5 Поиск идеальной структуры данных

Типы данных, обсуждавшиеся в главе 4, “Записная книжка сыщика”, охватывают отдельные схемы доступа к коллекциям данных. А поскольку коллекции могут сильно разрастаться, с практической точки зрения очень важно управлять ими эффективно. Как упоминалось ранее, в разных структурах данных поддерживается эффективность тех или иных операций, и они предъявляют разные требования к свободному пространству для хранения данных. Если построение и преобразование коллекций являются важными задачами, то поиск элементов в коллекции, вероятно, относится к наиболее часто востребованным операциям.

Нам постоянно приходится что-то искать. И зачастую это происходит неосознанно, хотя иногда мы болезненно воспринимаем данное обстоятельство, например в ходе такой обыденной деятельности, как мучительный поиск ключей от автомашины. Чем больше имеется мест для поиска и элементов для обхода, тем труднее найти именно то, что требуется найти. С годами у нас накапливается немало артефактов, ведь, в конце концов, мы — наследники охотников и собирателей. Кроме таких элементов реальных коллекций, как марки, монеты или коллекционные карточки с изображениями известных спортсменов, мы с годами накапливаем целые собрания книг, картин или предметов одежды. Иногда это происходит как побочный эффект какого-нибудь увлечения или страсти. Мне, например, известен ряд страстных любителей домашних ремонтов, накопивших впечатляющие коллекции инструментов.

Если книжная полка организована по алфавиту или по теме, а коллекция картин является электронной и имеет временны*е* отметки, то поиск конкретной

книги или изображения может упроститься. Если же коллекция насчитывает немало элементов, но не организована как следует, то поиск в ней может быть сильно затруднен.

Положение намного ухудшается, когда дело доходит до хранения данных в электронном виде. Ведь в данном случае имеется, по существу, неограниченный доступ к месту для хранения данных, и поэтому объем хранимых данных быстро растет. Например, по статистике веб-сайта YouTube каждую минуту на него выгружается около 300 часов видеозаписей [1].

Поиск является преобладающей задачей в реальной жизни и очень важным вопросом в информатике. Алгоритмы и структуры данных способны значительно ускорить процесс поиска. И то, что подходит для данных, иногда может помочь и в хранении и извлечении физических артефактов. Этот очень простой метод освобождает от повторного поиска ключей от автомашины, если, конечно, скрупулезно ему следовать.

Ключ к быстрому поиску

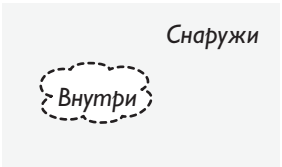
В истории из фильма *Индиана Джонс и последний крестовый поход* главный герой, Индиана Джонс, отправляется на поиски двух основных объектов. Сначала он пытается найти своего отца, сэра Генри Джонса, а затем вместе с ним — Священный Грааль. Будучи археологом, Индиана Джонс знает объекты своих поисков. На одной из своих лекций он фактически поясняет студентам следующее:

Археология — это поиск фактов.

Каким же образом действует поиск археологического артефакта (или человека)? Если местоположение искомого объекта известно, то никакого поиска, безусловно, не требуется. В противном случае процесс поиска зависит от следующих двух факторов: *места поиска* и потенциальных *путеводных нитей*, или просто *ключей*, сужающих область поиска.

В истории из фильма *Индиана Джонс и последний крестовый поход* главный герой получает по почте из Венеции записную книжку своего отца со сведениями о Священном Граале, что вынуждает его начать поиски именно оттуда. И здесь происхождение записной книжки служит Индиане Джонсу ключом к разгадке, значительно сужающим первоначальную область поиска от целого земного шара до одного города. В данном примере область поиска имеет двумерную геометрическую форму буквально, но в общем этот термин обозначает нечто более абстрактное. Например, любую структуру данных, представляющую коллекцию элементов, можно рассматривать в качестве области поиска, в которой можно искать конкретные элементы. Например, такой областью поиска служит составленный Шерлоком Холмсом список подозреваемых, в котором можно найти конкретное имя.

Насколько списки пригодны для поиска? Как обсуждалось в главе 4, "Записная книжка сыщика", в худшем случае приходится предварительно просматривать все элементы в списке, прежде чем будет найден нужный элемент, или же выяснять, что таковой элемент в списке отсутствует. Это означает, что список не позволяет эффективно воспользоваться ключом для сужения области поиска.



Чтобы стало понятнее, почему список не годится в качестве структуры данных для поиска, целесообразно рассмотреть подробнее принцип действия ключей. В двумерной области поиска ключ обеспечивает границу, отделяющую то, что находится "снаружи" и не содержит искомый элемент, от того, что находится "внутри" и может содержать искомый элемент [2]. Аналогично ключи обозначают в структуре данных границу, которая может разделять разные части этой структуры данных, а следовательно, ограничивать поиск пределами одной из ее частей. Кроме того, ключ представляет собой фрагмент информации, связанной с искомым объектом. В частности, ключ должен быть в состоянии выявить границу между элементами, относящимися к текущему поиску, и не имеющими к нему отношения.

В списке каждый элемент служит границей, отделяющей предшествующие ей элементы от следующих за ней. Но поскольку элементы в средине списка непосредственно недоступны и список вместо этого приходится всегда обходить из одного конца в другой, то его элементы, по существу, не способны отделить внешнее от внутреннего. Рассмотрим первый элемент в списке. Он не исключает никакой элемент из поиска, кроме него самого. Ведь если это не искомый элемент, то поиск придется продолжить среди всех остальных элементов списка. Но тогда при просмотре второго элемента в списке возникнет та же самая ситуация. Если же и второй элемент не окажется искомым, то придется снова продолжить поиск среди всех остальных элементов списка. Безусловно, второй элемент определяется как внешний по отношению к первому элементу списка, который не нужно проверять. Но тот факт, что он уже был проанализирован, еще не означает, что можно что-нибудь сэкономить на поиске. Это же справедливо для каждого элемента списка: чтобы достичь элемента в списке, необходимо проверить то, что определяется им как внешнее.

Приехав в Венецию, Индиана Джонс продолжает свои поиски в библиотеке. Поиск книги в библиотеке служит ярким примером, демонстрирующим применение понятий границы и ключа на практике. Когда книги располагаются на библиотечных полках упорядоченными по фамилиям авторов, каждая полка нередко обозначается фамилиями авторов первой и последней книг на этой полке. Фамилии этих двух авторов задают пределы, в которых можно искать книги указанной части авторов на полке. По существу, фамилии этих двух авторов определяют границу, отделяющую книги авторов в заданных пределах от всех остальных

книг. Допустим, что в библиотеке требуется найти повесть *Собака Баскервилей* Артура Конана Дойла. В качестве ключа можно воспользоваться фамилией данного автора, чтобы сначала найти полку с книгами в соответствующих пределах фамилий авторов. Как только такая полка будет найдена, поиск можно будет продолжить на самой полке. Такой поиск осуществляется в две стадии: на первой ищется полка, а на второй — книга на самой полке. Такая стратегия поиска оказывается вполне пригодной, поскольку разделяет область поиска на целый ряд небольших не перекрывающихся областей (полок), разделяемых фамилиями авторов как границами.

Поиск книг на библиотечных полках можно осуществлять по-разному. Один из способов состоит в том, чтобы проверять полки по очереди до тех пор, пока не будет найдена полка с фамилией "Дойл". В этом случае полки, по существу, выступают в качестве списка, а следовательно, им присущи такие же ограничения, как и спискам. Так, поиск по фамилии "Дойл" может отнять не очень много времени, но если искать книгу по фамилии "Ясперс", то ее поиск может отнять намного больше времени. Лишь немногие начнут поиск книги Ясперса, начиная не с буквы А, а с буквы Я, чтобы быстрее найти нужную полку. Такой способ основывается на предположении, что все библиотечные полки упорядочены по фамилиям авторов и что на каждую букву алфавита приходится одна полка, и поэтому вполне естественно начать поиск книг Ясперса с последней полки в библиотеке, а книги Дойла — с пятой полки.

Иными словами, библиотечные полки можно рассматривать в качестве элементов массива, проиндексированных буквами. Маловероятно, конечно, что в библиотеке есть столько же полок, сколько букв в алфавите, но рассматриваемый здесь способ можно расширить до любого количества полок, просто начав поиск книг Ясперса с последней одной тридцать второй части полок. А недостаток такого способа поиска заключается в том, что количество авторов, фамилии которых начинаются на одну и ту же букву, сильно разнится (например, авторов на букву "Д" намного больше, чем на букву "Я"). Иными словами, книги распределены неравномерно по всем полкам в библиотеке, и поэтому такая стратегия поиска неточна. Следовательно, посетителю библиотеки придется совершать поиск то в одном, то в другом направлении, чтобы найти целевую полку. Зная о неравномерном распределении фамилий авторов, можно было бы значительно повысить точность данной стратегии поиска, но даже столь простая стратегия оказывается вполне пригодной на практике, по существу, исключая как нечто "внешнее" большое количество полок, которые вообще не стоит просматривать.

Поиск книги на одной полке можно продолжать разными способами. Один из них состоит в том, чтобы просмотреть книги по очереди или прибегнуть к стратегии, аналогичной той, которая применяется для обнаружения полок, оценив местоположение книги по положению ключа в пределах фамилий авторов на

данной конкретной полке. В общем, двухэтапный способ поиска вполне пригоден и оказывается намного более быстродействующим, чем прямолинейный способ просмотра всех книг по очереди. Я провел эксперимент в поисках повести *Собака Баскервилей* в публичной библиотеке города Корваллис, шт. Орегон. И мне удалось найти нужную полку за пять шагов, а книгу на полке — еще за семь шагов. Это значительное улучшение по сравнению с прямолинейным способом поиска перебором, особенно если учесть, что на тот момент в данной библиотеке находилось 44 679 книг по художественной литературе для взрослых на 36 полках.

Решение Индианы Джонса отправиться в Венецию, чтобы найти своего отца, основано на аналогичной стратегии. В этом случае внешний мир можно рассматривать как массив элементов, соответствующих географическим регионам, проиндексированным по названиям городов. Обратный адрес на присланной по почте посылке с записной книжкой сэра Генри Джонса служит ключом для выбора элемента массива по индексу "Венеция", чтобы продолжить поиск. Поиск в Венеции приводит Индиану Джонса в библиотеку, где он ищет не книгу, а надгробие сэра Ричарда, одного из рыцарей, участвовавших в пятом крестовом походе. Он нашел надгробие после того, как разбил плитку на полу, помеченную знаком X, что не было лишено иронии, если принять во внимание следующее его прежнее заявление своим студентам:

Знак "X" едва ли вообще помечает место.

Ключом к быстрому поиску служит наличие структуры, позволяющей быстро сузить поиск. Чем меньше "внутреннее", обозначаемое ключом, тем лучше, поскольку это позволяет быстрее свести весь поиск к искомому результату. Так, если речь идет о поиске книг, то две стадии такого поиска разделяются на одном сужающем поиск главном шаге.

Выживание в боггле

Нередко поиск скрывается под какой-нибудь личиной в самых неожиданных обстоятельствах. К концу истории из фильма *Индиана Джонс и последний крестовый поход* главный герой, Индиана Джонс, прибывает в Храм Солнца, где ему предстоит преодолеть три препятствия, прежде чем войти, наконец, в помещение, где хранится Священный Грааль. Чтобы преодолеть второе препятствие, ему необходимо пересечь выложенный плиткой пол над пропастью. Трудность состоит в том, что лишь некоторые плитки пола надежны, а остальные осыпаются и ведут к ужасной смерти, если наступить на них. Пол состоит приблизительно из 50 плиток, неравномерно уложенных сеткой и обозначенных отдельными буквами алфавита. Доля археолога непроста: порой ему приходится ломать плитку пола, чтобы преуспеть, а иногда — избегать этого любой ценой.

Найти надежные плитки, на которые можно безопасно наступить, не так-то просто, поскольку в отсутствие каких-нибудь других ограничений вероятность попасть на такую плитку составляет один шанс из тысячи триллионов, т.е. число, состоящее из единицы с пятнадцатью нулями. Ключ к поиску жизнеспособной последовательности плиток, надежно ведущей к другой стороне пола, состоит в том, что буквы на плитках должны составлять имя *Iehova* (Иегова, т.е. "Сущий" — имя Бога в Ветхом Завете). И хотя эта информация, по существу, решает загадку, необходимо еще потрудиться, чтобы выявить правильную последовательность плиток. Как ни странно, для этого потребуется *поиск*, систематически сужающий область возможностей выбора.

Эта задача подобна игре в боггл, цель которой — найти строки связанных букв на сетке, образующей слова [3]. Задача Индианы Джонса кажется довольно простой, поскольку ему уже известно слово. Но в боггле последовательные символы должны находиться на соседних плитках; такого ограничения для задачи преодоления плиточного пола нет, а следовательно, имеется куда больше возможностей, которые необходимо рассмотреть Индиане Джонсу, что существенно затрудняет решение данной задачи.

Чтобы продемонстрировать решение задачи поиска, допустим ради простоты, что пол состоит из шести рядов, а каждый из них — из восьми плиток с разными буквами, что в итоге составляет 48 плиток. Если правильный путь состоит из одной плитки в каждом ряду, то всего имеется 8×8×8×8×8×8 = 262 144 возможных путей через все возможные комбинации плиток в шести рядах. И лишь один из этих путей оказывается правильным.

Как же тогда Индиане Джонсу найти правильный путь? Руководствуясь ключевым словом, он находит плитку с буквой *I* в первом ряду и становится на нее. Это само по себе обозначает процесс поиска, состоящий из нескольких стадий. Если буквы на плитках расположены не в алфавитном порядке, то Индиане Джонсу придется просматривать их по очереди до тех пор, пока не будет найдена плитка с буквой *I*. Став на нее, он продолжает поиск плитки с буквой *e* во втором ряду и т.д.

Если поочередный поиск напоминает вам поиск элемента в списке, вы совершенно правы. Именно это и происходит в данном случае. Отличие состоит в том, что поиск в списке повторяется в каждом ряду, т.е. каждого символа в ключевом слове. И в этом состоит истинная сила данного метода поиска. Рассмотрим, что происходит в области поиска, где насчитывается всего 262 144 возможных путей через сетку плиток. Каждая плитка в первом ряду обозначает разную отправную точку пути, который может быть продолжен 8×8×8×8×8 = 32 768 разными способами выбора разных сочетаний букв из пяти оставшихся рядок плиток. Если Индиана Джонс найдет первую плитку, обозначенную, скажем, буквой *K*, он,

конечно, не станет на нее, поскольку она не соответствует требуемой букве *I*. Этим единственным решением одним махом отметается в общем 32 768 остальных путей из области поиска, а именно — тех путей, которые могут быть сформированы, начиная с плитки, обозначенной буквой *K*. И то же самое сокращение возможных путей происходит с каждой последующей отвергаемой плиткой в первом ряду.

Как только Индиана Джонс найдет правильную плитку, область поиска сократится еще больше. Ведь как только он станет на плитку, процесс принятия решения в первом ряду завершится, и область поиска сразу же сократится до 32 768 путей, которые могут быть сформированы в оставшихся пяти рядах. Приняв в общем не больше семи решений, которые требуются, если плитка с буквой *I* оказывается последней в первом ряду, Индиана Джонс сокращает область поиска в восемь раз. И так он продолжает действовать во втором ряду плиток, чтобы найти букву *e*. И снова область поиска сокращается в восемь раз, т.е. до 4096 путей, с помощью не более семи решений. Как только Индиана Джонс достигнет последнего ряда, у него останется восемь возможных путей, и снова ему потребуется принять не более семи решений, чтобы завершить свой путь. В худшем случае для поиска ему потребуется сделать 6×7 = 42 "шага" (и лишь шесть буквальных шагов), и это замечательно эффективный способ поиска одного из 262 144 возможных путей.

Трудная задача, которую удалось решить Индиане Джонсу, имеет некоторое, хотя и не совсем очевидное, сходство с задачей Гензеля и Гретель. В обоих случаях главному герою приходится искать безопасный путь, который состоит из последовательного ряда отмеченных мест: в истории Гензеля и Гретель — камешками, а в истории Индианы Джонса — буквами на плитках пола, но поиск следующего местоположения на обоих путях разнится. В частности, Гензелю и Гретель достаточно найти любой камешек, хотя они должны следить за тем, чтобы не обходить повторно те же самые камешки, тогда как Индиане Джонсу требуется найти плитку с конкретной буквой. Оба эти примера еще раз подчеркивают роль представления в вычислении. Трудная задача, которую приходится решать Индиане Джонсу, демонстрирует, в частности, что последовательность означающих (букв) для отдельных плиток сама по себе служит еще одним означающим (слово *Iehova*) для пути. А тот факт, что слово *Iehova* обозначает искомый путь, наглядно показывает, насколько значимым и важным в реальном мире становится вычисление, которое просто состоит в поиске слова.

Способ, которым Индиана Джонс отыскивает ключевое слово на сетке плиток, точно соответствует тому, как можно эффективно найти слово в словаре, сузив поиск сначала до ряда страниц, содержащих слово, начинающееся на ту же самую букву, что и ключевое слово, а затем до еще более узкого ряда страниц, соответствующих первой и второй буквам искомого слова, и так далее до тех пор, пока слово не будет найдено.

Чтобы лучше понять структуру данных, скрывающуюся за сеткой плиток и делающей поиск Индианы Джонса столь эффективным, следует рассмотреть еще один способ представления слов и словарей, в которых для организации области поиска применяются деревья.

Словарный подсчет

В главе 4, "Записная книжка сыщика", была описана структура данных дерева. В частности, генеалогическое дерево послужило примером для того, чтобы продемонстрировать порядок вычисления списка наследников и приоритета их наследования. В ходе этого вычисления был выполнен обход всего дерева, при котором пришлось посетить каждый его узел. Деревья отлично подходят и для поддержки поиска элементов в коллекции. В этом случае узлы дерева используются с целью направить поиск по единственно верному пути и найти требуемый элемент.

Рассмотрим еще раз трудную задачу, которую пришлось решать Индиане Джонсу. В фильме *Индиана Джонс и последний крестовый поход* его первый шаг приводит к драматичной сцене, в которой он едва не погиб, став на ненадежную плитку. Используя написание *Jehova* ключевого слова, он становится на плитку с буквой *J*, которая ломается под его ступнями. Это ясно показывает, что рассматриваемая здесь трудная задача фактически оказывается еще более трудной, чем кажется на первый взгляд, поскольку у Индианы Джонса нет полной уверенности в правильности написания ключевого слова. Помимо букв *I* и *J*, ключевое слово может начинаться и с буквы *Y*. Кроме того, имеются другие возможные имена Бога, которые в принципе подходят для решения данной задачи, например *Yahweh* (Яхве) и *God* (Бог). Допустим, что все эти слова составляют возможные варианты выбора и что Индиана Джонс и его отец уверены, что одно из этих слов действительно указывает безопасный путь через плиточный пол.

Какая же стратегия подходит для выбора плитки, на которую следует стать? Если все упомянутые выше ключевые слова равнозначны, то Индиана Джонс мог бы улучшить свое положение, выбрав букву, присутствующую в нескольких словах. Допустим, что он выбрал букву *J*, как это и было на самом деле. Буква *J* присутствует только в одном из имен Бога, и поэтому есть лишь один из пяти шансов (т.е. вероятность 20%), что плитка с этой буквой устоит. С другой стороны, вероятность безопасно стать на плитку с буквой *v* составляет 60%, поскольку эта буква присутствует в трех именах Бога. И если какое-то из этих слов правильное, то плитка с буквой *v* окажется надежной, а поскольку все ключевые слова считаются в равной степени правильными, то шансы выжить возрастают для плитки с буквой *v* до трех из пяти [4].

Таким образом, подходящая стратегия состоит в том, чтобы вычислить сначала частотность букв в пяти словах, а затем попробовать стать на плитки с наибольшей встречаемостью. Такое соответствие букв их частотности называется *гистограммой*. Чтобы вычислить гистограмму букв, Индиане Джонсу придется организовать ведение подсчета частотности каждой буквы. Просмотрев все слова, он может инкрементировать подсчет каждого встретившегося слова. Ведение подсчета разных букв является задачей, решаемой с помощью типа данных словаря (см. главу 4, "Записная книжка сыщика"). В данном случае ключи служат отдельными буквами, а сведения, сохраняемые с каждым ключом, — частотностью буквы ключа.

С одной стороны, чтобы реализовать такой словарь, можно было бы воспользоваться массивом с буквами в качестве индексов, но это означало бы напрасно израсходовать более половины свободного места в массиве, поскольку подсчитать требуется всего одиннадцать разных букв. С другой стороны, для этой цели можно было бы воспользоваться списком, но и это не очень эффективно. Чтобы убедиться в этом, допустим, что слова просматриваются в алфавитном порядке. Следовательно, просмотр начинается со слова *God*, каждая буква которого вводится в список вместе с начальным подсчетом, равным 1. В итоге получается следующий список:

G:1 → o:1 → d:1

Следует, однако, иметь в виду, что если для ввода буквы *G* в список потребовался один шаг, то для ввода буквы *o* — два шага, поскольку ее пришлось ввести после буквы *G*, а для ввода буквы *d* — уже три шага, потому что она должна быть введена в список после обеих букв, *G* и *o*. Но нельзя ли вместо этого вводить новые буквы в начале списка? К сожалению, нельзя, поскольку необходимо убедиться в отсутствии буквы в списке, прежде чем ее вводить. Следовательно, придется просмотреть все присутствующие в списке буквы, прежде чем вводить в него новую букву. Если буква уже присутствует в списке, то вместо ее ввода следует просто увеличить ее счетчик. Таким образом, на первое слово уже затрачено 1+2+3 = 6 шагов.

Если перейти к следующему слову, *Iehova*, то для ввода букв *I*, *e* и *h* в список потребуется 4, 5 и 6 шагов соответственно, а в итоге — 21 шаг. Далее следует буква *o*, которая уже присутствует в списке. Чтобы найти эту букву и обновить ее счетчик до 2, потребуются лишь два шага. Буквы *v* и *a* являются новыми для данного списка, и поэтому для их добавления в конец списка потребуется 7+8 = 15 шагов. Итак, для заполнения данного списка потребовалось всего 38 шагов, а выглядит он следующим образом:

G:1 → o:2 → d:1 → I:1 → e:1 → h:1 → v:1 → a:1

Для обновления в словаре счетчиков букв из слова *Jehova* потребуется еще 9 шагов на ввод буквы *J* и далее 5, 6, 2, 7 и 8 шагов для увеличения счетчиков остальных букв, уже присутствующих в списке, что в итоге составит 75 шагов. Для обработки двух последних слов, *Yahweh* и *Yehova*, требуется 40 и 38 шагов соответственно, а в итоге получается следующий список, на заполнение которого потребовалось 153 шага:

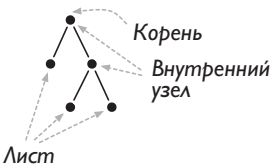
G:1 → o:4 → d:1 → I:1 → e:4 → h:4 → v:3 → a:4 → J:1 → Y:2 → w:1

Следует, однако, иметь в виду, что вводить вторую букву *h* из слова *Yahweh* не нужно, поскольку двойной ее учет для одного слова приведет (причем неверно) к увеличению вероятности встретить эту букву (в нашем случае — с 80% до 100%) [5]. Окончательный просмотр данного списка показывает, что буквы *o*, *e*, *h* и *a* оказываются самыми надежными, а вероятность встретить их в правильном слове составляет 80%.

Скудость — не всегда добродетель

Главный недостаток реализации словаря в виде структуры данных списка заключается в больших затратах на повторный доступ к элементам, находящимся ближе к концу списка. В структуре данных *бинарного дерева поиска* предпринимается попытка преодолеть подобный недостаток путем более равномерного разделения области поиска для поддержки ускоренного поиска. *Бинарным* называется такое дерево, у каждого узла которого имеются хотя бы два потомка. Как упоминалось ранее, узлы без потомков называются *листьями*, узлы с потомками — *внутренними узлами*, а узел, не имеющий родителя, — *корневым*.

Рассмотрим ряд примеров бинарного дерева. Так, на рис. 5.1, *слева*, показано дерево с единственным узлом, т.е. самое простое дерево, какое только можно себе представить, — это корень, одновременно являющийся листом этого дерева (помечен буквой *G*). На самом деле это дерево не очень похоже на подлинное дерево, подобно тому как список с единственным элементом — на настоящий список. На рис. 5.1 посредине показано дерево, состоящее из двух узлов: корневого и дочернего узла, помеченного буквой *o*, а на рис. 5.1, *справа*, — то же дерево, но содержащее дополнительный дочерний узел, помеченный буквой *d* и имеющий два потомка — листья *a* и *e*. Данный пример наглядно показывает, что если отсечь связь узла с его родителем, он станет корневым узлом отдельного дерева, называемого *поддеревом* своего родительского дерева. В данном случае дерево с корневым узлом *d* и двумя его потомками, *a* и *e*, является поддеревом узла *G*, как, врочем, и дерево с единственным (корневым) узлом *o*. Это означает, что дерево, по существу, является



рекурсивной структурой данных и может быть определено следующим образом: дерево состоит из единственного узла или же из узла с одним или двумя поддеревьями, корневые узлы которых являются потомками данного узла. На подобный рекурсивный характер структуры деревьев указывалось в главе 4, "Записная книжка сыщика", в которой был представлен рекурсивный алгоритм для обхода генеалогического дерева.

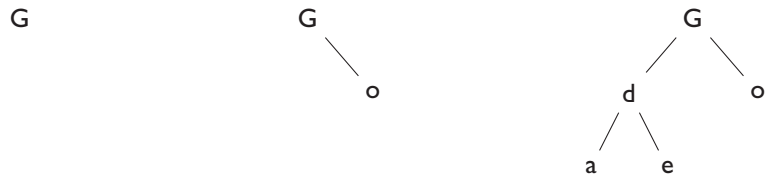


Рис. 5.1. Три примера двоичных деревьев. Слева: дерево с единственным узлом. Посредине: дерево с корневым узлом и правым поддеревом, которое представляет собой единственный узел. Справа: дерево с корневым узлом и двумя поддеревьями. Все три дерева обладают свойством бинарного поиска, которое означает, что узлы в левых поддеревьях содержат значения, меньшие значений в корневом узле, а узлы в правых поддеревьях — значения, большие, чем в корневом узле

Принцип организации бинарного дерева поиска состоит в том, чтобы использовать внутренние узлы в качестве границ, разделяющих все дочерние узлы на два класса: узлы со значением, меньшим, чем у граничного узла, содержатся в левом поддереве, а узлы с б*о*льшим значением — в правом поддереве.

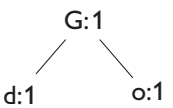
Таким расположением узлов дерева можно воспользоваться для поиска. Так, если требуется найти конкретное значение в дереве, оно сравнивается со значением в корневом узле дерева. Если обнаружено совпадение, то искомое значение найдено, и на этом поиск прекращается. В противном случае, если искомое значение оказывается меньше, чем значение в корневом узле дерева, дальнейший поиск можно ограничить только левым поддеревом. Это избавляет от необходимости искать значение в любых узлах правого поддерева, поскольку заранее известно, что все его узлы содержат большее значение, чем в корневом узле, а следовательно, оно больше искомого значения. Но в любом случае внутренний узел определяет границу, разделяющую то, что находится "внутри" (т.е. в левом поддереве), от того, что находится "снаружи" (т.е. в правом поддереве), подобно тому, как дневник о Святом Граале определяет то, что находится "внутри", на следующем шаге поиска Индианой Джонсом своего отца.

Все три дерева, показанные на рис. 5.1, являются бинарными деревьями поиска. В качестве значений они хранят буквы, которые можно сравнивать по их положению в алфавите. Чтобы найти значение в таком дереве, придется неоднократно сравнивать искомое значение со значениями в отдельных узлах дерева, опускаясь соответственно по правому или левому поддереву.

Допустим, что требуется выяснить, где находится буква *e* в правом дереве на рис. 5.1. Начнем поиск с корневого узла дерева и сравним буквы *e* и *G*. Поскольку буква *e* предшествует в алфавите букве *G*, она оказывается "меньше" буквы *G*, и поиск продолжается в левом поддереве, где находятся буквы, значения которых меньше значения буквы *G*. Сравнив далее букву *e* с буквой *d* в корневом узле левого поддерева, мы обнаружим, что буква *e* оказывается "больше" буквы *d*, а следовательно, поиск продолжается в правом поддереве, которое содержит единственный узел. Сравнение буквы *e* со значением в этом узле завершает поиск. Если бы мы искали букву *f*, то ее поиск привел бы к тому же самому узлу *e*, но поскольку у узла *e* правое поддерево отсутствует, то поиск завершился бы в узле *e* неудачей — выводом, что буква *f* в дереве отсутствует.

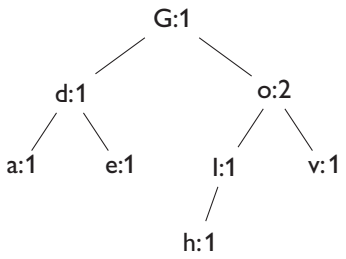
Любой поиск следует по некоторому пути в дереве, и поэтому время поиска элемента (или вывода об его отсутствии в дереве) никогда не превышает время следования по самому длинному пути от корня к листу дерева. Дерево, показанное на рис. 5.1, *справа*, состоит из пяти элементов, а пути к его листьям имеют длину, равную только двум или трем элементам. Это означает, что любой элемент можно найти не больше чем за три шага. Сравните это со списком из пяти элементов, в котором для поиска последнего элемента всегда требуется пять шагов.

Теперь мы готовы вычислить гистограмму букв, используя для представления словаря бинарное дерево поиска. И в этом случае мы также начнем со слова *God*, вводя каждую его букву в дерево вместе с начальным счетчиком этой буквы, равным 1. В итоге мы получим приведенное ниже дерево, отражающее порядок появления букв в дереве.

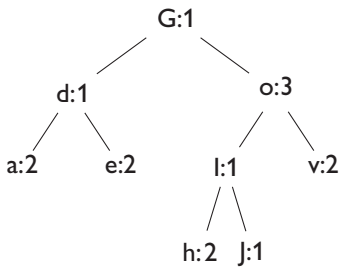


В данном случае для ввода буквы *G* в дерево потребовался один шаг, как и для ввода в список, но для ввода букв *o* и *d* потребовалось только два шага, поскольку их узлы являются потомками узла с буквой *G*. Таким образом, мы сэкономили один шаг на первом слове (1+2+2 = 5 шагов для его ввода в дерево в сравнении с 1+2+3 = 6 шагами для ввода в список).

Для ввода следующего слова, *Iehova*, требуются по три шага на каждую его букву, кроме букв *o* и *h*, для ввода которых требуются два и четыре шага соответственно. Как и при вводе в списки, обработка буквы *o* не приводит к созданию нового элемента, а лишь увеличивает на 1 счетчик существующего элемента. Следовательно, для обработки данного слова потребуется 18 шагов, а вместе с предыдущим словом — 24 шага по сравнению с 38 шагами, потребовавшимися для структуры данных списка. Слово *Jehova* лишь незначительно изменяет структуру дерева, вводя узел для буквы *J*, на что требуется четыре шага. Еще за 3+4+2+3+3 = 15 шагов обновляются счетчики существующих букв, что в итоге дает 39 шагов (сравните с 75 шагами, требовавшимися для списка).

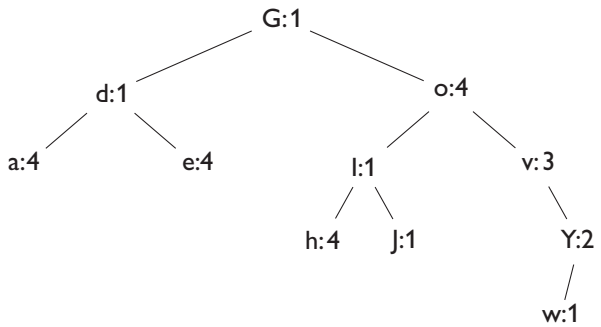


После ввода слова *Iehova*



После ввода слова *Jehova*

И наконец, для ввода каждого из слов *Yahwe(h)* и *Yehova* потребовалось по 19 шагов, так что в итоге полностью дерево было построено за 76 шагов. И это составляет лишь половину от 153 шагов, необходимых для построения списка.



Приведенное выше бинарное дерево поиска представляет тот же самый словарь, что и список, но в форме, поддерживающей ускоренный поиск и обновление, по крайней мере в общем случае. Пространственные формы списков и деревьев поясняют некоторые различия их эффективности. Длинная и узкая структура списка нередко вынуждает поиск следовать по долгому пути и просматривать ненужные элементы. Широкая же и плоская форма дерева эффективно направляет поиск и ограничивает количество рассматриваемых элементов и длину пройденного расстояния. Однако для беспристрастного сравнения списков и бинарных деревьев поиска требуется принять во внимание ряд дополнительных особенностей.

Балансирующая эффективность

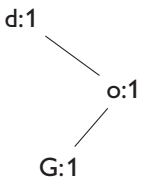
В информатике разные структуры данных обычно не сравниваются путем подсчета точного количества шагов для конкретных списков или деревьев, поскольку это может создать неверное впечатление (особенно это касается небольших структур данных). Но даже в столь простом анализе сравнивались операции, имеющие

неодинаковую сложность и требующие разного времени на выполнение. Например, чтобы выполнить сравнение в списке, достаточно проверить равенство двух элементов, тогда как в бинарном дереве поиска требуется определить, какой из сравниваемых элементов больше, чтобы направить поиск в соответствующее поддерево. Это означает, что выполнить большинство из 153 операций над списком оказывается быстрее и проще, чем некоторые из 76 операций над деревом, и поэтому не следует придавать особого значения непосредственному сравнению двух числовых показателей.

Вместо этого мы рассмотрим увеличение времени выполнения операций по мере укрупнения и усложнения структур данных. Соответствующие примеры были приведены в главе 2, "От слов к делу: когда действительно происходит вычисление". Что касается списков, то, как известно, время выполнения операции ввода новых элементов или поиска существующих элементов линейно, т.е. время ввода или поиска элементов в худшем случае пропорционально длине списка. Это совсем не так плохо, но если выполнять такую операцию многократно, то время ее выполнения будет накапливаться по квадратичному закону, что может оказаться недопустимым (вспомните пример, когда Гензелю приходилось за каждым новым камушком возвращаться домой).

Каково же в сравнении с этим время выполнения операций ввода и поиска элементов в дереве? Наше окончательное дерево поиска состоит из 11 элементов; и поиск, и ввод в него элементов отнимает от трех до пяти шагов. В действительности время поиска или ввода элемента ограничивается высотой дерева, которая зачастую оказывается намного меньше количества его элементов. В сбалансированном дереве, т.е. в дереве, в котором все пути от корня к листу имеют одинаковую длину (±1), высота дерева находится в *логарифмической* зависимости от его размера. Это означает, что высота вырастает лишь на 1, когда количество узлов в дереве удваивается. Например, сбалансированное дерево с 15 узлами имеет высоту 4, сбалансированное дерево с 1000 узлов — высоту 10, а с 1 000 000 узлов — высоту 20. Логарифмическое время выполнения оказывается намного лучше, чем линейное, и по мере увеличения размера словаря структура данных дерева становится все лучше в сравнении со структурой данных списка.

Такой анализ основывается на *сбалансированных* бинарных деревьях поиска. А можно ли гарантировать сбалансированный характер двоичных деревьев при их построении? И если нельзя, то каким будет время выполнения, если деревья окажутся несбалансированными? Окончательное дерево в рассматриваемом здесь примере *не* сбалансировано, поскольку длина пути к листу *e* равна 3, тогда как длина пути к листу *w* равна 5. Порядок, в котором буквы вводятся в дерево, имеет значение и может привести к построению разных деревьев. Так, если вводить буквы в том порядке, в каком они следуют в слове *doG*, то получится приведенное ниже бинарное дерево поиска, которое вообще не сбалансировано.



Такое дерево совершенно не сбалансировано и, по сути, мало чем отличается от списка. И это не какое-то редкое исключение. Две (*God* и *Gdo*) из шести возможных последовательностей, состоящих из трех букв, приводят к сбалансированному дереву, а четыре другие последовательности — к спискам. Правда, существуют методы балансирования бинарных деревьев поиска, теряющих свойство сбалансированности после операций добавления узлов. И хотя они и увеличивают время выполнения операции добавления узла, оно по-прежнему сохраняет свой логарифмический характер.

Наконец, бинарные деревья поиска пригодны для хранения лишь таких разновидностей элементов, которые всегда упорядочены, т.е. о двух любых элементах всегда можно сказать, какой из них больше, а какой меньше. Такое сравнение требуется для того, чтобы решить, в каком направлении двигаться по дереву, чтобы найти или сохранить в нем элемент. Но для некоторых видов данных такие сравнения недопустимы. Предположим, что требуется вести заметки о рисунках стеганых изделий и для каждого рисунка записывать требующиеся ткани и инструменты, особенности и время выполнения стежки. Чтобы сохранить сведения о рисунках стеганых изделий в бинарном дереве поиска, необходимо каким-то образом решить, какой из рисунков больше (или меньше) другого. А поскольку рисунки различаются количеством, формой и цветами фрагментов, совершенно не очевидно, как правильно упорядочить эти рисунки. Это не такая уж невыполнимая задача, поскольку отдельный рисунок можно сначала разложить на составляющие его фрагменты и описать с помощью перечня свойств (например, количества фрагментов, их форм и цветов), а затем сравнить рисунки по перечням их свойств (хотя это потребует определенных усилий и может быть не очень практичным). Таким образом, бинарное дерево поиска может оказаться не совсем пригодным для хранения словаря с рисунками стеганых изделий. Тем не менее для решения подобной задачи можно было бы воспользоваться списком, поскольку для списка достаточно выяснить, являются ли два рисунка одинаковыми, что куда проще, чем их упорядочивать.

Таким образом, бинарные деревья поиска используют стратегию, которая естественно и без особых усилий применяется людьми для разложения задач поиска на более мелкие подзадачи. Фактически бинарные деревья поиска основаны на систематизированном и более совершенном принципе и как следствие для представления словарей могут быть намного более эффективными, чем списки. Однако они требуют больше усилий, чтобы гарантировать сбалансированность, и пригодны не для всех разновидностей данных. Все это отвечает вашему обычному опыту поиска предметов на рабочем столе, на кухне или в гараже. Так, если вы регулярно тратите усилия на поддержание порядка и, попользовавшись, кладете документы или инструменты на место, то вы сможете намного легче и быстрее

найти нужные предметы, чем при поиске их в беспорядочной свалке, которую вы гордо называете рабочим столом.

Префиксное дерево

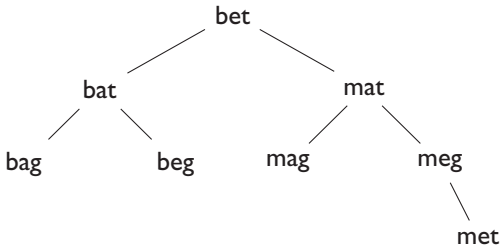
Бинарные деревья поиска служат эффективной альтернативой спискам, когда речь идет о сравнении гистограммы частотности букв в словах. Но это лишь одна из разновидностей вычислений, помогающих Индиане Джонсу решить трудную задачу пересечения плиточного пола. Другая разновидность вычисления состоит в том, чтобы выявить последовательность плиток, образующих конкретное слово из букв и ведущих по безопасному пути через плиточный пол.

Как мы уже определили, сетка состоит из шести рядов, а каждый из них — из восьми букв, образуя 262 144 разных пути по шести плиткам. А поскольку каждый путь соответствует одному слову, связь слов с путями можно было бы представить с помощью словаря. Список оказался бы неэффективным преставлением, поскольку ключевое слово *Iehova* могло бы располагаться ближе к концу списка, а следовательно, для его обнаружения потребовалось бы немало времени. Намного лучше было бы воспользоваться сбалансированным бинарным деревом поиска, поскольку его высота была бы равна 18, гарантируя относительно быстрое отыскание ключевого слова. Но у нас нет такого дерева, а его построение — дело долгое. Как и в дереве поиска, содержащем буквы, нам придется ввести каждый элемент по отдельности, а для этого мы должны будем многократно проходить в дереве пути от его корня к листьям. Даже без подробного анализа данного процесса должно быть ясно, что для построения такого дерева потребуется время, большее линейного [6].

Тем не менее последовательность плиток можно довольно эффективно выявить (не больше, чем за 42 шага) без всякой дополнительной структуры данных. Как же такое возможно? Дело в том, что пол, уложенный плитками с буквами, сам по себе является структурой данных, особенно хорошо поддерживающей вид поиска, совершаемого Индианой Джонсом. Такая структура данных называется *префиксным деревом* (trie) и в какой-то степени подобна бинарному дереву поиска, хотя и заметно отличается от него [7].

Напомним, что, становясь на очередную плитку, Индиана Джонс сокращает область поиска в восемь раз. Нечто подобное происходит при перемещении вниз по бинарному дереву поиска, в котором область поиска сокращается вдвое, поскольку в результате выбора одной из двух ветвей дерева половина узлов исключается из поиска. Аналогично, если не выбрать плитку, область поиска сократится на одну восьмую, а если выбрать плитку — на одну восьмую ее прежнего размера. Ведь выбор одной плитки означает, что не выбраны семь остальных плиток, а следовательно, область поиска сокращается на семь восьмых.

Но здесь происходит нечто иное, что и отличает происходящее от принципа действия бинарного дерева поиска. В бинарном дереве поиска каждый элемент хранится в отдельном узле, тогда как в узле префиксного дерева хранятся только отдельные буквы, а слова представлены как пути, соединяющие разные буквы. Чтобы понять отличие бинарного дерева поиска от префиксного дерева, рассмотрим следующий пример. Допустим, что требуется представить ряд слов *bat* (палка), *bag* (сумка), *beg* (мольба), *bet* (ставка), *mag* (сплетня), *mat* (коврик), *meg* (баба) и *met* (металл). Сбалансированное бинарное дерево поиска, содержащее эти слова, выглядит так, как показано ниже.



Чтобы найти слово *bag* в этом дереве, мы сравниваем его со словом *bet* в корневом узле. Результат этого сравнения подсказывает нам, что надо продолжить поиск в левом поддереве. Для такого сравнения потребуются два шага, в течение которых сравниваются две первые буквы обоих слов. Сравним далее слово *bag* со словом *bat* в корневом узле левого поддерева. Результат этого сравнения снова подскажет нам продолжить поиск в левом поддереве, но при этом нам потребуются уже три шага сравнения, поскольку придется сравнивать все три буквы обоих слов. И, наконец, сравнение слова *bag* со словом в крайнем слева узле дерева приведет к успешному завершению поиска. Для этого последнего сравнения также потребуются три шага, а для всего поиска в целом — восемь шагов сравнения.

b	m
a	e
g	t

Но ту же самую коллекцию слов можно представить как пол, уложенный плитками 2×3, каждый ряд которого содержит плитки с буквами на соответствующих местах в любом из слов. Так, если каждое слово начинается с буквы *b* или *m*, то в первом ряду должны находиться две плитки с этими буквами. Аналогично во втором ряду должны быть две плитки с буквами *a* и *e*, а в третьем ряду — плитки с буквами *g* и *t*.

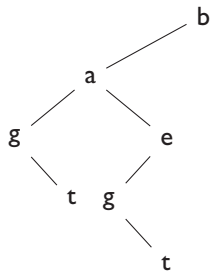
Поиск слова на плиточном полу осуществляется путем систематического обхода плиточного пола ряд за рядом. Для поиска каждой буквы слова обход соответствующего ряда плиток осуществляется слева направо до тех пор, пока буква

не будет найдена. Например, чтобы найти слово *bag* на таком плиточном полу, начнем поиск первой буквы *b* в первом ряду. Нужная плитка обнаруживается за один шаг. Далее мы ищем вторую букву *a* во втором ряду, для чего также требуется один шаг. И наконец, мы можем успешно завершить поиск, найдя букву *g* в третьем ряду, для чего также требуется лишь один шаг. А в целом для данного поиска требуются только три шага сравнения.

Для поиска на плиточном полу требуется меньше шагов, чем в бинарном дереве поиска, поскольку каждую букву приходится сравнивать лишь один раз, тогда как поиск в двоичном дереве приводит к повторяющимся сравнениям первоначальных частей слова. Слово *bag* представляет наиболее благоприятный вариант поиска на плиточном полу, поскольку каждую его букву можно найти на первой плитке. А для поиска слова *met* потребуется шесть шагов, поскольку каждая его буква находится на последней плитке в ряду. Но хуже этого варианта поиска быть не может, поскольку каждую плитку приходится проверять хотя бы один раз. (Для сравнения, чтобы найти слово *met* в бинарном дереве поиска, потребуется 2+3+3+3 = 11 шагов.) Наилучшим вариантом поиска в двоичном дереве является слово *bet*, поскольку для него требуются только три шага сравнения. Тем не менее по мере роста расстояния от корня дерева бинарный поиск приводит к постепенному увеличению шагов сравнения. А поскольку большинство слов располагаются ближе к листьям дерева, т.е. на большем расстоянии от его корня [8], в общем случае сравнение начальных частей слова приходится повторять многократно. Это означает, что, как правило, поиск в префиксном дереве осуществляется быстрее, чем в бинарном дереве поиска.

Аналогия с плиточным полом предполагает, что префиксное дерево может быть представлено в виде таблицы, хотя это верно не всегда. Рассматриваемый здесь пример достигает своей цели только потому, что все слова имеют одинаковую длину и на каждой позиции в слове могут находиться одни и те же буквы. Допустим, однако, что требуется также представить слова *bit* (кусочек) и *big* (большой). В этом случае во втором ряду потребуется дополнительная плитка для буквы *i*, что уже нарушает прямоугольную форму пола. Дополнение данного примера этими двумя словами демонстрирует еще одну закономерность, которая в общем случае не проявляется. Примеры слов выбраны таким образом, чтобы у разных префиксов одинаковой длины были одни и те же возможные суффиксы, что допускает совместное использование информации. Например, оба префикса, *ba* и *be*, могут быть дополнены суффиксом *g* или *t*, а это означает, что возможное продолжение обоих префиксов может быть описано одним рядом плиток с буквами *g* и *t*. Но этого больше нельзя утверждать, если добавить слова *bit* и *big*. Если начало слова с буквы *b* можно продолжить тремя возможными буквами, а именно: *a*, *e* и *i*, то начало слова с буквы *m* — только буквами *a* и *e*. Это означает, что в последнем случае потребуются два разных продолжения.

Таким образом, префиксное дерево, как правило, представлено особого рода двоичным деревом, в узлах которого находятся одиночные буквы, левые поддеревья представляют продолжения слов, а правые поддеревья — альтернативные варианты букв (плюс их продолжения). Например, префиксное дерево для хранения слов *bat*, *bag*, *beg* и *bet* выглядит так, как показано ниже.



У корневого узла дерева отсутствует правое ребро, и поэтому все слова начинаются с буквы *b*. А левое ребро, направленное от корневого узла *b* к поддереву с корневым узлом *a*, приводит к возможным продолжениям, которые начинаются с буквы *a* или буквы *e*, которая служит возможной альтернативой букве *a* и обозначается правым от нее ребром. Возможность продолжить префиксы *ba* и *be* буквой *g* или *t* представлена левым ребром, направленным от буквы *a* к поддереву с корневым узлом *g*, справа от которого находится его дочерний узел *t*. А то, что левые поддеревья узлов *a* и *e* одинаковы, означает, что они могут быть использованы совместно. Именно это обстоятельство используется в плиточном поле, представляющем рассматриваемое здесь дерево в виде единственного ряда плиток. Благодаря такому совместному использованию узлов в префиксном дереве обычно требуется хранить меньше информации, чем в бинарном дереве поиска.

Если в бинарном дереве поиска каждый ключ полностью содержится в отдельном узле, то в префиксном дереве хранящиеся ключи распределяются по всем его узлам. Любая последовательность узлов, исходящих из корня префиксного дерева, служит префиксом какого-нибудь ключа. Так, если речь идет о плиточном поле, то выбранные плитки служат префиксом для конечного пути. Именно поэтому префиксное дерево так и называется. Структура данных, представляющая в виде префиксного дерева плиточный пол, представший перед Индианой Джонсом, показана на рис. 5.2.

Как и у бинарных деревьев поиска и списков, у префиксных деревьев имеются свои преимущества и недостатки. В частности, они пригодны для хранения только тех ключей, которые могут быть разложены на последовательности составляющих элементов. Как в истории Индианы Джонса, которому так и не удалось получить Священный Грааль, так и у структур данных не существует своего Священного Грааля. Выбор наиболее подходящей структуры данных всегда зависит от подробностей ее применения.

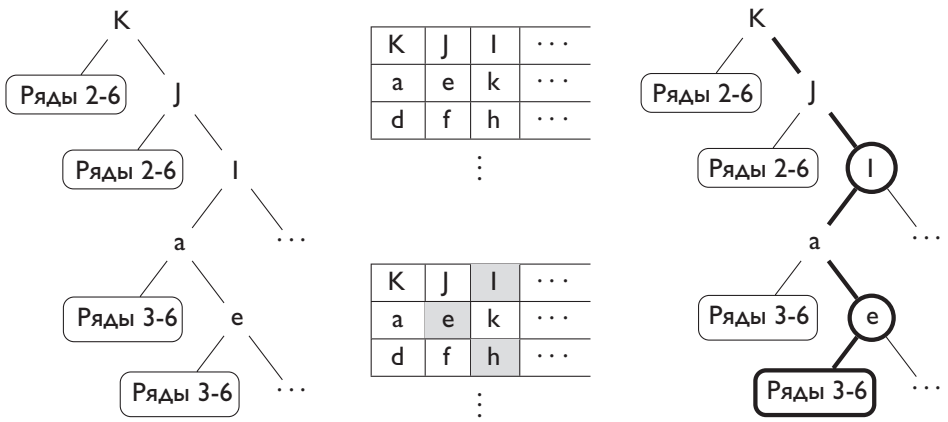


Рис. 5.2. Структура данных префиксного дерева и порядок поиска в ней. Слева: представление префиксного дерева, левые поддеревья которого обозначают (закругленными прямоугольниками) возможные продолжения слова от буквы в родительском узле, а правые поддеревья — их альтернативные варианты. Посредине сверху: представление левой части префиксного дерева в виде плиточного пола, общие поддеревья которого разделяют единые ряды плиток. Посредине внизу: три выбранные плиты, обозначающие начальные буквы слова Iehova. Справа: путь по префиксному дереву, помеченный выделенными полужирным ребрами, ведущими к выбранным плиткам, которые обозначены очерченными кругом узлами

Наблюдая упоминаемую здесь сцену в фильме *Индиана Джонс и последний крестовый поход*, мы понимаем трудность задачи поиска пути по ключевому слову или точного перемещения по целевым плиткам, но едва ли обращаем внимание на распознавание плиток, соответствующих буквам ключевого слова. Ведь это кажется нам настолько очевидным, что мы об этом вообще не думаем. И это обстоятельство еще раз показывает, насколько естественным является для нас выполнение эффективных алгоритмов. Как и в историях Гензеля и Гретель и Шерлока Холмса, спасение Индианы Джонса в его приключениях основывается на базовых принципах вычислений.

И наконец, если вас все еще интересует, как никогда не терять ключи от своей автомашины, знайте, что сделать это совсем не трудно. Придя домой, всегда кладите ключи в одно и то же отведенное для них место. Это место и будет служить ключом для поиска ключей от автомашины. Хотя об этом вам, вероятно, уже давно известно.

Приведение дел в порядок

Если вы преподаватель, то значительная часть вашей работы уходит на проверку сочинений или проведение экзаменов для учащихся в ваших классах. Ваша задача состоит не только в том, чтобы прочитать сочинения отдельных учащихся и принять у них экзамены. Например, прежде чем ставить окончательные оценки, вам потребуется иметь какое-то представление об уже выставленных оценках и кривой их распределения. После выставления оценок от вас потребуется внести их в классный журнал и, наконец, вернуть учащимся их работы. В некоторых случаях каждую из этих задач можно решить более эффективно, выполнив ту или иную сортировку.

Рассмотрим задачу внесения оценок в классный журнал. Даже для решения столь простой задачи можно применить три разных алгоритма с разными показателями времени выполнения. Во-первых, можно просмотреть всю стопку экзаменационных работ и внести оценку за каждую из них в классный журнал. Каждая экзаменационная работа извлекается из стопки за константное время, но поиск фамилии учащегося в классном журнале осуществляется за логарифмическое время, при условии, что он отсортирован по фамилиям, а сам поиск носит бинарный характер. Таким образом, время выполнения задачи по вводу всех оценок в классный журнал приобретает линейно-логарифмический характер [1], что намного лучше, чем квадратичный, хотя и не так хорошо, как линейный.

Во-вторых, можно просматривать классный журнал по фамилиям учащихся, а потом искать экзаменационную работу очередного ученика в стопке. В этом случае перейти к фамилии следующего учащегося в классном журнале можно за константное время, но поиск работы конкретного учащегося в стопке требует просмотра в среднем половины стопки, т.е. квадратичного времени в среднем [2]. Пользоваться таким алгоритмом явно не следует.

В-третьих, можно сначала отсортировать экзаменационные работы по фамилиям, а затем просматривать классный журнал и параллельно — стопку экзаменационных работ. Поскольку стопка отсортирована, а классный журнал упорядочен, для внесения каждой оценки потребуется лишь константное время. Все вместе приводит к тому, что к линейному времени проверки и выставления оценок добавляется время сортировки списка работ, которую можно выполнить за

линейно-логарифмическое (*линарифмическое*[1]) время. В общем времени выполнения данного алгоритма преобладает линейно-логарифмическая составляющая времени, требующегося для сортировки. Таким образом, общее время выполнения этого алгоритма также принимает линейно-логарифмический характер. Получается, такой алгоритм ничуть не лучше первого, так зачем вообще идти на дополнительные хлопоты, связанные с сортировкой экзаменационных работ? Дело в том, что иногда возникают ситуации, когда сортировку можно выполнить быстрее, чем за линейно-логарифмическое время, и тогда третий алгоритм способен повысить производительность и оказаться наиболее эффективным.

Для решения второй задачи — построения кривой распределения экзаменационных оценок — потребуется создание графика распределения баллов, т.е. отображения набранных баллов на количество учащихся, получивших эти баллы. Эта задача подобна задаче Индианы Джонса по вычислению гистограммы букв, поскольку распределение оценок также является гистограммой. Для вычисления такой гистограммы с помощью списка потребуется квадратичное время, а с помощью бинарного дерева поиска — линейно-логарифмическое. Эту гистограмму можно также вычислить, отсортировав сначала стопку экзаменационных работ по баллам, а затем просто просмотрев отсортированную стопку и подсчитав, как часто повторяется та или иная оценка. Такой способ вполне пригоден, поскольку в отсортированном списке за всеми экзаменационными работами с одинаковыми баллами следуют экзаменационные работы с другими, но тоже одинаковыми баллами. Как и при внесении оценок в классный журнал, сортировка не увеличивает время выполнения алгоритма, но потенциально может сократить его. Но совсем другое дело, если диапазон оценок небольшой. В этом случае обновление массива оценок в баллах, вероятно, окажется самым быстрым способом.

И наконец, раздача экзаменационных работ классу может стать чрезвычайно медленным делом. Представьте, что вы стоите перед учащимся, лихорадочно пытаясь найти его экзаменационную работу в большой стопке. Эта задача отнимет немало времени, даже если воспользоваться для ее решения бинарным поиском. Повторение поиска экзаменационных работ большого числа учащихся чрезмерно замедлит весь процесс, даже если он будет ускоряться по мере уменьшения стопки. В качестве альтернативы можно отсортировать экзаменационные работы по местоположению учащихся в классе. Если места учащихся пронумерованы и имеется схема их расположения в классе (или же просто известно, где они сидят), то экзаменационные работы можно отсортировать по номерам мест. В таком случае возврат экзаменационных работ превращается в линейный алгоритм: как и при внесении оценок в классный журнал, можно обойти места со стопкой в руках и

[1] Термин, построенный из слов “линейный” и “логарифмический”, по аналогии с использующимся в англоязычной литературе “linearithmic”, полученным из слов “linear” и “logarithmic”. — *Примеч. ред.*

вручить каждую экзаменационную работу за один шаг. И даже если сортировка отнимет линейно-логарифмическое время, она того стоит, поскольку экономится драгоценное время урока. Это характерный пример *предварительного вычисления*, когда некоторые данные, требующиеся для алгоритма, вычисляются до выполнения этого алгоритма. Именно так поступает почтальон, прежде чем разносить почту, или вы сами, когда составляете свой список покупок в соответствии с расположением товаров на полках магазина.

Сортировка оказывается более распространенным явлением, чем может показаться на первый взгляд. Помимо сортировки наборов предметов, нам регулярно приходится решать задачи сортировки в зависимости от обстоятельств. Рассмотрим в качестве примера процесс одевания, когда вы быстро понимаете, что следует придерживаться определенного порядка, надевая трусы до штанов, а носки — до обуви. Сборка мебели, ремонт и техническое обслуживание машины, оформление документов — б*о*льшая часть этих видов деятельности требует выполнения некоторых действий в правильном порядке. Даже дошкольникам дают задания отсортировать картинки в логичный ряд событий в сказках.