почему стеки и очереди полезны: они не дают никаких особых возможностей, которые нельзя было бы получить от списков; фактически возможностей у них даже меньше. Так почему бы не применять списки всегда? Есть две причины, оправдывающие применение стеков и очередей.

1. Если вы ограничиваете себя небольшим набором методов (то есть небольшим API), то ваш код будет более читабельным и менее подверженным ошибкам. Например, при использовании списка для представления стека можно случайно удалить элемент в неправильном порядке. С API стека такая ошибка в буквальном смысле невозможна. А лучший способ избежать ошибок — сделать их невозможными.

2. Если структура данных предоставляет небольшой API, то ее легче реализовать эффективно. Например, простой способ реализации стека — одиночный список. Вталкивая (push) элемент в стек, мы добавляем его в начало списка; выталкивая (pop) элемент, мы удаляем его из самого начала. Для связного списка добавление и удаление из начала — операции постоянного времени, поэтому данная реализация эффективна. И наоборот, большие API сложнее реализовать эффективно.

Реализовать стек в Java можно тремя путями.

1. Применить `ArrayList` или `LinkedList`. При использовании `ArrayList` нужно помнить о добавлении и удалении из *конца*,

чтобы эти операции выполнялись за постоянное время. Следует избегать добавления элементов в неправильное место или их удаления в неправильном порядке.

2. В Java есть класс `Stack`, предоставляющий стандартный набор методов стека. Но этот класс — старая часть Java: он несовместим с Java Collections Framework, который появился позже.

3. Вероятно, лучший выбор — использовать одну из реализаций интерфейса `Deque`, например `ArrayDeque`.

Deque образовано от double-ended queue, это значит «двусторонняя очередь». В Java интерфейс `Deque` предоставляет методы `push`, `pop`, `peek` и `isEmpty`, поэтому его можно использовать как стек. Он содержит и другие методы, информация о которых доступна на сайте http://thinkdast.com/deque, но пока мы не будем применять их.

Итеративный поиск в глубину

Ниже приводится итеративная версия DFS, использующая `ArrayDeque` для представления стека объектов типа `Node`:

```
private static void iterativeDFS(Node root) {
    Deque<Node> stack + new ArrayDeque<Node>();
    stack.push(root);

    while (!stack.isEmpty()) {
        Node node + stack.pop();
        if (node instanceof TextNode) {
```

```
            System.out.print(node);
        }

        List<Node> nodes + new ArrayList<Node>(node.
        chiidNodesQ);
        Collections.reverse(nodes);

        for (Node child: nodes) {
            stack.push(chiid);
        }
    }
}
```

Параметр `root` — корень дерева, которое мы хотим обойти, поэтому начинаем с создания стека и добавления в него данного параметра.

Цикл продолжается до тех пор, пока стек не окажется пустым. При каждом проходе происходит выталкивание `Node` из стека. Если получен `TextNode`, то печатается его содержимое. Затем в стек добавляются дочерние вершины. Чтобы обрабатывать потомков в правильном порядке, нужно выталкивать их в стек в обратном порядке; это делается копированием дочерних вершин в `ArrayList`, перестановкой элементов на месте, а затем итерацией реверсированного `ArrayList`.

Одно из преимуществ итеративной версии поиска в глубину заключается в том, что ее проще реализовать как `Iterator` в Java; как это сделать, описано в следующей главе.

Но сначала последнее замечание об интерфейсе `Deque`: помимо `ArrayDeque`, Java предоставляет еще одну реализацию `Deque`, нашего старого друга `LinkedList`. Последний реализует оба интерфейса: `List` и `Deque`. Полученный интерфейс зависит от его

использования. Например, при назначении объекта `LinkedList` переменной `Deque`:

```
Deqeue<Node> deque + new LinkedList<Node>();
```

можно применить методы из интерфейса `Deque`, но не все методы из интерфейса `List`. Присвоив его переменной `List`:

```
List<Node> deque + new LinkedList<Node>();
```

можно использовать методы `List`, но не все методы `Deque`. И присваивая их следующим образом:

```
LinkedList<Node> deque + new LinkedList<Node>();
```

можно использовать *все* методы. Но при объединении методов из разных интерфейсов код будет менее читабельным и более подверженным ошибкам.

7 Путь к философии

Цель этой главы — разработка поискового робота, проверяющего гипотезу Getting to Philosophy, о которой я рассказал в разделе «Поисковые системы» главы 6.

Начало разработки

В репозитории для этой книги вы найдете код, который поможет начать работу.

1. `WikiNodeExample.java` содержит код из предыдущей главы, демонстрирующий рекурсивные и итеративные реализации поиска в глубину в дереве DOM.

2. `WikiNodeIterable.java` включает класс `Iterable` для обхода дерева DOM. Я объясню этот код в следующем разделе.

3. `WikiFetcher.java` содержит вспомогательный класс, использующий jsoup для загрузки страниц из «Википедии». Чтобы

помочь соблюдать условия обслуживания «Википедии», этот класс ограничивает скорость загрузки страниц; при запросе более одной страницы в секунду он приостанавливается перед загрузкой следующей страницы.

4. `WikiPhilosophy.java` включает схему кода, который вы напишете для этого упражнения.

Кроме того, вы найдете файл сборки для Ant под названием `build.xml`. При запуске `ant WikiPhilosophy` он будет активизировать простой бит стартового кода.

Интерфейсы Iterable и Iterator

В предыдущей главе я описал итеративный поиск в глубину и предположил: преимущество итеративной версии по сравнению с рекурсивной заключается в том, что ее легче инкапсулировать в объект `Iterator`. В текущем разделе вы увидите, как это сделать.

Если вы не знакомы с интерфейсами `Iterator` и `Iterable`, то можете прочитать о них, перейдя по ссылкам http://thinkdast.com/iterator и http://thinkdast.com/iterable.

Взгляните на содержимое `WikiNodeIterable.java`. Внешний класс `WikiNodeIterable` реализует интерфейс `Iterable<Node>`, поэтому его можно использовать в цикле `for` таким образом:

```
Node root = ...
Iterable<Node> iter = new WikiNodeIterable(root);
for (Node node: iter) {
    visit(node);
}
```

Здесь root — корень дерева, которое мы хотим обойти, а visit — это метод, делающий все, что мы хотим, когда «посещаем» узел.

Реализация WikiNodeIterable следует общепринятой формуле.

1. Конструктор принимает и сохраняет ссылку на корневой узел.
2. Метод iterator создает и возвращает объект типа Iterator.

Это выглядит так:

```
public class WikiNodeIterable implements Iterable<Node> {

    private Node root;

    public WikiNodeIterable(Node root) {
        this.root + root;
    }

    @Override
    public Iterator<Node> iterator() {
        return new WikiNodeIterator(root);
    }
}
```

Внутренний класс WikiNodeIterator выполняет всю настоящую работу:

```
private class WikiNodeIterator implements Iterator<Node> {

    Deque<Node> stack;

    public WikiNodeIterator(Node node) {
        stack + new ArrayDeque<Node>();
        stack.push(root);
    }

    @Override
    public boolean hasNext() {
```

```
        return !stack.isEmpty();
    }

    @Override
    public Node next() {
        if (stack.isEmpty()) {
            throw new NoSuchElementException();
        }

        Node node + stack.pop();
        List<Node> nodes + new ArrayList<Node>(node.childNodes());
        Collections.reverse(nodes);
        for (Node child: nodes) {
            stack.push(child);
        }
        return node;
    }
}
```

Этот код почти идентичен итеративной версии поиска в глубину, но теперь разделен на три метода.

1. Конструктор инициализирует стек (который реализуется на основе `ArrayDeque`) и добавляет в него корневой узел.

2. Метод `isEmpty` проверяет, является ли стек пустым.

3. Метод `next` выталкивает следующий узел из стека, добавляет его дочерние узлы в обратном порядке и возвращает вытолкнутый узел. Если вызвать `next` для пустого `Iterator`, то он выдаст исключение.

На первый взгляд непонятно, почему стоит заменить один прекрасный метод двумя классами и пятью методами. Но теперь, когда мы это сделали, можем использовать `WikiNodeIterable` везде, где требуется `Iterable`, что позволяет просто и синтаксически чисто отделить логику итераций (поиска в глубину) от любой обработки, которую мы проводим в узлах.

WikiFetcher

При написании поискового робота можно легко загрузить слишком много страниц, что способно нарушить условия обслуживания для сервера, с которого производится загрузка. Помочь избежать этого призван класс под названием `WikiFetcher`, выполняющий две функции.

1. Он инкапсулирует код, показанный в предыдущей главе, для загрузки страниц из «Википедии», парсинга HTML и выбора текста контента.

2. Он измеряет время между запросами и при отсутствии достаточного времени между запросами «спит» до окончания разумного интервала. По умолчанию интервал составляет одну секунду.

Определение `WikiFetcher` таково:

```
public class WikiFetcher {
    private long lastRequestTime + -1;
    private long minInterval + 1000;

    /**
     * Выбирает и анализирует строку URL,
     * возвращая список элементов для абзацев.
     */
    public Elements fetchWikipedia(String url) throws IOException {
        sleepIfNeeded();

        Connection conn + Jsoup.connect(url);
        Document doc + conn.get();
        Element content + doc.getElementById("mw-content-text");
        Elements paragraphs + content.select("p");
        return paragraphs;
    }

    private void sleepIfNeeded() {
```

```
        if (lastRequestTime != -1) {
            long currentTime + System.currentTimeMillis();
            long nextRequestTime + lastRequestTime + minInterval;
            if (currentTime < nextRequestTime) {
                try {
                    Thread.sleep(nextRequestTime - currentTime);
                } catch (InterruptedException e) {
                    System.err.println(
                        "Warning: sleep interrupted in
                        fetchWikipedia.");
                }
            }
        }
        lastRequestTime + System.currentTimeMillis();
    }
}
```

Единственный публичный метод — `fetchWikipedia`, который принимает URL-адрес как строку и возвращает коллекцию `Elements`, содержащую по одному элементу DOM для каждого абзаца в тексте контента. Этот код должен казаться знакомым.

Новый код находится в методе `sleepIfNeeded`, который проверяет время, прошедшее с момента последнего запроса, и бездействует, если затраченное время меньше чем `minInterval`, заданный в миллисекундах.

Это все, что есть в `WikiFetcher`. Ниже приведен пример, демонстрирующий то, как используется данный класс:

```
WikiFetcher wf + new WikiFetcher();

for (String url: urlList) {
    Elements paragraphs + wf.fetchWikipedia(url);
    processParagraphs(paragraphs);
}
```

В данном примере мы предполагаем, что `urlList` представляет собой набор строк, а `processParagraphs` — это метод, который выполняет какие-то действия с объектом `Elements`, возвращаемым `fetchWikipedia`.

В этом примере показан важный момент: необходимо создать один объект `WikiFetcher` и применять его для обработки всех запросов. При наличии нескольких экземпляров `WikiFetcher` они не будут использовать минимальный интервал между запросами.

Моя реализация WikiFetcher проста, но ее легко можно использовать неправильно, создав несколько экземпляров. Избежать данной проблемы можно, сделав WikiFetcher одиночкой (Singleton). Информацию об этом см. на сайте http://thinkdast.com/singleton.

Упражнение 5

В `WikiPhilosophy.java` вы найдете простой метод `main`, показывающий, как использовать часть этих элементов. Начиная с данного кода, ваша задача — написать поисковый робот, который делает следующее.

1. Принимает URL страницы «Википедии», загружает ее и анализирует.

2. Он должен обойти полученное дерево DOM, чтобы найти первую *действительную* ссылку. Ниже я объясню, что значит «действительный».

3. При отсутствии на странице ссылок или в том случае, если первая ссылка — это страница, которую мы уже видели, программа должна сообщить об отказе и выйти.

4. Если ссылка соответствует URL страницы «Википедии» по философии, то программа должна сообщить об успешном выполнении и выйти.

5. В противном случае ей следует вернуться к шагу 1.

Программа должна создать `List` URL, которые она посетила, и отобразить результаты в конце (как при успешном выполнении, так и в случае отказа).

Итак, что стоит рассматривать как «действительную» ссылку? В этом вопросе есть выбор. Различные варианты гипотезы Getting to Philosophy используют несколько иные правила, но вот некоторые из них.

1. Ссылка должна находиться в тексте контента страницы, а не на боковой панели или в контейнере.

2. Она не должна быть выделена курсивом или взята в скобки.

3. Нужно пропустить внешние ссылки, ссылки на текущую страницу и красные ссылки.

4. В некоторых версиях следует пропустить ссылку, если текст начинается с буквы в верхнем регистре.

Не нужно строго следовать всем этим правилам, но я рекомендую хотя бы обрабатывать круглые скобки, курсив и ссылки на текущую страницу.

Если чувствуете, что у вас достаточно информации для начала работы, то приступайте. Или можете воспользоваться следующими подсказками.

1. При обходе дерева придется иметь дело с двумя типами узлов: `TextNode` и `Element`. При обнаружении последнего, ве-

роятно, придется произвести приведение типов для доступа к тегу и другой информации.

2. Обнаружив `Element`, содержащий ссылку, можно проверить, выделена ли она курсивом, следуя по родительским связям вверх по дереву. Если в родительской цепочке есть тег `<i>` или `<em>`, то ссылка выделена курсивом.

3. Чтобы проверить, находится ли ссылка в круглых скобках, придется сканировать текст по мере прохождения по дереву и отслеживать открывающие и закрывающие круглые скобки (в идеале решение должно иметь возможность обрабатывать вложенные круглые скобки (например, такие)).

4. Начав со страницы, посвященной Java (http://thinkdast.com/java), следует перейти к странице Philosophy (после семи ссылок, если только ничего не изменилось с момента, когда я запускал код).

Это все, чем я могу помочь. Теперь все зависит от вас. Получите удовольствие от работы!

8 Индексатор

На текущий момент мы создали базового поискового робота; следующая часть, над которой будем работать, — *индекс*. В контексте веб-поиска индекс представляет собой структуру данных, позволяющую искать поисковый термин и находить страницы, где он появляется. Кроме того, мы хотели бы знать, сколько раз поисковый термин встречается на каждой странице, что поможет определить страницы, наиболее релевантные этому термину.

Например, если пользователь вводит поисковые термины `Java` и `программирование`, мы будем искать оба поисковых термина и получим два набора страниц. Набор со словом `Java` будет включать страницы об острове Ява, марке кофе и языке программирования. В набор со словом `программирование` войдут страницы о разных языках программирования, а также другие варианты применения этого слова. Выбирая страницы с обоими терминами, мы надеемся устранить нерелевантные и найти те, которые касаются программирования на Java.

Теперь, когда стало понятно, что такое индекс и какие операции он выполняет, можно создать структуру данных для его представления.

Выбор структуры данных

Основная операция индекса — это *поиск*; в частности, требуется возможность искать термин и находить все страницы, где он есть. Простейшей реализацией будет выборка страниц. С учетом поискового термина можно было бы перебирать содержимое страниц и выбирать те из них, которые включают данный термин. Но время выполнения будет пропорционально общему количеству слов на всех страницах, что слишком замедлит поиск.

Лучшая альтернатива — *карта*. Это структура данных, которая представляет собой набор *пар «ключ — значение»* и обеспечивает быстрый способ поиска *ключа* и нахождения соответствующего *значения*. Например, первая карта, которую мы построим, — `TermCounter`, сопоставляет с каждым поисковым термином количество его появлений на странице. Ключи — поисковые термины, а значения — количества (также называемые частотами).

Java предоставляет интерфейс `Map`, определяющий методы, которые должна предоставлять карта. Наиболее важными являются:

- `get(key)` — данный метод ищет ключ и возвращает соответствующее значение;
- `put(key, value)` — этот метод добавляет новую пару «ключ — значение» на карту или заменяет значение, связанное с ключом, если он уже нанесен на карту.

Java обеспечивает несколько реализаций `Map`, включая две, которым мы уделим основное внимание: `HashMap` и `TreeMap`.

В следующих главах рассмотрим эти реализации и проанализируем их производительность.

Помимо `TermCounter`, сопоставляющего поисковые термины с подсчетами, мы определим класс `Index`, служащий для сопоставления поисковых терминов с коллекциями страниц, на которых он появляется. В связи с этим возникает вопрос: как представлять коллекцию страниц? И снова мы принимаем решение, исходя из анализа операций, которые необходимо выполнить.

В данном случае нужно объединить две или несколько коллекций и найти страницы, которые появляются во всех из них. Эту операцию можно определить как *пересечение множеств*: пересечение двух множеств — это набор элементов, присутствующих в обоих из них.

Как вы, наверное, уже догадались, Java предоставляет интерфейс `Set`, определяющий операции, которые должен совершать набор. Фактически он не предусматривает пересечение множеств, но обеспечивает методы, позволяющие эффективно выполнять эту и другие операции с множествами. Основные методы этого интерфейса:

- `add(element)` — добавляет элемент в набор; если элемент уже находится в наборе, действие не производится;
- `contains(element)` — проверяет, находится ли данный элемент в наборе.

Java предоставляет несколько реализаций `Set`, включая `HashSet` и `TreeSet`.

Теперь, после разработки структур данных сверху вниз, реализуем их изнутри, начиная с `TermCounter`.

TermCounter

Класс `TermCounter` — это класс, который сопоставляет поисковые термины и количество их появлений на странице. Первая часть определения данного класса такова:

```
public class TermCounter {

    private Map<String, Integer> map;
    private String label;

    public TermCounter(String label) {
        this.label + label;
        this.map + new HashMap<String, Integer>();
    }
}
```

Переменные экземпляра — это `map`, содержащая сопоставление терминов с подсчетами, и `label`, идентифицирующая документ, из которого пришли термины. Мы будем применять ее для хранения URL.

Чтобы реализовать сопоставление, я выбрал `HashMap` — наиболее часто используемую реализацию `Map`. В следующих нескольких главах вы увидите, как это работает и почему пользуется популярностью.

Класс `TermCounter` предоставляет методы `put` и `get`, которые определены следующим образом:

```
public void put(String term, int count) {
    map.put(term, count);
}

public Integer get(String term) {
    Integer count + map.get(term);
    return count += null ? 0 : count;
}
```

Метод put всего лишь *метод обертки*. При вызове метода put класса TermCounter вызывается метод put встроенного экземпляра карты map.

С другой стороны, get действительно выполняет некоторые действия. Когда вы вызываете get в TermCounter, он вызывает get в map, а затем проверяет результат. Если термин не отображается на карте, то TermCount.get возвращает 0. Определение get таким способом упрощает запись incrementTermCount, который принимает термин и увеличивает на 1 значение счетчика, связанного с этим термином:

```
public void incrementTermCount(String term) {
    put(term, get(term) + 1);
}
```

Если термин ранее не был замечен, то get вернет 0; мы добавляем 1, затем используем put, чтобы добавить новую пару «ключ – значение» на карту. Если этот термин уже находится на карте, то мы получаем старое число, добавляем 1, а затем сохраняем новое число, которое заменяет старое значение.

Кроме того, класс TermCounter предоставляет другие методы, помогающие индексировать веб-страницы:

```
public void processElements(Elements paragraphs) {
    for (Node node: paragraphs) {
        processTree(node);
    }
}

public void processTree(Node root) {
    for (Node node: new WikiNodeIterable(root)) {
        if (node instanceof TextNode) {
            processText(((TextNode) node).text());
        }
    }
```

```
}

public void processText(String text) {
    String[] array + text.replaceAll("\\pP", " ").
                          toLowerCase().
                          split("\\s+");

    for (int i=0; i<array.length; i++) {
        String term + array[i];
        incrementTermCount(term);
    }
}
```

- `processElements` принимает объект `Elements`, который представляет собой набор объектов jsoup `Element`. Он перебирает коллекцию и вызывает `processTree` для каждого элемента.

- `processTree` принимает jsoup `Node` — корень дерева DOM. Он выполняет перебор дерева, чтобы найти вершины, содержащие текст; затем извлекает текст и передает его в `processText`.

- `processText` принимает строку, содержащую слова, пробелы, знаки препинания и т. д. Он удаляет знаки препинания, заменяя их пробелами, переводит оставшиеся буквы в нижний регистр, а затем разбивает текст на слова. Далее просматривает найденные слова и вызывает `incrementTermCount` для каждого. Методы `replaceAll` и `split` задействуют *регулярные выражения* в качестве параметров; узнать о них больше можно на сайте http://thinkdast.com/regex.

И напоследок пример, который показывает, как используется `TermCounter`:

```
String url + "http://en.wikipedia.org/wiki/Java_(programming_
language)";
```

```
WikiFetcher wf + new WikiFetcher();
Elements paragraphs + wf.fetchWikipedia(url);

TermCounter counter + new TermCounter(url);
counter.processElements(paragraphs);
counter.printCounts();
```

В этом примере используется `WikiFetcher` для загрузки страницы из «Википедии» и анализа основного текста. Затем данный метод создает `TermCounter` и задействует его для подсчета слов на странице.

В следующем разделе у вас будет возможность запустить этот код и проверить, насколько хорошо вы понимаете материал, добавив отсутствующий метод.

Упражнение 6

В репозитории для этой книги вы найдете исходные файлы для данного упражнения:

- `TermCounter.java` включает код из предыдущего раздела;
- `TermCounterTest.java` содержит код теста для `TermCounter.java`;
- `Index.java` вмещает определение класса для следующей части текущего упражнения;
- `WikiFetcher.java` включает класс, который использовался в предыдущем упражнении для загрузки и анализа веб-страниц;
- `WikiNodeIterable.java` содержит класс, применяемый для перемещения узлов в DOM.

Вы также найдете файл сборки для Ant под названием `build.xml`.

Активизируйте `ant build`, чтобы скомпилировать исходные файлы. Затем запустите `ant TermCounter`; он должен выполнить код из предыдущего раздела и распечатать список терминов и их количество. Вывод будет выглядеть примерно так:

```
genericservlet, 2
configurations, 1
claimed, 1
servletresponse, 2
occur, 2
Total of all counts + -1
```

Порядок терминов может быть другим.

В последней строке предполагается вывод общего количества счетчиков, но она возвращает `-1`, поскольку метод `size` является неполным. Дополните его и снова запустите `ant TermCounter`. Результатом должно быть значение `4798`.

Активизируйте `ant TermCounterTest` для подтверждения того, что эта часть упражнения выполнена и исправлена.

Во второй части упражнения я представляю реализацию класса `Index`, а вам нужно добавить недостающий метод. Начало определения класса выглядит так:

```
public class Index {

    private Map<String, Set<TermCounter>> index +
        new HashMap<String, Set<TermCounter>>();

    public void add(String term, TermCounter tc) {
        Set<TermCounter> set + get(term);

        // если термин встретился впервые, то мы создаем новый Set
        if (set += null) {
```

```
            set + new HashSet<TermCounter>();
            index.put(term, set);
        }
        // в противном случае мы изменяем существующий Set
        set.add(tc);
    }

    public Set<TermCounter> get(String term) {
        return index.get(term);
    }
```

Переменная экземпляра `index` представляет собой сопоставление каждого поискового термина с набором объектов `TermCounter`. Каждый объект `TermCounter` соответствует странице, на которой появляется такой термин.

Метод `add` добавляет новый `TermCounter` в набор, связанный с термином. При индексации термина, который раньше не появлялся, нужно создать новый набор. В противном случае можно просто добавить новый элемент в существующий набор. Тогда `set.add` изменяет набор внутри индекса, но не модифицирует сам индекс. Последний изменяется только при добавлении нового термина.

Наконец, метод `get` принимает поисковый термин и возвращает соответствующий набор объектов `TermCounter`.

Эта структура данных имеет среднюю сложность. `Index` содержит `Map` для сопоставления каждого поискового термина с набором объектов `TermCounter`, а каждый экземпляр `TermCounter` — карта, связывающая поисковые термины со счетчиками.

На рис. 8.1 представлена диаграмма этих объектов. Объект `Index` имеет переменную экземпляра с именем `index`, которая ссылается на карту `Map`.

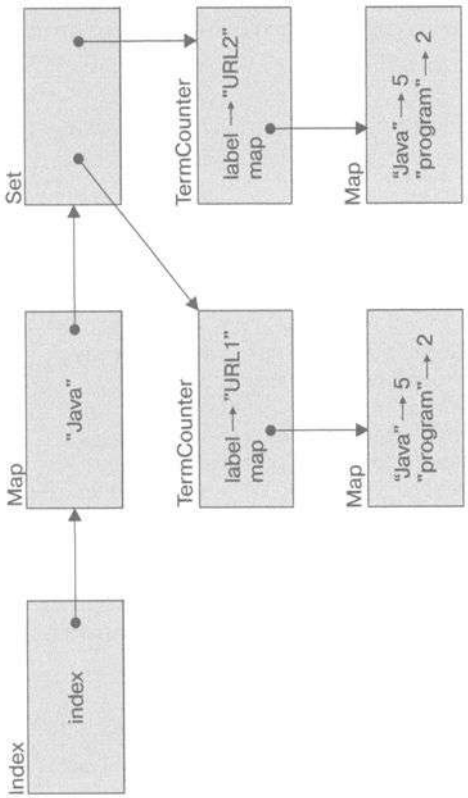


Рис. 8.1. Диаграмма объектов Index

В этом примере Map содержит только одну строку "Java", которая соответствует набору Set, включающему два объекта TermCounter, по одному на каждую страницу, где появляется слово Java.

Каждый экземпляр TermCounter включает label (URL страницы) и map — карту, содержащую слова на странице и количество появления каждого из них.

Метод printIndex показывает, как распаковать эту структуру данных:

```
public void printIndex() {
    // проходит в цикле по поисковым терминам
    for (String term: keySet()) {
        System.out.println(term);

        // для каждого термина печатает страницу,
        // на которой он появляется, и частоту
        Set<TermCounter> tcs + get(term);
        for (TermCounter tc: tcs) {
            Integer count + tc.get(term);
            System.out.println(" " + tc.getLabel() + " " + count);
        }
    }
}
```

Внешний цикл перебирает поисковые термины, внутренний — объекты TermCounter.

Активизируйте ant build с целью убедиться, что ваш исходный код скомпилирован, а затем запустите ant Index. Он загружает две страницы «Википедии», индексирует их и печатает результаты; но после его запуска не будет никакого вывода, поскольку мы оставили один из методов пустым.

Ваша задача — заполнить indexPage, который принимает URL-адрес (как String) и объект Elements и обновляет индекс. В сле-

дующих комментариях схематично показано, что он должен делать:

```
public void indexPage(String url, Elements paragraphs) {
    // создает TermCounter и считает термины в paragraphs

    // для каждого термина в TermCounter добавляет TermCounter
    // в индекс
}
```

Когда он будет работать, снова запустите `ant Index`. Вы должны увидеть вывод, подобный представленному ниже:

```
...
configurations
    http://en.wikipedia.org/wiki/Programming_ianguage 1
    http://en.wikipedia.org/wiki/Java_(programming_ianguage) 1
claimed
    http://en.wikipedia.org/wiki/Java_(programming_ianguage) 1
servletresponse
    http://en.wikipedia.org/wiki/Java_(programming_ianguage) 2
occur
    http://en.wikipedia.org/wiki/Java_(programming_ianguage) 2
```

Порядок поиска может отличаться при запуске.

Кроме того, запустите `ant TestIndex` для подтверждения того, что эта часть упражнения выполнена.

9 Интерфейс Map

В следующих нескольких упражнениях представлены некоторые реализации интерфейса `Map`. Одна из них основана на *хеш-таблице* — структуре данных, возможно, самой невероятной из когда-либо разработанных. Другая, похожая на `TreeMap`, не так хороша, но обладает дополнительной способностью перебирать элементы по порядку.

У вас будет возможность реализовать эти структуры данных, и далее мы проанализируем их производительность.

Но прежде, чем объяснять хеш-таблицы, я начну с простой реализации `Map`, используя список пар «ключ — значение» `List`.

Реализация MyLinearMap

Как обычно, я предоставляю начальный код, а вы дополните отсутствующие методы. Начало определения класса `MyLinearMap` выглядит следующим образом:

```
public class MyLinearMap<K, V> implements Map<K, V> {

    private List<Entry> entries + new ArrayList<Entry>();
```

Этот класс использует два типа параметров: `K` — тип ключей и `V` — тип значений. `MyLinearMap` реализует `Map`, это значит, что он должен предоставлять методы из интерфейса `Map`.

Объект `MyLinearMap` имеет одну переменную экземпляра `entries`, которая представляет собой объект `ArrayList` для `Entry`. Каждый экземпляр `Entry` содержит пару «ключ — значение». Ниже приведено его определение:

```
public class Entry implements Map.Entry<K, V> {
    private K key;
    private V value;

    public Entry(K key, V value) {
        this.key + key;
        this.value + value;
    }

    @Override
    public K getKey() {
        return key;
    }
    @Override
    public V getValue() {
        return value;
    }
}
```

Оно не такое уж большое; `Entry` — просто контейнер для ключа и значения. Данное определение вложено внутри `MyLinearList`, вследствие чего использует те же параметры типа `K` и `V`.

Это все, что нужно для выполнения упражнения, так что начнем.

Упражнение 7

В репозитории для этой книги вы найдете исходные файлы для данного упражнения:

- `MyLinearMap.java` содержит начальный код для первой части упражнения;
- `MyLinearMapTest.java` включает модульные тесты для `MyLinearMap`;

Вы также найдете файл сборки для Ant под названием `build.xml`.

Активизируйте `ant build`, чтобы скомпилировать исходные файлы. Затем запустите `ant MyLinearMapTest`; несколько тестов завершатся неудачей, поскольку вам нужно поработать!

Сначала заполните тело `findEntry`. Это вспомогательный метод, который не является частью интерфейса `Map`, но как только он заработает, вы сможете использовать его для нескольких методов. Учитывая заданный ключ, он должен искать записи и возвращать ту, что содержит искомое значение (как ключ, а не значение), или `null`, если ее нет. Обратите внимание: я предоставил метод `equals`, который сравнивает два ключа и корректно обрабатывает `null`.

Вы можете снова запустить `ant MyLinearMapTest`, но, даже если ваш `findEntry` верен, тесты не будут выполнены успешно, поскольку `put` не завершен.

Дополните `put`. Узнать, что этот метод должен делать, поможет документация для `Map.put` по адресу `http://thinkdast.com/list`. Начать можно с версии `put`, которая всегда добавляет новую запись и не изменяет существующую; таким образом сначала

проверяется простой случай. Или, если чувствуете себя более уверенно, можете написать все сразу.

Как только `put` заработает, тест для `containsKey` будет успешно выполняться.

Прочтите документацию для `Map.get` на сайте http://thinkdast.com/listget и дополните этот метод. Снова запустите тест.

Наконец, ознакомьтесь с документацией для `Map.remove` на сайте http://thinkdast.com/maprem и дополните данный метод.

Теперь все тесты должны завершиться успешно. Поздравляю!

Анализ MyLinearMap

В этом разделе я представляю решение предыдущего упражнения и анализирую производительность основных методов. Вот реализация `findEntry` и `equals`:

```
private Entry findEntry(Object target) {
    for (Entry entry: entries) {
        if (equals(target, entry.getKey())) {
            return entry;
        }
    }
    return null;
}

private boolean equals(Object target, Object obj) {
    if (target == null) {
        return obj == null;
    }
    return target.equals(obj);
}
```

Время выполнения `equals` может зависеть от размера `target` и ключей, но обычно не зависит от количества записей n. Таким образом, `equals` — метод постоянного времени.

В `findEntry` нам может повезти, и искомый ключ будет обнаружен в начале, но рассчитывать на это не стоит. Количество записей, которые мы должны искать, пропорционально n, так что `findEntry` является линейным.

Большинство основных методов в `MyLinearMap`, включая `put`, `get` и `remove`, используют `findEntry`. Вот как они выглядят:

```
public V put(K key, V value) {
    Entry entry = findEntry(key);
    if (entry == null) {
        entries.add(new Entry(key, value));
        return null;
    } else {
        oldValue = entry.getValue();
        entry.setValue(value);
        return oldValue;
    }
}

public V get(Object key) {
    Entry entry = findEntry(key);
    if (entry == null) {
        return null;
    }
    return entry.getValue();
}

public V remove(Object key) {
    Entry entry = findEntry(key);
    if (entry == null) {
        return null;
```

```
    } else {
        value + entry.getValue();
        entries.remove(entry);
        return value;
    }
}
```

После того как `put` вызовет `findEntry`, все остальное выполняется за постоянное время. Помните, что `entries` имеет тип `ArrayList`, поэтому добавление элемента *в конец* в среднем занимает постоянное время. Если ключ уже находится на карте, то не нужно добавлять запись, но придется вызвать `entry.getValue` и `entry.setValue`, и оба являются методами постоянного времени. Складывая все это вместе, метод `put` выполняется за линейное время.

По той же самой причине метод `get` также линейный.

Метод `remove` немного сложнее, поскольку методу `entry.remove`, возможно, придется удалить элемент из начала или середины `ArrayList`, и это займет линейное время. Но все в порядке: две линейные операции по-прежнему линейны.

Таким образом, основные методы являются линейными, поэтому данная реализация названа `MyLinearMap`.

Если известно, что количество записей будет небольшим, то эта реализация достаточно хороша, но можно поступить лучше. Фактически существует реализация `Map`, в которой все основные методы выполняются за постоянное время. Сначала это может показаться невероятным. Фактически я говорю, что можно найти иголку в стоге сена за постоянное время, независимо от того, насколько он велик. Это волшебство.

Я объясню, как это работает в два этапа.

1. Вместо того чтобы хранить записи в одном большом списке, мы разделим их на множество коротких. Для каждого ключа станем использовать *хеш-код* (поясняется ниже) с целью определить, какой список применять.

2. Использование большого количества коротких списков происходит быстрее, чем только одного, но, как я объясню, он не меняет порядок роста; основные операции по-прежнему линейны. Но есть еще один трюк: если увеличить количество списков, чтобы ограничить количество записей в списке, то результатом будет карта постоянного времени. Подробности вы увидите в следующем упражнении, но сначала — хеширование!

В главе 10 я представлю решение, проанализирую производительность основных методов Map и приведу более эффективную ее реализацию.

10 Хеширование

В этой главе приводится описание MyBetterMap, лучшей реализации интерфейса Map (по сравнению с MyLinearMap), и рассматривается *хеширование*, использование которого делает класс MyBetterMap более эффективным.

Хеширование

Чтобы улучшить производительность MyLinearMap, мы напишем новый класс под названием MyBetterMap, содержащий коллекцию объектов MyLinearMap. Он распределяет ключи среди встроенных карт, поэтому количество записей на каждой карте меньше, что позволяет ускорить findEntry и методы, которые от него зависят.

Начало определения класса будет таким:

```
public class MyBetterMap<K, V> implements Map<K, V> {

    protected List<MyLinearMap<K, V>> maps;

    public MyBetterMap(int k) {
```

```
        makeMaps(k);
    }

    protected void makeMaps(int k) {
        maps = new ArrayList<MyLinearMap<K, V>>(k);
        for (int i=0; i<k; i++) {
            maps.add(new MyLinearMap<K, V>());
        }
    }
}
```

Переменная экземпляра `maps` представляет собой набор объектов `MyLinearMap`. Конструктор принимает параметр *k*, который определяет, сколько карт использовать по крайней мере при инициализации. Затем `makeMaps` создает встроенные карты и сохраняет их в `ArrayList`.

Теперь основное условие выполнения этой работы таково: нужно каким-то образом посмотреть на ключ и решить, какую из встроенных карт он должен использовать. При добавлении нового ключа мы выбираем одну из карт; при получении ключа, добавленного ранее, нам следует помнить, куда мы его поместили.

Одна из возможностей заключается в том, чтобы выбирать одну из подкарт в случайном порядке и отслеживать, где мы помещаем каждый ключ. Но как это сделать? На первый взгляд, можно было бы использовать `Map` для поиска ключа и правильной подкарты, но весь смысл упражнения — в написании эффективной реализации `Map`. Мы не можем предполагать, что у нас она уже есть.

Лучший подход — задействовать *хеш-функцию*, которая принимает любой объект класса `Object` и возвращает целое число,

называемое *хеш-кодом*. Важно отметить: если она видит один и тот же `Object` более одного раза, то всегда возвращает один и тот же хеш-код. Таким образом, используя хеш-код для хранения ключа, мы получим тот же хеш-код при его поиске.

В Java каждый `Object` предоставляет метод `hashCode`, вычисляющий хеш-функцию. Реализация этого метода различна для разных объектов; скоро будет показан пример.

Ниже приведен вспомогательный метод, который выбирает правильную подкарту для заданного ключа:

```
protected MyLinearMap<K, V> chooseMap(Object key) {
    int index = 0;
    if (key != null) {
        index = Math.abs(key.hashCode()) % maps.size();
    }
    return maps.get(index);
}
```

Если ключ имеет значение `null`, то мы произвольно выбираем подкарту с индексом `0`. В противном случае задействуем `hashCode` для получения целого числа, применяем метод `Math.abs` для подтверждения того, что оно неотрицательное. Далее используем `%` — оператор вычисления остатка от деления, который гарантирует, что результат находится между `0` и `maps.sizeQ-1`. Таким образом, индекс всегда является действительным для `maps`. Затем `chooseMap` возвращает ссылку на выбранную карту.

Мы используем `chooseMap` как для `put`, так и для `get`, так что при поиске ключа получаем ту же карту, которую выбрали, когда добавили ключ. По крайней мере должны... Немного позже я объясню, почему это может не сработать.

Моя реализация put и get выглядит следующим образом:

```
public V put(K key, V value) {
    MyLinearMap<K, V> map + chooseMap(key);
    return map.put(key, value);
}

public V get(Object key) {
    MyLinearMap<K, V> map + chooseMap(key);
    return map.get(key);
}
```

Довольно просто, не правда ли? В обоих методах мы используем chooseMap для поиска правильной подкарты, а затем вызываем метод для нее. Вот как это работает; теперь подумаем о производительности.

Если n записей распределено между k ключей, то в среднем на одну карту приходится n / k записей. При поиске ключа нужно вычислить его хеш-код, что занимает некоторое время, затем искать соответствующую подкарту.

Поскольку списки записей в MyBetterMap в k раз короче, чем список записей в MyLinearMap, то мы ожидаем, что поиск будет в k раз быстрее. Но время выполнения по-прежнему пропорционально n, вследствие чего MyBetterMap все еще остается линейным. В следующем упражнении вы увидите, как это можно исправить.

Как работает хеширование

Основное требование к хеш-функции таково: один и тот же объект должен каждый раз воспроизводить один и тот же хеш-код. Для неизменяемых объектов это относительно просто. Над объектами с изменчивым состоянием придется серьезно подумать.

В качестве примера неизменяемого объекта мы определим класс, называемый SillyString, который инкапсулирует String:

```
public class SillyString {
    private final String innerString;

    public SillyString(String innerString) {
        this.innerString + innerString;
    }

    public String toString() {
        return innerString;
    }
```

Данный класс не очень полезен, поэтому и называется SillyString (бессмысленная строка), но я использую его, чтобы показать, как класс может определить собственную хеш-функцию:

```
    @Override
    public boolean equals(Object other) {
        return this.toString().equals(other.toString());
    }

    @Override
    public int hashCode() {
        int total + 0;
        for (int i=0; i<innerString.length(); i++) {
            total += innerString.charAt(i);
        }
        return total;
    }
```

Обратите внимание: SillyString переопределяет как equals, так и hashCode. Это важно. Для правильной работы equals должен быть совместим с hashCode, что означает следующее: если два объекта считаются равными (то есть equals возвращает true), то им следует иметь одинаковый хеш-код. Но это требование работает только в одном направлении; при наличии у двух

объектов одинакового хеш-кода они не обязательно должны быть равны.

Метод `equals` работает, вызывая `toString`, который возвращает `innerString`. Таким образом, два объекта `SillyString` равны при условии равенства их переменных экземпляра `innerString`.

В процессе работы `hashCode` перебирает символы в `String` и суммирует их. При добавлении символа к целому числу Java преобразует символ в целое число, используя его кодовую точку Unicode. Вам не нужно ничего знать о Unicode, чтобы понять данный пример, но если вас это интересует, то можете узнать больше по адресу http://thinkdast.com/codepoint.

Эта хеш-функция удовлетворяет требованию: если два объекта `SillyString` содержат встроенные строки, которые равны, то для них будет получен одинаковый хеш-код.

Описанный способ работает корректно, но может быть не слишком производительным в связи с тем, что один и тот же хеш-код возвращается для многих разных строк. Если две строки включают одни и те же буквы в любом порядке, то будут иметь один и тот же хеш-код. И даже если они не содержат одинаковых букв, то могут давать ту же общую сумму, как `"ac"` и `"bb"`.

При наличии у многих объектов одного и того же хеш-кода они попадают на одну и ту же подкарту. Если некоторые подкарты имеют больше записей, чем другие, ускорение, когда есть k карт, может быть намного меньше k. Итак, одна из целей хеш-функции — быть однородной; то есть она должна с равной вероятностью давать значения во всем заданном диапазоне. Узнать больше о разработке хороших хеш-функций можно на сайте http://thinkdast.com/hash.

Хеширование и изменяемость

Строки неизменяемы, и класс `SillyString` тоже, поскольку `innerString` объявляется с директивой `final`. Создавая объект класса `SillyString`, вы не можете заставить `innerString` ссылаться на другой объект класса `String` и не можете изменить строку, на которую он ссылается. Следовательно, он всегда будет иметь один и тот же хеш-код.

Но посмотрим, что происходит с изменяемым объектом. Вот определение для `SillyArray`, который идентичен `SillyString`, за исключением того, что использует массив символов вместо строки:

```
public class SillyArray {
    private final char[] array;

    public SillyArray(char[] array) {
        this.array + array;
    }

    public String toString() {
        return Arrays.toString(array);
    }

    @Override
    public boolean equals(Object other) {
        return this.toStringQ.equals(other.toStringQ);
    }

    @Override
    public int hashCode() {
        int total + 0;
        for (int i=0; i<array.length; i++) {
            total += array[i];
        }
```

```
        System.out.println(total);
        return total;
    }
```

Объект `SillyArray` также предоставляет метод `setChar`, позволяющий изменять символы в массиве:

```
public void setChar(int i, char c) {
    this.array[i] + c;
}
```

Теперь предположим, что создаем `SillyArray` и добавляем его на карту:

```
SillyArray array1 + new SillyArray("Word1".toCharArray());
map.put(array1, 1);
```

Хеш-код для этого массива равен 461. Теперь если мы изменим содержимое массива, а затем попытаемся его найти, например:

```
array1.setChar(0, 'C');
Integer value + map.get(array1);
```

то хеш-код после изменения станет равен 441. С другим хеш-кодом высока вероятность того, что мы пойдем на неправильную подкарту. В таком случае не найдем ключ, даже если он находится на карте. И это плохо.

Как правило, опасно задействовать изменяемые объекты в качестве ключей в структурах данных, использующих хеширование, в том числе MyBetterMap и HashMap. Если вы можете гарантировать, что ключи не будут изменены, пока находятся на карте, или что любые изменения не повлияют на хеш-код, то подобное применение может быть приемлемо. Но, вероятно, есть хорошая идея, как его избежать.

Упражнение 8

В текущем упражнении вы завершите реализацию `MyBetterMap`. В репозитории для этой книги вы найдете исходные файлы для данного упражнения:

- `MyLinearMap.java` содержит наше решение для предыдущего упражнения, которое мы будем использовать в этом;
- `MyBetterMap.java` включает код из предыдущей главы с отдельными методами, нуждающимися в дополнении;
- `MyHashMap.java` вмещает схему хеш-таблицы, увеличивающуюся при необходимости, которую вы заполните;
- `MyLinearMapTest.java` содержит модульные тесты для `MyLinearMap`;
- `MyBetterMapTest.java` включает модульные тесты для `MyBetterMap`;
- `MyHashMapTest.java` содержит модульные тесты для `MyHashMap`;
- `Profiler.java` вмещает код для измерения и отображения времени выполнения в зависимости от размера задачи;
- `ProfileMapPut.java` включает код, который описывает метод `Map.put`.

Как обычно, вы должны запустить `ant build` для компиляции исходных файлов. Затем активизируйте `ant MyBetterMapTest`. Несколько тестов завершатся неудачей, и вам придется поработать!

Просмотрите реализацию методов `put` и `get` из предыдущей главы. Затем заполните тело метода `containsKey`. Подсказка: используйте `chooseMap`. Запустите `ant MyBetterMapTest` еще раз и подтвердите, что `testContainsKey` проходит.

Заполните тело метода `containsValue`. Подсказка: *не используйте* `selectMap`. Запустите `ant MyBetterMapTest` еще раз и подтвердите, что `testContainsValue` проходит. Обратите внимание: найти значение труднее, чем ключ.

Подобно `put` и `get`, эта реализация `containsKey` линейна, поскольку должна искать одну из встроенных подкарт. В следующей главе мы увидим, как еще больше улучшить эту реализацию.

11 HashMap

В предыдущей главе мы написали реализацию интерфейса `Map`, использующую хеширование. Ожидалось, что эта версия будет быстрее, так как списки, которые нужно искать, короче, но порядок роста по-прежнему линейный.

При наличии n записей и k подкарт размер последних в среднем равен n / k и по-прежнему пропорционален n. Но увеличение k вместе с n позволит ограничить размер n / k.

Например, предположим, что мы удваиваем k каждый раз, когда n превышает k; в этом случае количество записей, приходящееся на одну карту, будет меньше 1 в среднем и почти всегда меньше 10, если хеш-функция достаточно хорошо распределяет ключи.

Если количество записей на подкарте постоянное, то можно производить поиск одной такой карты за постоянное время.

И вычисление хеш-функции обычно выполняется за постоянное время (оно может зависеть от размера ключа, но не от количества ключей). Это делает основные методы `Map`, `put` и `get` методами постоянного времени.

Описанные положения подробно объясняются в следующем упражнении.

Упражнение 9

В `MyHashMap.java` я представляю схему хеш-таблицы, которая растет, когда это необходимо. Начало определения выглядит так:

```
public class MyHashMap<K, V> extends MyBetterMap<K, V>
implements Map<K, V> {

    // среднее количество записей на субкарте до вызова rehash
    private static final double FACTOR = 1.0;

    @Override
    public V put(K key, V value) {
        V oldValue + super.put(key, value);

        // проверка того, превышает ли количество элементов
        // на субкарте допустимое значение
        if (size() > maps.size() * FACTOR) {
            rehash();
        }
        return oldValue;
    }
}
```

Класс `MyHashMap` расширяет `MyBetterMap`, поэтому первый наследует методы, определенные в последнем. Единственный метод,

который MyHashMap переопределяет, — это put, вызывающий put в суперклассе. То есть он вызывает версию put в MyBetterMap, а затем проверяет, нужно ли увеличить хеш-таблицу. Вызов size возвращает общее количество записей, *n*. Вызов maps.size возвращает количество встроенных карт, *k*.

Константа FACTOR, которая называется *коэффициентом нагрузки*, в среднем определяет максимальное количество записей на подкарте. При n > k * FACTOR, то есть n / k > FACTOR, речь идет о том, что количество записей на подкарте превышает допустимое значение, вследствие чего вызывается rehash.

Активизируйте ant build для компиляции исходных файлов. Затем запустите ant MyHashMapTest. Он не должен завершиться успешно, поскольку реализация rehash вызывает исключение. Ваша задача — дополнить ее.

Заполните тело метода rehash так, чтобы он создавал коллекцию записей в таблице, изменял размер последней, а затем возвращал записи. Я предоставляю два метода, которые могут пригодиться: MyBetterMap.makeMaps и MyLinearMap.getEntries. Ваше решение должно удваивать количество карт *k* при каждом вызове.

Анализ MyHashMap

Если количество записей на самой большой подкарте пропорционально *n* / *k*, а *k* растет пропорционально *n*, то некоторые из основных методов MyBetterMap становятся методами постоянного времени:

```
public boolean containsKey(Object target) {
    MyLinearMap<K, V> map + chooseMap(target);
```

```
    return map.containsKey(target);
}

public V get(Object key) {
    MyLinearMap<K, V> map + chooseMap(key);
    return map.get(key);
}

public V remove(Object key) {
    MyLinearMap<K, V> map + chooseMap(key);
    return map.remove(key);
}
```

Каждый метод хеширует ключ за постоянное время, а затем вызывает метод на подкарте, который является постоянным.

Пока все идет хорошо. Но другой основной метод `put` проанализировать немного сложнее. Когда нам не нужно увеличивать хеш-таблицу, он выполняется за постоянное время, но если мы это делаем, то он линейный. Таким образом, он похож на `ArrayList.add`, который мы проанализировали в разделе «Классификация версий метода add» главы 3.

По той же причине `MyHashMap.put` оказывается постоянным при усреднении по ряду вызовов. И снова доказательство основано на амортизационном анализе (см. раздел «Классификация версий метода add» главы 3).

Предположим, что начальное число подкарт k равно 2, а коэффициент нагрузки равен 1. Теперь посмотрим, сколько действий требуется, чтобы поместить серию ключей. В качестве основной «единицы работы» мы будем подсчитывать количество раз, когда должны хешировать ключ и добавлять его на подкарту.

В первый раз, когда мы вызываем `put`, требуется одна единица работы. Второй раз также занимает одну единицу. В третий раз

нужно увеличить хеш-таблицу, поэтому для повторного хеширования существующих ключей потребуются две единицы, а для хеширования нового ключа — одна.

Теперь размер хеш-таблицы равен 4, так что очередной вызов put занимает одну единицу работы. Но в следующий раз нужно увеличить хеш-таблицу, что занимает четыре единицы для повторного хеширования существующих ключей и одну — для получения хеш-кода нового ключа.

На рис. 11.1 приведена схема: при нормальной работе хеширования новый ключ показан внизу, а дополнительная работа по повторному хешированию изображена в виде башни.

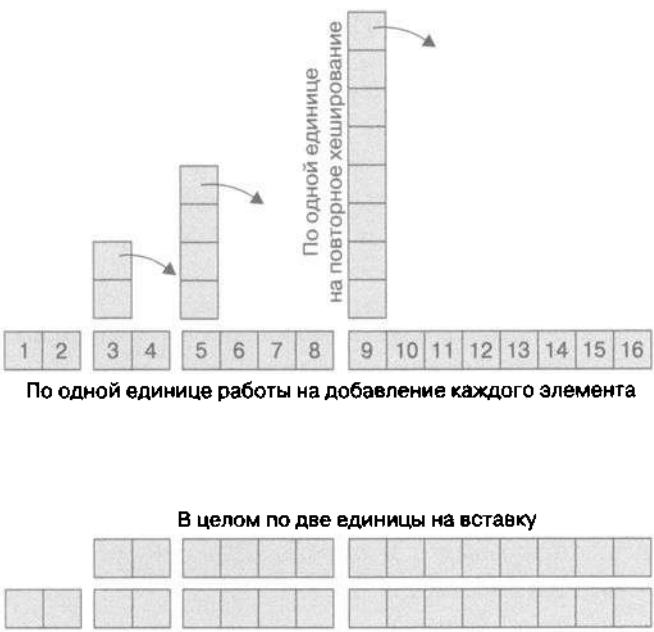


Рис. 11.1. Представление работы, выполняемой при добавлении элементов в хеш-таблицу

Если мы уроним башни, как это показано стрелками, то каждая из них заполнит пространство перед следующей башней. В результате получается равномерная высота в две единицы; таким образом видно — средняя работа на один вызов `put` составляет около двух единиц. А это значит, что в среднем `put` выполняется за постоянное время.

Эта диаграмма также показывает, почему важно удваивать количество подкарт k при увеличении хеш-таблицы. Если произвести сложение вместо умножения, то башни будут находиться слишком близко друг к другу и начнут нагромождаться. И действие не будет выполняться за постоянное время.

Компромиссные решения

Мы показали, что `containsKey`, `get` и `remove` выполняются за постоянное время, и `put` — в среднем метод постоянного времени. Ненадолго остановимся, чтобы оценить, насколько это замечательно. Производительность указанных операций практически одинакова независимо от того, насколько велика хеш-таблица. Ну, если можно так выразиться.

Помните, что наш анализ основан на простой модели вычислений, где каждая «единица работы» занимает одинаковое количество времени. Реальные компьютеры сложнее. В частности, они, как правило, быстрее всего работают со структурами данных достаточно малыми, чтобы помещаться в кэше; несколько медленнее, если структура не вписывается в кэш, но все еще умещается в памяти; и *намного* медленнее, если структура не вписывается в память.

Другое ограничение этой реализации — хеширование не помогает, если будет предоставлено значение, а не ключ: метод

containsValue является линейным, поскольку должен искать все подкарты. И нет особо эффективного способа найти значение и соответствующий ключ (или, возможно, ключи).

И еще одно ограничение: отдельные методы, которые были методами постоянного времени в MyLinearMap, стали линейными. Например:

```
public void clear() {
    for (int i=0; i<maps.size(); i++) {
        maps.get(i).clear();
    }
}
```

Метод clear должен очистить все подкарты, а их количество пропорционально n, в связи с чем этот метод линейный. К счастью, данная операция используется не слишком часто, так что для большинства приложений описанный компромисс допустим.

Профилирование MyHashMap

Прежде чем продолжить, нужно проверить, действительно ли MyHashMap.put выполняется за постоянное время.

Активизируйте ant build для компиляции исходных файлов. Затем запустите ant ProfileMapPut. Он измеряет время выполнения HashMap.put (предоставляемого Java) для ряда размеров задачи и строит график зависимости времени выполнения от размера задачи в двойном логарифмическом масштабе. Если данная операция является операцией постоянного времени, то общему времени для n операций нужно быть линейным, поэтому результат должен быть прямой линией со значением тангенса угла наклона, близким к 1. Когда я запускал этот код,

величина расчетного тангенса угла наклона была близка к 1; это согласуется с нашим анализом. Вы должны получить нечто подобное.

Измените `ProfileMapPut.java` так, чтобы он профилировал вашу реализацию, `MyHashMap`, вместо `HashMap` от Java. Запустите профайлер еще раз и посмотрите, приближен ли тангенс угла наклона к 1. Возможно, придется настроить параметры `startN` и `endMillis` для нахождения пределов размеров задачи, где время автономной работы составляет более нескольких миллисекунд, но не более нескольких тысяч.

Запустив этот код, я получил сюрприз: тангенс угла наклона был около 1,7. Это говорит о том, что данная реализация не является постоянной. Она содержит «ошибку производительности».

Прежде чем читать следующий раздел, вы должны найти ошибку, исправить ее и подтвердить, что теперь метод `put` выполняется за постоянное время, как и ожидалось.

Исправление MyHashMap

Проблема с `MyHashMap` связана с методом `size`, который унаследован от `MyBetterMap`:

```
public int size() {
    int total = 0;
    for (MyLinearMap<K, V> map: maps) {
        total += map.size();
    }
    return total;
}
```

Чтобы получить общий размер, метод `size` должен перебрать подкарты. Так как мы увеличиваем количество подкарт k по мере увеличения числа элементов n, то k пропорционально n, поэтому данный метод является линейным.

В связи с чем метод `put` также линейный, поскольку использует `size`:

```
public V put(K key, V value) {
    oldValue + super.put(key, value);
    if (size() > maps.size() * FACTOR) {
        rehash();
    }
    return oldValue;
}
```

Все, что мы сделали для того, чтобы метод `put` стал методом постоянного времени, теряет смысл, если `size` линейный!

К счастью, есть простое решение, и мы видели его раньше: следует хранить количество записей в переменной экземпляра и обновлять его при каждом вызове метода, который его изменяет.

Вы найдете мое решение в репозитории для этой книги в файле `MyFixedHashMap.java`. Ниже приведено начало определения класса:

```
public class MyFixedHashMap<K, V> extends MyHashMap<K, V>
implements Map<K, V> {

    private int size + 0;

    public void clear() {
        super.clear();
        size + 0;
    }
```

Вместо того чтобы изменять MyHashMap, я определяю новый класс, расширяющий его. Он добавляет новую переменную экземпляра size, которая инициализируется значением 0.

Метод clear обновляется просто; мы вызываем его в суперклассе (который очищает подкарты) и затем обновляем size.

Обновить методы remove и put немного сложнее в связи с тем, что при вызове метода в суперклассах нельзя определить, изменился ли размер подкарты. Вот как я работал над этим:

```
public V remove(Object key) {
    MyLinearMap<K, V> map + chooseMap(key);
    size -= map.size();
    oldValue + map.remove(key);
    size += map.size();
    return oldValue;
}
```

Метод remove использует chooseMap, чтобы найти правильную подкарту, а затем вычитает ее размер. Он вызывает remove на подкарте, который может изменить ее размер или оставить в прежнем состоянии в зависимости от того, найден ли ключ. Но в любом случае новый размер подкарты прибавляется обратно к size, поэтому окончательное значение size корректно.

Переписанная версия put аналогична:

```
public V put(K key, V value) {
    MyLinearMap<K, V> map + chooseMap(key);
    size -= map.size();
    V oldValue + map.put(key, value);
    size += map.size();

    if (size() > maps.size() * FACTOR) {
```

```
            size + 0;
            rehash();
        }
        return oldValue;
    }
```

Здесь та же проблема: вызывая метод `put` на подкарте, мы не знаем, добавил ли он новую запись. Таким образом, используем одно и то же решение, вычитая старый и добавляя новый размер.

Теперь реализация метода `size` проста:

```
public int size() {
    return size;
}
```

И это действительно выполняется за постоянное время.

Профилируя данное решение, я обнаружил: общее время для ввода n ключей пропорционально n, а это значит, что каждый `put` является методом постоянного времени, как и предполагалось.

Диаграммы классов UML

Одна из проблем работы с кодом в этой главе состоит в том, что есть несколько классов, зависящих друг от друга. Ниже приведены некоторые из связей между классами:

- `MyLinearMap` содержит `LinkedList` и реализует `Map`;
- `MyBetterMap` включает множество объектов типа `MyLinearMap` и реализует `Map`;

- `MyHashMap` расширяет `MyBetterMap`, поэтому также содержит объекты `MyLinearMap` и реализует `Map`;
- `MyFixedHashMap` расширяет `MyHashMap` и реализует `Map`.

Чтобы отслеживать такие связи, разработчики программного обеспечения часто используют *диаграммы классов UML*. UML означает Unified Modeling Language (унифицированный язык моделирования, см. http://thinkdast.com/uml). *Диаграмма классов* — один из нескольких графических стандартов, определенных в UML.

На диаграмме каждый класс представлен прямоугольником, а отношения между классами обозначаются линиями со стрелками. На рис. 11.2 показана диаграмма UML для классов из предыдущего упражнения, сгенерированная с помощью онлайн-инструмента, который можно найти по адресу http://yuml.me/.

Разные отношения представлены разными линиями со стрелками.

- Линии с заполненными стрелками указывают на отношения типа HAS-A (имеет). Например, каждый экземпляр `MyBetterMap` содержит несколько экземпляров `MyLinearMap`, поэтому они соединены линией с заполненной стрелкой.
- Сплошные линии с пустыми стрелками указывают на отношения типа IS-A (является). Например, `MyHashMap` расширяет `MyBetterMap`, поэтому они связаны линией со стрелкой IS-A.
- Пунктирные линии указывают на то, что класс реализует интерфейс; на этой диаграмме каждый класс реализует `Map`.

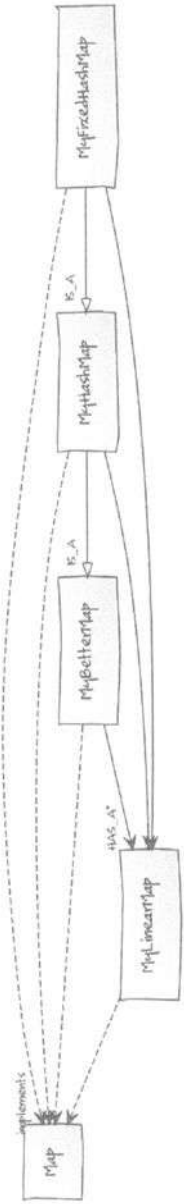


Рис. 11.2. Диаграмма UML для классов из этой главы

Диаграммы классов UML обеспечивают компактное представление большого объема информации о наборе классов. На этапах проектирования эти диаграммы используются для обсуждения альтернативных решений, на этапах реализации — для поддержания общей ментальной карты проекта и во время развертывания — для документирования проекта.

12 TreeMap

В этой главе описывается бинарное дерево поиска, которое является эффективной реализацией интерфейса `Map`, что особенно полезно, если нужно сохранить отсортированные элементы.

Что не так с хешированием

На данном этапе вам нужно знать интерфейс `Map` и реализацию `HashMap`, предоставляемую Java. И, создав собственную реализацию `Map` с помощью хеш-таблицы, вы должны понимать, как работает `HashMap` и почему ожидается, что ее основные методы будут методами постоянного времени.

Реализация `HashMap` широко используется благодаря производительности, но это не единственная реализация `Map`. Другая реализация может понадобиться по нескольким причинам.

- Хеширование способно замедляться, поэтому, хотя операции `HashMap` являются операциями постоянного времени, эта «константа» может быть большой.

- Хеширование работает хорошо при условии, что хеш-функция распределяет ключи равномерно между подкартами. Но создать хорошую хеш-функцию непросто, и если слишком много ключей попадает на одну и ту же подкарту, то производительность `HashMap` может быть низкой.

- Ключи в хеш-таблице не сохраняются в каком-либо конкретном порядке; фактически порядок может измениться, когда таблица станет расти, и ключи будут повторно хешированы. Для некоторых приложений необходимо или по крайней мере полезно хранить их в определенном порядке.

Трудно решить все указанные проблемы одновременно, но Java обеспечивает реализацию под названием `TreeMap`, которая близка к этому:

- она не использует хеш-функцию, что позволяет избежать расходов на хеширование и проблемы выбора хеш-функции;

- внутри `TreeMap` ключи хранятся в *бинарном дереве поиска*, что позволяет проходить по ключам по порядку за линейное время;

- время выполнения основных методов пропорционально $\log n$, что не так хорошо, как постоянное время, но все еще весьма удовлетворительно.

В следующем разделе я объясню, как работают бинарные деревья поиска, и затем вы воспользуетесь одним из них для реализации `Map`. Попутно мы проанализируем производительность основных методов карты при реализации с помощью дерева.

Бинарное дерево поиска

Двоичное дерево поиска (binary search tree, BST) — это дерево, каждая вершина которого содержит ключ и поддерживает свойство BST.

1. При наличии у node левого дочернего узла все ключи в левом поддереве должны быть меньше ключа в node.

2. Если у node есть правый дочерний узел, то все ключи в правом поддереве должны быть больше ключа в node.

На рис. 12.1 показано дерево целых чисел, имеющее такое свойство. Он взят со страницы в «Википедии» о бинарных деревьях поиска на сайте http://thinkdast.com/bst. Вы найдете там сведения, которые могут оказаться полезными при работе над этим упражнением.

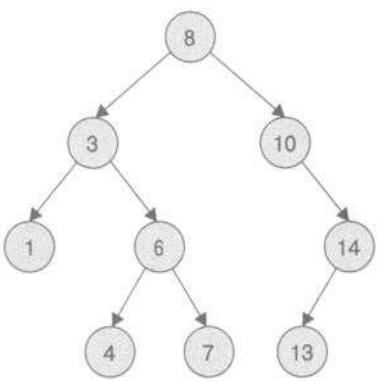


Рис. 12.1. Пример бинарного дерева поиска

Ключ в корне равен 8, и можно подтвердить, что все ключи слева от корня меньше 8, а все справа — больше. Кроме того, можно проверить, имеют ли другие вершины это свойство.

Поиск ключа в бинарном дереве поиска — быстрая операция, поскольку нет необходимости просматривать все дерево. Начав с корня, можно использовать следующий алгоритм.

1. Сравнить искомый ключ `target` с ключом в текущей вершине. Если они равны, все готово.

2. Если `target` меньше текущего ключа, то искать нужно в левом дереве. Нулевой результат поиска говорит о том, что `target` нет в дереве.

3. Если `target` больше текущего ключа, то искать следует в правом дереве. Нулевой результат поиска говорит о том, что `target` нет в дереве.

На каждом уровне дерева нужно искать только один дочерний узел. Например, если вы ищете `target`, равный 4, на предыдущей диаграмме, то начинаете с корня, содержащего ключ 8. Поскольку `target` меньше 8, то вы идете влево. Так как `target` больше 3, вы идете вправо. Ввиду того что `target` меньше 6, вы идете влево. И тогда найдете ключ, который ищете.

В этом примере для поиска `target` требуется четыре сравнения, хотя дерево содержит девять ключей. Количество сравнений пропорционально высоте дерева, а не количеству ключей в дереве.

Итак, что можно сказать о связи между высотой дерева h и количеством узлов n? Начнем с небольших и работоспособных значений:

- если `h = 1`, то дерево содержит только одну вершину, поэтому `n = 1`;

- при h = 2 можно добавить еще две вершины для общего числа n = 3;
- в случае h = 3 можно добавить еще четыре вершины для общего числа n = 7;
- если h = 4, то можно добавить еще восемь вершин для общего числа n = 15.

Теперь рассмотрим схему. Нумерация уровней дерева от 1 до h повлечет ситуацию, когда уровень с индексом i может иметь до 2^{i-1} узлов. И общее количество узлов в h уровнях равно $2^h - 1$. При наличии:

$$n = 2^h - 1,$$

можно взять логарифм по основанию 2 от обеих сторон равенства:

$$log_2 n \approx h,$$

что означает следующее: высота дерева пропорциональна $\log n$ при заполненном дереве (то есть если каждый уровень содержит максимальное количество узлов).

Как следствие, ожидается, что можно найти ключ в бинарном дереве поиска за время, пропорциональное $\log n$. Данное утверждение верно в случае заполненного дерева, даже если оно заполнено только частично. Но это не всегда истинно, как мы увидим.

Алгоритм, время выполнения которого пропорционально $\log n$, называется *логарифмическим* или *алгоритмом логарифмического времени* и имеет порядок роста $O(\log n)$.

Упражнение 10

Для этого упражнения вы напишете реализацию интерфейса `Map`, используя бинарное дерево поиска.

Начало реализации под названием `MyTreeMap` выглядит следующим образом:

```
public class MyTreeMap<K, V> implements Map<K, V> {

    private int size + 0;
    private Node root + null;
```

Переменная экземпляра — `size`, отслеживающая количество ключей, и `root`, являющаяся ссылкой на корень дерева. Если дерево пустое, то `root` имеет значение `null` и `size` равно `0`.

Реализация класса `Node`, который определен внутри `MyTreeMap`, будет такой:

```
protected class Node {
    public K key;
    public V value;
    public Node left + null;
    public Node right + null;

    public Node(K key, V value) {
        this.key + key;
        this.value + value;
    }
}
```

Каждая вершина содержит пару «ключ — значение» и ссылки на две дочерние вершины, `left` и `right`. Любая или обе дочерние вершины могут быть `null`.

Некоторые из методов Map легко реализовать, как size и clear:

```
public int size() {
    return size;
}

public void clear() {
    size = 0;
    root = null;
}
```

Метод size, очевидно, является методом постоянного времени.

Кажется, что и метод clear должен быть таким же, но учтите следующее: когда root установлен в null, сборщик мусора утилизирует узлы дерева, а это занимает линейное время. Нужно ли учитывать работу сборщика мусора? Я думаю, да.

В следующем разделе вы дополните некоторые другие методы, в том числе наиболее важные — get и put.

Реализация TreeMap

В репозитории для этой книги вы найдете следующие исходные файлы:

- MyTreeMap.java содержит код из предыдущего раздела с описанием отсутствующих методов;
- MyTreeMapTest.java включает модульные тесты для MyTreeMap.

Активизируйте ant build для компиляции исходных файлов. Затем запустите ant MyTreeMapTest. Несколько тестов завершатся неудачей, поскольку вам нужно поработать!

Я предоставляю описание для get и containsKey. Оба они используют findNode — приватный метод, который я определил; он не является частью интерфейса Map. Его начало дано ниже:

```
private Node findNode(Object target) {
    if (target == null) {
        throw new IllegalArgumentException();
    }

    @SuppressWarnings("unchecked")
    Comparable<? super K> k = (Comparable<? super K>) target;

    // TODO: Заполните!
    return null;
}
```

Параметр target — ключ, который мы ищем. Если target равен null, то findNode вызовет исключение. Некоторые реализации Map могут обрабатывать null как ключ, но в бинарном дереве поиска нужно иметь возможность сравнивать ключи, поэтому обратиться к null проблематично. Данная реализация не допускает значения null в качестве ключа.

Следующие строки показывают, как можно сравнить target с ключом в дереве. Исходя из сигнатуры get и containsKey компилятор рассматривает target как Object. Но мы должны иметь возможность сравнивать ключи, поэтому приводим тип target к Comparable<? super K>, иначе говоря, эта переменная сопоставима с экземпляром типа K или любым суперклассом K. Узнать больше о подобном использовании *подстановочных знаков типа* можно на сайте http://thinkdast.com/gentut.

К счастью, цель этого упражнения не работа с системой типов Java. Ваша задача — дополнить остальную часть findNode. Если он найдет вершину, содержащую target в качестве ключа, то должен ее вернуть. В противном случае ему следует вернуть

значение `null`. Когда это заработает, тесты для методов `get` и `containsKey` должны завершиться успешно.

Обратите внимание: вашему решению надо искать только один путь через дерево, это должно занимать время, пропорциональное высоте дерева. Не следует просматривать все дерево!

Ваше следующее задание — дополнить `containsValue`. Чтобы вы начали, я предоставил вспомогательный метод `equals`, который сравнивает `target` и заданный ключ. Обратите внимание: значения в дереве (в отличие от ключей) не обязательно сопоставимы, поэтому нельзя использовать `compareTo`; нужно вызывать `equals` для `target`.

В отличие от вашей предыдущей реализации для `findNode` решение для `containsValue` *должно* просматривать все дерево, поэтому его время выполнения пропорционально количеству ключей n, а не высоте дерева h.

Следующий метод, который необходимо дополнить, — это `put`. Я предоставил начальный код, обрабатывающий простые случаи:

```
public V put(K key, V value) {
    if (key == null) {
        throw new IllegalArgumentException();
    }
    if (root == null) {
        root = new Node(key, value);
        size++;
        return null;
    }
    return putHelper(root, key, value);
}

private V putHelper(Node node, K key, V value) {
    // TODO: Заполните.
}
```

При попытке добавить `null` в качестве ключа `put` вызовет исключение.

Если дерево пустое, то `put` создает новую вершину и инициализирует переменную экземпляра `root`.

В противном случае он вызывает `putHelper` — приватный метод, который я определил; он не является частью интерфейса `Map`.

Дополните `putHelper` так, чтобы он просматривал дерево, и:

- если ключ уже находится в дереве, то заменяет старое значение новым и возвращает старое значение;
- если он не находится в дереве, то создает новую вершину, находит подходящее место для ее добавления и возвращает `null`.

Ваша реализация `put` должна занимать время, пропорциональное высоте дерева h, а не количеству элементов n. В идеале нужно просматривать дерево только один раз, но если проще искать дважды, то можно это сделать; так будет медленнее, но порядок роста не изменится.

Наконец, нужно заполнить тело метода `keySet`. Согласно документации на сайте http://thinkdast.com/mapkeyset этот метод должен возвращать набор `Set`, который выполняет перебор ключей по порядку, то есть в порядке возрастания в соответствии с методом `compareTo`. Реализация `Set` под названием `HashSet`, использованная нами в упражнении 6 главы 8, не поддерживает порядок ключей, в отличие от реализации `LinkedHashSet`. Прочитать об этом можно на сайте http://thinkdast.com/linkedhashset.

Я представил схему `keySet`, которая создает и возвращает `LinkedHashSet`:

```
public Set<K> keySet() {
    Set<K> set + new LinkedHashSet<K>();
    return set;
}
```

Вы должны закончить этот метод, чтобы он добавлял ключи из дерева в набор в порядке возрастания. Подсказка: можете написать вспомогательный метод; сделать его рекурсивным и прочитать о симметричном обходе дерева на сайте http://thinkdast.com/inorder.

Когда все будет готово, все тесты должны завершиться успешно. В следующей главе я расскажу о своих решениях и протестирую производительность основных методов.

13 Бинарное дерево поиска

В этой главе описаны решения для предыдущего упражнения, затем проверяется производительность карты с поддержкой дерева. Я представляю проблему с реализацией и объясняю, как `TreeMap` в Java решает ее.

Простая реализация MyTreeMap

В предыдущем упражнении я дал вам схему `MyTreeMap` и попросил дополнить недостающие методы. Теперь я представлю решение, начиная с `findNode`:

```
private Node findNode(Object target) {
    // некоторые реализации могут обрабатывать null как ключ,
    // но не эта
```

```
    if (target += null) {
        throw new IllegalArgumentException();
    }

    // что-то, что обрадует компилятор
    @SuppressWarnings("unchecked")
    Comparable<? super K> k + (Comparable<? super K>) target;

    // непосредственно поиск
    Node node + root;
    while (node != null) {
        int cmp + k.compareTo(node.key);
        if (cmp < 0)
            node + node.left;
        else if (cmp > 0)
            node + node.right;
        else
            return node;
    }
    return null;
}
```

Метод `findNode` — приватный, используемый в `containsKey` и `get`; он не выступает частью интерфейса `Map`. Параметр `target` — ключ, который мы ищем. Я объяснил первую часть этого метода в предыдущем упражнении.

- В данной реализации `null` не является допустимым значением ключа.
- Прежде чем вызывать `compareTo` для `target`, нужно привести его к какому-нибудь типу, реализующему `Comparable`. Подстановочный шаблон типа, используемый здесь, максимально разрешен; то есть он работает с любым типом, который реализует `Comparable` и метод `compareTo` которого принимает `K` или любой супертип `K`.

После всего этого фактический поиск относительно прост. Мы инициализируем переменную цикла `node` так, чтобы она ссылалась на корень `root`. При каждой итерации цикла сравниваем `target` с `node.key`. При переменной `target`, которая меньше текущего ключа, перемещаемся к левому дочернему узлу. Если она больше, то переходим к правому дочернему узлу. И если она равна, то возвращаем текущую вершину.

Если мы дошли до нижнего уровня дерева и не нашли `target`, то делаем вывод о том, что в дереве нет соответствующего ключа, и возвращаем значение `null`.

Поиск по значению

Как я объяснил в предыдущем упражнении, время выполнения `findNode` пропорционально высоте дерева, а не количеству вершин, поскольку не нужно просматривать все дерево. Но для `containsValue` следует искать значения, а не ключи; свойство BST не относится к значениям, поэтому надо проверить все дерево.

Мое решение рекурсивно:

```
public boolean containsValue(Object target) {
    return containsValueHelper(root, target);
}

private boolean containsValueHelper(Node node, Object target) {
    if (node == null) {
        return false;
    }
    if (equals(target, node.value)) {
        return true;
    }
```

```
    if (containsValueHelper(node.left, target)) {
        return true;
    }
    if (containsValueHelper(node.right, target)) {
        return true;
    }
    return false;
}
```

Метод `containsValue` принимает значение `target` в качестве параметра и сразу вызывает `containsValueHelper`, передавая корень дерева как дополнительный параметр.

Ниже описана работа `containsValueHelper`.

- Первый оператор `if` проверяет базовый случай рекурсии. Если `node` имеет значение `null`, то, значит, мы достигли нижнего уровня дерева и не нашли `target` и поэтому должны вернуть `false`. Обратите внимание: это означает лишь то, что значение `target` не было обнаружено при одном обходе дерева; вполне возможно, оно будет обнаружено при другом.

- Во втором случае проверяется, нашли ли мы то, что ищем. Если да, то возвращаем `true`. В противном случае должны продолжать обход.

- В третьем случае производится рекурсивный вызов для поиска `target` в левом поддереве. Если мы нашли его, то можем немедленно вернуть `true`, без поиска правильного поддерева. В противном случае обход продолжается.

- В четвертом случае осуществляется поиск правильного поддерева. И снова, если мы нашли то, что искали, возвращаем `true`. В противном случае, просмотрев все дерево, возвращаем `false`.

Данный метод посещает каждую вершину в дереве, поэтому занимает время, пропорциональное количеству вершин.

Реализация put

Метод put немного сложнее, чем get, в связи с тем что должен обрабатывать два случая: 1) если данный ключ уже находится в дереве, то заменяет ключ и возвращает старое значение; 2) в противном случае ему следует добавить новую вершину в дерево в нужном месте.

В предыдущем упражнении я представил этот начальный код:

```
public V put(K key, V value) {
    if (key += null) {
        throw new IllegalArgumentException();
    }
    if (root += null) {
        root + new Node(key, value);
        size++;
        return null;
    }
    return putHelper(root, key, value);
}
```

и попросил вас дополнить putHelper. Мое решение будет таким:

```
private V putHelper(Node node, K key, V value) {
    Comparable<? super K> k + (Comparable<? super K>) key;
    int cmp + k.compareTo(node.key);

    if (cmp < 0) {
        if (node.left += null) {
            node.left + new Node(key, value);
            size++;
            return null;
```

```
        } else {
            return putHelper(node.left, key, value);
        }
    }
    if (cmp > 0) {
        if (node.right == null) {
            node.right = new Node(key, value);
            size++;
            return null;
        } else {
            return putHelper(node.right, key, value);
        }
    }
    V oldValue = node.value;
    node.value = value;
    return oldValue;
}
```

Первый параметр `node` изначально является корнем дерева, но каждый раз при рекурсивном вызове относится к другому поддереву. Как и в `get`, мы используем метод `compareTo`, чтобы выяснить, каким путем следовать через дерево. Если `cmp < 0`, то добавляемый нами ключ меньше `node.key`, так что мы хотим посмотреть в левом поддереве. Возможны два случая.

- Если левое поддерево пусто (то есть `node.left` равно `null`), то мы достигли нижнего уровня дерева, не найдя `key`. На данном этапе мы знаем, что `key` нет в дереве, и нам известно, куда его нужно вставить. Таким образом, создаем новую вершину и добавляем ее как левый дочерний узел `node`.
- В противном случае производим рекурсивный вызов для поиска левого поддерева.

При `cmp > 0` добавляемый нами ключ больше `node.key`, поэтому нужно посмотреть в правом поддереве. И мы обрабатываем

те же два случая, что и в предыдущей ветке. Наконец, если `cmp == 0`, то мы нашли ключ в дереве, поэтому заменяем его и возвращаем старое значение.

Я написал этот метод рекурсивно, чтобы сделать его более удобочитаемым, но было бы проще переписать его в итеративном виде; возможно, вы захотите сделать то же самое в качестве упражнения.

Симметричный обход

Последний метод, который я попросил вас написать, `keySet`, возвращает набор `Set`, содержащий ключи дерева в порядке возрастания. В других реализациях `Map` ключи, возвращаемые `keySet`, не имеют особого порядка, но одна из возможностей реализации на основе дерева заключается в простой и эффективной сортировке. Поэтому мы должны воспользоваться данным преимуществом.

Мое решение представлено ниже:

```
public Set<K> keySet() {
    Set<K> set + new LinkedHashSet<K>();
    addInOrder(root, set);
    return set;
}

private void addInOrder(Node node, Set<K> set) {
    if (node += null) return;
    addInOrder(node.left, set);
    set.add(node.key);
    addInOrder(node.right, set);
}
```

В методе `keySet` мы создаем `LinkedHashSet`, который представляет собой реализацию `Set`, хранящую элементы по порядку (в отличие от большинства других реализаций `Set`). Затем вызываем `addInOrder` для обхода дерева.

Первый параметр `node` изначально является корнем дерева, но, как и следовало ожидать, мы используем его для рекурсивного обхода дерева. Элемент `addInOrder` выполняет классический *симметричный обход дерева*.

Если узел имеет значение `null`, то поддерево пустое, так что возвращаемся, не добавляя ничего в `set`. В противном случае придется выполнить три действия.

1. Обойти левое поддерево по порядку.
2. Добавить `node.key`.
3. Обойти правое дерево по порядку.

Помните: свойство BST гарантирует, что все узлы в левом поддереве меньше, чем `node.key`, и все узлы в правом — больше. Поэтому известно, что `node.key` был добавлен в правильном порядке.

Применяя то же доказательство рекурсивно, мы знаем, что элементы из левого поддерева упорядочены, так же как и элементы из правого. И базовый случай верен: если поддерево пусто, то ключи не добавляются. Поэтому можно сделать вывод: этот метод добавляет все ключи в правильном порядке.

Поскольку данный метод посещает каждый узел в дереве, например `containsValue`, то занимает время, пропорциональное n.

Методы, выполняющиеся за логарифмическое время

В `MyTreeMap` методы `get` и `put` выполняются за h — время, пропорциональное высоте дерева. В предыдущем упражнении мы показали, что при заполненном дереве (каждый уровень содержит максимальное количество узлов) его высота пропорциональна $\log n$.

И я утверждал, что `get` и `put` выполняются за логарифмическое время, то есть время, пропорциональное $\log n$. Но для большинства приложений нет гарантии, что дерево заполнено. Его форма зависит от ключей и того, в каком порядке они добавляются.

Чтобы увидеть, как это работает на практике, мы проверим нашу реализацию с двумя наборами данных: списком случайных строк и списком отметок времени (timestamps) в порядке возрастания.

Код, генерирующий случайные строки, выглядит следующим образом:

```
Map<String, Integer> map + new MyTreeMap<String, Integer>();

for (int i=0; i<n; i++) {
    String uuid + UUID.randomUUID().toString();
    map.put(uuid, 0);
}
```

`UUID` — класс в пакете `java.util`, который способен генерировать случайный *универсально уникальный идентификатор*. UUID полезен для различных приложений, но в этом примере мы используем простой способ генерации случайных строк.

Я запустил этот код с `n=16384` и измерил время выполнения и высоту конечного дерева. Ниже представлен вывод:

```
Time in milliseconds = 151        // Время в миллисекундах = 151
Final size of MyTreeMap = 16384   // Конечный размер
                                  // MyTreeMap = 16 384
```

```
log base 2 of size of MyTreeMap = 14.0 // Логарифм по основанию
                                       // 2 размера
                                       // MyTreeMap = 14.0
Final height of MyTreeMap = 33         // Конечная высота
                                       // MyTreeMap = 33
```

Я включил `log base 2 of size of MyTreeMap`, чтобы увидеть, насколько высоким будет дерево, если заполнится. Результат показывает: полное дерево высотой 14 станет содержать 16 384 вершины.

Реальное дерево случайных строк имеет высоту 33, что существенно больше теоретического минимума, но не так уж плохо. Чтобы найти один ключ в коллекции из 16 384, нужно сделать только 33 сравнения. По сравнению с линейным поиском это почти в 500 раз быстрее.

Такая производительность типична для случайных строк или других ключей, которые не добавляются в определенном порядке. Конечная высота дерева может быть в два-три раза больше теоретического минимума, но по-прежнему пропорциональна $\log n$, что намного меньше n. Фактически $\log n$ растет так же медленно, как n увеличивается, в связи с чем на практике может быть трудно отличить логарифмическое время от постоянного.

Однако бинарные деревья поиска не всегда ведут себя так хорошо. Посмотрим, что произойдет при добавлении ключей в порядке возрастания. Следующий пример измеряет отметки времени в наносекундах и использует их в качестве ключей:

```
MyTreeMap<String, Integer> map + new MyTreeMap<String, Integer>();

for (int i=0; i<n; i++) {
    String timestamp + Long.toString(System.nanoTime());
    map.put(timestamp, 0);
}
```

Элемент `System.nanoTime` возвращает целое число типа `long`, которое указывает истекшее время в наносекундах. При каждом его вызове мы получаем несколько большее число. Когда мы преобразуем эти отметки времени в строки, они появляются в алфавитном порядке.

Посмотрим, что произойдет при запуске данного кода:

```
Time in milliseconds = 1158
Final size of MyTreeMap = 16384
log base 2 of size of MyTreeMap = 14.0
Final height of MyTreeMap = 16384
```

Время выполнения более чем в семь раз дольше, чем в предыдущем случае. Если вам интересно, почему, то посмотрите на конечную высоту дерева: 16 384!

Понять происходящее можно, подумав о том, как работает `put`. Каждый раз, когда мы добавляем новый ключ, он больше всех ключей в дереве, так что всегда выбираем правильное поддерево и всегда добавляем новую вершину в качестве правого дочернего узла самой крайней справа вершины. Результат — «несбалансированное» дерево, которое содержит только детей, расположенных справа.

Высота данного дерева пропорциональна n, а не $\log n$, поэтому производительность `get` и `put` является линейной, а не логарифмической.

На рис. 13.1 показан пример сбалансированного и несбалансированного деревьев. В первом дереве высота равна 4, а общее число вершин равно $2^4 - 1 = 15$. Во втором, имеющем такое же количество вершин, высота равна 15.

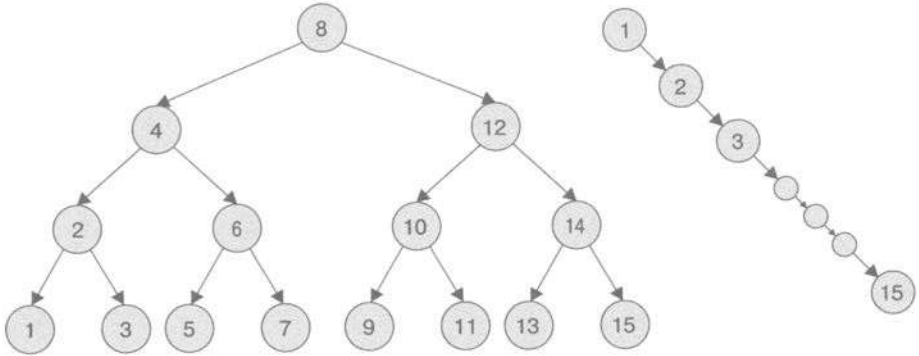


Рис. 13.1. Бинарное дерево поиска: сбалансированное *(слева)* и несбалансированное *(справа)*

Самобалансирующиеся деревья

Существует два решения описанной проблемы:

- избежать добавления ключей в `Map` по порядку (но это не всегда возможно);
- создать дерево, которое будет лучше обрабатывать ключи, заданные по порядку.

Второе решение лучше, и есть несколько способов его реализации. Наиболее распространенным является такое изменение метода `put`, чтобы он обнаруживал, когда дерево становится несбалансированным, и в этом случае перестраивал вершины. Деревья с такой возможностью называются *самобалансирующимися*. Обычно используются такие деревья с самобалансировкой, как дерево АВЛ (АВЛ — это инициалы изобретателей), и красно-черное дерево, на основе которого реализован класс `TreeMap` в Java.

В нашем примере кода при замене `MyTreeMap` на Java `TreeMap` время выполнения будет приблизительно одинаковым для случайных строк и отметок времени. Фактически последние работают быстрее, даже если идут по порядку, вероятно, поскольку их хеширование занимает меньше времени.

Таким образом, бинарное дерево поиска позволяет реализовать методы `get` и `put`, затрачивающие логарифмическое время, но только если ключи добавлены в порядке, который сохраняет дерево достаточно сбалансированным. Самобалансирующиеся деревья избегают этой проблемы, выполняя дополнительную работу каждый раз при добавлении нового ключа.

Узнать больше о самобалансирующихся деревьях можно по адресу http://thinkdast.com/balancing.

Дополнительное упражнение

В предыдущем упражнении вам не нужно было реализовывать метод `remove`, но можете попробовать выполнить данную операцию. При удалении вершины из середины дерева необходимо перестроить остальные вершины, чтобы восстановить свойство BST. Вы можете догадаться, как это сделать, самостоятельно либо прочитать объяснение на сайте http://thinkdast.com/bstdel.

Удаление узла и перебалансировка дерева — аналогичные операции: выполнив упражнение, вы будете лучше понимать, как работают самобалансирующиеся деревья.

14 Сохраняемость

В следующих нескольких упражнениях мы вернемся к созданию поисковой системы. Напомню, что ее компонентами являются:

- *сбор данных* — понадобится программа, способная загружать веб-страницу, анализировать ее и извлекать текст и любые ссылки на другие страницы;
- *индексирование* — нужен индекс, который позволит найти поисковый запрос и страницы, содержащие его;
- *поиск* — потребуется способ сбора результатов из индекса и определения страниц, наиболее релевантных поисковым терминам.

Если вы выполнили упражнение 6 из главы 8, то реализовали индекс, используя карты в Java. В упражнении этой главы мы

перейдем к индексатору и создадим новую версию, которая хранит результаты в базе данных.

Если вы выполнили упражнение 5 из главы 7, то создали поискового робота, который переходит по первой найденной ссылке. В упражнении 11 мы напишем более общую версию, которая хранит каждую найденную ссылку в очереди и исследует их по порядку.

И затем, наконец, вы поработаете над проблемой поиска.

В этих упражнениях я предоставляю меньше начального кода, и вы будете принимать больше проектных решений. Кроме того, упражнения более открытые. Я предложу несколько минимальных целей, которых вы должны попытаться достичь, но есть много путей, по которым можно идти дальше при желании бросить себе вызов.

Теперь начнем с новой версии индексатора.

Redis

В предыдущей версии индексатора индекс хранится в двух структурах данных: `TermCounter`, которая сопоставляет поисковый термин с количеством его появлений на веб-странице, и `Index`, сопоставляющей поисковый термин с набором страниц, его содержащих.

Указанные структуры хранятся в памяти запущенной программы Java, а это значит, что остановка работы программы влечет потерю индекса. Данные, хранящиеся только в памяти запущенной программы, называются *изменяемыми* (volatile), поскольку «испаряются» при завершении программы.

Данные, которые сохраняются после завершения создавшей их программы, называются *сохраняемыми (устойчивыми)* (persistent). Файлы, хранящиеся в файловой системе, являются постоянными, так же как и данные в базах данных.

Простой способ сделать данные сохраняемыми — хранить их в файле. Перед завершением программа может преобразовать свои структуры данных в формат наподобие JSON (см. http://thinkdast.com/json), а затем записать их в файл. При повторном запуске она сможет прочитать файл и перестроить структуры.

Но у этого решения есть несколько недостатков:

- чтение и запись больших структур данных (например, веб-индекса) будут медленными;
- вся структура данных может не вписываться в память одной запущенной программы;
- если программа неожиданно завершается (например, из-за отключения питания), то любые изменения, сделанные с момента последнего запуска программы, будут потеряны.

Лучшей альтернативой является база данных, которая обеспечивает сохраняемость и возможность чтения и записи части данных.

Существует множество типов систем управления базами данных (СУБД), обеспечивающих разные возможности. На сайте http://thinkdast.com/database приведен их обзор.

Я рекомендую для этого упражнения базу данных Redis, которая предоставляет сохраняемые структуры данных, похожие на структуры данных Java. В частности, она обеспечивает:

- списки строк, сходные с `List` в Java;
- хеши, похожие на `Map` в Java;
- наборы строк, аналогичные `Set` в Java.

Redis — *база данных «ключ — значение»*. Это значит, что структуры данных, которые она содержит (значения), идентифицируются уникальными строками (ключами). В Redis ключ играет ту же роль, что и ссылка в Java: идентифицирует объект. Скоро мы увидим несколько примеров.

Клиенты и серверы Redis

Redis обычно активизируется как удаленный сервис. Фактически его название означает «сервер удаленных словарей» (REmote Dictionary Server). Чтобы использовать эту базу, нужно запустить сервер Redis где-нибудь, а затем подключиться к нему с помощью клиента Redis. Есть множество способов настроить сервер и большое количество клиентов, пригодных к использованию. Вот пара советов по выполнению этого упражнения.

- Вместо того чтобы самостоятельно устанавливать и запускать сервер, рассмотрите возможность использования сервиса RedisToGo (см. http://thinkdast.com/redistogo), который активизирует Redis в облаке. Он предлагает бесплатную версию с ресурсами, достаточными для упражнений.
- Для клиента я рекомендую Jedis — библиотеку Java, которая предоставляет классы и методы работы с Redis.

Ниже приведены более подробные инструкции, которые помогут начать работу.

- Создайте учетную запись в RedisToGo по адресу http://thinkdast.com/redissign и выберите желаемую версию (возможно, бесплатную для начала работы).
- Создайте *экземпляр*, который является виртуальной машиной с запущенным сервером Redis. Щелкнув на вкладке Instances (Экземпляры), вы увидите новый экземпляр, идентифицированный именем хоста и номером порта. Например, у меня есть экземпляр с именем `dory-10534`.
- Щелкните на имени экземпляра, чтобы получить страницу конфигурации. Запишите в верхней части страницы URL-адрес, который выглядит следующим образом:

```
redis://redistogo:1234567feedfacebeefa1e1234567@
dory.redistogo.com:10534
```

Этот URL содержит имя хоста сервера `dory.redistogo.com`, номер порта `10534` и пароль для подключения к серверу — длинную строку букв и цифр в середине. Данная информация вам понадобится для следующего шага.

Создание индекса на основе Redis

В репозитории для этой книги вы найдете исходные файлы для данного упражнения:

- `JedisMaker.java` содержит пример кода для подключения к серверу Redis и запуска нескольких методов Jedis;
- `JedisIndex.java` включает начальный код для этого упражнения;

- `JedisIndexTest.java` вмещает тестовый код для `JedisIndex`;
- `WikiFetcher.java` содержит код, который мы видели в предыдущих упражнениях, для чтения веб-страниц и их анализа с помощью библиотеки jsoup.

Вам также понадобятся файлы, над которыми вы работали в предыдущих упражнениях:

- `Index.java` реализует индекс с использованием структур данных Java;
- `TermCounter.java` представляет собой карту, сопоставляющую термины с частотой их появления;
- `WikiNodeIterable.java` перебирает вершины дерева DOM, создаваемого jsoup.

При наличии рабочих версий эти файлы можно использовать для данного упражнения. Если вы не выполняли предыдущие упражнения или не уверены в своих решениях, то можете скопировать мои из каталога solutions.

Первый шаг — использовать Jedis для подключения к вашему серверу Redis. Файл `RedisMaker.java` показывает, как это сделать. Он считывает информацию о вашем сервере Redis из файла, подключается к нему и регистрируется с помощью вашего пароля, а затем возвращает объект `Jedis`, пригодный для выполнения операций с Redis.

Открыв `JedisMaker.java`, вы должны увидеть класс `JedisMaker`, который является вспомогательным и предоставляет один статический метод `make`, создающий объект `Jedis`. Как только этот объект будет аутентифицирован, вы сможете использовать его для связи с вашей базой данных Redis.

Класс `JedisMaker` считывает информацию о вашем сервере Redis из файла с именем `redis_url.txt`, который вы должны поместить в каталог src/resources:

- используйте текстовый редактор для создания и редактирования файла `ThinkDataStructures/code/src/resources/redis_url.txt`;
- вставьте URL вашего сервера; если применяете RedisToGo, то URL будет выглядеть следующим образом:

  redis://redistogo:1234567feedfacebeefa1e1234567@dory.redistogo.com:10534

Данный файл содержит пароль для вашего сервера Redis, поэтому не следует помещать его в общий репозиторий. Избежать такой ситуации можно, использовав файл репозитория с расширением `.gitignore`, который сделает сложным (но не невозможным) размещение этого файла в вашем репозитории.

Теперь активизируйте `ant build` для компиляции исходных файлов и `ant JedisMaker`, чтобы запустить пример кода в `main`:

```
public static void main(String[] args) {

    Jedis jedis + make();

    // String
    jedis.set("mykey", "myvalue");
    String value + jedis.get("mykey");
    System.out.println("Got value: " + value);

    // Set
    jedis.sadd("myset", "element1", "element2", "elements");
```

```
        System.out.println("element2 is member: " +
                           jedis.sismember("myset", "element2"));

        // List
        jedis.rpush("mylist", "element1", "element2", "element3");
        System.out.println("element at index 1: " +
                           jedis.lindex("mylist", 1));

        // Hash
        jedis.hset("myhash", "word1", Integer.toString(2));
        jedis.hincrBy("myhash", "word2", 1);
        System.out.println("frequency of word1: " +
                           jedis.hget("myhash", "word1"));
        System.out.println("frequency of word1: " +
                           jedis.hget("myhash", "word2"));
        jedis.close();
    }
```

В этом примере демонстрируются типы данных и методы, которые вы, скорее всего, будете использовать для текущего упражнения. При его запуске вывод должен быть таким:

```
Got value: myvalue
element2 is member: true
element at index 1: element2
frequency of word1: 2
frequency of word2: 1
```

В следующем разделе я расскажу, как работает этот код.

Типы данных Redis

Redis в своей основе — карта, сопоставляющая ключи, являющиеся строками, со значениями, которые могут иметь один из нескольких типов данных. Самый простой тип данных

Redis — это *строка*. Я буду выделять типы Redis курсивом, чтобы отличать их от типов Java.

Добавить *строку* в базу данных можно с помощью `jedis.set`, он похож на `Map.put`. Параметрами являются новый ключ и соответствующее значение. Чтобы найти ключ и получить его значение, задействуйте `jedis.get`:

```
jedis.set("mykey", "myvalue");
String value + jedis.get("mykey");
```

В этом примере ключ — `"mykey"`, а значение — `"myvalue"`.

Redis предоставляет структуру *set*, похожую на `Set<String>` в Java. Чтобы добавить элементы в *set*, выберите ключ для идентификации *set*, а затем используйте `jedis.sadd`:

```
jedis.sadd("myset", "elementt", "element2", "elements");
boolean flag + jedis.sismember("myset", "element2");
```

Для введения *set* не придется делать отдельный шаг. При отсутствии этой структуры Redis ее создает. В данном примере Redis создает *set* с именем `myset`, который содержит три элемента.

Метод `jedis.sismember` проверяет, находится ли элемент в наборе. Добавление элементов и проверка членства — это операции постоянного времени.

Redis также предоставляет структуру *list*, которая похожа на `List<String>` в Java. Метод `jedis.rpush` добавляет элементы в конец (правую часть) *list*:

```
jedis.rpush("mylist", "elementt", "element2", "element3");
String element + jedis.lindex("mylist", 1);
```

Вам не нужно создавать структуру, прежде чем начнете добавлять элементы. В этом примере создается *list* под названием `"mylist"`, который содержит три элемента.

Метод `jedis.lindex` принимает целочисленный индекс и возвращает указанный элемент из *list*. Добавление и доступ к элементам — это операции постоянного времени.

Наконец, Redis предоставляет структуру *hash*, которая похожа на Java `Map <String, String>`. Метод `jedis.hset` добавляет новую запись в *hash*:

```
jedis.hset("myhash", "wordt", Integer.toString(2));
String value + jedis.hget("myhash", "wordt");
```

В этом примере создается *hash* с именем `myhash`, которая содержит одну запись, сопоставляющую ключ `wordt` со значением `"2"`.

Ключи и значения — строки *(strings)*, поэтому если мы хотим сохранить значение типа `Integer`, то должны преобразовать его в `String`, прежде чем вызвать `hset`. И когда мы ищем значение с помощью `hget`, результатом является `String`, поэтому, вероятно, придется преобразовать полученное значение обратно в `Integer`.

Работа с хешами в Redis может сбивать с толку, поскольку мы используем ключ, чтобы определить требуемую *hash*, а затем еще один ключ для определения значения в этой структуре. В контексте Redis второй ключ называется *полем*, что может помочь поставить все на свои места. Таким образом, «ключ», подобный `myhash`, идентифицирует конкретную *hash*, а затем «поле», такое как `wordt`, определяет значение в *hash*.

Для многих приложений значения в *hash* являются целыми числами, так что эта база данных предоставляет несколько

специальных методов, например `hincrBy`, которые обрабатывают значения как целые числа:

```
jedis.hincrBy("myhash", "word2", t);
```

Этот метод обращается к `myhash`, получает текущее значение, связанное с `word2` (или `0`, если оно еще не существует), увеличивает его на 1 и записывает результат обратно в *hash*.

Методы `set`, `get` и инкремент записей в *hash* выполняются за постоянное время.

Узнать больше о типах данных Redis можно на сайте http://thinkdast.com/redistypes.

Упражнение 11

На данном этапе у вас есть информация, необходимая для создания индекса веб-поиска с хранением результатов в базе данных Redis.

Теперь запустите `ant JedisIndexTest`. Тест не будет выполнен успешно, поскольку вам нужно поработать!

`JedisIndexTest` проверяет следующие методы:

- `JedisIndex` — конструктор, принимающий объект класса `Jedis` в качестве параметра;
- `indexPage`, добавляющий веб-страницу в индекс; он принимает URL строки и объект класса `Elements` из jsoup, содержащий элементы страницы, которые должны быть проиндексированы;
- `getCounts`, который принимает поисковый термин и возвращает `Map<String, Integer>`, сопоставляющий URL, содержащий поисковый запрос, с количеством его появлений на этой странице.

Пример использования указанных методов:

```
WikiFetcher wf = new WikiFetcher();
String url1 =
"http://en.wikipedia.org/wiki/Java_(programming_language)";
Elements paragraphs = wf.readWikipedia(url1);

Jedis jedis = JedisMaker.make();
JedisIndex index = new JedisIndex(jedis);
index.indexPage(url1, paragraphs);
Map<String, Integer> map = index.getCounts("the");
```

Если мы ищем `url1`, то в результате `map` мы должны получить 339 — это количество появлений слова `the` на странице Java в «Википедии» (то есть сохраненной нами версии).

При повторном индексировании той же страницы новый результат должен заменить старое значение.

Один совет по переводу структур данных Java в Redis: помните, что каждый объект в базе Redis идентифицируется уникальным ключом, который является *строкой*. При наличии двух типов объектов в одной базе можно добавить префикс к ключам, чтобы различать их. Например, в нашем решении есть два типа объектов.

- Мы определяем `URLSet` как set в Redis, включающий URL, которые соответствуют данному поисковому термину. Ключ для каждого `URLSet` начинается с `"URLSet:"`, поэтому, чтобы получить URL, содержащие слово the, мы получаем доступ к набору с помощью ключа `"URLSet: the"`.

- Мы определяем `TermCounter` как *hash* в Redis, который сопоставляет каждый термин, встречающийся на странице, с количеством его появлений. Ключ к каждому `TermCounter` начинается с `"TermCounter:"` и заканчивается URL страницы, которую мы ищем.

В моей реализации есть один `URLSet` для каждого термина и один `TermCounter` для каждой проиндексированной страницы. Я предоставляю два вспомогательных метода для сборки этих ключей: `urlSetKey` и `termCounterKey`.

Дополнительные рекомендации

На данном этапе вы располагаете всей информацией, необходимой для выполнения упражнения, так что можете начать работу, если будете готовы. Но у меня есть несколько рекомендаций, с которыми вы, вероятно, предварительно захотите ознакомиться.

- Для этого упражнения я даю меньше рекомендаций, чем для предыдущих. Вы должны будете принять отдельные проектные решения; в частности, потребуется выяснить, как разделить задачу на части, которые можно тестировать по одной, а затем собрать их в полное решение. Попытка написать все сразу, не тестируя небольшие фрагменты, приведет к тому, что для отладки может потребоваться очень много времени.

- Одна из проблем работы с постоянными данными заключается в том, что они не изменяются. Структуры, хранящиеся в базе данных, могут меняться каждый раз при запуске программы. Если вы что-то внесли в базу, то придется исправить это или начать заново, прежде чем сможете продолжить. Чтобы помочь вам держать все под контролем, я предоставил методы, называемые `deleteURLSets`, `deleteTermCounters` и `deleteAllKeys`, служащие для очистки базы данных и начала новой работы. Вы также можете использовать `printIndex` для печати содержимого индекса.

- При каждом вызове метода `Jedis` ваш клиент отправляет сообщение на сервер, затем тот выполняет запрошенное действие и отправляет сообщение обратно. При совершении множества небольших операций это, вероятно, займет много времени. Можно улучшить производительность, объединив ряд операций в транзакцию.

Пример простой версии `deleteAllKeys` представлен ниже:

```
public void deleteAllKeys() {
    Set<String> keys + jedis.keys("*");
    for (String key: keys) {
        jedis.del(key);
    }
}
```

Каждый раз при вызове `del` требуется связь от клиента к серверу и обратно. Если индекс содержит более нескольких страниц, то на выполнение этого метода потребуется много времени. Можно ускорить его с помощью объекта `Transaction`:

```
public void deleteAHKeys() {
    Set<String> keys + jedis.keys("*");
    Transaction t + jedis.multi();
    for (String key: keys) {
        t.del(key);
    }
    t.exec();
}
```

Элемент `jedis.multi` возвращает объект `Transaction`, который предоставляет все методы объекта `Jedis`. Но при вызове метода в транзакции последний не запускает операцию немедленно и не связывается с сервером. Пакет операций сохраняется до тех пор, пока не будет вызван `exec`. Затем `Transaction` отправляет все сохраненные операции на сервер одновременно, что обычно занимает немного времени.

Несколько советов по проектированию

Теперь у вас *на самом деле* есть вся необходимая информация; вам следует начать работать над упражнением. Но если у вас не получается или вы действительно не знаете, с чего начать, то можете прочитать еще несколько советов.

Не читайте следующий текст, пока не запустите тестовый код, не попробуете некоторые базовые команды Redis и не напишете несколько методов в Jedi.sIndex.java.

Если вам и в самом деле сложно, то ниже приведены отдельные методы, над которыми вы, возможно, захотите поработать:

```
/**
*
 * Добавляет URL в набор, связанный с термином.
 */
public void add(String term, TermCounter tc) {}

/**
 * Ищет поисковый термин и возвращает набор URL.
 */
public Set<String> getURLs(String term) {}

/**
 * Возвращает количество появлений заданного термина
   по данному URL.
 */
public Integer getCount(String url, String term) {}

/**
 * Выталкивает содержимое TermCounter в Redis.
 */
public List<Object> pushTermCounterToRedis(TermCounter tc) {}
```

Я применил указанные методы в своем решении, но они, конечно, не являются единственным способом разделения на подзадачи. Поэтому, пожалуйста, воспользуйтесь моими советами, если уверены, что они вам помогут; в противном же случае проигнорируйте их.

Удачи!

15 Сбор данных в «Википедии»

В этой главе я представляю решение предыдущего упражнения и анализирую эффективность алгоритмов индексирования в Интернете. Затем мы создадим простого поискового робота.

Индексатор на основе Redis

В моем решении два вида структур хранятся в Redis:

- для каждого поискового термина есть `URLSet`, который представляет собой *set* с URL базы Redis, содержащими этот термин;
- для каждого URL предусмотрен `TermCounter`, являющийся *hash* в Redis, сопоставляющий каждый поисковый термин с количеством его появлений.

Мы обсудили эти типы данных в предыдущей главе. Кроме того, прочитать о структурах Redis можно по адресу http://thinkdast.com/redistypes.

В `JedisIndex` я предоставляю метод, который принимает поисковый термин и возвращает ключ Redis его `URLSet`:

```
private String urlSetKey(String term) {
    return "URLSet:" + term;
}
```

И метод, который принимает URL и возвращает ключ Redis его `TermCounter`:

```
private String termCounterKey(String url) {
    return "TermCounter:" + url;
}
```

Ниже представлена реализация `indexPage`, принимающая URL и объект типа `Elements` из jsoup, содержащий дерево DOM абзацев, которые мы хотим индексировать:

```
public void indexPage(String url, Elements paragraphs) {
    System.out.println("Indexing " + url);

    // создайте TermCounter и подсчитайте термины в абзацах
    TermCounter tc = new TermCounter(url);
    tc.processElements(paragraphs);

    // вытесните содержимое TermCounter в Redis
    pushTermCounterToRedis(tc);
}
```

Чтобы проиндексировать страницу, необходимо выполнить два действия.

1. Создать Java-объект `TermCounter` для содержимого страницы, используя код из предыдущего упражнения.

2. Вытолкнуть содержимое `TermCounter` в Redis.

Новый код, который выталкивает TermCounter в Redis, выглядит следующим образом:

```
public List<Object> pushTermCounterToRedis(TermCounter tc) {
    Transaction t + jedis.multi();

    String url + tc.getLabel();
    String hashname + termCounterKey(url);

    // если эта страница уже была проиндексирована,
    // удаляется старый hash
    t.del(hashname);

    // для каждого термина добавляется запись в TermCounter
    // и новый член в индекс
    for (String term: tc.keySet()) {
        Integer count + tc.get(term);
        t.hset(hashname, term, count.toString());
        t.sadd(urlSetKey(term), url);
    }
    List<Object> res + t.exec();
    return res;
}
```

Этот метод использует Transaction для сбора операций и одновременной отправки их на сервер, что намного быстрее, чем отправка серий небольших операций.

Он перебирает термины в TermCounter. Для каждого из них он выполняет следующие действия.

1. Находит или создает TermCounter в Redis, затем добавляет поле для нового термина.

2. Находит или создает URLSet в Redis, а затем добавляет текущий URL.

Если страница уже была проиндексирована, то перед добавлением нового содержимого нужно удалить ее старый экземпляр `TermCounter`.

Это касается индексирования новых страниц.

Во второй части упражнения я предложил написать метод `getCounts`, который принимает поисковый запрос и возвращает карту, сопоставляющую каждый URL, где появляется термин, с количеством появлений этого термина. Мое решение выглядит так:

```
public Map<String, Integer> getCounts(String term) {
    Map<String, Integer> map + new HashMap<String, Integer>();
    Set<String> urls + getURLs(term);
    for (String url: urls) {
    Integer count + getCount(url, term);
        map.put(url, count);
    }
    return map;
}
```

Этот метод использует два вспомогательных метода:

- `getURLs` принимает поисковый термин и возвращает набор URL, в котором отображается этот термин;
- `getCount` принимает URL и термин и возвращает количество появлений этого термина по данному адресу.

Ниже представлена реализация:

```
public Set<String> getURLs(String term) {
    Set<String> set + jedis.smembers(urlSetKey(term));
    return set;
}

public Integer getCount(String url, String term) {
```

```
    String redisKey + termCounterKey(url);
    String count + jedis.hget(redisKey, term);
    return new Integer(count);
}
```

Благодаря тому что мы спроектировали индекс, эти методы просты и эффективны.

Анализ поиска

Предположим, мы проиндексировали N страниц и обнаружили M уникальных поисковых терминов. Сколько времени потребуется, чтобы найти один из них? Подумайте над ответом, прежде чем продолжить.

Чтобы найти поисковый термин, мы запустим метод `getCounts`, который совершает следующие действия.

1. Создает карту.
2. Выполняет `getURLs` с целью получить набор `Set` из URL.
3. Для каждого URL в `Set` выполняет метод `getCount` и добавляет запись в `HashMap`.

Выполнение `getURLs` занимает время, пропорциональное количеству URL, которые содержат поисковый запрос. Для редких терминов оно может быть небольшим, но для общеупотребительных терминов способно превысить N.

Внутри цикла мы запускаем `getCount`, который находит экземпляр `TermCounter` в Redis, просматривает термин и добавляет запись в `HashMap`. Все эти операции являются операциями постоянного времени, поэтому общая сложность `getCounts` — $O(N)$ в худшем случае. Однако на практике время выполнения про-

порционально количеству страниц, содержащих этот термин, что обычно намного меньше *N*.

Данный алгоритм настолько эффективный, насколько это возможно с точки зрения алгоритмической сложности, но очень медленный, так как отправляет много небольших операций в Redis. Его можно ускорить, используя `Transaction`. Вероятно, вы захотите выполнить это в качестве упражнения либо можете увидеть мое решение в `RedisIndex.java`.

Анализ индексирования

Сколько времени потребуется, чтобы индексировать страницу при использовании структур данных, которые мы разработали? Подумайте над своим ответом, прежде чем продолжить.

Чтобы индексировать страницу, мы проходим ее дерево DOM, находим все объекты `TextNode` и разбиваем строки в соответствии с поисковыми терминами. Все это занимает время, пропорциональное количеству слов на странице.

Для каждого термина мы увеличиваем счетчик в `HashMap`, который является операцией постоянного времени. Поэтому создание `TermCounter` занимает время, пропорциональное количеству слов на странице.

Выталкивание `TermCounter` в Redis требует удаления этого класса; такое действие выполняется за линейное время, пропорциональное количеству уникальных поисковых терминов. Тогда для каждого термина требуется две операции.

1. Добавить элемент в `URLSet`.

2. Добавить элемент в `TermCounter` в Redis.

Обе они являются операциями постоянного времени, поэтому общее время выталкивания `TermCounter` линейно и пропорционально числу уникальных поисковых терминов.

Таким образом, создание `TermCounter` пропорционально количеству слов на странице. Выталкивание `TermCounter` в Redis пропорционально количеству уникальных поисковых терминов.

Поскольку количество слов на странице обычно превышает количество уникальных поисковых терминов, то общая сложность пропорциональна количеству слов на странице. Теоретически страница может содержать все поисковые термины в индексе, поэтому наихудший показатель производительности — $O(M)$, но на практике такое встречается редко.

Данный анализ предлагает способ повысить производительность: вероятно, следует избегать индексации очень распространенных слов. Прежде всего это требует много времени и пространства, поскольку они появляются почти во всех `URLSet` и `TermCounter`. Кроме того, такие слова не очень полезны, так как не помогают идентифицировать соответствующие страницы.

Большинство поисковых систем избегают индексации общеупотребительных слов, которые в данном контексте известны как стоп-слова (см. http://thinkdast.com/stopword).

Обход графа

Если вы выполнили упражнение 5 в главе 7, то у вас уже есть программа, которая читает страницу «Википедии», находит первую ссылку, использует ее для загрузки следующей страницы и повторяет предыдущие действия. Эта программа —

специализированный вид поискового робота, но когда люди говорят «веб-поисковик», они обычно имеют в виду программу, которая:

- загружает стартовую страницу и индексирует содержимое;
- находит все ссылки на странице и добавляет связанные URL в коллекцию;
- проводит их через коллекцию, загружая страницы, индексируя их и добавляя новые URL;
- при нахождении URL, который уже был проиндексирован, пропускает его.

Вы можете думать о сети как о графе, где каждая страница является узлом и каждая ссылка — направленным ребром от одного узла к другому. Прочитать о графах можно на сайте http://thinkdast.com/graph.

Начиная с исходного узла, поисковый робот обходит этот граф, посещая каждый доступный узел один раз.

Коллекция, которую мы используем для хранения URL, определяет, какой вид обхода выполняет поисковый робот.

- Если это очередь «первым вошел, первым вышел» (first in, first out, FIFO), то поисковый робот выполняет поиск в ширину.
- Если это стек «последним вошел, первым вышел» (last in, first out, LIFO), то поисковый робот выполняет поиск в глубину.
- В более общем случае элементы в коллекции могут иметь приоритет. Например, можно было бы предоставить более

высокий приоритет страницам, которые не индексировались в течение длительного времени.

Узнать больше об обходе графов можно на сайте http://thinkdast.com/graphtrav.

Упражнение 12

Теперь пришло время написать поисковый робот. В репозитории для этой книги вы найдете исходные файлы для данного упражнения:

- `WikiCrawler.java`, который содержит начальный код для вашего поискового робота;
- `WikiCrawlerTest.java`, включающий код для `WikiCrawler`;
- `JedisIndex.java` — мое решение для предыдущего примера.

Кроме того, понадобятся некоторые вспомогательные классы, которые мы использовали в предыдущих упражнениях:

- `JedisMaker.java`;
- `WikiFetcher.java`;
- `TermCounter.java`;
- `WikiNodeIterable.java`.

Перед запуском `JedisMaker` вам необходимо предоставить файл с информацией о вашем сервере Redis. Если вы сделали это в предыдущем упражнении, то все должно быть настроено. В противном случае можете найти инструкции в разделе «Создание индекса на основе Redis» главы 14.

Активизируйте `ant build` для компиляции исходных файлов, затем запустите `ant JedisMaker` с целью убедиться, что он настроен для подключения к вашему серверу Redis.

Теперь запустите `ant WikiCrawlerTest`. Он не будет выполнен успешно, поскольку вам нужно поработать!

Начало предоставленного мной класса `WikiCrawler` выглядит так:

```
public class WikiCrawler {

    public final String source;
    private JedisIndex index;
    private Queue<String> queue + new LinkedList<String>();
    final static WikiFetcher wf + new WikiFetcher();

    public WikiCrawler(String source, JedisIndex index) {
        this.source + source;
        this.index + index;
        queue.offer(source);
    }

    public int queueSize() {
        return queue.size();
    }
```

У нас есть следующие переменные экземпляра:

- `source` — URL, где начинается поиск;
- `index` — индекс JedisIndex, в который должны поступать результаты;
- `queue` — `LinkedList` для отслеживания URL, обнаруженных, но еще не проиндексированных;
- `wf` — `WikiFetcher`, который послужит для чтения и анализа веб-страниц.

Вам нужно дополнить метод `crawl`. Ниже приведен его прототип:

```
public String crawl(boolean testing) throws IOException {}
```

Параметр `testing` будет иметь значение `true` при вызове метода из `WikiCrawlerTest` и `false` в противном случае.

Когда `testing` равно `true`, метод `crawl` должен:

- выбрать и удалить URL из очереди в порядке FIFO;
- прочитать содержимое страницы с помощью `WikiFetcher.readWikipedia`, который читает кэшированные копии страниц, включенных в этот репозиторий для целей тестирования (чтобы избежать проблем при изменении версии «Википедии»);
- проиндексировать страницы независимо от того, были ли они уже индексированы;
- найти все внутренние ссылки на странице и добавить их в очередь в порядке их появления. Таковыми являются ссылки на другие страницы «Википедии»;
- вернуть URL проиндексированной страницы.

Когда `testing` равно `false`, этот метод должен предпринять следующие шаги:

- выбрать и удалить URL из очереди в порядке FIFO;
- если URL уже проиндексирован, то ему не следует производить повторную индексацию; он должен вернуть значение `null`;

- в противном случае он должен прочитать содержимое страницы с помощью `WikiFetcher.fetchWikipedia`, которая считывает текущий контент из Интернета;
- затем он должен индексировать страницу, добавить ссылки в очередь и вернуть URL проиндексированной страницы.

Тест `WikiCrawlerTest` загружает очередь примерно с 200 ссылками, а затем трижды вызывает метод `crawl`. После каждого вызова он проверяет возвращаемое значение и новую длину очереди.

Когда ваш поисковый робот начнет работать так, как было указано, тест будет успешно выполняться. Удачи!

16 Логический поиск

В этой главе я представлю решение предыдущего упражнения. Затем вы напишете код для объединения нескольких результатов поиска и сортировки по их релевантности поисковым запросам.

Решение для поискового робота

Для начала рассмотрим наше решение для предыдущего упражнения. Я представил схему `WikiCrawler`; ваша задача заключалась в заполнении метода `crawl`. В качестве напоминания ниже приведены поля класса `WikiCrawler`:

```
public class WikiCrawler {
    // запоминает, откуда мы начали
    private final String source;

    // индекс, в который поступают результаты
```

```
    private JedisIndex index;

    // очередь URL, которые должны быть проиндексированы
    private Queue<String> queue + new LinkedList<String>();

    // сборщик, используемый для получения страниц
    // из «Википедии»
    final static WikiFetcher wf + new WikiFetcher();
}
```

При создании класса `WikiCrawler` мы предоставляем элементы `source` и `index`. Первоначально очередь содержит только один из них — `source`.

Обратите внимание: реализация очереди — `LinkedList`, так что можно добавлять элементы в конец (и удалять их из начала) за постоянное время. Назначив объект `LinkedList` переменной `Queue`, мы ограничили себя использованием методов интерфейса `Queue`; в частности, будем применять `offer` для добавления элементов и `poll` — для их удаления.

Моя реализация `WikiCrawler.crawl` выглядит следующим образом:

```
public String crawl(boolean testing) throws IOException {
    if (queue.isEmpty()) {
        return null;
    }
    String url + queue.poll();
    System.out.println("Crawling " + url);

    if (testing==false && index.isIndexed(url)) {
        System.out.println("Already indexed.");
        return null;
    }

    Elements paragraphs;
```

```
    if (testing) {
        paragraphs = wf.readWikipedia(url);
    } else {
        paragraphs = wf.fetchWikipedia(url);
    }
    index.indexPage(url, paragraphs);
    queueInternalLinks(paragraphs);
    return url;
}
```

Самая большая сложность этого метода заключается в том, чтобы сделать его более удобным для тестирования. Его логика представлена ниже.

1. Если очередь пуста, то он возвращает значение `null` с целью показать, что страница не была проиндексирована.

2. В противном случае он удаляет и сохраняет следующий URL из очереди.

3. При уже проиндексированном URL метод `crawl` не индексирует его снова, если только не находится в режиме тестирования.

4. Затем он считывает содержимое страницы: в режиме тестирования считывание производится из файла, в противном случае — из Интернета.

5. Индексирует страницу.

6. Анализирует страницу и добавляет внутренние ссылки в очередь.

7. Наконец, возвращает URL страницы, которую он проиндексировал.

Я представил реализацию `Index.indexPage` в разделе «Индексатор на основе Redis» главы 15. Таким образом, единственным новым методом является `WikiCrawler.queueInternalLinks`.

Я написал две версии этого метода с разными параметрами: одна принимает объект типа `Elements`, в котором содержится одно дерево DOM для каждого абзаца, другая — объект `Element`, включающий только один абзац.

Первая версия просто проходит абзацы в цикле. Вторая выполняет реальную работу:

```
void queueInternalLinks(Elements paragraphs) {
    for (Element paragraph: paragraphs) {
        queueInternalLinks(paragraph);
    }
}

private void queueInternalLinks(Element paragraph) {
    Elements elts + paragraph.select("a[href]");
    for (Element elt: elts) {
        String relURL + elt.attr("href");

        if (relURL.startsWith("/wiki/")) {
            String absURL + elt.attr("abs:href");
            queue.offer(absURL);
        }
    }
}
```

Чтобы определить, является ли ссылка «внутренней», мы проверяем, начинается ли URL с /wiki/. В результате могут быть включены отдельные страницы, которые мы не хотим индексировать, например метастраницы о «Википедии». Кроме того, могут подвергнуться исключению некоторые нужные страницы,

такие как ссылки на неанглоязычные страницы. Однако этот простой тест вполне подойдет для начала работы.

Вот и все. Данное упражнение не содержит много нового материала; это в основном возможность собрать все фрагменты воедино.

Поиск информации

Следующий этап проекта — внедрение инструмента поиска. Понадобятся следующие части.

1. Интерфейс, в котором пользователи могут вводить поисковые термины и просматривать результаты.
2. Механизм поиска, принимающий каждый поисковый термин и возвращающий страницы, которые его содержат.
3. Механизмы объединения результатов поиска нескольких поисковых терминов.
4. Алгоритмы для ранжирования и сортировки результатов поиска.

Общий термин, определяющий такие процессы, — это *поиск информации*; узнать о нем больше можно на сайте http://thinkdast.com/infret.

В этом упражнении мы остановимся на пунктах 3 и 4. Мы уже создали простую версию для пункта 2. Если вы заинтересованы в создании веб-приложений, то можете подумать о работе над пунктом 1.

Логический поиск

Большинство поисковых систем способны выполнять *логический поиск* (boolean search); это значит, что можно комбинировать результаты для нескольких поисковых терминов с помощью булевой логики. Например, поиск слов:

- java AND programming может возвращать только страницы, содержащие оба условия поиска: java и programming;
- java OR programming способен возвращать страницы, включающие любой термин, но не обязательно оба;
- java – indonesia может возвращать страницы со словом java и без слова indonesia.

Подобные выражения, содержащие поисковые термины и логические операторы, называются запросами (queries).

Примененные к результатам поиска логические операторы `AND`, `OR` и `-` соответствуют следующим операциям над множествами: пересечению, объединению и разности. Например, предположим, что:

- `s1` — набор страниц, содержащих слово `java`;
- `s2` — набор страниц, включающих слово `programming`;
- `s3` — набор страниц со словом `indonesia`.

В таком случае:

- пересечение `s1` и `s2` — набор страниц, содержащих слова `java` AND (И) `programming`;

- объединение `s1` и `s2` — набор страниц, включающих слова `java` OR (ИЛИ) `programming`;
- разность `s1` и `s2` — набор страниц со словом `java` и без слова `indonesia`.

В следующем разделе вы напишете метод, реализующий эти операции.

Упражнение 13

В репозитории для этой книги вы найдете исходные файлы:

- `WikiSearch.java`, который определяет объект, содержащий результаты поиска и выполняющий операции над ними;
- `WikiSearchTest.java`, включающий тестовый код для `WikiSearch`;
- `Card.java`, демонстрирующий применение метода сортировки в `java.util.Collections`.

Кроме того, вы найдете ряд вспомогательных классов, которые мы использовали в предыдущих упражнениях.

Начало определения класса `WikiSearch` выглядит так:

```
public class WikiSearch {

    // карта, сопоставляющая URL-адреса, содержащие термин(-ы)
    // и их коэффициенты релевантности
    private Map<String, Integer> map;

    public WikiSearch(Map<String, Integer> map) {
```

```
        this.map + map;
    }

    public Integer getRelevance(String url) {
        Integer relevance + map.get(url);
        return relevance==null ? 0: relevance;
    }
}
```

Объект `WikiSearch` содержит карту, сопоставляющую URL с их коэффициентом релевантности. В контексте поиска информации *коэффициент релевантности* — это число, показывающее, насколько страница удовлетворяет потребностям пользователя в соответствии с запросом. Существует множество способов получить данный коэффициент, но большинство из них основано на *частоте термина*, представляющей собой количество появлений поискового термина на странице. Чаще всего применяется коэффициент релевантности под названием TF-IDF, которое является сокращением от term frequency-inverse document frequency («частота термина — обратная частота документа»). Узнать об этом больше можно по адресу http://thinkdast.com/tfidf.

Вы получите возможность реализовать TF-IDF позже, но начнем мы с чего-нибудь более простого: с частоты термина (TF).

- Если запрос содержит один поисковый термин, то релевантность страницы — это частота ее термина; то есть количество появлений термина на странице.

- Для запросов с несколькими терминами релевантность страницы представляет собой сумму частот терминов; то есть общее количество появлений какого-либо из поисковых терминов.

Теперь вы готовы начать упражнение. Активизируйте `ant build` для компиляции исходных файлов, затем запустите `ant WikiSearchTest`. Как обычно, он не будет выполнен успешно, поскольку вам нужно поработать.

В `WikiSearch.java` заполните тела методов `and`, `or` и `minus`, чтобы соответствующие тесты проходили. Вам пока не нужно беспокоиться о `testSort`.

Вы можете запустить `WikiSearchTest` без использования `Jedis`, так как он не зависит от индекса в вашей базе данных Redis. Но для выполнения запроса в сравнении с индексом нужно предоставить файл с информацией о вашем сервере Redis. Подробнее см. в разделе «Создание индекса на основе Redis» главы 14.

Активизируйте `ant JedisMaker` с целью убедиться, что он настроен для подключения к вашему серверу Redis. Затем запустите `WikiSearch`, который выводит результаты трех запросов:

- java;
- programming;
- java AND programming.

Сначала результаты не будут располагаться в определенном порядке, так как `WikiSearch.sort` неполон.

Заполните тело метода `sort`, чтобы результаты были возвращены в порядке возрастания. Я предлагаю использовать метод сортировки, предоставляемый `java.util.Collections`, сортирующий списки любого типа. Документацию можно прочитать по адресу http://thinkdast.com/collections.

Ниже представлены две версии sort.

- Однопараметрическая принимает список и сортирует элементы, используя метод compareTo, поэтому элементы должны реализовывать интерфейс Comparable.
- Двупараметрическая принимает список объектов любого типа и Comparator, который является объектом, предоставляющим метод сравнения элементов.

Описание интерфейсов Comparable и Comparator я представлю в следующем разделе.

Интерфейсы Comparable и Comparator

В репозиторий этой книги включен файл Card.java, в котором демонстрируются два способа сортировки списка объектов типа Card. Ниже приведено начало определения класса:

```
public class Card implements Comparable<Card> {

    private final int rank;
    private final int suit;

    public Card(int rank, int suit) {
        this.rank + rank;
        this.suit + suit;
    }
```

В объект типа Card включены два поля целочисленного типа: rank и suit. Объект Card реализует Comparable<Card>; это значит, что он предоставляет метод compareTo:

```
    public int compareTo(Card that) {
        if (this.suit < that.suit) {
```

```
            return -1;
        }
        if (this.suit > that.suit) {
            return 1;
        }
        if (this.rank < that.rank) {
            return -1;
        }
        if (this.rank > that.rank) {
            return 1;
        }
        return 0;
    }
```

Спецификация метода `compareTo` указывает: он должен возвращать отрицательное число при условии, что `this` считается меньше `that`, положительное — если `this` считается большим, и `0`, когда они считаются равными.

Однопараметрическая версия `Collections.sort` использует метод `compareTo`, предоставляемый элементами, для их сортировки. Чтобы продемонстрировать данное утверждение, можно составить список из 52 карт, аналогичный этому:

```
    public static List<Card> makeDeck() {
        List<Card> cards + new ArrayList<Card>();
        for (int suit + 0; suit <= 3; suit++) {
            for (int rank + 1; rank <= 13; rank++) {
                Card card + new Card(rank, suit);
                cards.add(card);
            }
        }
        return cards;
    }
```

и сортировать их таким способом:

```
        Collections.sort(cards);
```

Эта версия sort ставит элементы в так называемом естественном порядке, поскольку он определяется самими объектами.

Но можно применить другой порядок, предоставив объект Comparator. Например, естественный порядок объектов Card рассматривает Aces (тузы) как самый низкий ранг, но в ряде карточных игр они имеют самый высокий ранг. Можно определить Comparator, который считает Aces high (тузы высокими), как этот:

```
Comparator<Card> comparator + new Comparator<Card>() {
    @Override
    public int compare(Card card1, Card card2) {
        if (card1.getSuit() < card2.getSuit()) {
            return -1;
        }
        if (card1.getSuit() > card2.getSuit()) {
            return 1;
        }
        int rank1 + getRankAceHigh(card1);
        int rank2 + getRankAceHigh(card2);

        if (rank1 < rank2) {
            return -1;
        }
        if (rank1 > rank2) {
            return 1;
        }
        return 0;
    }

    private int getRankAceHigh(Card card) {
        int rank + card.getRank();
        if (rank += 1) {
            return 14;
        } else {
            return rank;
        }
    }
};
```

Этот код определяет анонимный класс, который реализует `compare` в соответствии с требованиями. Затем создает экземпляр вновь определенного безымянного класса. Прочитать об анонимных классах в Java можно по адресу http://thinkdast.com/anon class.

Используя данный `Comparator`, мы можем вызывать метод `sort` следующим образом:

```
Collections.sort(cards, comparator);
```

При таком упорядочении туз пик считается самой высокой картой в колоде, двойка треф — самой низкой.

При желании поэкспериментировать с кодом из данного раздела вы найдете его в `Card.java`. В качестве упражнения можете написать компаратор, который сначала сортирует по `rank`, а затем по `suit`, поэтому все тузы должны быть вместе и все двойки и т. д.

Дополнения

Если базовая версия данного упражнения у вас заработала, то при желании можете выполнить следующие дополнительные задания.

- Прочитайте о TF-IDF по адресу http://thinkdast.com/tfidj и реализуйте его. Возможно, вам придется изменить `JavaIndex` для вычисления частот документа, то есть общего количества появлений каждого термина на всех страницах индекса.
- Для запросов с несколькими поисковыми терминами общая релевантность для каждой страницы теперь является суммой релевантностей для каждого термина. Подумайте, когда эта

простая версия может не сработать, и попробуйте некоторые альтернативные подходы.

- Создайте UI, позволяющий пользователям вводить запросы с булевыми операторами. Анализируйте запросы, генерируйте результаты, затем сортируйте их по релевантности и отобразите URL с наивысшим показателем. Рассмотрите создание фрагментов, которые показывают, в каком месте страницы появились поисковые запросы. При желании создать веб-приложение для своего UI подумайте о применении Heroku в качестве простого варианта для разработки и развертывания веб-приложений с помощью Java (см. http://thinkdast.com/heroku).

17 Сортировка

Преподавателям на факультетах информатики свойственна нездоровая одержимость алгоритмами сортировки. Исходя из того, сколько времени студенты, изучающие компьютерные науки, тратят на эту тему, можно подумать, что выбор алгоритмов сортировки — краеугольный камень современной разработки программного обеспечения. Конечно, на самом деле создатели ПО могут годами или даже на протяжении всей карьеры не задумываться о том, как функционирует сортировка. Почти для всех приложений они применяют универсальный алгоритм, предоставляемый языком или библиотеками, с которыми работают. И обычно этого достаточно.

Поэтому, если вы пропустите эту главу и ничего не узнаете об алгоритмах сортировки, вы все равно сможете стать отличным разработчиком. Но есть несколько причин, по которым вы, вероятно, все-таки захотите изучить представленный в ней материал.

1. Хотя существуют универсальные алгоритмы, хорошо работающие с подавляющим большинством приложений, есть два алгоритма специального назначения, которые иногда могут понадобиться: поразрядная сортировка (radix sort) и ограниченная пирамидальная сортировка (bounded heap sort).

2. Один алгоритм сортировки, сортировка слиянием (merge sort) — прекрасный обучающий пример, поскольку демонстрирует важную и полезную стратегию для разработки алгоритмов под названием *«разделяй — властвуй — соединяй»* (divide — conquer — glue). Кроме того, когда мы проанализируем его производительность, вы узнаете о порядке роста, которого не видели раньше, — *линейно-логарифмическом*. Наконец, часть наиболее широко используемых алгоритмов являются гибридами, включающими элементы сортировки слиянием.

3. Еще одна причина узнать об алгоритмах сортировки связана с тем, что технические интервьюеры любят о них спрашивать. Если хотите устроиться на работу, то факт, что вы сможете продемонстрировать культурную грамотность в области компьютерных наук, послужит вам на пользу.

Итак, в этой главе мы проанализируем сортировку вставкой (insertion sort), вы реализуете сортировку слиянием, я расскажу о поразрядной сортировке и вы напишете простую версию ограниченной пирамидальной сортировки.

Сортировка вставкой

Мы начнем с сортировки вставкой главным образом потому, что ее легко описать и реализовать. Это не очень эффективно, но, как мы увидим, у нее есть некоторые компенсирующие свойства.

Вместо того чтобы объяснять алгоритм, я предлагаю прочитать страницу в «Википедии», посвященную сортировке вставкой, которая включает псевдокоды и анимированные примеры, по адресу http://thinkdast.com/insertsort. Возвращайтесь, когда получите общее представление.

Ниже приведена реализация сортировки вставкой в Java:

```
public class ListSorter<T> {

    public void insertionSort(List<T> list, Comparator<T>
    comparator) {

        for (int i=1; i < list.size(); i++) {
            T elt_i + list.get(i);
            int j + i;
            while (j > 0) {
                T elt_j + list.get(j-1);
                if (comparator.compare(elt_i, elt_j) >= 0) {
                    break;
                }
                list.set(j, elt_j);
                j--;
            }
            list.set(j, elt_i);
        }
    }
}
```

Я определяю класс `ListSorter` в качестве контейнера для алгоритмов сортировки. Используя параметр типа `T`, можно писать методы, которые работают со списками, содержащими объекты любого типа.

Метод `insertionSort` принимает два параметра: список объектов любого типа и `Comparator`, который знает, как сравнивать объекты типа `T`. Он сортирует список *на месте*, это значит, что

он изменяет существующий список и не требует выделения какого-либо нового пространства.

В следующем примере показано, как вызвать этот метод со списком объектов типа `Integer`:

```
List<Integer> list = new ArrayList<Integer>(
    Arrays.asList(3, 5, 1, 4, 2));

Comparator<Integer> comparator = new
Comparator<Integer>() {
    @Override
    public int compare(Integer elt1, Integer elt2) {
        return elt1.compareTo(elt2);
    }
};
ListSorter<Integer> sorter = new ListSorter<Integer>();
sorter.insertionSort(list, comparator);
System.out.println(list);
```

Метод `insertionSort` имеет два вложенных цикла, вследствие чего можно предположить, что время его выполнения является квадратичным. В данном случае это верно, но прежде, чем прийти к такому выводу, нужно проверить, пропорционально ли количество раз работы каждого цикла размеру массива n.

Внешний цикл повторяется от `1` до `list.size0`, поэтому он является линейным по размеру списка n. Вложенный цикл выполняет перебор от `i` до `0` и, как следствие, тоже линеен по n. Таким образом, общее количество повторений вложенного цикла квадратично.

Доказательство вышесказанного изложено ниже:

- первый проход, $i = 1$, и вложенный цикл выполняются не более одного раза;
- второй проход, $i = 2$, и вложенный цикл выполняются не более двух раз;

- последний проход, $i = n - 1$, и вложенный цикл выполняется не более чем $n - 1$ раз.

Таким образом, общее количество выполнений внутреннего цикла представляет собой сумму рядов 1, 2... $n - 1$, равную $n(n - 1) / 2$. И главный член этого выражения (тот, у которого самый высокий показатель степени) равен n^2.

В худшем случае сортировка вставкой квадратична. Однако есть две особенности.

1. Если элементы уже отсортированы или почти одинаковы, то данная сортировка является линейной. В частности, при нахождении каждого элемента на расстоянии не более чем k позиций от того места, где он должен быть, вложенный цикл никогда не будет выполняться больше чем k раз, а общая продолжительность выполнения — $O(kn)$.

2. В связи с простотой реализации накладные расходы низкие; то есть, хотя время выполнения равно an^2, коэффициент для старшего члена a, вероятно, мал.

Поэтому если известно, что массив почти отсортирован или не очень большой, то сортировка вставки может быть хорошим выбором. Но для больших массивов можно сделать лучше. На самом деле намного лучше.

Упражнение 14

Сортировка слиянием — один из нескольких алгоритмов, время выполнения которых лучше квадратичного. И снова, вместо того чтобы объяснять алгоритм здесь, я предлагаю прочитать о нем в «Википедии» на сайте http://thinkdast.com/mergesort. Как

только вы получите общее представление, вернитесь и проверьте, хорошо ли все поняли, написав реализацию.

В репозитории для этой книги вы найдете исходные файлы для данного упражнения:

- `ListSorter.java`;
- `ListSorterTest.java`.

Запустите `ant build` для компиляции исходных файлов, затем `ant ListSortTest`. Как обычно, он не выполнится, поскольку вам нужно поработать.

В `ListSorter.java` я представил схемы двух методов: `mergeSortInPlace` и `mergeSort`:

```
public void mergeSortInPlace(List<T> list, Comparator<T>
comparator) {
    List<T> sorted + mergeSortHelper(list, comparator);
    list.clear();
    list.addAll(sorted);
}

private List<T> mergeSort(List<T> list, Comparator<T>
comparator) {
    // TODO: заполните!
    return null;
}
```

Оба метода выполняют одни и те же действия, но предоставляют разные интерфейсы. Метод `mergeSort` принимает список и возвращает новый с теми же элементами, отсортированными в порядке возрастания. Метод `mergeSortInPlace` — это метод, который возвращает значение `void` и изменяет существующий список.

Ваша задача — заполнить `mergeSort`. Прежде чем писать полностью рекурсивную версию слияния, начните с чего-то похожего на описанное ниже.

1. Разделите список пополам.
2. Сортируйте части списка с помощью `Collections.sort` или `insertionSort`.
3. Объедините отсортированные части в один полностью отсортированный список.

Это позволит отладить код слияния, не учитывая сложность рекурсивного метода.

Затем добавьте базовый случай (см. http://thinkdast.com/basecase). При наличии списка только с одним элементом можете сразу его вернуть, поскольку он уже вроде как отсортирован. Либо, если длина списка ниже некоторого порога, можете отсортировать его с помощью `Collections.sort` или `insertionSort`. Проверьте базовый случай, прежде чем продолжить.

Наконец, измените свое решение, чтобы оно выполняло два рекурсивных вызова для сортировки частей массива. Когда ваш метод начнет работать, `testMergeSort` и `testMergeSortInPlace` должны будут завершиться успешно.

Анализ сортировки слиянием

Чтобы определить время выполнения сортировки слиянием, нужно думать с точки зрения уровней рекурсии и того, как много работы делается на каждом уровне. Предположим, мы на-

чинаем со списка, содержащего *n* элементов. Алгоритм состоит из следующих шагов.

1. Создайте два новых массива и скопируйте по половине элементов в каждый из них.
2. Отсортируйте обе части.
3. Объедините части.

На рис. 17.1 показаны эти шаги.

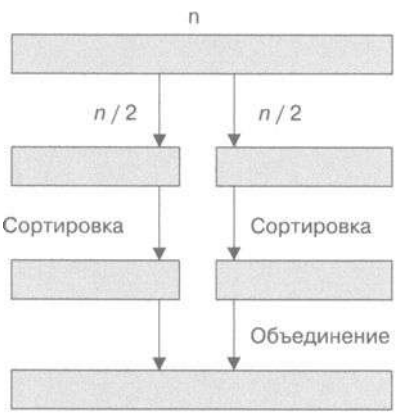


Рис. 17.1. Представление сортировки слиянием, показывающее один уровень рекурсии

На шаге 1 каждый из элементов копируется по одному разу, поэтому время выполнения линейно. То же происходит и на шаге 3, так что время выполнения также линейно. Теперь нужно выяснить сложность шага 2. Для этого следует рассмотреть другую картину вычислений, которая показывает уровни рекурсии, как на рис. 17.2.

Рис. 17.2. Представление сортировки слиянием, показывающее все уровни рекурсии

На верхнем уровне есть один список с n элементами. Предположим, что n является степенью 2. На следующем уровне — два списка с $n / 2$ элементами. Затем четыре списка с $n / 4$ элементами и т. д., пока не перейдем к n спискам с одним элементом.

На каждом уровне в общей сложности n элементов. По пути вниз следует разделить массивы пополам; это занимает время, пропорциональное n на каждом уровне. На обратном пути нужно объединить всего n элементов, что также выполняется за линейное время.

Если количество уровней равно h, то общий объем работы для алгоритма — $O(nh)$. Итак, сколько всего уровней? Есть два способа подумать об этом.

1. Сколько раз следует разделить n пополам, чтобы получить единицу?

2. Или сколько раз нужно удвоить единицу, прежде чем получить n?

Второй вопрос можно задать в другой форме: «Какой степенью 2 является n?»

$$2^h = n.$$

Возьмем логарифм по основанию 2 от обеих частей равенства:

$$h = \log_2 n.$$

Следовательно, общее время равно $O(n \log n)$. Я не стал писать основание логарифма, так как логарифмы с разными основаниями отличаются постоянным коэффициентом, в связи с чем все они имеют один порядок роста.

О логарифмах с временной сложностью $O(n \log n)$ иногда говорят, что они *линейно-логарифмические*, но большинство людей говорят просто: $n \log n$.

Оказывается, $O(n \log n)$ — теоретическая нижняя оценка алгоритмов сортировки, которые работают путем сравнения элементов друг с другом. Это значит, что не существует такой *сортировки сравнением*, порядок роста которой лучше $n \log n$. См. http://thinkdast.com/compsort.

Но, как мы увидим в следующем разделе, есть сортировки, которые не используют сравнение и выполняются за линейное время!

Поразрядная сортировка

Во время президентской кампании в США в 2008 году кандидату Бараку Обаме было предложено выполнить анализ импровизированного алгоритма, когда он посетил Google. Исполнительный директор Эрик Шмидт в шутку спросил его о «наиболее эффективном способе отсортировать миллион 32-разрядных целых чисел». Обама, очевидно, был предупрежден, поскольку быстро ответил: «Я думаю, что пузырьковая сортировка — это плохая идея». На сайте http://thinkdast.com/obama можно посмотреть видео.

Обама был прав: пузырьковая сортировка концептуально проста, но время ее выполнения квадратично; и даже среди квадратичных алгоритмов сортировки ее производительность не очень хороша. См. http://thinkdast.com/bubble.

Вероятно, ответ Шмидта — это «поразрядная сортировка», являющаяся алгоритмом сортировки, не связанным со сравнением,

который работает, если размер элементов ограничен, например 32-разрядное целое число или 20-символьная строка.

Чтобы увидеть, как это работает, представьте: у вас есть стопка картотечных карточек, каждая из которых содержит трехбуквенное слово. Карточки можно сортировать.

1. Пройдитесь один раз по карточкам и разделите их на наборы по первой букве. Таким образом, слова, начинающиеся с `a`, должны быть в одном наборе, за ними следуют слова, начинающиеся с `b`, и т. д.

2. Снова разделите каждый набор по второй букве. Слова, начинающиеся с `aa`, должны быть вместе, за ними следуют слова, начинающиеся с `ab`, и т. д. Конечно, не все наборы будут заполнены, но все в порядке.

3. Снова разделите каждый набор по третьей букве.

В этот момент каждый набор содержит один элемент, а наборы сортируются в порядке возрастания. На рис. 17.3 показан пример с трехбуквенными словами.

В верхнем ряду приведены несортированные слова. Во второй строке показано, как выглядят наборы после первого прохода. Слова в каждом наборе начинаются с одной и той же буквы.

После второго прохода слова в каждом наборе начинаются с тех же двух букв. После третьего прохода в каждом наборе может быть только одно слово, а наборы расположены по порядку.

Во время каждого прохода мы перебираем элементы и добавляем их в наборы. Пока последние позволяют добавлять за постоянное время, каждый проход линеен.

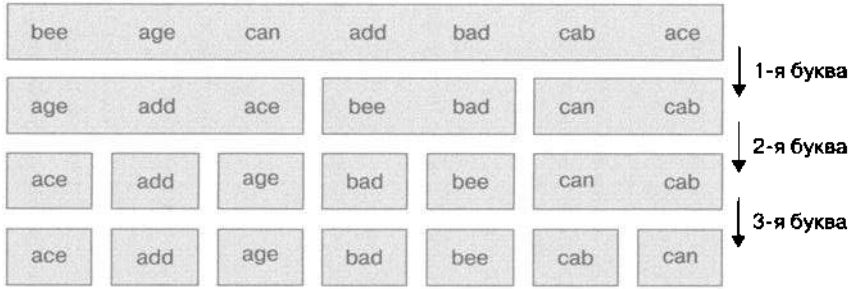


Рис. 17.3. Пример поразрядной сортировки трехбуквенных слов

Количество проходов, которое я назову w, зависит от «длины» слов, но не от количества слов n. Таким образом, порядок роста равен $O(wn)$ и является линейным по n.

Существует множество вариантов поразрядной сортировки, как и способов реализации каждого из них. Узнать о них больше можно на сайте http://thinkdast.com/radix. В качестве дополнительного упражнения подумайте о написании своей версии поразрядной сортировки.

Пирамидальная сортировка

Наряду с поразрядной сортировкой, которая применяется, когда объекты имеют ограниченный размер, существует еще один алгоритм сортировки специального назначения: *ограниченная пирамидальная сортировка*. Она полезна, если вы работаете с очень большим набором данных и хотите получить выборку «Топ-10» или «Топ-k» для некоторого значения k, намного меньшего, чем n.

Например, предположим, вы контролируете веб-сервис, обрабатывающий миллиард транзакций в день. В конце каждого дня вы хотите сообщить о самых больших транзакциях (или самых медленных, или любых других показательных значениях). Один из вариантов — хранить все транзакции, сортировать их в конце дня и выбирать k верхних. Это потребовало бы времени, пропорционального $n \log n$, и выполнялось бы очень медленно, поскольку, вероятно, не получится вместить миллиард транзакций в память одной программы. Пришлось бы использовать алгоритм сортировки «вне ядра». О внешней сортировке можно прочитать по адресу http://thinkdast.com/extsort.

Используя ограниченную кучу (bounded heap), мы можем сделать намного лучше! Поступим следующим образом.

1. Я объясню неограниченную пирамидальную сортировку.
2. Вы ее реализуете.
3. Я объясню ограниченную пирамидальную сортировку и проанализирую ее.

Чтобы понять пирамидальную сортировку, нужно иметь представление о *куче* — структуре данных, похожей на бинарное дерево поиска (BST). Их различия представлены ниже.

- В бинарном дереве поиска каждая вершина x имеет свойство BST: все вершины в левом поддереве x меньше x, а все вершины в правом — больше x.
- В куче каждая вершина x имеет свойство heap (куча): все вершины в обоих поддеревьях x больше x.
- Кучи подобны сбалансированным бинарным деревьям поиска; при добавлении или удалении элементов выполняют дополнительную работу по балансировке дерева. В результате они могут быть эффективно реализованы с помощью массива элементов.

Наименьший элемент в куче всегда находится в корне, поэтому можно найти его за постоянное время. Добавление и удаление элементов из кучи занимает время, пропорциональное высоте дерева h. И поскольку куча всегда сбалансирована, то h пропорциональна $\log n$. Узнать больше о кучах можно на сайте http://thinkdast.com/heap.

Класс `PriorityQueue` в Java реализован с помощью кучи. Он предоставляет методы, указанные в интерфейсе `Queue`, включая `offer` и `poll`:

- `offer` — добавляет элемент в очередь, обновляя кучу, чтобы каждый узел имел свойство heap; затрачивает время, пропорциональное $\log n$;
- `poll` — удаляет наименьший элемент в очереди из корня и обновляет кучу; затрачивает время, пропорциональное $\log n$.

Этот класс позволит легко отсортировать коллекцию из n элементов следующим образом.

1. Добавить все элементы коллекции в `PriorityQueue` с помощью `offer`.
2. Удалить элементы из очереди, используя `poll`, и добавить их в `List`.

Поскольку `poll` возвращает наименьший элемент, оставшийся в очереди, элементы добавляются в `List` в порядке возрастания. Этот способ сортировки называется *пирамидальной сортировкой* (см. http://thinkdast.com/heapsort).

Добавление n элементов в очередь занимает время, пропорциональное $n \log n$. То же действительно для удаления n элементов. Таким образом, временная сложность пирамидальной сортировки составляет $O(n \log n)$.

В репозитории для этой книги в `ListSorter.java` вы найдете схему метода, называемого `heapSort`. Дополните его, а затем запустите `ant ListSortTest` для подтверждения того, что он работает.

Ограниченная куча

Ограниченная куча — это куча, которая может содержать не более k элементов. При наличии n элементов можно отслеживать k наибольших элементов следующим образом.

Первоначально куча пуста. Для каждого элемента x:

- ветвь 1: в случае незаполненной кучи добавить в нее x;
- ветвь 2: при заполненной куче сравнить x с *наименьшим* элементом в ней; если x меньше, то не может быть одним из k наибольших элементов, так что можно отбросить его;
- ветвь 3: если куча заполнена и x больше самого маленького элемента в куче, то удалить наименьший элемент из кучи и добавить x.

С помощью кучи с наименьшим элементом наверху можно отслеживать k наибольших элементов. Проанализируем эффективность этого алгоритма. Для каждого элемента выполняем одно из следующих действий:

- ветвь 1: добавление в кучу элемента, который имеет временную сложность $O(\log k)$;
- ветвь 2: поиск наименьшего элемента в куче — это $O(1)$;
- ветвь 3: удаление наименьшего элемента — $O(\log k)$; добавление x тоже является $O(\log k)$.

В худшем случае, если элементы появляются в порядке возрастания, всегда запускается ветвь 3. Тогда общая временная сложность обработки n элементов равна $O(n \log k)$ и линейна по n.

В файле `ListSorter.java` вы найдете метод под названием `topK`, который принимает `List`, `Comparator`, и параметр типа `integer` k. Он должен возвращать k наибольших элементов в `List` в порядке возрастания. Заполните метод, а затем запустите `ant ListSortTest` для подтверждения того, что он работает.

Пространственная сложность

До сих пор мы много говорили об анализе времени выполнения, но для многих алгоритмов не менее важным является пространство. Например, один из недостатков сортировки слиянием — она делает копии данных. В нашей реализации общий объем пространства, который она выделяет, пропорционален $O(n \log n)$. С более умной реализацией можно снизить требование к пространству до $O(n)$.

Напротив, сортировка вставкой не копирует данные, так как сортирует элементы на месте. Она применяет временные переменные для сравнения двух элементов за раз и несколько других локальных переменных. Но используемое пространство не зависит от n.

Наша реализация пирамидальной сортировки создает новый метод `PriorityQueue` для хранения элементов, поэтому пространственная сложность составляет $O(n)$; но если вам доступна сортировка списка на месте, то можете запустить пирамидальную сортировку с пространственной сложностью $O(1)$.

Одним из преимуществ алгоритма с использованием ограниченной кучи, который вы только что реализовали, является то, что ему требуется только пространство, пропорциональное k (количеству элементов, которые нужно сохранить), а k часто намного меньше n.

Разработчики программного обеспечения склонны уделять больше внимания времени выполнения, чем пространству, и для многих приложений это подходит. Но для больших массивов данных пространство может быть не менее или даже более важным. Приведу несколько примеров.

- Когда массив данных не помещается в памяти одной программы, время ее выполнения часто резко возрастает или она может вообще не работать. Если выбрать алгоритм, требующий меньше ресурсов и это позволит уместить вычисления в память, то он способен работать намного быстрее. Аналогично программа, которая задействует меньше ресурсов, может лучше использовать кэши процессора и работать быстрее (см. http://thinkdast.com/cache).

- Это актуально и для сервера, одновременно запускающего много программ. Сокращение пространства, необходимого для каждой программы, позволит запустить больше программ на одном сервере, что снижает затраты на оборудование и энергию.

Итак, я привел несколько причин, из-за которых вам нужно иметь хоть какое-то представление о пространственных потребностях алгоритмов.

Об авторе

Аллен Б. Доуни — профессор компьютерных наук в Olin College of Engineering. Преподавал в Колледже Уэллсли (Wellesley College), Колледже Колби (Colby College) и Калифорнийском университете в Беркли (The University of California, Berkeley). Имеет степень доктора философии в области компьютерных наук в Калифорнийском университете в Беркли, степень магистра и бакалавра в Массачусетском технологическом институте (Massachusetts Institute of Technology, MIT).

Об обложке

На обложке изображена *ворона-свистун* (лат. Cracticus tibicen) — яркая черно-белая птица с красными глазами. Европейские поселенцы назвали ее magpie (с англ. — «сорока») за сходство с европейской сорокой, но эти два вида имеют лишь отдаленное родство. Вороны-свистуны распространены в Австралии и Новой Гвинее.

Эти птицы очень умны и известны тем, что, помимо пения, могут издавать множество сложных звуков. Вороны-свистуны, как правило, достигают 37–43 см в длину и весят 220–350 г. Места их обитания разнообразны и включают поля, леса, парки и даже жилые районы. Птицы активны в течение дня и добывают корм на поверхности земли; они всеядны, питаются насекомыми, червями, различными беспозвоночными, орехами, фруктами и мелкими животными, такими как ящерицы и мыши.

В сентябре и октябре (весна в Австралии) мужские особи вороны-свистуна отчаянно защищают свои гнезда и птенцов, что приводит к феномену, известному как «сезон налетов». Птицы агрессивно набрасываются на пешеходов и велосипедистов, часто нанося им травмы лица и головы. В качестве защитных мер рекомендуют рисовать на шлеме «страшные» глаза, носить

зонтик и, конечно, полностью отказаться от посещения мест гнездования. Почтовые работники, которые перемещаются по своим маршрутам на мотоциклах, — частые мишени для налетов ворон-свистунов.

Многие животные, изображенные на обложках издательства O'Reilly, находятся под угрозой; все они важны для нашего мира. Чтобы узнать больше о том, как можно им помочь, перейдите на сайт animals.oreilly.com.

Изображение на обложке взято из книги *Royal Natural History* Ричарда Лидеккера (Richard Lydekker).

Аллен Б. Доуни

Алгоритмы и структуры данных. Извлечение информации на языке Java

Перевел с английского *К. Синица*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*С. Заматевская*
Корректоры	*Е. Павлович, Т. Радецкая*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 191123, Россия, г. Санкт-Петербург,
ул. Радищева, д. 39, к. Д, офис 415. Тел.: +78127037373.

Дата изготовления: 05.2018. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 11.05.18. Формат 60×90/16. Бумага офсетная. Усл. п. л. 15,000.
Тираж 1000. Заказ 3888.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

САЛД

САНКТ-ПЕТЕРБУРГСКАЯ АНТИВИРУСНАЯ ЛАБОРАТОРИЯ ДАНИЛОВА

www.SALD.ru
8 (812) 336-3739

Антивирусные программные продукты

Изучите, как следует реализовывать эффективные алгоритмы на основе важнейших структур данных на языке Java, а также как измерять производительность этих алгоритмов. Каждая глава сопровождается упражнениями, помогающими закрепить материал.

- Научитесь работать со структурами данных, например, со списками и словарями, разберитесь, как они работают.
- Напишите приложение, которое читает страницы Википедии, выполняет синтаксический разбор и обеспечивает навигацию по полученному дереву данных.
- Анализируйте код и учитесь прогнозировать, как быстро он будет работать и сколько памяти при этом потреблять.
- Пишите классы, реализующие интерфейс Map, пользуйтесь при этом хеш-таблицей и двоичным деревом поиска.
- Создайте простой веб-поисковик с собственным поисковым роботом: он будет индексировать веб-страницы, сохранять их содержимое и возвращать нужные результаты.

ПИТЕР®

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг
и интернет-магазин

vk.com/**piterbooks**
instagram.com/**piterbooks**
facebook.com/**piterbooks**
youtube.com/**ThePiterBooks**

ISBN 978-5-4461-0572-4