РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ

С. М. Окулов

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

Лаборатория ЗНАНИЙ

РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ

С. М. Окулов

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

3-е издание, электронное

Москва
Лаборатория знаний
2020

УДК 519.85(023)
ББК 22.18
О-52

Серия основана в 2008 г.

Окулов С. М.

О-52 Абстрактные типы данных / С. М. Окулов. — 3-е изд., электрон. — М. : Лаборатория знаний, 2020. — 253 с. — (Развитие интеллекта школьников). — Систем. требования: Adobe Reader XI ; экран 10". — Загл. с титул. экрана. — Текст : электронный.

ISBN 978-5-00101-891-9

Абстракция, абстрагирование — одна из составляющих мыслительного процесса творческой личности. Для развития этого компонента мышления в процессе обучения информатике есть дополнительные возможности, так как знание абстрактных типов данных, умение оперировать ими — необходимый элемент профессиональной культуры специалиста, связанного с разработкой программных комплексов.

Для школьников, преподавателей информатики и студентов младших курсов университетов. Книга может быть использована при проведении факультативных занятий и при углубленном изучении информатики.

УДК 519.85(023)
ББК 22.18

Деривативное издание на основе печатного аналога: Абстрактные типы данных / С. М. Окулов. — М. : БИНОМ. Лаборатория знаний, 2009. — 250 с. : ил. — (Развитие интеллекта школьников). — ISBN 978-5-94774-869-7.

ISBN 978-5-00101-891-9

Оглавление

Сыновьям: Алексею, Михаилу
и Сергею Окуловым — посвящаю

Предисловие

Когда человек познает самого себя,
сфинкс засмеется.

Надпись на постаменте сфинкса

Выдвинем *утверждение* о том, что информатика обладает исключительным, присущим только этому предмету ресурсом по интеллектуальному развитию школьника.

Схема доказательства этого утверждения: 1) дадим неформальное определение информатики как школьного предмета; 2) выделим особенности учебной деятельности, присущие только предмету «Информатика»; 3) установим соответствие между деятельностью в информатике и интеллектуальным развитием — например, на базе основных положений теории интеллектуального развития Ж. Пиаже[1)].

1.

Во-первых, в школьном курсе информатики должны изучаться фундаментальные основы этого предмета[2)].

Во-вторых, между понятиями «информатика» и «computer science» мы практически ставим знак эквивалентности. Из этого вытекает, например, то, что в информатике не

1) Жан Пиаже (1896–1980) — психолог, чье влияние на развитие психологии XX века было определяющим. В генетической эпистемологии (как называют его теорию) отражено глубокое убеждение в том, что развитие интеллекта проходит определенные стадии и является результатом динамического взаимодействия ребенка и окружающей среды.

2) О фундаментальных основах информатики вы можете прочитать короткое эссе в предыдущей книге серии «Развитие интеллекта школьников»: *Окулов С. М., Лялин А. В.* «Ханойские башни». М.: БИНОМ. Лаборатория знаний, 2008.

следует изучать информационные процессы в целом (везде и всюду), а только те из них, которые присущи компьютерной науке.

Очевидно, каждому виду человеческой деятельности, каждой науке свойственны свои способы и методы выражения мысли, свои методы формализации и создания моделей, свой язык описания процессов. С внедрением же компьютера в какую-то предметную область задачи и проблемы этой предметной области должны быть представлены в системе понятий компьютерной науки. Однако обратное утверждение неверно.

В-третьих, синтезирующим видом деятельности в нашем понимании информатики является программирование[1)].

Таким образом, утверждение о том, *что информатика может изучаться на основе программирования, через программирование*, имеет право на существование. Не путем освоения информационных технологий как таковых (это уже задача, например, отдельного предмета «Прикладная информатика»), не путем разбора понятий «информационный процесс», «моделирование», «формализация» в целом, — а в ходе самостоятельной разработки некоего целого с использованием, на основе фундаментальных понятий предметной области. Ключевое положение здесь — *самостоятельная деятельность*[2)], которой присущи некие черты (назовем их, например, *A*, *B*, *C*).

2.

Программирование — это многогранная деятельность. Кратко охарактеризуем только один ее аспект. Программу можно рассматривать как *формализованную запись метода решения* некой сложной проблемы. Программа попадает и под понятие *модели*, ибо последняя — не что иное, как имитация структуры или функционирования объекта (например, коммивояжера, которому требуется определенным образом обойти заданную совокупность городов). Но главное даже не в этом, а в том, что программа есть *динамичес-*

[1)] Обоснование этого положения дано в монографии: *Окулов С. М.* Информатика: развитие интеллекта школьника. М.: БИНОМ. Лаборатория знаний, 2005.

[2)] Самостоятельная деятельность не исключает учителя. Она понимается по М. Мамардашвили: человек может что-то понять, узнать только сам, никому не дано пребывать в состоянии чужой мысли и заставить другого мыслить.

кая модель со всеми вытекающими из этого положения следствиями, принципиальным из которых является появление возможности исследовать эту модель. И именно в данной возможности состоит принципиальное отличие информатики от любого другого школьного предмета!

Что происходит при решении сложной проблемы (задачи), при разработке и исследовании динамической модели? Сложная задача разбивается на взаимосвязанные подзадачи. Последние, в свою очередь, опять разделяются на свои подзадачи и т. д. вплоть до самых низших уровней нашего понимания задачи.

Что мы при этом получаем? Во-первых, сложная задача (проблема) описывается некой иерархической структурой; во-вторых, определяются принципы ее декомпозиции; в-третьих, так как каждый уровень — это определенный *уровень абстрагирования*, создается инструментарий для описания абстракций. *Иерархия* — это ранжированная или упорядоченная система абстракций, расположение частей или элементов целого в порядке от высшего к низшему. В результате деятельности программиста создается иерархическая структура из абстракций (*А*). Но ведь «абстракция — это такие существенные характеристики исследуемого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа»[1]. Г. Буч (и не только он[2]) разделяет *абстракцию сущности объекта* — когда объект представляет собой модель существенных сторон предметной области, и *абстракцию поведения* — когда объект состоит из обобщенного множества операций, каждая из которых выполняет определенную функцию. Другими словами, сущность задачи (проблемы) и метода ее решения мы описываем (очевидно, на каком-то языке, а это уже и есть формализованная запись!) на уровне данных в программе, а ее поведение — действиями над данными.

Но как осуществляется движение от проблемы (задачи) к модели, к динамической модели? С помощью логических приемов мышления, основными из которых являются ана-

---

1) *Буч Г.* Объектно-ориентированное проектирование с примерами применения. М.: Конкорд, 1992.

2) Здесь можно сослаться на работы В. М. Глушкова, известного специалиста XX в. по кибернетике, и на работы родоначальника школьной информатики А. П. Ершова.

лиз, всегда сочетающийся с абстрагированием[1], и синтез! Но описывают ли они полностью этот процесс движения к результату? Вряд ли! Говорят, что этими дефинициями отражается рациональная сторона мышления — поэтому далее вводится понятие метода. *Метод* (в самом широком смысле слова) — это «путь к чему-либо», способ достижения определенной цели, совокупность приемов или операций практического или теоретического освоения действительности. Р. Декарт писал в своем философском учении: «Под методом же я разумею достоверные и легкие правила, строго соблюдая которые, человек никогда не примет ничего ложного за истинное» и сможет добывать новое знание — все, что он способен познать, — «без излишней траты умственных сил»[2]. Р. Декарт считал, что для реализации правил его рационалистического метода необходима *интуиция*, с помощью которой усматриваются первые начала (принципы), а затем с помощью дедукции получаются следствия из этих начал. Но нужна не только интуиция, но и *инсайт*, и *воображение*, — другими словами, все свойства (или качества) интеллекта, не укладывающиеся в его рациональную составляющую (B).

В чем отличие хорошей программы от плохой? Чем отличается добротно сделанная динамическая модель (даже миниатюрная) от «поделки»? В хорошей программе нет никаких излишеств ни на уровне абстракций сущности, ни на уровне абстракций поведения. Хорошая программа стремится (состоит) к возникновению и организации структур с простыми и четкими формами, к простым и устойчивым состояниям. То есть она функционирует «без излишней траты умственных сил»! И опять-таки выделим главное: в результате исследования (экспериментальной деятельности с программой) достигается соответствие между динамической моделью и объектом, для которого она строится (C).

---

[1] Имеются в виду все типы абстракции: абстракция отождествления, или обобщающая абстракция; абстракция аналитическая, или изолирующая; абстракция идеализирующая, или идеализация; абстракция актуальной бесконечности (отвлечение от принципиальной невозможности зафиксировать каждый элемент бесконечного множества, т. е. когда бесконечные множества рассматриваются как конечные); абстракция потенциальной осуществимости (предполагается, что может быть осуществлено любое, но конечное число операций в процессе деятельности).

[2] *Декарт Р.* Избранные произведения. М.: Госполитиздат, 1950.

3.

По Ж. Пиаже, интеллект, подобно всем биологическим функциям, является продуктом эволюционной адаптации, поэтому для его понимания следует изучать умственную деятельность от момента рождения, наблюдая за его развитием и изменением в процессе взросления.

Два основных принципа интеллектуального роста ребенка: организация (*A*) и адаптация (*B*). *Адаптация* (приспособление к условиям окружения) состоит из ассимиляции и аккомодации. *Организация* — это структуризация интеллекта. Наиболее простой уровень такой структуризации — *схема*, являющаяся мысленной репрезентацией некоторого действия (физического или мысленного), выполняемого над объектом. Для новорожденного сосание, хватание, смотрение — это схемы, т. е. его способы познания окружающего мира и воздействия на этот мир. С развитием интеллект ребенка структурируется, или организуется, со все возрастающей сложностью и степенью интеграции. Кроме того, как схемы действий, так и вновь образуемые структуры все более интериоризуются, т. е. начинают совершаться в его голове как быстрые, короткозамкнутые мысленные последовательности.

Еще один фундаментальный аспект теории Ж. Пиаже — *знание есть действие*: «Познание начинается с действия, а всякое действие повторяется или обобщается (генерализуется) через применение к новым объектам, порождая тем самым некоторую «схему»... Основная связь, лежащая в основе всякого знания, состоит не в простой «ассоциации» между объектами (поскольку это понятие отрицает активность субъекта), а в «ассимиляции» объектов по определенным схемам, которые присущи субъекту. Этот процесс является продолжением различных форм биологической ассимиляции, среди которых когнитивная ассимиляция представляет лишь частный случай и выступает как процесс функциональной интеграции. В свою очередь, когда объекты ассимилированы схемами действий, возникает необходимость приспособления («аккомодации» — (*C*)) к особенностям этих объектов, это приспособление (аккомодация) является результатом внешних воздействий, т. е. результатом опыта»[1)]. Знания об объекте определяют в

[1)] *Пиаже Ж.* Психогенез знаний и его эпистемологическое значение // Семиотика: Антология / Сост. Ю. С. Степанов. М.: Академ-Проект, Екатеринбург: Деловая книга, 2001.

конечном счете те действия, которые вы над ним можете совершить.

Согласно Ж. Пиаже, в развитии интеллекта человека можно условно выделить четыре главных периода развития: *сенсомоторная стадия* (от рождения до 2 лет), *дооперациональная стадия* (от 2 до 7 лет); *стадия конкретных операций* (от 7 до 11 лет) и *стадия формальных операций* (от 11 лет и подростковый период). Последний период имеет наибольшее отношение к рассматриваемой нами проблеме. На предыдущей стадии ребенок ограничен координацией конкретных объектов в действительной ситуации. Он все еще не может координировать вероятностные события в гипотетической или более абстрактной формализованной ситуации, не может решать задачи без привязки к непосредственно воспринимаемой реальности. Главным же результатом освоения такой координации является то, что подросток может вызвать в уме системы операций, не присущие конкретной наблюдаемой ситуации, он способен координировать мысленные системы в системы более высокого порядка (*иерархия абстракций*). Наступает период умственной зрелости. Именно об этом периоде (стадии) формирования интеллекта идет речь: о периоде формирования и становления гипотетического, абстрактного мышления.

Рассмотрим особенности стадии формальных операций. Это — период формирования того, что называют *теоретическим мышлением*. Его логическими формами (способами отражения действительности посредством взаимосвязанных абстракций) являются *понятия*, *суждения* и *умозаключения*. Они отражают как бы результат логических действий ума при решении проблемы. *Проблема* — ситуация, в которой возникают задачи, связанные с интеллектуальной деятельностью; говоря философским языком, — *форма теоретического знания*. Ее содержанием является то, что еще не познано человеком, но что нужно познать (знание о незнании; вопрос, возникший в ходе познания и требующий ответа). Проблемная ситуация, согласно М. Вертгеймеру[1)], не является чем-то замкнутым в себе, поэтому она ведет нас к решению, к структурному завершению. Точ-

[1)] Макс Вертгеймер (1880–1943) — немецкий и американский психолог, один из основателей и главных теоретиков *гештальт-психологии*. «Гештальт» — это немецкий термин, не имеющий точного английского и русского аналога, но вошедший в научный язык. Этот термин используется для обозначения целого, полной структуры, множества, природа которого не обнаруживается с помощью простого анализа отдельных частей, его составляющих.

но так же решенная задача не должна быть завершенной «вещью в себе». Она снова может функционировать как часть, которая заставляет нас выходить за ее пределы, побуждает рассматривать и осмысливать более широкое поле.

Итак, у нас есть вход (проблема) и есть результат (понятия, суждения и умозаключения, или новый уровень организации интеллекта, по Ж. Пиаже). Запускается («принцип действия», по Ж. Пиаже) процесс мышления («процесс адаптации», по Ж. Пиаже) для достижения этого результата. Этот процесс — его логическая составляющая — состоит из таких основных приемов мышления, как *анализ*, *синтез*, *абстрагирование*, *сравнение*, *обобщение*. К этому классу понятий относятся также *индукция* (индуктивное обобщение) и *дедукция* (дедуктивный вывод). Но в этом процессе в обязательном порядке есть и нерациональная составляющая, без которой (ибо это процесс адаптации, а не ассоциации с уже познанным!) результат, как правило, недостижим.

А теперь, согласно ранее приведенной схеме доказательства утверждения, установим соответствие (которое было намечено в предыдущем тексте буквами A, B и C)[1)]. Мы конструируем программу как иерархию абстракций, порождаем некую схему — возможно ли это без структуризации интеллекта (A)? Ответ однозначен! Причем интеллект постоянно как бы тренируется в порождении схем. Мы используем методы, а, по Ж. Пиаже, это не что иное, как адаптация созданной схемы (иерархии абстракций) к исследуемому объекту (B). У нашей динамической модели нет согласования с объектом исследования? Начинается творческий процесс поиска причин несоответствия. При этом происходит или аккомодация, или новый виток генерации иерархии абстракций (C).

Афоризм, порожденный этой идеей: «Целое (т. е. гештальт) отличается от суммы его частей».

1) Вероятно, существует весьма ограниченное количество законов, управляющих мирозданием и, в частности, процессом познания человеком окружающего мира и себя. Люди говорят о своем, на своем языке, на языке своей предметной области. Но говорят ли они о разном?

Введение

> Инструменты, которые мы применяем, оказывают глубокое (и тонкое) влияние на наши способы мышления и, следовательно, на нашу способность мыслить.
>
> *Э. Дейкстра*

Структура данных — это не что иное, как способ взаимосвязи элементов данных. Другими словами, из элементов данных конструируется некое целое, с некоторыми свойствами, и это целое есть структура данных. Дальнейший шаг — в единое целое связываются как структуры данных, так и выполняемые над ними операции. Это новое целое носит в информатике название «*абстрактные типы данных*» (*abstract data type*), т. е. является множеством абстрактных объектов, представляющих неким образом организованные элементы данных, и определенного на этом множестве набора операций, которые могут быть выполнены над его элементами. Зачем нужна эта цепь абстрагирования? Она позволяет более рационально мыслить при решении проблем с использованием компьютера и в итоге получать результат «с наименьшей затратой умственных сил», — ибо уже не конкретное данное (число, символ и т. д.), а абстрактные типы данных становятся элементарными «кирпичиками» конструирования программного продукта[1)]. И эта книга — о таких элементарных кирпичиках!

Следует отметить, что данный основополагающий раздел информатики не представлен должным образом в рекомендациях по преподаванию информатики, разработанных

---

1) Например, если школьнику достаточно сказать при обсуждении задачи фразу вроде «требуется использовать очередь с приоритетом», и этого оказывается достаточно (продолжать не надо), то вызывает восхищение, с какой скоростью и с каким качеством реализует этот школьник такую очередь с приоритетом. Автор уже не способен «на такие подвиги»…

в ведущем вузе России — МГУ[1], хотя он есть в сокращенном варианте (правда, под названием «Фундаментальные структуры данных») в зарубежном аналоге[2] этой работы. В школьном же курсе информатики он если и затрагивается, то только косвенно.

Структура книги

В этой книге описываются абстрактные типы данных — матрицы, списки, стек, очередь, двоичные деревья, множества, очереди с приоритетом, сбалансированные деревья. Материал по каждому из этих типов данных разбит на разделы, и практически все они сопровождаются упражнениями разной степени сложности. Упражнения, отмеченные символом «*», — повышенной сложности.

Необходимые условия для работы с книгой

Начального курса информатики, проводимого на основе программирования[3], достаточно для понимания и свободного освоения материала данной книги. Для записи алгоритмов используется ограниченное подмножество возможностей (сведенное до минимума) языка программирования Паскаль. Это подмножество можно назвать «псевдокодом», — но «живым» псевдокодом, ибо при его воспроизведении в системе программирования, основанной на языке Паскаль, получается работоспособный код программы!

Преподавателю

Практически каждый раздел — это материал для одного занятия. Особенность текста книги — в том, что автор старался его сделать «под голос», под реальное изложение материала в аудитории (большинство материалов неоднократно проговаривалось при работе со школьниками и, в рамках предмета «Информатика», со студентами первого курса). Автор стремился достичь в тексте синтеза в единое целое слов, иллюстраций и фрагментов логики. Текст фрагментов кода здесь не есть некое приложение к словам, а является его необходимой и обязательной составляющей. (Сам автор при обсуждении материала книги использовал аудиторию с

---

1) Преподавание информатики и математических основ информатики для непрофильных специальностей классических университетов. М.: Интернет-ун-т информ. технологий, 2005.

2) Рекомендации по преподаванию информатики в университетах. Computer Curricula 2001: Computer Science: Пер. с англ. СПб., 2002. С. 162.

3) Например, на основе первых частей книги: *Окулов С. М.* Основы программирования. 2-е изд. М.: БИНОМ. Лаборатория знаний, 2005.

проекционным оборудованием. Доска применялась только для написания вспомогательного материала (экспромтов), возникающих при обсуждении. Текст занятия обычно предоставлялся слушателю.)

Школьникам и студентам младших курсов

Информатика как учебный предмет обладает удивительной особенностью. Можно прослушать тысячу лекций, при этом все понимать, и — не знать предмета! Нет, — на вербальном уровне вы, конечно, сможете что-то рассказать, что-то воспроизвести и получить очередную оценку. Но вы не сможете свободно выражать свои мысли на языке, понятном компьютеру, вы не сможете окунуться в мир отладки и поиска ошибок, в мир экспериментов, т. е. в ту атмосферу творчества, мучений, озарений, которая присуща деятельности при создании работающих программ, пусть даже и миниатюрных! А ведь только после этих самостоятельных мучений (а мучение, как и мысль, может быть только ваше!) приходит истинное знание предмета, когда все эти списки, очереди, деревья становятся элементарными конструкциями вашей мыслительной деятельности при решении проблем. Материал книги предоставляет возможность для такой самостоятельной деятельности, ибо, во-первых, он написан простым и доступным языком и, во-вторых, текст программных фрагментов хотя и работоспособен, но не является завершенным («набрал, запустил, получил»), и для получения результата вам необходимо будет еще приложить определенные усилия.

Благодарности

Хочу сказать огромное спасибо всем студентам и школьникам, проявившим терпение и выдержку при виде ошибок, которые автор допускал (не всегда по злому умыслу) при чтении этого курса, а также выносившим все экспромты автора и его заблуждения.

Глава 1

Матрицы

> Пожелав, чтобы все было хорошо, а худого, по возможности, ничего не было, бог каким-то образом все подлежащее зрению, что застал не в состоянии покоя, а в нестройном и беспорядочном движении, из беспорядка привел в порядок, полагая, что последний всячески лучше первого.
>
> *Платон*

1.1. Основные понятия

> Многие вещи нам непонятны не потому, что наши понятия слабы, но потому, что сии вещи не входят в круг наших понятий.
>
> *Козьма Прутков*

Матрица A размера $n \times m$ есть прямоугольная таблица чисел, содержащая n строк одинаковой длины (или m столбцов одинаковой длины). Матрица A записывается в виде

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}$$

или, сокращенно,

$$A = (a_{ij}),$$

где $i = 1, \dots, n$ — номер строки, а $j = 1, \dots, m$ — номер столбца. Матрица, у которой $n = m$, называется *квадратной*.

Элементы $a_{11}, a_{22}, \dots, a_{nn}$ образуют *главную диагональ* матрицы A.

Матрица, все элементы которой равны нулю, называется *нулевой*, а квадратная матрица, на главной диагонали которой стоят единицы, а все остальные элементы равны нулю, называется *единичной* (обозначим ее как I).

Представление матриц в памяти компьютера

Напомним уже известные вам факты. Память компьютера представляет собой последовательность ячеек, имеющих уникальные адреса, и возможен прямой доступ к содержимому ячейки памяти, если известен ее адрес. Способ вычисления адреса любого элемента одномерного массива определяется на основании *дескриптора*, который создается при описании этого массива и содержит данные о его физической структуре.

Пусть имеется описание

```
Var V: Array[i..k] Of Integer.
```

Тогда дескриптор для этого массива имеет структуру, показанную в табл. 1.1.

Таблица 1.1

Имя	V
Адрес начального элемента (физический адрес первого элемента массива)	*address*(*V*[*i*])
Индекс начального элемента	*i*
Индекс конечного элемента	*k*
Тип элемента	`Integer`
Длина элемента	*w* (=2)

Адрес элемента *V* с индексом *j* вычисляется по следующей формуле: *address*(*V*[*j*]) = *address*(*V*[*i*]) + (*j* – *i*)**w*.

Для представления матрицы в памяти компьютера обычно используется двумерный массив, каждый элемент которого идентифицируется парой индексов, где первый индекс определяет номер строки, а второй — номер столбца, на пересечении которых расположен заданный элемент.

Существует два способа отображения логической структуры массива в физическую: *отображение по строкам* и *отображение по столбцам*. В первом случае элементы массива располагаются в памяти последовательно строка за строкой, а во втором — столбец за столбцом. Какое именно представление используется — по строкам или столбцам, — зависит от конкретной системы программирования, так как оно однозначно определяет функцию вычисления адреса

элемента матрицы. При отображении по строкам адрес элемента массива с индексами (j, k) вычисляется как

$$address(A[j,k]) = address(A[1,1]) + w*(m*(j-1) + (k-1)),$$

а при отображении по столбцам — как

$$address(A[j,k]) = address(A[1,1]) + w*(n*(k-1) + (j-1)).$$

Пример:

Дана матрица целых чисел $A(20, 10)$. Мы описываем ее как двумерный массив:

```
Var V: Array[1..20,1..10] Of Integer,
```

считая, что $w = 2$ и $address(V[1,1]) = 400$. Тогда адрес элемента $V[13,5]$ при отображении по строкам равен

$$address(V[13,5]) = 400 + 2*(10*(13-1) + (5-1)) = 648.$$

Дескриптор массива V (`V:Array[`i_1`..`k_1`,`i_2`..`k_2`] Of Integer`) — а он обязательно создается — в этом случае может иметь вид, представленный в табл. 1.2.

Таблица 1.2

Имя	V
Адрес начального элемента (физический адрес первого элемента массива)	$address(V[i_1,k_1])=400$
Индекс начального элемента по строкам (i_1)	1
Индекс конечного элемента по строкам (k_1)	20
Индекс начального элемента по столбцам (i_2)	1
Индекс конечного элемента по столбцам (k_2)	10
Произведение $w \times m$ при отображении по строкам	20
Тип элемента	`Integer`
Длина элемента (w)	2

1.2. Операции над матрицами

> Одна знакомая попросила Альберта Эйнштейна позвонить ей по телефону, но предупредила, что ее телефон очень трудно запомнить: 24361.
>
> — И чего же тут трудного? — удивился Эйнштейн. — Две дюжины и 19 в квадрате.
>
> *Анекдот*

Сложение

Сумма двух матриц $A_{n\times m} = (a_{ij})$ и $B_{n\times m} = (b_{ij})$ есть матрица $C_{n\times m} = (c_{ij})$ такая, что $c_{ij} = a_{ij} + b_{ij}$ для $i = 1, \ldots, n$ и $j = 1, \ldots, m$.

Пример:

$$\begin{bmatrix} 2 & 5 & 3 \\ 4 & 6 & 9 \\ 7 & 8 & 13 \end{bmatrix} + \begin{bmatrix} 3 & 8 & 2 \\ 6 & 13 & 5 \\ 1 & 7 & 4 \end{bmatrix} = \begin{bmatrix} 5 & 13 & 5 \\ 10 & 19 & 14 \\ 8 & 15 & 17 \end{bmatrix}$$

Умножение на число

Произведение матрицы $A_{n\times m} = (a_{ij})$ на число k есть матрица $B_{n\times m} = (b_{ij})$ такая, что $b_{ij} = k \cdot a_{ij}$ для $i = 1, \ldots, n$ и $j = 1, \ldots, m$. **Вычитание матриц** тогда можно определить как $A - B = A + (B \cdot -1)$.

Произведение матриц

Операция умножения двух матриц выполняется, только если количество столбцов первой матрицы равно количеству строк второй. В этом случае произведение матрицы $A_{n\times m} = (a_{ij})$ на матрицу $B_{m\times p} = (b_{ij})$ есть матрица $C_{n\times p} = (c_{ij})$ такая, что $c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \ldots + a_{im} \cdot b_{mj} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj}$, где $i = 1, \ldots, n$, $j = 1, \ldots, p$. Другими словами, элемент i-й строки и j-го столбца матрицы C равен сумме произведений элементов i-й строки матрицы A на соответствующие элементы j-го столбца матрицы B.

Пример:

$$A=\begin{bmatrix}1 & 4\\ 2 & 3\end{bmatrix},\quad B=\begin{bmatrix}2 & 0 & 1\\ 5 & 6 & 3\end{bmatrix}$$

$$\Downarrow$$

$$A\cdot B=\begin{bmatrix}1\cdot 2+4\cdot 5 & 1\cdot 0+4\cdot 6 & 1\cdot 1+4\cdot 3\\ 2\cdot 2+3\cdot 5 & 2\cdot 0+3\cdot 6 & 2\cdot 1+3\cdot 3\end{bmatrix}=\begin{bmatrix}22 & 24 & 13\\ 19 & 18 & 11\end{bmatrix}$$

Вычисление произведения двух квадратных матриц A и B размера n выполняется с помощью следующего программного фрагмента (предполагаем, что первоначально элементы C равны нулю):

```
For i:=1 To n Do
 For j:=1 To n Do
  For k:=1 To n Do
   C[i,j]:=C[i,j]+A[i,k]*B[k,j];
```

При этом требуется n^3 умножений и $n^2(n-1)$ сложений, время работы программы — $O(n^3)$.

Метод В. Штрассена (V. Strassen, 1969)

Пусть A и B — две матрицы размера $(n \times n)$, где n — степень числа 2. Тогда матрицы A и B разбиваются каждая на четыре матрицы $((n/2) \times (n/2))$, и через них выражается произведение матриц C.

Пример:

$$\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}\cdot\begin{bmatrix}B_{11} & B_{12}\\ B_{21} & B_{22}\end{bmatrix}=\begin{bmatrix}C_{11} & C_{12}\\ C_{21} & C_{22}\end{bmatrix},$$

где $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$;
$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$;
$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$;
$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$.

Из вышеприведенных формул следует, что произведение A и B как двух матриц (2×2), элементами которых являются матрицы $((n/2) \times (n/2))$, можно выразить через суммы и произведения матриц $((n/2) \times (n/2))$. Допустим, что

для вычисления C_{ij} требуется m умножений и a сложений матриц ($(n/2) \times (n/2)$). Тогда, рекурсивно применяя эту схему, получим, что произведение двух матриц $(n \times n)$ можно вычислить за время $T(n)$, для которого справедливо неравенство

$$T(n) \leqslant m \cdot T\left(\frac{n}{2}\right) + \frac{a \cdot n^2}{4},\ n > 2.$$

Здесь первое слагаемое определяет время умножения m пар матриц ($(n/2) \times (n/2)$), а второе — сложность выполнения a сложений (вычитаний), каждое из которых требует времени $n^2/4$.

Известно, что при $m > 4$ решение неравенства ограничено сверху величиной $k \cdot n^{\log_2 m}$, где k — некоторая константа[1]. При обычном же умножении $m = 8$, $T(n) = O(n^3)$, и ничего нового нет.

Добавим также, что В. Штрассен изобрел искусный метод умножения двух матриц (2×2) за семь умножений и восемнадцать сложений (вычитаний)[2]:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix},$$

где

$$\begin{aligned}
m_1 &= (a_{12} - a_{22}) \cdot (b_{21} + b_{22}),\\
m_2 &= (a_{11} + a_{22}) \cdot (b_{11} + b_{22}),\\
m_3 &= (a_{11} - a_{21}) \cdot (b_{11} + b_{12}),\\
m_4 &= (a_{11} + a_{12}) \cdot b_{22},\\
m_5 &= a_{11} \cdot (b_{12} - b_{22}),\\
m_6 &= a_{22} \cdot (b_{21} - b_{11}),\\
m_7 &= (a_{21} + a_{22}) \cdot b_{11};\\
c_{11} &= m_1 + m_2 - m_4 + m_6,\\
c_{12} &= m_4 + m_5,\\
c_{21} &= m_6 = m_7,\\
c_{22} &= m_2 - m_3 + m_5 - m_7.
\end{aligned}$$

---

[1] Это следует из основной теоремы о рекуррентных соотношениях; см., например, книгу *Миллер Р., Боксер Л.* Последовательные и параллельные алгоритмы. Общий подход. М.: БИНОМ. Лаборатория знаний, 2006. С. 63–76.

[2] *Strassen V.* Gaussian Elimination is not Optimal // Numer. Math. 1969. Vol. 13. P. 354–356.

Если при умножении матриц порядка n (где n — степень числа 2) использовать результат Штрассена для умножения матриц порядка 2, то получим

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + 18 \cdot \left(\frac{n}{2}\right)^2 \text{ для } n \geqslant 2.$$

То есть $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$.

Рассмотрим особенности программной реализации метода умножения матриц.

Описание данных в этом случае очевидно:

```
Const nn=...;
{Указывается конкретный размер матриц, например 256}
Type TInt=Array[1..nn,1..nn] Of Integer;
Var A,B,C:TInt;
    n:Integer;
```

Сложение (вычитание) матриц размером $n \times n$, начиная с позиций $ka + 1$, $la + 1$ в матрице A и $kb + 1$, $lb + 1$ в матрице B, осуществляется с помощью процедуры `MatrixSumm`. Результат формируется в матрице C с позиции $kc + 1$, $lc + 1$. Значение логической переменной *IsSumm* при этом определяет тип операции — сложение или вычитание.

```
Procedure MatrixSumm(n:Integer; Const A:TInt;
ka,la:Integer; Const B:TInt; kb,lb:Integer;
  Var C:TInt; kc,lc:Integer; IsSumm:Boolean);
  Var i,j:Integer;
  Begin
    For i:=1 To n Do
      For j:=1 To n Do
        If IsSumm Then
          C[kc+i,lc+j]:=A[ka+i,la+j]+B[kb+i,lb+j]
        Else
          C[kc+i,lc+j]:=A[ka+i,la+j]-B[kb+i,lb+j];
  End;
```

В случае когда n не является степенью числа 2, обрабатываемая матрица вкладывается в матрицу, порядок которой равен наименьшей степени числа 2, большей n. Эта операция увеличивает порядок матрицы не более чем вдвое, а константу в оценке времени работы — не более чем в семь раз.

При реализации этого способа в основной процедуре определяется наименьшая степень двойки, большая n, и осуществляется вызов рекурсивной процедуры MatrixMult:

```
Procedure Solve;
  Var m:Integer;
  Begin
    m:=1;
    While m<n Do m:=m Shl 1;
    <Элементам матрицы C присвоить значение 0>;
    MatrixMult(m,A,0,0,B,0,0,C,0,0);
  End;
```

Здесь процедура MatrixMult — это не что иное, как прямая реализация формул В. Штрассена. Параметры *ka*, *la*, *kb*, *lb*, *kc* и *lc* при этом определяют позицию верхнего левого элемента соответствующей матрицы, подвергающейся разбивке на четыре части:

```
Procedure MatrixMult(n:Integer; Const A:TInt;
ka,la:Integer; Const B:TInt; kb,lb:Integer; Var
C:TInt; kc,lc:Integer);
  Var m1,m2,m3,m4,m5,m6,m7,tmp1,tmp2:TInt;
  Begin
    If (n=1) Then
             c[kc+1,lc+1]:=a[ka+1,la+1]*b[kb+1,lb+1]
             Else Begin
      n:=n Shr 1;
          {m1=(a12-a22)*(b21+b22)}
      MatrixSumm(n,A,ka,la+n,A,ka+n,la+n,tmp1,0,0,
                 False);
      MatrixSumm(n,B,kb+n,lb,B,kb+n,lb+n,tmp2,0,0,
                 True);
      MatrixMult(n,tmp1,0,0,tmp2,0,0,m1,0,0);
          {m2=(a11+a22)*(b11+b22)}
      MatrixSumm(n,A,ka,la,A,ka+n,la+n,tmp1,0,0,
                 True);
      MatrixSumm(n,B,kb,lb,B,kb+n,lb+n,tmp2,0,0,
                 True);
      MatrixMult(n,tmp1,0,0,tmp2,0,0,m2,0,0);
          {m3=(a11-a21)*(b11+b12)}
```

```
      MatrixSumm(n,A,ka,la,A,ka+n,la,tmp1,0,0,
                 False);
      MatrixSumm(n,B,kb,lb,B,kb,lb+n,tmp2,0,0,True);
      MatrixMult(n,tmp1,0,0,tmp2,0,0,m3,0,0);
          {m4=(a11+a12)*b22}
      MatrixSumm(n,A,ka,la,A,ka,la+n,tmp1,0,0,True);
      MatrixMult(n,tmp1,0,0,b,kb+n,lb+n,m4,0,0);
          {m5=a11*(b12-b22)}
      MatrixSumm(n,B,kb,lb+n,B,kb+n,lb+n,tmp2,0,0,
                 False);
      MatrixMult(n,A,ka,la,tmp2,0,0,m5,0,0);
          {m6=a22*(b21-b11)}
      MatrixSumm(n,B,kb+n,lb,B,kb,lb,tmp2,0,0,
                 False);
      MatrixMult(n,A,ka+n,la+n,tmp2,0,0,m6,0,0);
          {m7=(a21+a22)*b11}
      MatrixSumm(n,A,ka+n,la,A,ka+n,la+n,tmp1,0,0,
                 True);
      MatrixMult(n,tmp1,0,0,b,kb,lb,m7,0,0);
          {c11=m1+m2-m4+m6}
      MatrixSumm(n,m1,0,0,m2,0,0,C,kc,lc,True);
      MatrixSumm(n,C,kc,lc,m4,0,0,C,kc,lc,False);
      MatrixSumm(n,C,kc,lc,m6,0,0,C,kc,lc,True);
          {c12=m4+m5}
      MatrixSumm(n,m4,0,0,m5,0,0,C,kc,lc+n,True);
          {c21=m6+m7}
      MatrixSumm(n,m6,0,0,m7,0,0,C,kc+n,lc,True);
          {c22=m2-m3+m5-m7}
      MatrixSumm(n,m2,0,0,m3,0,0,C,kc+n,lc+n,
                 False);
      MatrixSumm(n,C,kc+n,lc+n,m5,0,0,C,kc+n,lc+n,
                 True);
      MatrixSumm(n,C,kc+n,lc+n,m7,0,0,C,kc+n,lc+n,
                 False);
    End;
  End;
```

Как нетрудно заметить, выигрыш по времени в алгоритме Штрассена достигается за счет уменьшения количества операций умножения при увеличении числа операций сложения (вычитания). Положительный эффект достигается

начиная с некоторого значения n (примерно начиная с размеров 45×45[1]), тогда как при меньших значениях n разумнее перемножать матрицы традиционным способом.

Упражнения

1. Вычислите произведение $\begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 7 \\ 8 & 5 \end{bmatrix}$ прямым методом и с помощью алгоритма Штрассена.

2. Убедитесь, что формулы Штрассена при умножении двух матриц 2×2 при их раскрытии дают тот же результат, что и прямое перемножение этих матриц.

3. Проверьте правильность следующей модификации алгоритма Штрассена для вычисления произведения двух матриц второго порядка за 7 умножений и 15 сложений[2]:

$$
\begin{array}{lll}
s_1 = a_{21} + a_{22}, & m_1 = s_2 \cdot s_6, & t_1 = m_1 + m_2, \\
s_2 = s_1 - a_{11}, & m_2 = a_{11} \cdot b_{11}, & t_2 = t_1 + m_4. \\
s_3 = a_{11} - a_{21}, & m_3 = a_{12} \cdot b_{21}, & \\
s_4 = a_{12} - s_2, & m_4 = s_3 \cdot s_7, & \\
s_5 = b_{12} - b_{11}, & m_5 = s_1 \cdot s_5, & \\
s_6 = b_{22} - s_5, & m_6 = s_4 \cdot b_{22}, & \\
s_7 = b_{22} - b_{12}, & m_7 = a_{22} \cdot s_8, & \\
s_8 = s_6 - b_{21}, & &
\end{array}
$$

Элементы произведения матриц при этом вычисляются так:

$$
\begin{aligned}
c_{11} &= m_2 + m_3, \\
c_{12} &= t_1 + m_5 + m_6, \\
c_{21} &= t_2 - m_7, \\
c_{22} &= t_2 + m_5.
\end{aligned}
$$

Покажите, что они равны значениям, вычисляемым при прямом умножении матриц.

4. Разработайте программу умножения матриц по алгоритму Штрассена.

---

[1] *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 683.

[2] *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. С. 280.

Примечание. При этом целесообразно сначала разработать вспомогательную программу для генерации матриц заданного порядка n.

5. Заполните табл. 1.3.

Таблица 1.3

Значение n	Время умножения матриц прямым методом	Время умножения матриц по алгоритму Штрассена
10		
20		
30		
40		
50		
60		
70		

6. Экспериментально найдите значение n, при котором разумнее (с точки зрения времени выполнения) использовать прямой метод перемножения матриц, а не делать это с помощью алгоритма Штрассена.

7*. *Математическое упражнение*. Выведите формулы Штрассена.

8. Пусть A, B и C — матрицы порядка n. Проверьте и докажите истинность следующих равенств:
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$;
$A \cdot (B + C) = A \cdot B + A \cdot C$;
$(A + B) \cdot C = A \cdot C + B \cdot C$.

9*. Рассмотрим произведение четырех матриц $A = A_1 \cdot A_2 \cdot A_3 \cdot A_4$. Матрицы имеют размеры: A_1 — 10×12, A_2 — 12×25, A_3 — 25×50, A_4 — 50×10. Обычный алгоритм умножения матриц размером $n \times m$ и $m \times t$ потребует $n \cdot m \cdot t$ умножений. При выполнении умножения в порядке $A = ((A_1 \cdot A_2) \cdot A_3) \cdot A_4$ требуется $10 \cdot 12 \cdot 25 + 10 \cdot 25 \cdot 50 + 10 \cdot 50 \cdot 10 = 5000$ операций, а при

умножении $A = A_1 \cdot (A_2 \cdot (A_3 \cdot A_4))$ – $10 \cdot 12 \cdot 10 + 12 \cdot 25 \cdot 10 + 25 \cdot 50 \cdot 10 = 1200$ операций. Таким образом, количество умножений зависит от порядка перемножения матриц. Обобщите эту задачу на случай n матриц и ответьте на вопрос: как определить порядок перемножения матриц, при котором выполняется минимальное количество операций умножения?

Указание. Для расчета минимального количества операций умножения следует найти элементы матрицы $Q[n \times n]$, вид которой показан на рис. 1.1. Элементы матрицы заполняются по диагоналям, параллельным главной, по принципу:

$$q[i,j] = \begin{cases} 0, & \text{если } i \geqslant j, \\ \min\limits_{i \leqslant k < j} (q[i,k] + q[k,j] + r_{i-1} \cdot r_k \cdot r_j), & \text{если } i < j, \end{cases}$$

где число r_{i-1} — количество строк матрицы A_i. Для рассмотренного нами примера числа r равны: $r_0 = 10$, $r_1 = 12$, $r_2 = 25$, $r_4 = 50$. Ответом в задаче является значение $q[1,n]$ — найденное минимальное количество операций умножения. Для определения порядка умножения матриц требуется хранить в дополнительной структуре данных для каждого значения $q[i,j]$ значение k, на котором достигается минимум.

Q	1	2	3	...	$n-2$	$n-1$	n
1	0	$q_{1,2}$	$q_{1,3}$	...	$q_{1,n-2}$	$q_{1,n-1}$	$q_{1,n}$
2	0	0	$q_{2,3}$	...	$q_{2,n-2}$	$q_{2,n-1}$	$q_{2,n}$
3	0	0	0	...	$q_{3,n-2}$	$q_{3,n-1}$	$q_{3,n}$
...	...	...	...	...	...	...	...
$n-2$	0	0	0	0	0	$q_{n-2,n-1}$	$q_{n-2,n}$
$n-1$	0	0	0	0	0	0	$q_{n-1,n}$
N	0	0	0	0	0	0	0

Рис. 1.1. Вид матрицы Q

1.3. Элементарные преобразования матриц

Изучите азы науки, прежде чем взойти на ее вершины.

И. П. Павлов

Пусть A и $\overline{A}$ — произвольные матрицы одинакового порядка $n \times m$. Обозначим через $a_1, \ldots, a_k, \ldots, a_l, \ldots, a_n$ последовательные строки матрицы A. Будем говорить, что матрица $\overline{A}$ получена из матрицы A:

- перестановкой двух строк, если $a_1, \ldots, a_l, \ldots, a_k, \ldots, a_n$ — последовательные строки матрицы $\overline{A}$;
- умножением строки на не равное нулю число t, если $a_1, \ldots, t \cdot a_k, \ldots, a_l, \ldots, a_n$ — последовательные строки матрицы $\overline{A}$;
- прибавлением к строке матрицы A другой ее строки, умноженной на число t, если $a_1, \ldots, a_k, \ldots, a_l + t \cdot a_k, \ldots, a_n$ — последовательные строки матрицы $\overline{A}$.

Здесь мы перечислили элементарные преобразования строк матрицы A. Аналогичные преобразования можно определить и для столбцов этой матрицы.

Пример. Изменение матрицы в процессе выполнения элементарных преобразований:

$$\begin{bmatrix} 2 & 3 & 1 \\ 0 & 2 & -1 \\ 4 & 0 & 5 \end{bmatrix} \overset{(1)}{\Rightarrow} \begin{bmatrix} 1 & 3 & 2 \\ -1 & 2 & 0 \\ 5 & 0 & 4 \end{bmatrix} \overset{\substack{(2)\\(3)}}{\Rightarrow} \begin{bmatrix} 1 & 3 & 2 \\ 0 & 5 & 2 \\ 0 & -15 & -6 \end{bmatrix} \overset{(4)}{\Rightarrow} \begin{bmatrix} 1 & 0 & 2 \\ 0 & 5 & 2 \\ 0 & -15 & -6 \end{bmatrix} \overset{\substack{(5)\\\Rightarrow\\(6)}}{} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 2 \\ 0 & 0 & 0 \end{bmatrix} \overset{(7)}{\Rightarrow} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix} \overset{(8)}{\Rightarrow} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Выполняемые при этом преобразования:

(1) — перестановка 1-го и 3-го столбцов;
(2) — прибавление 1-й строки ко 2-й строке;

(3) — прибавление к 3-й строке 1-й строки, умноженной на число –5;
(4) — прибавление ко 2-му столбцу 1-го столбца, умноженного на число –3;
(5) — прибавление к 3-му столбцу 1-го столбца, умноженного на число –2;
(6) — прибавление к 3-й строке 2-й строки, умноженной на число 3;
(7) — умножение 2-го столбца на 1/5;
(8) — прибавление к 3-му столбцу 2-го столбца, умноженного на число –2.

Оказывается, что любая матрица с помощью элементарных преобразований приводится к матрице, у которой в начале главной диагонали записано какое-то количество единиц, а все остальные элементы равны нулю. Такие матрицы называются *каноническими*. Если же квадратная матрица A сведена к канонической и оказалось, что на всей ее главной диагонали записаны единицы, то такая матрица A считается *невырожденной*.

Определим матрицы трех типов.

Первый тип матрицы получается из единичной путем перестановки i-й и j-й строк. Все остальные элементы этой матрицы равны нулю.

$$P_{ij} = \begin{bmatrix} 1 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 1 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ 0 & \dots & 0 & 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ 0 & \dots & 0 & 0 & 1 & \dots & 0 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & 0 & \dots & 1 & 0 & 0 & \dots & 0 \\ 0 & \dots & 0 & 1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

Второй тип матрицы получается из единичной при замене некоторого (i-го) элемента на главной диагонали (единицы) на число t.

$$D_i = \begin{bmatrix} 1 & \dots & 0 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 1 & 0 & 0 & \dots & 0 \\ 0 & \dots & 0 & t & 0 & \dots & 0 \\ 0 & \dots & 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

Третий тип матриц получается из единичной добавлением лишь одного элемента, не равного нулю (t), не на главной диагонали матрицы:

$$L_{ij} = \begin{bmatrix} 1 & \dots & 0 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 1 & \dots & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & t & \dots & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

или

$$R_{ij} = \begin{bmatrix} 1 & \dots & 0 & 0 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 1 & \dots & t & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & 0 & \dots & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

В первом случае элемент добавлен ниже главной диагонали на пересечении i-й строки и j-го столбца, а во втором — выше главной диагонали.

Элементарные преобразования произвольной матрицы A равносильны ее умножению на соответствующие матрицы. В частности:

- перестановка строк i и $j \Leftrightarrow P_{ij} \cdot A$;
- перестановка столбцов i и $j \Leftrightarrow A \cdot P_{ij}$;

- умножение строки i на число $t \Leftrightarrow D_i \cdot A$;
- умножение столбца i на число $t \Leftrightarrow A \cdot D_i$;
- прибавление к j-й строке i-й строки, умноженной на $t \Leftrightarrow L_{ij} \cdot A$;
- прибавление к j-му столбцу i-го столбца, умноженного на $t \Leftrightarrow A \cdot R_{ij}$.

Определим еще три типа матриц.

Матрица перестановки P — это квадратная матрица, имеющая ровно одну единицу в каждой строке и в каждом столбце, тогда как все остальные ее элементы равны нулю.

Верхнетреугольная матрица U — это квадратная матрица, у которой все элементы под главной диагональю равны нулю.

Нижнетреугольная матрица L — это квадратная матрица, у которой все элементы над главной диагональю равны нулю.

Известно, что для всякой невырожденной квадратной матрицы A можно построить ее разложение по трем типам введенных нами выше матриц: $P \cdot A = L \cdot U$. Рассмотрим эту задачу; сначала — ее частный случай, когда $P = I$.

Пример. На рис. 1.2 показан процесс преобразования матрицы A.

После выполненных действий

$$A = \begin{bmatrix} 2 & 3 & 5 & 4 \\ 4 & 8 & 10 & 12 \\ 8 & 14 & 16 & 18 \\ 6 & 20 & 22 & 24 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 4 & 1 & 1 & 0 \\ 3 & 5.5 & -1.75 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 3 & 5 & 4 \\ 0 & 2 & 0 & 4 \\ 0 & 0 & -4 & -2 \\ 0 & 0 & 0 & -13.5 \end{bmatrix},$$

т. е. найдено разложение матрицы A типа $A = L \cdot U$.

В чем же суть выполненных преобразований? Пусть A есть квадратная матрица $n \times n$. Представляем ее в виде

$$A = \begin{bmatrix} a_{11} & w \\ v & A' \end{bmatrix},$$

где v — матрица с размерностью $(n-1) \times 1$ (или, другими словами, — $(n-1)$-компонентный вектор), w — матрица с размерностью $1 \times (n-1)$, A' — матрица с размерностью

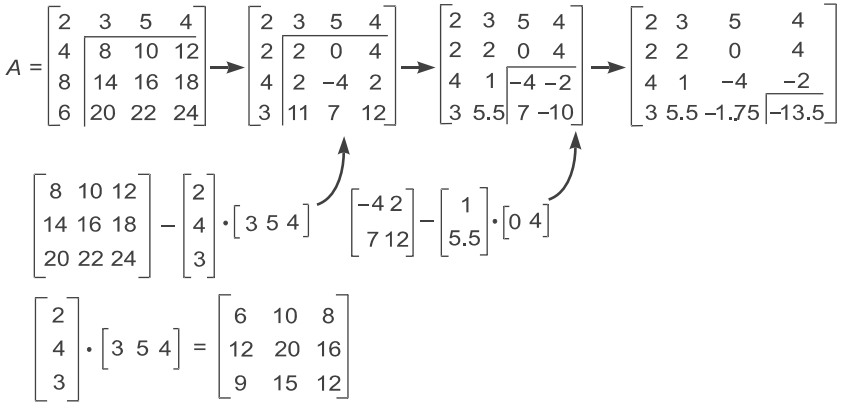


Рис. 1.2. Процесс преобразования матрицы A

$(n-1) \times (n-1)$. А затем выполняем шаги преобразований: элементы вектора v делим на a_{11}; находим произведение матриц $(v/a_{11}) \cdot w$ — получаем матрицу с размерностью $(n-1) \times (n-1)$; вычитаем из A' полученную матрицу. Далее процесс преобразований продолжается, но уже с вновь полученной матрицей A' (см. рис. 1.2). Полученную в результате матрицу можно представить в виде нижнетреугольной и верхнетреугольной, произведение которых равно исходной матрице A.

Для чего мы все это делаем? Мы сводим матрицу A к верхнетреугольной матрице U и при этом данные о преобразованиях фиксируем в матрице L.

Пусть требуется сделать нулевыми все элементы первого столбца A, кроме a_{11} (предполагаем, что $a_{11} \neq 0$). Для этого достаточно выполнить элементарные действия: элементы первой строки умножить на число a_{i1}/a_{11}, а затем полученную строку вычесть из строки i матрицы A. Если интегрировать все эти действия по обнулению элементов первого столбца в единое целое, то мы получим первый шаг преобразования, представленный на рис. 1.2. Затем в полученной матрице A' размерностью $(n-1) \times (n-1)$ мы аналогичным образом обнуляем элементы ее первого столбца, опять же в предположении, что элемент a_{11} не равен нулю. Это будет вторым шагом преобразований. И так до тех пор, пока матрица не будет преобразована или не окажется, что этого сделать нельзя.

Вышеприведенная схема преобразования матрицы A работоспособна только в случае, если $a_{11} \neq 0$ и все последующие элементы в левом верхнем углу матриц A' тоже не равны нулю. Эти элементы называют *ведущими*, а перестановочная матрица P в разложении $P \cdot A = L \cdot U$ предотвращает выбор нулевого элемента в качестве ведущего (если, конечно, такая возможность есть).

Пример. Пусть дана матрица

$$A = \begin{bmatrix} 3 & 9.5 & 5.75 & 7.375 \\ 6 & 3 & 5.5 & 6.75 \\ 8 & 4 & 2 & 1 \\ 4 & 6 & 4 & 2.5 \end{bmatrix}$$

На начальном этапе матрица P равна единичной матрице. Находим максимальный элемент в первом столбце. Он располагается в третьей строке и равен 8. Переставляем строки 1 и 3, отражая это изменение в матрице P перестановкой этих же строк. Затем выполняем первый шаг преобразований по нахождению разложения A. В матрице A' на месте a_{22} оказался нуль. Находим максимальный элемент во втором столбце ниже главной диагонали (третья строка) и переставляем вторую и третью строки, фиксируя изменения в P. Выполняем второй шаг преобразований, и так — до его очевидного завершения. Процесс преобразований нельзя продолжить, если все элементы очередного столбца ниже главной диагонали равны нулю, т. е. матрица A — вырожденная. Для рассматриваемого примера такой процесс имеет вид:

$$\begin{bmatrix} 3 & 9.5 & 5.75 & 7.375 \\ 6 & 3 & 5.5 & 6.75 \\ 8 & 4 & 2 & 1 \\ 4 & 6 & 4 & 2.5 \end{bmatrix} \Rightarrow \begin{bmatrix} 8 & 4 & 2 & 1 \\ 6 & 3 & 5.5 & 6.75 \\ 4 & 9.5 & 5.75 & 7.35 \\ 3 & 6 & 2.5 & 2.5 \end{bmatrix} \Rightarrow$$

$$\Rightarrow \begin{bmatrix} 8 & 4 & 2 & 1 \\ 0.75 & 0 & 4 & 6 \\ 0.375 & 8 & 5 & 7 \\ 0.5 & 4 & 3 & 2 \end{bmatrix} \Rightarrow \begin{bmatrix} 8 & 4 & 2 & 1 \\ 0.375 & 8 & 5 & 7 \\ 0.75 & 0 & 4 & 6 \\ 0.5 & 4 & 3 & 2 \end{bmatrix} \Rightarrow$$

$$\Rightarrow\begin{bmatrix} 8 & 4 & 2 & 1 \\ 0.375 & 8 & 5 & 7 \\ 0.75 & 0 & 4 & 6 \\ 0.5 & 0.5 & 0.5 & -1.5 \end{bmatrix}\Rightarrow\begin{bmatrix} 8 & 4 & 2 & 1 \\ 0.375 & 8 & 5 & 7 \\ 0.75 & 0 & 4 & 6 \\ 0.5 & 0.5 & 0.125 & -2.25 \end{bmatrix}$$

Матрица P и ее изменение на первом и втором шагах преобразования:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\Rightarrow\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\Rightarrow\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Получим:

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\cdot\begin{bmatrix} 3 & 9.5 & 5.75 & 7.375 \\ 6 & 3 & 5.5 & 6.75 \\ 8 & 4 & 2 & 1 \\ 4 & 6 & 4 & 2.5 \end{bmatrix}=$$

$$=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.375 & 1 & 0 & 0 \\ 0.75 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0.125 & 1 \end{bmatrix}\cdot\begin{bmatrix} 8 & 4 & 2 & 1 \\ 0 & 8 & 5 & 7 \\ 0 & 0 & 4 & 6 \\ 0 & 0 & 0 & -2.25 \end{bmatrix}$$

При программной реализации матрицу P можно заменить вектором, инициализируя его при вводе исходных данных следующим образом:

```
For i:=1 To n Do P[i]:=i;
```

Рекурсивный вариант реализации разложения невырожденной квадратной матрицы A по трем типам введенных матриц P, L и U тогда имеет вид:

```
Procedure Solve(k:Integer);
{k - номер столбца; для матриц L и U память не
выделяется - они хранятся в матрице A}
```

```
Var i,j,imax:Integer;
    q2:Tmas;
    {Type Tmas = Array[1..nn] Of Real;
     nn - константа}
    max,q1:Real;
Begin
  If (k>n) Then print {Вывод результата}
  Else Begin
    max:=0;
    For i:=k To n Do
    {Поиск максимального элемента}
      If (Abs(A[i,k])>max) Then Begin
        max:=Abs(A[i,k]);
        imax:=i;
      End;
    If max=0 Then Exit;
    {Преобразование выполнить нельзя - матрица A
    вырожденная}
    q1:=p[k];p[k]:=p[imax];p[imax]:=q1;
    q2:=A[k];A[k]:=A[imax];A[imax]:=q2;
    {Переставляем строки}
    For i:=k+1 To n Do Begin
      A[i,k]:=A[i,k]/A[k,k];
      For j:=k+1 To n Do
        A[i,j]:=A[i,j]-A[i,k]*A[k,j];
    End;
    Solve(k+1);
   {Переход к следующему столбцу}
  End;
End;
```

Упражнения

1. Дана матрица

$$A=\begin{bmatrix}3 & -1 & 3 & 2\\ 5 & -3 & 2 & 3\\ 1 & -3 & -5 & 0\\ 7 & -5 & 1 & 4\end{bmatrix}$$

Выполняя последовательности элементарных преобразований, выясните, вырождена она или нет.

2. Разработайте алгоритм и напишите программу преобразования матрицы A к виду, когда только в верхней части главной диагонали (или на всей главной диагонали) записаны единицы. Элементарные преобразования при этом следует представить в матричном виде и найти произведение заданных матриц.

3. Найдите разложение вида $A = L \cdot U$ для матрицы

$$A = \begin{bmatrix} 3 & 5 & 7 & 8 \\ -1 & 7 & 0 & 1 \\ 0 & 5 & 3 & 2 \\ 1 & -7 & 7 & 4 \end{bmatrix}$$

4. Найдите разложение вида $P \cdot A = L \cdot U$ для матрицы из упражнения 1.

5. Разработайте нерекурсивный вариант реализации поиска разложения $P \cdot A = L \cdot U$.

Методические комментарии

В этой главе автором не ставилась задача изучения проблематики численных методов и, в частности, вопросов, связанных с линейной алгеброй. Целью же здесь является начальное знакомство с матрицей как с абстрактной структурой данных и рассмотрение в связи с этим содержательных задач, имеющих «продолжение», а не традиционных задач типа «найти максимальный элемент в заданном двумерном массиве». Численные методы и линейная алгебра — это отдельные курсы в системе подготовки специалистов по информатике, и по ним существует обширная учебная литература, приводить ссылки на которую автор воздержится (поскольку, как уже было сказано, рассмотрение этих тем не являлось задачей при написании данной главы).

Метод Штрассена (перенесемся мысленно в 1969 год) в очередной раз показывает, что, казалось бы, даже очевидные вещи не относятся к разряду абсолютно познанных. Их требуется исследовать, а результат, если он будет получен, может иметь для развития информатики такое же огромное значение, как и метод Штрассена.

На элементарных преобразованиях матриц основаны все построения соответствующего раздела теории численных методов, поэтому их понимание и уверенное использование — это гарантия успешности освоения курса.

Глава 2

Списки

> Все сплетено друг с другом… и едва ли найдется что-нибудь чуждое всему остальному. Ибо все объединено общим порядком и служит к украшению одного и того же мира.
>
> *Сенека*

2.1. Основные понятия о ссылочном типе данных (указателях)

> Пушка… Они заряжают пушку…
> Зачем? А! Они будут стрелять!
>
> *Из м/ф «Остров сокровищ»*

Обычные типы данных являются *статическими*. Область памяти для их размещения выделяется на стадии компиляции, и ее перераспределение на стадии выполнения программы не допускается.

Выделение же памяти для переменных уже на стадии выполнения программы возможно с использованием нового типа данных — *указателей* (*ссылок*). Значением указателя (переменной ссылочного типа) является *адрес области памяти* (первой ячейки) переменной заданного базового типа. Таким образом, здесь задается не значение переменной (величины), а адрес памяти, в которой находится переменная (*принцип косвенной адресации*). При этом для указателей область памяти выделяется статически (как обычно), а для переменных, на которые они указывают, — *динамически*, т. е. на стадии выполнения программы (поэтому они и называются *динамическими*). Для хранения динамических переменных выделяется специальная область памяти, называемая «*кучей*».

Работая с указателями, мы работаем с *адресами величин*, а не с их *именами*.

Структуры данных, сконструированные с использованием указателей, часто называют динамическими (по прин-

ципу их размещения в памяти). Отметим ряд преимуществ и недостатков динамических структур. При этом, будучи программистами, оценим их с точки зрения конкретной решаемой задачи.

Преимущества:

- разумное использование динамических структур данных приводит к сокращению объема памяти, необходимого для работы программы;
- динамические данные, в противоположность статическим и автоматически размещаемым данным, не требуют их объявления как данных фиксированного размера, причем в большинстве систем программирования для «кучи» выделяется достаточно большой объем памяти;
- целый ряд алгоритмов более эффективен при их реализации именно с использованием динамических структур. Например, вставка элемента на определенное место в массиве требует перемещения части других элементов массива, но при использовании динамических структур данных для вставки элемента в середину списка достаточно нескольких операторов присваивания.

Недостатки:

- алгоритмы на динамических структурах обычно более сложны, трудны для отладки по сравнению с аналогичными алгоритмами на статических данных;
- использование динамических структур данных требует дополнительных затрат памяти для хранения ссылок (так что в некоторых задачах объем памяти, отводимой для ссылок, превосходит объем памяти, выделяемой непосредственно для данных);
- существуют алгоритмы, реализация которых более эффективна именно на обычных данных. Например, в ряде задач индекс элемента в массиве можно просто вычислять, в то время как использование списковых структур требует обхода списка.

Однако вернемся к основной теме этого раздела.

Для объявления указателей (переменных ссылочного типа) используется специальный символ «^», после которого указывается тип динамической (базовой) переменной:

```
Type <имя_типа>=^ <базовый тип>;
Var <имя_переменной>: <имя_типа>;
```

или

```
<имя_переменной>: ^ <базовый тип>;
```

Пример:

```
Type ss = ^Integer;
Var x, y: ss; {Указатели на переменные целого типа}
a: ^Real; {Указатель на переменную вещественного типа}
```

Зарезервированное слово `nil` обозначает константу ссылочного типа, которая ни на что не указывает.

Выделение оперативной памяти (в «куче») для динамической переменной базового типа осуществляется с помощью оператора `New(x)`, где `x` определен как соответствующий указатель.

Обращение к динамическим переменным выполняется по правилу:

```
<имя_переменной>^
```

Например, оператор `x^:=15` записывает число 15 в область памяти (два байта), адрес которой является значением указателя `x`.

Оператор `Dispose(x)` освобождает память, занятую динамической переменной; при этом значение указателя `x` становится неопределенным.

Все описанные выше действия показаны на рис. 2.1.

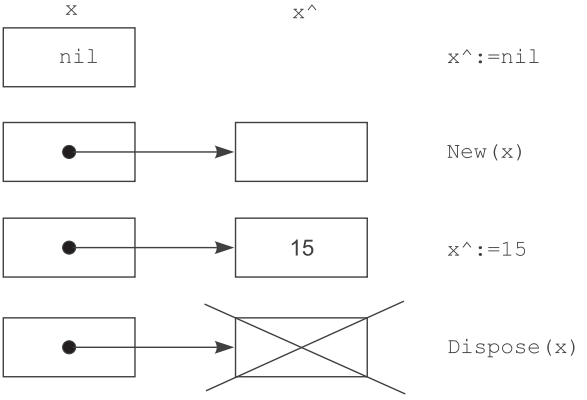


Рис. 2.1. Основные операторы для работы с указателями

2.2. Линейный список

— Анизотропное шоссе, — заявил Антон. Анка стояла к нему спиной. — Движение только в одну сторону.

— Мудры были предки, — задумчиво сказал Пашка. — Этак едешь-едешь километров двести, вдруг — хлоп! — «кирпич».

Аркадий и Борис Стругацкие

Списком называется структура данных, каждый элемент которой при помощи указателя связывается со следующим элементом. Из этого определения следует, что каждый элемент списка содержит, как минимум, поле ссылки на следующий элемент (назовем его `next`) и одно поле данных (назовем его `data` и для простоты будем считать, что оно имеет тип `Integer`), которое может иметь сложную структуру. Поле ссылки последнего элемента списка имеет значение `nil`. Указатель на начало списка (первый элемент) является значением отдельной переменной.

Пример. На рис. 2.2 показан линейный список из четырех элементов, содержащий в полях данных целые числа 3, 5, 1 и 9.

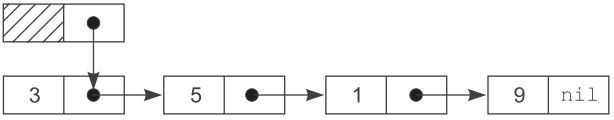


Рис. 2.2. Пример линейного списка

Описание элемента списка имеет вид:

```
Type pt=^elem; {Указатель на элемент списка}
     elem=Record
       data:Integer; {Поле данных (ключ)}
       next:pt; {Указатель на следующий элемент}
     End;
Var first:pt; {Указатель на первый элемент списка}
```

Основные операции с элементами списка:

- вывод элементов списка;
- вставка элемента в список;
- удаление элемента из списка.

Вывод элементов списка (если он создан) — это достаточно очевидная процедура: элементы последовательно просматриваются друг за другом, начиная с первого. Вызов процедуры: `Print(first)`, где `first` — указатель на первый элемент списка.

```
Procedure Print(t:pt);
  Begin
    While t<>nil Do Begin
      Write(t^.data,' ');
      t:=t^.next;
    End;
  End;
```

Ее рекурсивная реализация:

```
Procedure Print(t:pt);
  Begin
    If t<>nil Then Begin
      Write(t^.data,' ');
      Print(t^.next);
    End;
  End;
```

Вставка элемента в список. Пусть дано значение указателя `p` на элемент списка, после которого требуется вставить элемент `y` (рис. 2.3). Список до вставки элемента `y` показан на рис. 2.3*а*, а после вставки — на рис. 2.3*б*. Изменен-

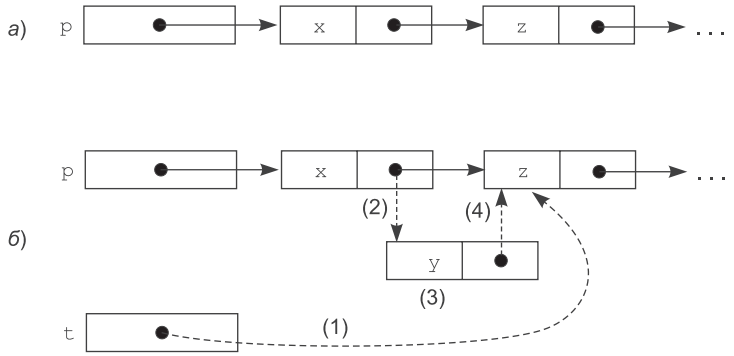


Рис. 2.3. Вставка элемента в список:
а) до операции; б) после операции

ные значения указателей показаны на рисунке пунктирной линией. Последовательность действий обозначена цифрами 1, 2, 3 и 4 и реализована в процедуре Insert:

```
Procedure Insert(p:pt; y:Integer);
  Var t:pt;
  Begin
    t:=p^.next;        {1}
    New(p^.next);      {2}
    p^.next^.data:=y; {3}
    p^.next^.next:=t; {4}
  End;
```

Прежде всего здесь в рабочей переменной t запоминается значение указателя (поля next) элемента, после которого вставляется новый элемент. Вторым шагом выделяется область памяти для нового элемента, а ее адрес становится значением указателя элемента, после которого осуществляется вставка. На третьем шаге формируется поле data нового элемента, а на четвертом — в его поле next переписывается запомненное в t значение.

Логически возможны три случая вставки — в начало, в середину и в конец списка. Для всех ли случаев применима процедура Insert? Очевидно, что во втором и третьем случаях — да. Причем в третьем случае значением указателя является адрес последнего элемента списка, он имеет ссылку nil, и это значение переписывается в поле next вставляемого элемента. В первом же случае эта процедура не работает! Ведь нам требуется изменить значение глобальной переменной first, а мы не передаем в вызывающую логику измененное значение p. Но если изменить параметры процедуры на

```
Procedure Insert(Var p:pt; y:Integer)
```

то проблема будет решена, хотя и останется вопрос вставки первого элемента в пустой список (p = nil): в этом случае действие p^.next «выбрасывает» нас в неопределенную область памяти. В процедуре же Ins_List решается и эта проблема:

```
Procedure Ins_List(Var p:pt;y:Integer);
  Begin
    If p=nil Then Begin
```

```
      New(p);
      p^.data:=y;
      p^.next:=nil;
    End
      Else Insert(p,y); {Параметр p - глобальный
      в процедуре Insert. Четыре оператора этой
      процедуры лучше просто "прописать" здесь}
  End;
```

Удаление элемента из списка. Действия при удалении элемента из списка сводятся к его поиску, а затем — к переадресации от элемента, предшествующего удаляемому, к элементу, следующему за удаляемым (рис. 2.4*а*). Единственная сложность здесь заключается в том, чтобы предусмотреть случай удаления первого элемента списка, когда изменяется значение переменной `first` (рис. 2.4*б*).

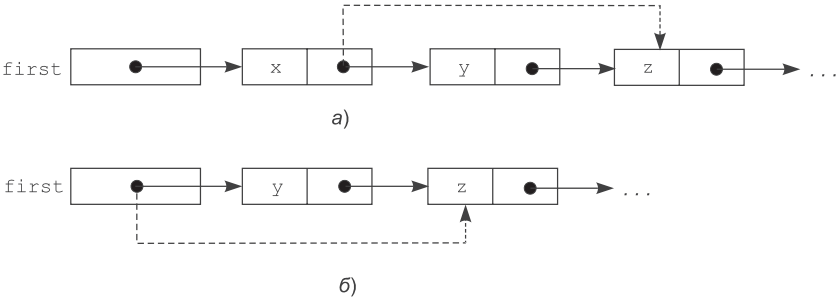


Рис. 2.4. Удаление элемента из списка

В процедуре `Del_List` из списка удаляются все элементы, равные `y`:

```
Procedure Del_List(Var first:pt; y:Integer);
  Var t,x,dx:pt;
  Begin
    t:=first; {Переменная цикла}
    While t<>nil Do {Пока список не просмотрен}
      If t^.data=y Then {Есть совпадение}
        If t=first Then Begin
         {Удаляем первый элемент списка}
         x:=first;
         {Запоминаем, ибо "кучу" засорять не следует}
```

```
        first:=first^.next; {Изменяем значение
        указателя на первый элемент списка}
        Dispose(x); {Освобождаем место в "куче"}
        t:=first;
        {Переменная цикла изменила свое значение}
        End
      Else Begin
        x:=t;
        {Запоминаем адрес удаляемого элемента}
        t:=t^.next;
        dx^.next:=t; {Удаление элемента не должно
        нарушать структуру списка}
        Dispose(x);
      End
    Else Begin
      dx:=t; {Переход к следующему элементу списка.
      Адрес текущего элемента списка запоминается
      в переменной dx}
      t:=t^.next;
    End
End;
```

Упражнения

1. Список из элементов a_1, a_2, ..., a_n (значения поля `data`) назовем упорядоченным по неубыванию, если выполняется следующее условие: $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n$. Напишите функцию проверки упорядоченности такого списка.

2. Верно ли, что следующая функция возвращает адрес (указатель) первого элемента списка, равного значению `x`?

```
Function Locate(t:pt;x:Integer):pt;
  Begin
    While (t<>nil) And (t^.data<>x) Do t:=t^.next;
    If t=nil Then Locate:=nil
    Else Locate:=t;
  End;
```

3. Список состоит не менее чем из пяти элементов. Мы вызываем функцию `Retrieve(first,x)`, где `x` — некото-

рое целое число. Каков результат работы этой функции? Что она делает?

```
Function Retrieve(t:pt;p:Integer):Integer;
{Предполагаем, что p <> 0}
  Var q:Integer;
  Begin
    q:=1;
    While (t<>nil) And (t^.data<>p) Do Begin
      q:=q+1;
      t:=t^.next;
    End;
    If t=nil Then Retrieve:=0
    Else Retrive:=q;
  End;
```

4. Определите, что делает функция `Succ` (при ее вызове значение `t` равно `first`).

```
Function Succ(t:pt; p:Integer):pt;
  Var q:Integer;
  Begin
    q:=0;
    While (t<>nil) And (q<>p) Do t:=t^.next;
    If (t<>nil) And (p=q) Then Retrieve:=t^.next
    Else Retrive:=nil;
  End;
```

5. Определите, что делает функция `Pred` (при ее вызове значение `t` равно `first`).

```
Function Pred(t:pt; p:Integer):pt;
  Var q:Integer;
      dt:pt;
  Begin
    q:=0;
    dx:=nil;
    While (t<>nil) And (q<>p) Do Begin
      dt:=t;
      t:=t^.next;
    End;
    If (t<>first) And (p=q) Then Retrieve:=dx
    Else Retrive:=nil;
  End;
```

6. Напишите функцию, возвращающую указатель на последний элемент списка. Ответьте на вопрос: можно ли использовать ее, как в упражнениях 4 и 5?

7. Разработайте:
- функцию, вычисляющую среднее арифметическое значение элементов непустого списка;
- рекурсивную функцию проверки наличия в списке заданного элемента;
- процедуру перестановки первого и последнего элементов непустого списка;
- процедуру вставки нового элемента перед (после) каждым вхождением заданного элемента;
- функцию проверки совпадения списков L_1 и L_2;
- функцию проверки вхождения списка L_1 в список L_2;
- процедуру переноса первого элемента непустого списка L в его конец;
- процедуру переноса последнего элемента непустого списка в его начало;
- процедуру копирования в список L всех элементов списка L_1 вслед за каждым вхождением заданного элемента;
- процедуру объединения двух упорядоченных по неубыванию списков L_1 и L_2 в один упорядоченный по неубыванию список путем построения нового списка L и изменения соответствующим образом ссылок в L_1 и L_2;
- функцию подсчета количества слов в списке (поле `data` имеет тип `String`), начинающихся и заканчивающихся одним и тем же символом.

8. Разработайте процедуру удаления из списка L:
- второго элемента, если такой есть;
- всех элементов, равных x;
- первого отрицательного элемента, если такой есть;
- всех отрицательных элементов.

9. Разработайте процедуру формирования списка L путем включения в него по одному разу элементов:
- входящих хотя бы в один из списков L_1 и L_2;
- входящих одновременно в оба списка L_1 и L_2;
- входящих в список L_1, но не входящих в список L_2;
- входящих в один из списков L_1 и L_2, но не входящих в другой из них.

10. «*Считалочка*». Пусть n ребят расположены по кругу. Начав отсчет от первого, удаляем каждого k-го участника, смыкая круг. Напишите программу определения порядка удаления ребят из круга.
Примечание. Для хранения данных об участниках игры целесообразно использовать *кольцевой список*, где значением поля `next` последнего элемента является адрес первого элемента списка — рис. 2.5.

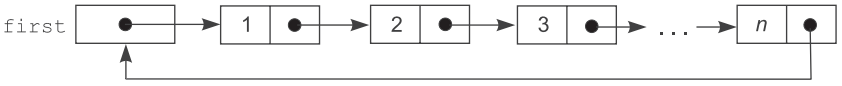


Рис 2.5. Первоначальное состояние кольцевого списка

Исследуйте эту задачу для различных значений n. Заполните табл. 2.1 для оставшихся значений n от 1 до 64 (где t — номер оставшегося участника).

Таблица 2.1

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...
t	1	1	3	1	3	5	7	1	3	5	7	9	11	13	15	1	3	...

Экспериментальным путем установите закономерность:
- $t(1) = 1$ при $n = 1$;
- $t(2 \cdot n) = 2 \cdot t(n) - 1$ при $n \geqslant 1$;
- $t(2 \cdot n + 1) = 2 \cdot t(n) - 1$ при $n \geqslant 1$.

Если $n = 2^m + q$, где 2^m — наибольшая степень двойки, не превосходящая n, а q — разность $n - 2^m$, то номер оставшегося участника вычисляется по формуле: $t(2^m + q) = 2 \cdot q + 1$ при $m \geqslant 0$ и $0 \leqslant q < 2^m$.
Экспериментально проверьте правильность этой формулы и напишите версию программы, вычисляющей номер оставшегося участника игры без использования ссылочного типа данных. Сравните результаты работы обеих написанных вами программ.

11. Пусть в задаче о «считалочке» удаляется каждый второй участник. Напишите программу для определения номеров двух последних оставшихся ребят.

12*. Пусть в задаче о «считалочке» участник находится на месте с номером i. Существует ли такое значение k (k-й удаляемый участник), что соответствующий участник останется последним в круге? Напишите программу для поиска значения k.

2.3. Реализация линейного списка с использованием массивов

> В последнее время в прессе муссируются структуры данных... Как известно всякому *Настоящему Программисту*, единственная полезная структура данных — массив. Строки, списки, структуры и множества — все они лишь частные случаи массивов, и их легко можно обрабатывать как массивы без усложнения вашего языка программирования.
>
> *Эд Пост*

Работу со списком можно организовать и без использования ссылочного типа данных (указателей). *Массив* — это, в определенной степени, универсальная структура данных.

Первым вариантом такой реализации является использование двух массивов либо *массива записей* (`list`), который мы и рассмотрим.

Элемент массива (запись), как минимум, должен содержать два поля: ключ (`data`) и адрес следующего элемента списка (`next`). Адресом следующего элемента списка при этом является номер (индекс) этого элемента массива. Кроме того, нам необходимы еще две переменные: `first` — указывает на местоположение первого элемента списка и `free` — определяет первое свободное место в массиве для записи нового элемента. До начала работы со списком элементы массива «размечаются» как свободные при помощи программного фрагмента:

```
For i:=1 To NMax-1 Do list.next[i]:=i+1,
```

где `NMax` — максимальный размер массива.

Пусть нам необходимо организовать работу с упорядоченным по значению ключей списком (по неубыванию). Описание данных в этом случае может иметь вид:

```
Const NMax=...;
Type elem=Record
       data:Integer; {Поле данных (ключ)}
       next:Integer;
       {Указатель на следующий элемент списка}
     End;
     MyArray=Array[1..NMax] Of elem;
Var  list:MyArray;
     first, free:Integer;
```

Пример. На рис. 2.6 показан пример такого списка в некоторый момент времени работы с ним. Первый его элемент записан в 7-й ячейке, второй — в 1-й и т. д. Первой свободной является ячейка с номером 8.

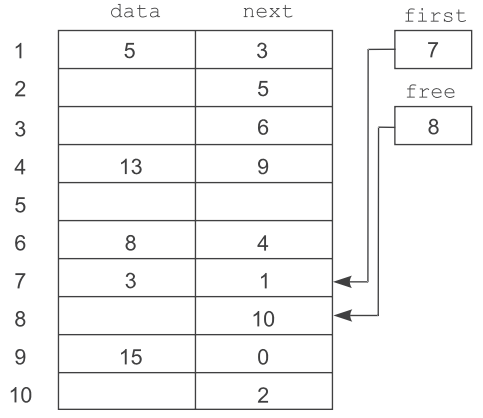


Рис. 2.6. Пример представления списка с использованием массивов

Рассмотрим теперь процедуру вставки элемента в список. Для этого надо взять первую свободную ячейку и изменить значение переменной `free`, а затем найти место в упорядоченном списке для нового элемента. При этом для корректной вставки необходимо знать адрес предшествующего вставляемому элемента `j`. Последний момент, который следует учесть, — это вставка в начало списка. В этом случае корректировке подлежит значение переменной `first`.

```
Procedure Ins_List(Var first,free:Integer;x:Integer);
{Имена first и free в заголовке процедуры приведены
для наглядности}
  Var i,j,t:Integer;
  Begin
    t:=free; {Предполагаем, что free <> 0, т. е.
              есть свободное место}
    list.data[t]:=x;
    {Формируем поле data нового элемента}
    free:=list.next[t];
    i:=first;
    j:=first;
    {В j храним адрес предыдущего элемента списка}
    While (i<>0) And (x>list.data[i]) Do Begin
      j:=i; {Просмотр элементов списка для
      определения места вставки нового элемента}
      i:= list.next[i];
    End;
    list.next[t]:=i; {Номер следующего элемента
    списка является значением переменной i}
    If i=first Then first:=t
    Else list.next[j]:=t; {Если элемент вставляется
    в начало списка, то изменяется значение
    переменной first, иначе корректируется поле
    ссылки предыдущего элемента}
  End;
```

При удалении элемента x из списка требуется найти его в списке, зная адрес предшествующего элемента, а затем, после традиционного изменения значений поля next, сделать эту ячейку первой в числе свободных:

```
Procedure Del_List(Var first,free:Integer;
                   x:Integer);
  Var i,j:Integer;
  Begin
    i:=first;
    j:=first;
    While(i<>0) And (x<>list.data[i]) Do Begin
    {Поиск удаляемого элемента}
      j:=i;
      i:=list.next[i];
    End;
```

```
    If i<>first Then list.next[j]:=list.next[i];
    {Изменение значения ссылки на следующий элемент
    списка}
    Else first:=list.next[i];
    list.data[i]:=0;
    list.next[i]:=free;
    {Освобождаемый элемент делаем первым свободным}
    free:=i;
End;
```

Второй вариант реализации списка с помощью массивов заключается в расположении его элементов в смежных ячейках. В этом случае достаточно одного массива (`data`) и одной переменной (например `last`), для фиксации номера (индекса) первой свободной ячейки массива. Тогда при вставке элемента в такой список потребуется перемещение всех последующих элементов на одну ячейку к концу массива, а при удалении — к началу массива.

Пример. На рис. 2.7 показано представление списка: а) исходного; б) после вставки одного элемента; в) после удаления двух элементов списка.

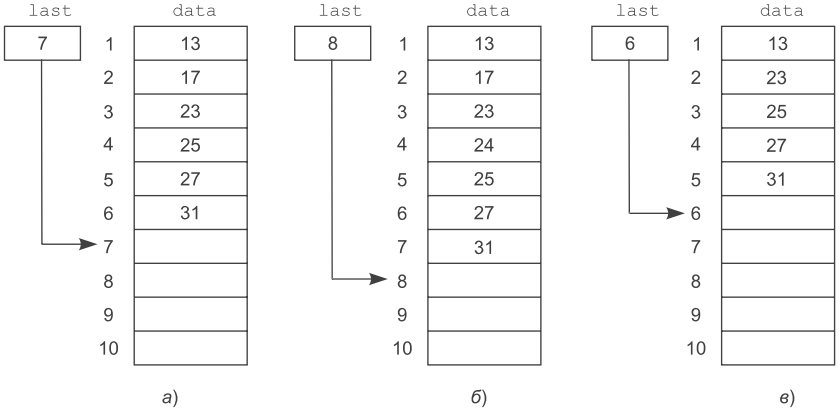


Рис. 2.7. Представление списка с использованием одного одномерного массива: а) исходный список; б) после вставки элемента с ключом 24; в) после удаления элементов с ключами 17 и 25

Вставка элемента в упорядоченный список требует сдвига части его элементов во всех случаях, кроме вставки в самый конец списка. Текст процедуры при этом имеет вид:

```
Procedure Ins_List1(Var last:Integer;x:Integer);
  Var i,j:Integer;
  Begin
    i:=1;
    While (i<last) And (x>data[i]) Do i:=i+1;
    {Находим место для вставки элемента}
    If i<>last Then
      For j:=last DownTo i Do data[j+1]:=data[j];
      {Если элемент вставляется не в конец списка,
      то требуется выполнить сдвиг}
    data[i]:=x;
    last:=last+1;
  End;
```

Упражнения

1. Какие действия необходимо предпринять, чтобы исключить вставку элемента в список (работу процедуры `Ins_List`) при отсутствии места в массиве `list`?
2. Как будет работать процедура `Del_List` при отсутствии элемента `x` в списке? Если необходимо, внесите изменения в текст этой процедуры.
3. Выполните упражнение 2 при втором представлении списка с помощью массива.
4. Напишите процедуру для удаления `q`-го элемента упорядоченного списка.
5. Напишите программу для слияния двух упорядоченных списков в один.

2.4. Двусвязные списки

Тоской по пониманию — вот чем я болен, тоской по пониманию...

Аркадий и Борис Стругацкие

Линейный список (см. раздел 2.2) обеспечивает эффективный просмотр элементов только в одном направлении. Если же в решаемой задаче необходимо использовать список, а операции по определению предшествующего элемен-

та применяются достаточно часто, то целесообразно использовать *двусвязные списки* (рис. 2.8), в которых у каждого элемента имеется связь не только со следующим, но и с предшествующим элементом.

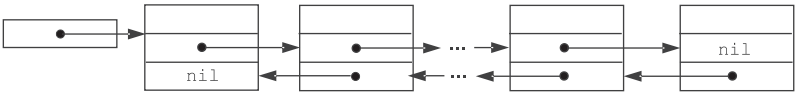


Рис. 2.8. Двусвязный список

Описание элемента списка в этом случае имеет вид:

```
Type pt=^elem; {Указатель на элемент списка}
     elem=Record
       data:Integer; {Поле данных (ключ)}
       next, prev:pt; {Указатели на следующий
       и на предшествующий элементы списка}
     End;
Var head:pt; {Указатель на первый элемент списка}
```

Рассмотрим логику вставки элемента в двусвязный список. На рис. 2.9 показано, как осуществляется вставка нового элемента в середину списка. Адрес элемента, после которого вставляется новый элемент, является значением указателя `p`. На рис. 2.9 цифрами отмечены действия, а пунктирными линиями показано, адреса каких элементов и куда записываются.

Оформленные в виде процедуры, эти действия выглядят так:

```
Procedure Insert(p:pt; y:Integer);
  Var t:pt;
  Begin
    t:=p^.next;
    {Шаг 1 - запоминаем адрес следующего элемента}
    New(p^.next); {Шаг 2 - формируем новый элемент,
    а его адрес записываем в поле next элемента,
    после которого осуществляется вставка}
    p^.next^.data:=y;
    p^.next^.prev:=p; {Шаг 3 - предыдущим для
    вставляемого элемента является элемент,
    на который указывает p}
```

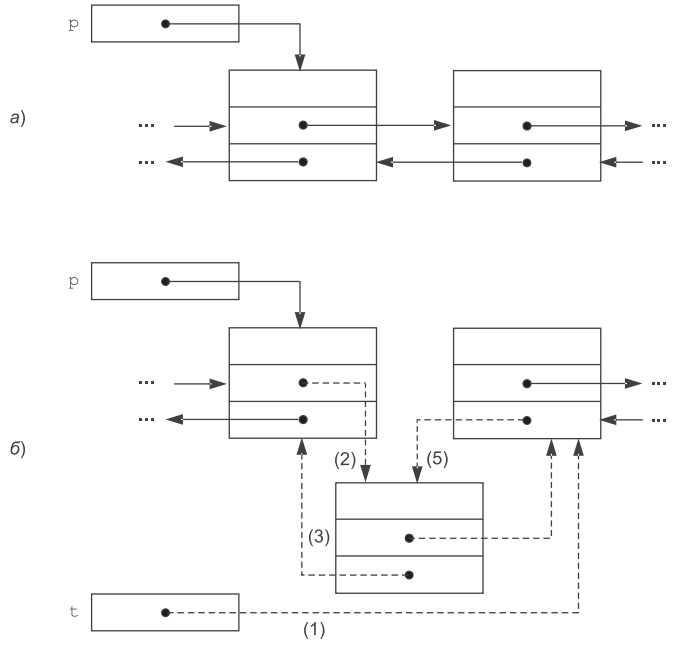


Рис. 2.9. Вставка в середину двусвязного списка: а) до вставки; б) после вставки

```
    p^.next^.next:=t; {Шаг 4 - следующим за
    вставляемым элементом является элемент, адрес
    которого был запомнен}
    t^.prev:=p^.next; {Шаг 5 - предыдущим для
    элемента, следовавшего в списке за элементом,
    после которого осуществляется вставка,
    является вставленный элемент}
End;
```

Далее мы должны рассмотреть различные случаи вставки: в пустой список (вставка первого элемента), в начало списка и в конец списка. В каких из этих случаев указанная логика (процедура `Insert`) неработоспособна? Очевидно, что вставка первого элемента приводит к ошибке — ведь значение `p` равно `nil`. Тогда для исправления ситуации достаточно операторов:

```
New(p); {2}
p^.data:=y;
p^.prev:=nil; {3}
p^.next:=nil; {4}
```

где p имеет значение head, возвращаемое в вызывающую программу.

При вставке в начало уже существующего (не пустого) списка значение p равно head и не равно nil. Тогда вставку реализует такая последовательность операторов:

```
t:=p;          {1}
New(p);        {2}
p^.data:=y;
p^.prev:=nil; {3}
p^.next:=t;    {4}
t^.prev:=p;    {5}
```

Остался последний случай — вставка в конец списка. Он характеризуется тем, что значение p^.next равно nil, т. е. значению t, так что пятое действие (рис. 2.9) процедуры Insert становится некорректным. Для решения проблемы достаточно выполнить следующие действия:

```
New(p^.next);        {2}
p^.next^.data:=y;
p^.next^.prev:=p;    {3}
p^.next^.next:=nil; {4}
```

Синтезируя все разобранные случаи в единое целое, мы и получим процедуру вставки в двусвязный список.

Рассмотрим теперь логику удаления из двусвязного списка. Пусть у нас уже есть значение указателя на удаляемый элемент, и он не является ни первым, ни последним в списке. С помощью указателя prev тогда определяется адрес предшествующего элемента и в его поле next записывается указатель на ячейку, следующую за удаляемым элементом. Далее аналогично находится следующий за удаляемым элемент списка и в его поле prev записывается адрес элемента, предшествующего удаляемому элементу t. Эти действия отражены в следующей процедуре:

```
Procedure Del(t:pt);
  Begin
    If t^.prev<>nil Then t^.prev^.next:=t^.next;
    If t^.next<>nil Then t^.next^.prev:=t^.prev;
    Dispose(t);
  End;
```

Первый элемент списка характеризуется тем, что значение `t^.prev` равно `nil`. В этом случае необходимо изменить значение переменной `head`, для чего потребуется замена первого оператора процедуры `Del` на:

```
If t^.prev<>nil Then t^.prev^.next:=t^.next
                Else head:=t^.next;
```

У последнего элемента списка поле `t^.next` равно `nil`, а потому поле `prev` следующего элемента не требует корректировки (поскольку его нет). Поэтому текст процедуры `Del` работоспособен.

Случай же удаления последнего элемента из списка автор намеренно оставил неразобранным. Самостоятельно ответьте на вопрос: требуется ли вносить изменения в процедуру `Del` для его реализации?

Упражнения

1. Напишите процедуру вставки элемента в двусвязный список.

2. Изобразите на рисунке логику удаления элемента из списка во всех случаях: из начала, середины и из конца списка. Напишите полный текст процедуры удаления элемента из двусвязного списка.

3. Добавим в двусвязный список «фиктивный» элемент так, как показано на рис. 2.10. Значением `prev` первого элемента и `next` последнего элемента при этом является адрес «фиктивного» элемента. Упростятся ли в этом случае процедуры вставки и удаления из списка? Напишите их и сравните с процедурами из упражнений 1 и 2.

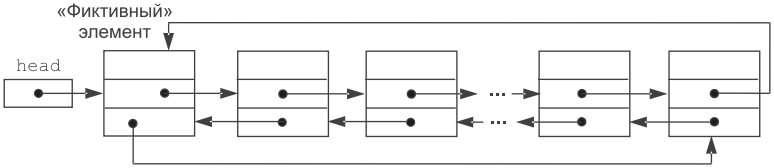


Рис. 2.10. Двусвязный список с «фиктивным» элементом

4. Напишите для двусвязного списка процедуры вставки и удаления элемента перед заданным элементом списка.

5. Напишите процедуру обмена значениями (поля `data`) элементов двусвязного списка с указателями `t` и `t^.next`. Рассмотрите все особые случаи.

6. Напишите процедуру обмена значениями (поля `data`) элементов двусвязного списка с указателями `t` и `t^.prev`. Рассмотрите все особые случаи.

7. Двоичное число $a_1, a_2, \ldots, a_n$, где $a_i = 0$ или 1 (значение поле `data`), можно представить в виде двусвязного списка $a_1, a_2, \ldots, a_n$. Напишите процедуру прибавления единицы к такому двоичному числу.

8. Напишите процедуру слияния двух упорядоченных двусвязных списков в один (также упорядоченный двусвязный).

9. Даны два двоичных числа, представленные в виде двусвязных списков (см. упражнение 7). Напишите программу для сложения этих чисел.

Методические комментарии

Тема «Указатели и списки» является классической в информатике. Приведем следующую цитату из книги Д. Кнута[1]:

«Идея размещения линейных списков в непоследовательных ячейках, по-видимому, впервые появилась в связи с проектированием ЭВМ с запоминающими устройствами на магнитных барабанах; из существующих машин такого типа самой примечательной и последней была машина IBM 650. После выполнения команды из ячейки n такая машина обычно не могла взять следующую команду из ячейки $n + 1$, так как барабан уже повернулся и прошел эту точку. В зависимости от выполняемой команды наиболее предпочтительной позицией для следующей команды может быть $n + 7$ или $n + 18$ и так далее, и машина может работать в шесть, семь раз быстрее, если команды расположены оптимальным образом, а не последовательно. Поэтому при проектировании машины в каждой машинной команде предусматривалось дополнительное адресное поле, которое служило связью со следующей командой».

[1] *Кнут Д.* Искусство программирования для ЭВМ. Т. 1. Основные алгоритмы. М.: Мир, 1976. С. 558–559.

Далее Д. Кнут указывал, что методы связанной памяти фактически возникли в 1956 г. в работах Э. Ньюэлла, К. Шоу и Х. Саймона, когда эти авторы исследовали эвристические методы решения задач с помощью ЭВМ (в настоящее время такую проблематику называют «искусственным интеллектом»). Под влиянием работ этих ученых идеи «связанной памяти» стали *основным инструментом программирования* на ЭВМ. Первоисточник же списков с циклическими и двойными связями установить достаточно сложно, ибо эти идеи были естественным развитием и продолжением идей, заложенных в линейном списке.

Данная тема так или иначе присутствует в любой книге по фундаментальным основаниям информатики, но степень ее доступности для школьника и учителя в таких изданиях весьма различна. Наша цель — дать простое и ясное описание темы с заданиями для самостоятельной и коллективной работы школьников.

Глава 3

Стек

Тихо, тихо ползи, улитка,
по склону Фудзи,
вверх, до самых высот!

Кобаяси Исса

3.1. Основные понятия

Одним из примеров абстрактного типа данных является *стек* (*stack*) — упорядоченный набор элементов, в котором добавление новых элементов и удаление существующих производится с одного конца, называемого *вершиной стека*.

Простой пример стека — детская пирамидка. Процесс сборки и разборки такой пирамидки подобен процессу функционирования стека: в любой момент времени доступен лишь один элемент пирамидки — самый верхний. Другой пример — коробка с листами бумаги: новый (последний) лист кладется в стопку сверху, и только верхний лист может быть извлечен из коробки; для извлечения же некоторого листа из середины стопки необходимо сначала вынуть из коробки все листы, лежащие над ним.

Из приведенного выше определения следует, что извлекать элементы из стека можно только в порядке, обратном порядку их добавления в стек. Этот принцип формулируется так: «первым пришел, последним ушел» или «последним пришел — первым ушел» (*last-in, first-out — LIFO*).

Основные операции со стеком: запись («вталкивание» — *Push*) элемента в стек; извлечение («выталкивание» — *Pop*) элемента из стека; проверка наличия элементов в стеке («стек пустой» — *Empty*).

3.2. Реализация стека через линейный список

Мостовая пусть качнется, как очнется.
Пусть начнется, что еще не началось.

Булат Окуджава

Если использовать линейный список для представления данных стека, то его можно определить как список, в котором добавление новых элементов и извлечение имеющихся происходит с начала (или конца) списка. Значением указателя, представляющего стек, тогда является ссылка на вершину стека, а каждый элемент стека содержит поле ссылки на следующий элемент.

Таким образом, описать стек (элементом данных здесь являются целые числа) можно следующим образом:

```
Type pt=^elem;
     elem=Record
       data: Integer;
       next: pt;
     End;
Var  head:pt;
```

Если стек пуст, то значение указателя `head` равно `nil`. Эта проверка реализуется функцией `Empty` с результатом логического типа:

```
Function Empty(head:pt):Boolean;
  Begin
    If head=nil Then Empty:=False
    Else Empty:= True;
  End;
```

Процедура записи элемента в стек должна содержать два параметра: первый определяет указатель на начало стека, а второй — записываемое в стек значение. Запись в стек производится аналогично вставке нового элемента в начало списка:

```
Procedure Push(Var head:pt; x:Integer);
  Var t:pt;
  Begin
```

```
    New(t);
    t^.data:=x;
    t^.next:=head;
    head:=t;
  End;
```

Извлечение элемента из непустого стека. В результате выполнения этой операции некоторой переменной x должно быть присвоено значение первого элемента стека, а затем изменено значение указателя на начало списка:

```
Procedure Pop(Var head:pt; Var x:Integer);
  Var t:pt;
  Begin
    x:=head^.data;
    t:=head;
    head:=head^.next;
    Dispose(t);
  End;
```

3.3. Реализация стека с использованием массива

Хорошо выраженная мысль звучит
умно на всех языках.

Джон Драйден

Стек — это упорядоченный набор данных, и кажется, было бы естественным использовать для его реализации массив, ибо последний также является упорядоченным набором данных. Отличие же между ними в том, что в первом случае запись и чтение осуществляются по правилам стека, а во втором — есть возможность обращаться к любому элементу по значению его индекса.

Рассмотрим, как можно реализовать стек на базе массива. Будем считать, что для реализации стека используется массив статического типа с именем A (`A:Array[1..n] Of Integer`). Однако тут же возникают вопросы: каков должен быть размер этого массива (значение n); сколько ячеек следует зарезервировать под стек. Однозначного ответа на них

нет — величина n определяется из условий конкретной задачи и ограничений на используемую память.

Пример. На рис. 3.1 приведено состояние стека в определенный момент времени. Элементы записывались в стек в следующей очередности: 13, 5, 20, 7, 18, 21 и 3, начиная с конца массива A. Доступный для чтения (извлечения) элемент стека — 3. Начальное значение указателя на вершину стека (head) равно n + 1.

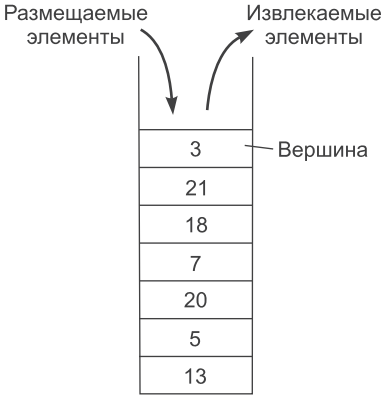


Рис. 3.1. Пример размещения стека в массиве

Операции «вытолкнуть» и «втолкнуть» в этом случае очевидны:

```
Procedure Pop(Var x:Integer);
  Begin
    x:=A[head];
    head:=head+1;
    {Элемент A[head] после выталкивания
    не обнуляется}
  End;

Procedure Push(x:Integer);
  Begin
    head:=head-1;
    A[head]:=x;
  End;
```

Кроме функции `Empty`, проверяющей пустоту стека, при реализации стека через массив возникает необходимость проверки *переполнения стека* (функция `Full`). Если значение `head` равно единице, то была осуществлена запись в первую ячейку, и стек заполнен полностью.

```
Function Empty:Boolean;
  Begin
    If head=n+1 Then Empty:=True
    Else Empty:=False;
  End;

Function Full:Boolean;
  Begin
    If head=1 Then Full:=True
    Else Full:=False;
  End;
```

Теоретически возможна и другая реализация стека через массив. Пусть вершина стека зафиксирована (ее значение постоянно в течение всего времени работы со стеком), например `head = n`, а запись в эту ячейку и чтение из нее приводят к сдвигу элементов массива на одну позицию вверх (к началу массива) и вниз соответственно.

Пример. На рис. 3.2 показано состояние стека для тех же данных, что и на рис. 3.1.

Рис. 3.2. Вершина стека зафиксирована

Начальное состояние элементов массива равно нулю. Это существенно при данной реализации стека, а фиксация пустоты и полноты стека осуществляется с помощью логических переменных Empty и Full, значения которых определяются при «вталкивании» и «выталкивании» элементов (начальные значения: Empty = **True**, Full = **False**).

Одна из возможных реализаций процедур Push и Pop тогда может иметь следующий вид:

```
Procedure Push(x:Integer);
  Var i,j:Integer;
  Begin
    i:=n;
    While (i>0) And (A[i]<>0) Do i:=i-1; {Определяем
    позицию первого (снизу) нулевого элемента A}
    If i=0 Then Full:=True {Стек заполнен}
    Else Begin
      Full:=False;
      For j:=i To n-1 Do A[j]:=A[j+1]; {Осуществляем
      сдвиг "вверх" на одну позицию}
      A[n]:=x; {Записываем элемент}
    End;
  End;

Procedure Pop(Var x:Integer);
  Var i:Integer;
  Begin
    If A[n]=0 Then Empty:=True {Стек пуст}
    Else Begin
      Empty:=False;
      x:=A[n]; {Читаем элемент - "выталкиваем" его}
      i:=n;
      While (i>0) And (A[i]<>0) Do Begin
        A[i]:=A[i-1];
        {Верно ли работает верно логика при
        полностью заполненном стеке? Может быть,
        следует определить массив A как A:Array[0..n]
        Of Integer и считать, что A[0]=0?}
        i:=i-1;
      End;
    End;
  End;
```

Как видим, этот вариант реализации стека через массив проигрывает первому не только по времени выполнения операций `Pop` и `Push` (оно не константное, выполняемое за фиксированное количество операций, а равно $O(n)$), но и по сложности реализации логики.

Упражнения

1. На рис. 3.3 представлены два способа реализации стека с использованием массива: когда запись в массив начинается с первой ячейки и идет «вниз» (рис. 3.3*а*) и когда запись всегда осуществляется в первую ячейку массива (рис. 3.3*б*). Напишите процедуры `Pop` и `Push` для этих способов. Сформулируйте особенности каждой реализации.

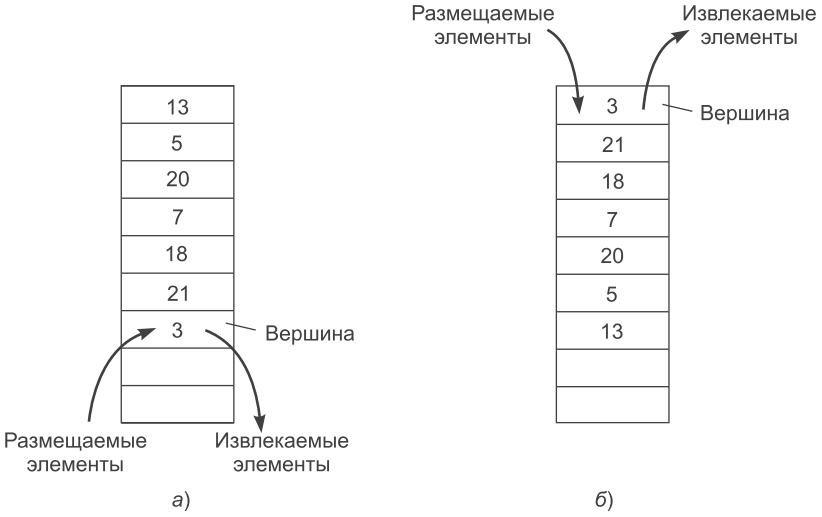


Рис. 3.3. Примеры реализации стека:
а) вершина стека «перемещается сверху вниз»;
б) вершина стека зафиксирована в первой ячейке

2. Рассмотрите вариант размещения двух стеков в одном массиве. Выберите способ его реализации и определите особенности процедур для реализации операций со стеком: `Pop`, `Push`, `Empty` и `Full`.

3. Как организовать хранение k стеков в одном массиве? Что необходимо сделать в случае наложения («конфликта») стеков при записи нового элемента? Какие изменения требуется в этом случае сделать в массиве A (как реорганизовать память)?

4. Напишите программу, проверяющую своевременность закрытия скобок типа (,), {, }, [,] в строке символов (если строка состоит из одних скобок этих типов). Так, строка `(([]{}))` является правильной, а строки `(([]{))` и `(([{]}))` — неверны: в первой из них не хватает одной скобки, а во второй нарушена логика (закрывающая скобка должна быть того же типа, что и последняя открывающая).

Рекомендации.

1) для решения задачи следует определить стек, элементами которого являются символы:

```
Type pt=^el;
     el=Record
       data:Char;
       next:pt;
     End;
```

2) можно использовать факт, что коды соответствующих друг другу скобок отличаются не более чем на 2 (в соответствии с кодировкой ASCII): скобки { и } имеют коды 123 и 125; [и] — 91 и 93; (и) — 40 и 41, причем код открывающей скобки меньше кода закрывающей;

3) в процессе решения задачи анализируются символы строки. Если встречена одна из открывающих скобок, то она записывается в стек. При обнаружении закрывающей скобки, соответствующей скобке, находящейся в вершине стека, последняя из него удаляется. При несоответствии скобок выдается сообщение об ошибке, которое фиксируется в логической переменной.

Следует рассмотреть и другие случаи, например когда обработка строки закончена, а стек — не пуст.

5. Дана строка вида: `s*w` (где s и w — строки из символов, не содержащих символа *). Чтение разрешено по одному символу. Проверьте, является ли строка w обратной строке s. Например, для случая s = ABCDEF, w = FEDCBA ответ будет положительным.

3.4. Постфиксная, префиксная и инфиксная формы записи выражений

> Человек не станет изучать абстрактные модели некоторой проблемы, если его не заинтересовали примеры.
>
> *Дональд Кнут*

Рассмотрим использование такой структуры данных, как стек на примере обработки записей, представленных в различном виде. Отметим значимость этой проблемы в информатике, хотя мы ограничимся здесь ее простейшей интерпретацией.

Пусть a и b — некоторые числа, а выражение $a + b$ обозначает сумму этих чисел. Такую запись называют *инфиксной*, а записи типа «$+\ a\ b$» и «$a\ b\ +$» — *префиксной* и *постфиксной*[1] соответственно. Префиксы «пре», «пост» и «ин» здесь фиксируют относительную позицию знака операции по отношению к операндам: в префиксной записи знак операции предшествует обоим операндам, в постфиксной — следует за двумя операндами, а в инфиксной — разделяет эти два операнда.

Пусть требуется преобразовать выражение из инфиксной в постфиксную запись. Единственным правилом, используемым в процессе такого преобразования, является то, что операции с высшим приоритетом преобразуются первыми, а после того как операция преобразована к постфиксной форме, она рассматривается как один операнд. Для бинарных операций (с двумя операндами) «+», «–», «*», «/» и «^» приоритет считается традиционным (от высшего к низшему): возведение в степень → умножение/деление → сложение/вычитание. В инфиксной записи он может быть изменен с помощью скобок.

Примеры.

1. Дано выражение $a + b * c$. Последовательность действий по его преобразованию в постфиксную форму:

---

[1] Эту форму записи называют также *польской* в честь польского математика Яна Лукасевича.

$a + (b * c)$ — скобками выделяется операция;
$a + (b\ c\ *)$ — преобразована операция умножения;
$a\ (b\ c\ *) +$ — преобразована операция сложения;
$a\ b\ c\ * +$ — постфиксная форма.

2. Для выражения $(a + b) * c$ последовательность преобразований в постфиксную форму имеет вид:
$(a + b) * c$ — инфиксная форма;
$(a\ b\ +) * c$ — преобразована операция сложения;
$(a\ b\ +)\ c\ *$ — преобразована операция умножения;
$a\ b + c\ *$ — постфиксная форма.

В табл. 3.1 приведен еще ряд примеров преобразования инфиксного представления в постфиксную и префиксную формы. При их анализе следует считать, что при возведении в степень вычисления производятся справа налево: $a\hat{}b\hat{}c = a\hat{}(b\hat{}c)$, а для операций с одинаковым приоритетом — слева направо: $a - b - c = (a - b) - c$.

Таблица 3.1

Инфиксное представление	Постфиксное представление	Префиксное представление
$a+b-c$	$a\,b+c-$	$-+a\,b\,c$
$(a+b)*(c-d)$	$a\,b+c\,d-*$	$*+a\,b-c\,d$
$a\hat{}b*c-d+e/f/(g+h)$	$a\,b\hat{}c*d-e\,f/g\,h+/+$	$+-*\hat{}a\,b\,c\,d/e\,f+g\,h$
$a-b/(c*d\hat{}e)$	$a\,b\,c\,d\,e\hat{}*/-$	$-a/b*c\hat{}d\,e$
$((a+b)*c-(d-e))\hat{}(f+g)$	$a\,b+c*d\,e--f\,g+\hat{}$	$\hat{}-*+a\,b\,c-d\,e+f\,g$

Логика использования стека в процессе преобразования выражения, например $(a + b) * (c + d)$, в постфиксную форму отражена в табл. 3.2. В первом столбце показан читаемый символ из исходной строки, во втором столбце — результирующая строка, а в третьем столбце — текущее состояние стека.

Идентификаторы переменных (предполагаем, что они состоят из одного символа) мы сразу переписываем в результирующую строку, а знаки операций и открывающую круглую скобку записываем в стек. Если встречается закрывающая скобка, то из стека символы операций (до открывающей скобки) переписываются в результат, а обе скобки отбрасываются. После того как вся исходная строка про-

Таблица 3.2

Символ строки	Результат	Состояние стека
(		(
a	*a*	(
+	*a*	(+
b	*a b*	(+
)	*a b* +	
*	*a b* +	*
(	*a b* +	*(
c	*a b* + *c*	*(
+	*a b* + *c*	*(+
d	*a b* + *c d*	*(+
)	*a b* + *c d* +	*
	a b + *c d* + *	

смотрена, остается дописать из стека знаки операций (если они есть) в строку, содержащую выражение в постфиксной форме.

```
Procedure Solve(a:String;Var z:String);
  Var head:pt;
      i:Integer;
      w:Char;
  Begin
    <инициализация стека, head – указатель
    на вершину стека>;
    z:=''; {Результат – пустая строка}
    i:=1;
    While i<=<длина строки a> Do Begin
      If <a[i] - не пробел> Then
        Begin {Пропускаем пробелы}
          If Not(a[i] In ['+','-','*','/','^',
                          '(',')'])
          {Операция In проверяет принадлежность
          элемента заданному множеству, в данном
          случае - символа строки множеству из
          знаков операций и круглых скобок}
          Then z:=z+a[i] {Очередной символ a
          не является знаком операции или скобкой}
          Else
```

```
      If a[i] In ['+','-','*','/','^','(']
       Then Push(head,a[i])
       Else
        If a[i]=')' Then Begin
       {Считываем из стека до символа "("}
         Pop(head,w);
         While w<>'(' Do Begin
          z:=z+w;
          Pop(head,w);
         End;
        End;
     End;
    i:=i+1;
   End;
   While Not Empty Do Begin {Дополняем строку
   символами операций, запомненных в стеке}
    Pop(head,w);
    z:=z+w;
   End;
End;
```

Пусть нам теперь требуется вычислить выражение, представленное в постфиксном виде. Упростим эту задачу предположением о том, что выражение состоит из цифр от 1 до 9 и знаков операций «+», «–», «*», «/», ибо наша основная цель — понять логику использования стека. В этом случае (как и в предыдущем), просматривая строку, мы анализируем очередной символ:

- если это цифра, то записываем ее в стек;
- если это знак, то считываем два элемента из стека, выполняем математическую операцию, определяемую этим знаком, и заносим результат в стек.

После просмотра всей строки в стеке должен оставаться только один элемент, который и является значением выражения.

Прежде чем рассматривать основную логику решения этой задачи, решим чисто технические вопросы. Во-первых, для преобразования символьного представления цифры в соответствующее числовое значение воспользуемся процедурой `Val(s,x,k)`, которая символьное представление числа `s` (в том числе вещественное) переводит в соот-

ветствующее числовое выражение x. При этом k = 0, если такое преобразование возможно; в противном случае k <> 0. Во-вторых, выполнение операции (символ ch) с двумя операндами (a, b — вещественные числа) может быть реализовано так:

```
Procedure Operation(ch:Char;a,b:Real; Var c:Real);
  Begin
    Case ch Of
      '+':c:=a+b;
      '-':c:=b-a;
      '*':c:=a*b;
      '/':c:=b/a;
    End;
  End;
```

Несмотря на практическое совпадение логики вычисления выражения с логикой перевода выражения из инфиксной в префиксную форму, мы приведем ее, ибо она служит основой для выполнения упражнений, приведенных в конце раздела:

```
Procedure Solve(a:String;Var pp:Boolean;Var z:Real);
{Значение переменной pp, равное False, говорит о том,
 что выражение не может быть вычислено корректно}
  Var head:pt; {Предполагаем, что для реализации
                стека использован список}
      i,k:Integer;
      r,w:Real;
  Begin
    head:=nil;
    pp:=True;
    i:=1;
    While (i<=Length(a)) And pp Do Begin
    {Пока значение i меньше или равно длине строки a
    и эта строка правильная, выполняем операторы
    тела цикла}
      If a[i]<>' ' Then Begin {Пропускаем пробелы}
        If Not(a[i] In ['+','-','*','/']) Then Begin
        {Если символ - не знак операции, то
        преобразуем его и в случае корректности
        преобразования записываем в стек}
```

```
          Val(a[i],r,k);
          If k=0 Then Push(head,r)
          Else pp:=False;
        End
        Else Begin {Обрабатываем операцию: берем из
        стека два операнда, выполняем операцию и
        результат записываем в стек}
          Pop(head,r)
          Pop(head,w)
          Operation(a[i],r,w,r);
          Push(head,r);
        End;
      End;
      i:=i+1;
    End;
    Pop(head,z); {Считываем результат}
  End;
```

Упражнения

1. Преобразуйте из инфиксной в префиксную и постфиксную формы следующие выражения:
   - $a + b - c$
   - $(a + b) * (c - d) \hat{} e * f$
   - $(a + b) * (c \hat{} (d - e) + f) - g$
   - $a + ((b - c) * (d - e) + f) / g) \hat{} (h - j)$
2. Преобразуйте префиксные выражения в инфиксную форму:
   - $+ - a\ b\ c$
   - $+ a - b\ c$
   - $+ + a - \hat{} * b\ c\ d / + e\ f * g\ h\ i$
   - $+ - \hat{} a\ b\ c * d * e\ f\ g$
3. Преобразуйте постфиксные выражения в инфиксную форму:
   - $a\ b + c -$
   - $a\ b\ c + -$
   - $a\ b - c + d\ e\ f - + \hat{}$
   - $a\ b\ c\ d\ e - + \hat{} * e\ f * -$
   - $a\ b + c - b\ a + c \hat{} -$
   - $a\ b\ c + * c\ b\ a - + *$

4. В рассмотренной логике преобразования выражения из инфиксной в постфиксную форму мы считали, что идентификатор переменной состоит из одного символа. Снимите это ограничение.

5. В логике вычисления выражения в постфиксной форме обработка выражения: «3.1 6 *» дает ответ 6. Измените ее так, чтобы в подобных случаях выдавалось сообщение об ошибке.

6. В логике вычисления выражения в постфиксной форме обработка выражения «3 4 6 5 +» дает результат 11, что явно не соответствует действительности, так как выражение записано ошибочно. Доработайте программу для отслеживания таких ошибок.

7. Доработайте логику вычисления выражения в постфиксной форме так, чтобы обрабатывалась и операция возведения в степень: «^».

8. Доработайте логику вычисления выражения в постфиксной форме так, чтобы обрабатывались вещественные числа.

9. Напишите программу для преобразования выражения из инфиксной формы в префиксную.
Примечание. Инфиксная форма анализируется справа налево; префиксная форма создается аналогичным образом.

10. Напишите программу для вычисления выражения в префиксной форме.

11. Напишите программу, вычисляющую строку в инфиксной форме.
Примечание. Здесь не требуется преобразовывать выражение в постфиксную форму! Необходимо вычислять его без всякого преобразования. Целесообразно использовать для этого два стека: один — для операндов, а другой — для операций.

12. Напишите программы для преобразования строки:
    - строки в префиксной форме в строку в постфиксной форме;
    - строки в постфиксной форме в строку в префиксной форме;

- строки в префиксной форме в строку в инфиксной форме;
- строки в постфиксной форме в строку в инфиксной форме.

3.5. Стек и рекурсивные процедуры

И обнаружил микроскоп,
Что на клопе бывает клоп,
Питающийся паразитом.
На нем — другой, ad infinitum...

Джонатан Свифт

Такая структура данных, как стек, является основополагающей при реализации *рекурсивных процедур* в языках программирования. Рекурсивные вызовы процедур упрощают логику программных решений, но рекурсивная реализация, по сравнению с итерационной, увеличивает время работы. В стеке (его называют *стеком адресов возврата*[1]) при рекурсивном вызове сохраняется информация о текущих локальных переменных и значениях параметров процедуры, а также адрес возврата, т. е. адрес оператора (*точки возврата*), на который должно перейти управление после завершения очередного прохода процедуры. При выходе из процедуры эта информация о последнем проходе процедуры восстанавливается, а управление вычислительным процессом передается на точку возврата, и выполнение процедуры, прерванной рекурсивным вызовом, продолжается.

Если рекурсия доступна и эффективна в конкретной системе программирования, то в принципе любая итерационная управляющая структура может быть заменена на соответствующую рекурсивную логику. На рис. 3.4 продемонстрирована схема замены цикла `While` (а значит — и любой циклической конструкции) на его рекурсивный эквивалент.

Каждый вызов такой процедуры эквивалентен одному из повторений цикла `While`. Если «*условие цикла*» истинно, то перед рекурсивным вызовом выполняются операторы из «*тела цикла*». Если же это условие ложно, то процедура и все ее предыдущие активации завершают работу.

[1] Правда, этот термин нельзя считать общепринятым.

Произвольный цикл While

```
While <условие цикла> Do
  <тело цикла>
```

а)

Эквивалентная рекурсивная процедура

```
Procedure Рекурсивный цикл;
  Begin
    if  <условие цикла> Then Begin
      <тело цикла>;
      Рекурсивный цикл;
    End;
  End;
```

б)

Рис. 3.4. Цикл While (*а*) и его рекурсивный эквивалент (*б*)

Итак, любой итеративный цикл может быть заменен рекурсией. Однако обратное утверждение, вероятно, неверно.

При работе с рекурсивной логикой следует быть очень внимательным. Скажем, переставив местами в ветви Then процедуры Рекурсивный цикл (см. рис. 3.4*б*) вызов процедуры и операторы тела цикла, мы получим бесконечную рекурсию: у процедуры нет условий (точнее, они вырабатываются в теле цикла, идущем уже после рекурсивного вызова) для завершения собственной работы.

Пример. Поиск наибольшего общего делителя двух натуральных чисел с помощью алгоритма Евклида сводится к последовательному вычитанию из большего числа меньшего до тех пор, пока одно из них не станет равным нулю. Так, если $a = 36$, $b = 24$, то на первой итерации мы получаем $a = 12$, $b = 24$; на второй — $a = 12$, $b = 12$ и на третьей — $a = 12$, $b = 0$. Цикл завершает работу, а наибольший общий делитель равен 12.

```
Function Nod(a,b:Integer):Integer;
  Begin
    While (a<>0) And (b<>0) Do
      If a>b Then a:=a-b
      Else b:-b-a;
    Nod:=a+b;
  End;
```

Рекурсивный вариант повторяет эти действия, но в силу простоты операторов из тела цикла (операция вычитания) они просто включены в параметры вызываемой функции:

```
Function Nod(a,b:Integer):Integer;
  Begin
    If (a=0) Or (b=0) Then Nod:=a+b
    Else If a>b Then Nod:=Nod(a-b,b)
                Else Nod:=Nod(a,b-a);
  End;
```

Для данного примера в стеке адресов возврата хранятся значения `a`, `b` и адрес оператора `End`.

Рассмотрим теперь более сложный пример, введя для большей содержательности ряд новых понятий.

Граф нестрого определяется как совокупность точек плоскости, называемых *вершинами*, часть из которых соединена отрезками, называемыми *ребрами* или *дугами*.

Строгое определение: *граф* $G = (V,E)$ определяется парой множеств: конечным множеством V, называемым *вершинами*, и множеством E, содержащим пары вершин, называемых *ребрами*. Количество вершин (*мощность множества*) мы будем обозначать как $|V| = n$, а количество ребер — $|E| = m$. Вершины графа обычно помечаются целыми числами (метками) от 1 до n. Иногда, когда этого требует задача, помечаются и ребра графа. Вершины u и v, связанные между собой ребром (u, v), называют *смежными*.

Путь, или *маршрут*, от вершины u к вершине v определяется как последовательность смежных вершин, начинающаяся в u и заканчивающаяся в v. Если все составляющие путь ребра графа различны, то такой путь называется *простым*. *Длина пути* равна количеству ребер, через которые он проходит.

Одним из способов описания графа в памяти компьютера является *матрица смежности*, например A — двумерный массив размерности $n \times n$, определяемый как:

$$A[i,j] = \begin{cases} 1, \text{если вершины с номерами } i \text{ и } j \text{ смежны}, \\ 0, \text{если вершины с номерами } i \text{ и } j \text{ не смежны}. \end{cases}$$

Нашей задачей является просмотр вершин графа. Допустим, что с каждой вершиной связана некая информация, и нам требуется найти определенные сведения.

Идея такого поиска (он называется *поиском в глубину*) следующая. Поиск начинается с некоторой фиксирован-

ной вершины v. Находится и выбирается вершина u, смежная с v. Этот процесс повторяется с вершиной u (а это — уже рекурсивная схема, ибо действия с вершиной u точно такие же, как и с вершиной v). Если на очередном шаге мы работаем с вершиной q, и нет вершин, смежных с q и не рассмотренных ранее (новых), то мы возвращаемся из вершины q к вершине, которая была до нее. Если это вершина v, то процесс просмотра закончен. Очевидно, что для фиксации признака, просмотрена вершина графа или нет, нам потребуется структура данных типа `Nnew: Array[1..n] Of Boolean`.

Пример. Пусть дан граф, приведенный на рис. 3.5 и описанный матрицей смежности `A`. Поиск начинается с первой вершины — помечаем ее как просмотренную (`Nnew[1]:= False`). Находим первую смежную непросмотренную вершину — это вершина с номером 4. Помечаем ее и продолжаем поиск вершин из четвертой вершины. Переходим к вершине с номером 6, а затем — 2 и 3. Из вершины с номером 3 продолжить процесс просмотра нельзя — мы не находим непросмотренных вершин. Осуществляем возврат к вершине с номером 2 и из нее продолжаем поиск не про-

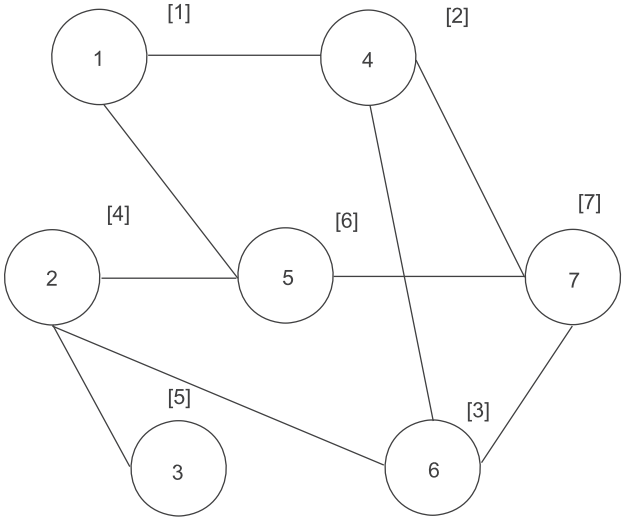


Рис. 3.5. Пример графа (в квадратных скобках указана очередность просмотра вершин при поиске в глубину)

смотренных ранее вершин. Переходим к вершине с номером 5, а затем — 7. После этого осуществляется возврат к вершине 5, затем 2, 6, 4 и 1. Мы вернулись к той вершине, с которой начали просмотр, — значит, обработка по методу поиска в глубину завершена.

Формализованная запись этой рекурсивной логики имеет вид:

```
Procedure Pg(v:Integer);
{Массивы Nnew и A - глобальные}
  Var j:Integer;
  Begin
    Nnew[v]:=False;
    Write(v:3);
    For j:=1 To n Do
      If (A[v,j]<>0) And Nnew[j] Then Pg(j);
  End;
```

Приведем также фрагмент основной логики:

```
For i:=1 To n Do Nnew[i]:=True;
For i:=1 To n Do
  If Nnew[i] Then Pg(i);
```

А сейчас реализуем нерекурсивную реализацию поиска в глубину. Глобальные структуры данных — прежние: `A` — матрица смежностей, `Nnew` — массив признаков. При рекурсивном вызове процедуры `Pg` ей передается номер найденной и еще не просмотренной вершины (`j`). Если таковой нет, то на основании последней записи в стек адресов возврата осуществляется выход в процедуру, инициировавшую вызов, и в ней продолжается выполнение цикла `For` со следующим значением `j`.

Промоделировать эту ситуацию можно путем введения стека (`St`, указатель на вершину `head`) для хранения найденных вершин `j` и организации цикла по элементам стека (до тех пор, пока стек не пуст, — продолжаем обработку).

Возможный вариант реализации такой процедуры имеет вид:

```
Procedure Pgn(v:Integer);
{Просмотр начинаем с вершины v}
  Var St:Array[1..n] Of Integer;
      head, t, j:Integer;
      pp:Boolean;
  Begin
    For j:=1 To n Do St[j]:=0;
    head:=1;
    St[head]:=v;
    Nnew[v]:=False;
    While head<>0 Do Begin
    {Пока стек не пуст}
     t:=St[head];
     {Выбор "самой верхней" вершины из стека}
     j:=1;
     pp:=False;
     Repeat
      If (A[t,j]<>0) And Nnew[j] Then pp:=True
      Else j:=j+1;
     Until pp Or (j>=n);
     {Найдена новая вершина, либо все вершины,
     связанные с данной вершиной, просмотрены}
     If pp Then Begin
      head:=head+1;
      St[head]:=j;
      Nnew[j]:=False;
      {Добавляем номер вершины в стек}
     End
     Else head:=head-1;
     {Возвращаемся к предыдущей вершине}
    End;
  End;
```

Упражнения

1. Работа процедуры `Pgn` не полностью идентична работе процедуры `Pg`. Так, при возврате к ранее просмотренной вершине цикл по `j` в `Pg` продолжается со следующего значения `j`, а в `Pgn` — всегда с единицы. Каких ресурсов потребует устранение этого несоответствия? Доработайте программу.

2. Напишите нерекурсивный вариант следующей процедуры вывода двоичного представления натурального числа:

```
Procedure Rec(n:Integer);
  Begin
    If n>1 Then Rec(n Div 2);
    Write(n Mod 2);
  End;
```

3. Напишите рекурсивный вариант следующей процедуры поиска максимального элемента в массиве A (n элементов):

```
Procedure Search_Max(Var x:Integer);
 Var i:Integer;
 Begin
  x:=A[1];
  For i:=2 To n Do
   If A[i]>x Then x:=A[i];
 End;
```

4. Каждое число Фибоначчи равно сумме двух предыдущих чисел, при условии, что первые два равны единице: 1, 1, 2, 3, 5, 8, 13, 21, ... В общем виде n-е число Фибоначчи определяется по формуле:

$$F(n)=\begin{cases}1 \text{ при } n=1 \text{ или } n=2,\\ F(n-1)+F(n-2) \text{ при } n>2.\end{cases}$$

Рекурсивная функция его вычисления имеет вид:

```
Function Fib(n:Integer):Integer;
  Begin
    If n<=2 Then Fib:=1
    Else Fib:=Fib(n-1)+Fib(n-2);
  End;
```

Напишите нерекурсивный вариант функции вычисления n-го числа Фибоначчи.

5. Рекурсивный вариант процедуры Gen выводит все последовательности длины n, составленные из чисел от 1 до k. Последовательности хранятся в массиве A. Напишите нерекурсивный вариант этой процедуры.

```
Procedure Gen(t:Integer);
  Var i:Integer;
  Begin
    If t=n+1 Then <вывод элементов массива A>
    Else
      For i:=1 To k Do Begin
        A[t]:=i;
        Gen(t+1);
      End;
  End;
```

6. Рекурсивный вариант процедуры `Gen` выводит все возрастающие последовательности длины `n`, составленные из чисел от **1** до `k` (`n` ⩽ `k`). Последовательности хранятся в массиве `A[0..n]` (`A[0]` = **0**). Напишите нерекурсивный вариант этой процедуры:

```
Procedure Gen(t:Integer);
  Var i:Integer;
  Begin
    If t=n+1 Then <вывод элементов массива A>
    Else
      For i:=A[t-1]+1 To t-n+k Do Begin
        A[t]:=i;
        Gen(t+1);
      End;
  End;
```

7. *Задача о лабиринте*. Дано клеточное поле, часть клеток которого занята препятствиями. Необходимо попасть из некоторой заданной клетки в другую заданную клетку, последовательно перемещаясь по клеткам. Классический перебор клеток при этом выполняется по правилам, предложенным в 1891 г. Э. Люка в «Математических досугах»:

- в каждой клетке выбирается еще не исследованный путь;
- если из исследуемой в данный момент клетки нет таких путей, то возвращаемся на один шаг назад (в предыдущую клетку) и пытаемся выбрать другой путь.

Схема рекурсивного решения:

```
Const NMax=...; {Максимальный размер поля}
      dx:Array[1..4] Of Integer=(1,0,-1,0);
      dy:Array[1..4] Of Integer=(0,1,0,-1);
Var A:=Array[0..Nmax+1,0..Nmax+1] Of Integer;
    {На границе поля при инициализации данных
    записываются ненулевые значения.
    Препятствие - ненулевой элемент A,
    свободная клетка - нулевой элемент A}
    xn,yn,xk,yk,n:Integer; {xn, yn - координаты
    начальной клетки (входа в лабиринт);
    xk, yk - координаты конечной клетки
    (выхода из лабиринта)}

Procedure Solve(x,y,k:Integer);
{k - номер шага, x,y - координаты клетки}
  Var i:Integer;
  Begin
    A[x,y]:=k;
    If (x=xk) And (y=yk) Then <вывод решения>
    Else
      For i:=1 To 4 Do
        If A[x+dx[i],y+dy[i]]=0 Then
          Solve(x+dx[i],y+dy[i],k+1);
    A[x,y]:=0;
  End;
```

Разработайте нерекурсивный вариант решения этой задачи.

8*. *Задача коммивояжера*. Классическая формулировка этой задачи известна уже более 200 лет: имеется n городов, расстояния между которыми заданы; коммивояжеру необходимо выйти из какого-то города, посетить остальные $n - 1$ городов точно по одному разу и вернуться в исходный город. При этом маршрут коммивояжера должен иметь минимальную длину (стоимость).

Рекурсивный вариант решения:

```
Const Max=...;
Var A:Array[1..Max,1..Max] Of Integer;
```

```
    {Матрица расстояний между городами}
    Way, BestWay:Array[1..Max] Of Byte;
    {Хранится текущее решение и лучшее решение}
    Nnew:Array[1..Max] Of Boolean;
    {Значение элемента массива False говорит
    о том, что в соответствующем городе
    коммивояжер уже побывал}
    BestCost:Integer;{Стоимость лучшего решения}

Procedure Solve(v,Count:Byte;Cost:Integer);
{v - номер текущего города; Count - счетчик
 количества пройденных городов; Cost -
 стоимость текущего решения}
  Var i:Integer;
  Begin
    If Cost>BestCost Then Exit;
    {Стоимость текущего решения превышает
    стоимость лучшего из ранее полученных}
    If Count=n Then Begin
      Cost:=Cost+A[v,1];
      Way[n]:=v;
      {Последний город пути. Добавляем к решению
      стоимость перемещения в первый город
      и сравниваем его с лучшим из ранее
      полученных}
      If Cost<BestCost Then Begin
      {Найдено лучшее решение}
        BestCost:=Cost;
        BestWay:=Way;
      End;
      Exit;
    End;
    Nnew[v]:=False; {Город с номером v пройден,
    записываем его номер в путь коммивояжера}
    Way[Count]:=v;
    For i:=1 To n Do
      If Nnew[i] Then Solve(i,Count+1,Cost+A[v,i]);
      {Поиск города, в который коммивояжер может
      пойти из города с номером v}
      Nnew[v]:=True; {Возвращаем город с номером
      v в число непройденных}
  End;
```

Разработайте нерекурсивный вариант реализации этой логики.

Методические комментарии

В этом разделе дан новый вариант[1)] изложения этой обязательной темы, присутствующий в любой книге по информатике, связанной со структурами данных. В качестве основы для написания раздела 3.4 автором использован материал из книги Й. Лэнгсама, М. Огейстайна и А. Тененбаума[2)].

---

1) *Окулов С. М.* Основы программирования. 2-е изд. М.: БИНОМ. Лаборатория знаний, 2005.

2) *Лэнгсам Й., Огейстайн М., Тененбаум А.* Структуры данных для персональных ЭВМ. М.: Мир, 1989. С. 152–168.

Глава 4

Очередь

> Словом, можно согласиться с Алисой: играть в крокет здесь было трудновато! Игроки били по шарам все сразу, никто не соблюдал очереди хода, зато все непрерывно скандалили и спорили.
>
> *Льюис Кэрролл*

4.1. Определение и реализация очереди с использованием списков

> — Итак, с чего же мы начнем, мистер Сайрес? — спросил Пенкроф на следующее утро.
>
> — С самого начала, — ответил Сайрес Смит.
>
> *Жюль Верн, «Таинственный остров»*

Очередь — это упорядоченный набор элементов, в котором извлечение элементов происходит с одного его конца, а добавление новых элементов — с другого (аббревиатура *FIFO* — *first-in-first-out*: «первым вошел — первым вышел»).

Основные операции с очередью:

- запись элемента в очередь (если это возможно, т. е. нет переполнения структуры данных, выбранной для хранения элементов очереди);
- чтение элемента из очереди (если очередь не пуста).

В очереди, в силу ее определения, доступны две позиции: ее начало, откуда извлекаются (обслуживаются) элементы, и ее конец, куда заносятся новые элементы (ставятся в очередь на обслуживание). Вследствие этого для работы с очередью независимо от выбранной структуры данных для хранения ее элементов целесообразно ввести две переменные, которые мы назовем `head` («голова») и `tail` («хвост») и значения которых определяют, соответствен-

но, первый элемент для обслуживания и место для записи нового элемента.

Выбор линейного списка в качестве структуры для реализации такого абстрактного типа данных, как очередь, сводит операцию обслуживания к чтению первого элемента списка, а постановку в очередь — к записи элемента в конец списка. Разумеется, можно было бы отказаться от использования указателя `tail`, но тогда при каждой записи потребовалось бы находить последний элемент списка, что вряд ли разумно по временны́м характеристикам операции.

Описание элемента очереди (аналогично тому, как это было сделано для стека):

```
Type pt=^elem;
     elem=Record
       data: Integer;
       next: pt;
     End;
Var head, tail:pt;
```

Если очередь пуста (в ней нет элементов), то значение указателей `head` и `tail` равно `nil`. Эта проверка реализуется функцией `Empty` с результатом логического типа:

```
Function Empty(head:pt):Boolean;
  Begin
    If head=nil Then Empty:=False
    Else Empty:=True;
  End;
```

Процедура записи элемента в непустую очередь должна содержать два параметра, первый из которых определяет указатель на конец очереди, а второй — записываемое значение:

```
Procedure Queue_Write(Var tail:pt; x:Integer);
  Var t:pt;
  Begin
    New(t);
    t^.data:=x;
    t^.next:=nil;
    tail^.next:=t;
    tail:=t;
  End;
```

Извлечение элемента из непустой очереди сводится к чтению первого элемента списка:

```
Procedure Queue_Read(Var head:pt; Var x:Integer);
  Var t:pt;
  Begin
    x:=head^.data;
    t:=head;
    head:=head^.next;
    Dispose(t);
  End;
```

Упражнения

1. Измените процедуру `Queue_Write` так, чтобы запись в очередь была корректна и для случая, когда очередь пуста.
2. Предположим, что указателя `tail` нет, и каждый раз при записи элемента в очередь приходится находить последний элемент списка. Реализуйте операцию записи в очередь для этого случая.
3. Измените операции работы с очередью при ее реализации с помощью двусвязного списка.
4. Измените операции работы с очередью при ее реализации с помощью двусвязного списка с «фиктивным» элементом.

4.2. Реализация очереди с помощью массива

> Чем фундаментальнее закономерность, тем проще ее можно сформулировать.
>
> *Петр Капица*

Очередь — это упорядоченный определенным образом набор данных. Пусть для ее реализации используется массив статического типа с именем `A` (`A:Array[0..n-1] Of Integer`), запись и чтение в который осуществляются по правилам очереди (обратите внимание: индексация элементов массива начинается с нулевого элемента).

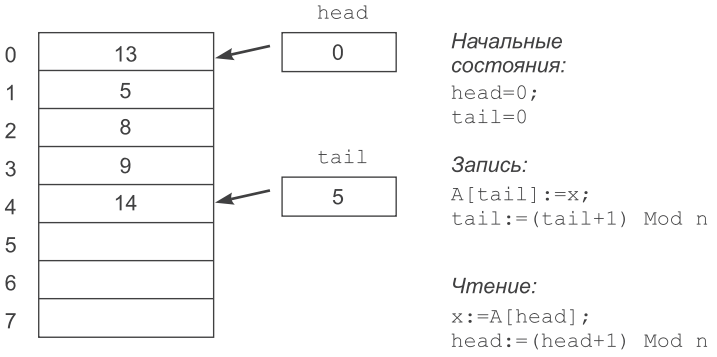


Рис. 4.1. Пример очереди

Пример. На рис. 4.1 приведено состояние очереди в определенный момент времени (n = 8). Элементы записывались в очередь так: 13, 5, 8, 9, 14, начиная с нулевого элемента массива A. Чтения — не было. Доступный для извлечения (обслуживания) элемент очереди — 13, он определяется значением указателя (head). Место для записи (постановки в очередь) определяется значением указателя tail.

Естественным желанием является отказ от сдвигов элементов массива A при осуществлении операций с очередью, — например, после чтения числа 13 элементы массива сдвигаются вверх на одну позицию. Это приводит к понятию «кольца», или «циклического массива», т. е. вычисление значений head и tail при чтении и записи элементов осуществляется по модулю числа n (рис. 4.1). Так, в нашем примере после записи в ячейку с номером 7 осуществляется запись в ячейку с индексом 0, если она, конечно, свободна. Аналогично — и при чтении элементов.

Открытым является вопрос о фиксации пустоты (Empty) и полноты (Full) очереди. Пусть функция Empty фиксирует факт совпадения значений head и tail — очередь пуста, если head = tail.

Продолжим рассмотрение примера очереди (рис. 4.1). На рис. 4.2*а* показано состояние очереди после чтения элементов 13, 5, 8, 9 и 14. Значение указателей — head = tail. На рис. 4.2*б* первоначально выполнено чтение элементов 13, 5, а затем запись 25, 6, 9, 10, 45. Очередь полна, значения указателей — head = tail. Тем самым,

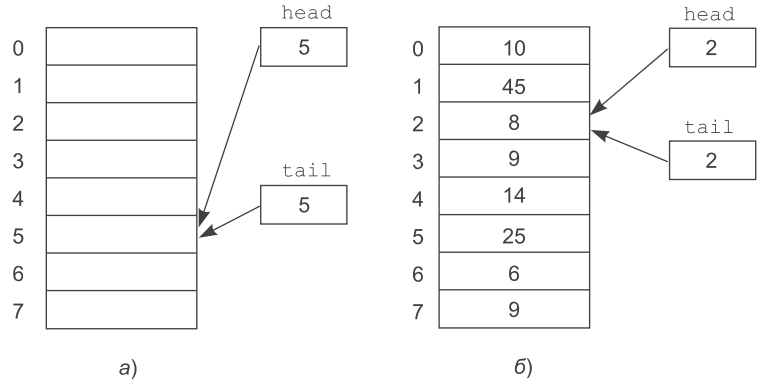


Рис. 4.2. Очередь: *а*) пуста — `head = tail`; *б*) заполнена — `head = tail`

мы имеем два неразличимых на уровне значений переменных `head` и `tail` состояния очереди.

Одним из возможных способов разрешения этой ситуации является (что естественно) ввод «флага» или, другими словами, дополнительной переменной логического типа (`p`) для фиксации факта «кто кого догнал». Если вы обратили внимание, то в первом случае — когда очередь пуста — `head` «догнал» `tail` (рис. 4.2*а*), а во втором — когда очередь полна — `tail` «догнал» `head` (рис. 4.2*б*). Начальное значение переменой `p` равно `True`, а при переходе как того, так и другого указателей (`heap` и `tail`) через значение `n` она изменяет свое значение на противоположное. Так, для рассматриваемого примера, в первом случае (рис. 4.2*а*) `p = True`, а во втором (рис. 4.2*б*) — `p = False`, так как мы один раз перешли через значение `n` (указатель `tail`).

Итак, операции работы с очередью могут иметь следующий вид:

```
Function Queue_Full(head,tail:Integer;p:Boolean):
Boolean;
  Begin
    If (head=tail) And Not p Then Queue_Full:=True
    Else Queue_Full:=False;
  End;

Function Queue_Empty(head,tail:Integer; p:Boolean):
Boolean;
```

```
  Begin
    If (head=tail) And p Then Queue_Empty:=True
    Else Queue_Empty:=False;
  End;

Procedure Queue_Write(Var tail:Integer;
Var p:Boolean; x:Integer);
  Begin
    A[tail]:=x;
    tail:=(tail+1) Mod n;
    If tail=0 Then p:=Not p;
  End;

Procedure Queue_Read(Var head:Integer;
Var p:Boolean; Var x:Integer);
  Begin
    x:=A[head];
    head:=(head+1) Mod n;
    If head=0 Then p:=Not p;
  End;
```

Упражнения

1. На рис. 4.3 показан пример реализации очереди с помощью массива, когда началом очереди считается последний элемент. Первоначально в эту очередь были последовательно записаны элементы 13, 5, 8, 9 и 14, а затем из нее прочитали два элемента — 13 и 5.

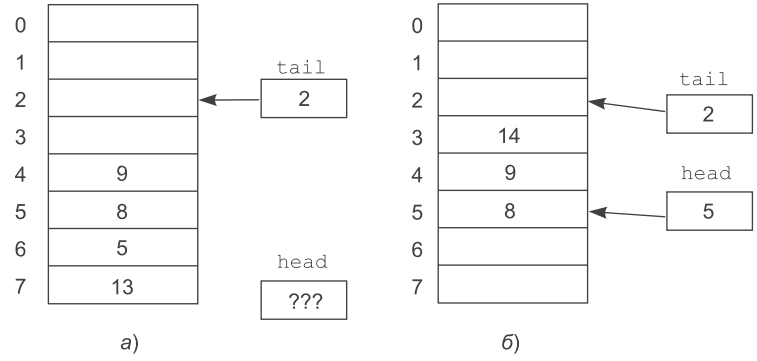


Рис. 4.3. Пример реализации очереди: а) записаны элементы 13, 5, 8, 9 и 14; б) прочитаны элементы 13 и 5

Определите начальные значения переменных `head`, `tail` и `p` и напишите вариант реализации операций `Queue_Full`, `Queue_Empty`, `Queue_Write` и `Queue_Read` для этого случая.

2. На рис. 4.4 показан еще один способ реализации очереди. Чтение элемента из этой очереди всегда осуществляется из первой ячейки и приводит к сдвигу элементов очереди вверх на одну позицию по массиву. В этом случае отпадает необходимость в указателе `head`, а логика анализа пустоты и полноты очереди становится «прозрачной». Однако время обслуживания элемента при этом уже не константное, а равно $O(n)$.

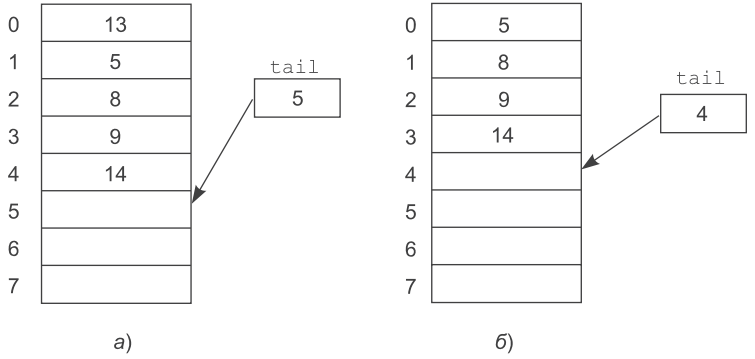


Рис. 4.4. Способ реализации очереди, при котором чтение элементов всегда производится из первой ячейки массива

Реализуйте операции работы с очередью для данного способа ее организации.

3. На рис. 4.5 приведен пример, иллюстрирующий способ реализации очереди, при котором чтение элемента всегда осуществляется из последней ячейки массива.

Реализуйте операции работы с очередью для данного способа ее организации.

4. Можно ли решить проблему проверки пустоты и полноты очереди в случае использования для ее реализации циклического массива, задавая при этом другие начальные значения для `head` и `tail`? Например, если

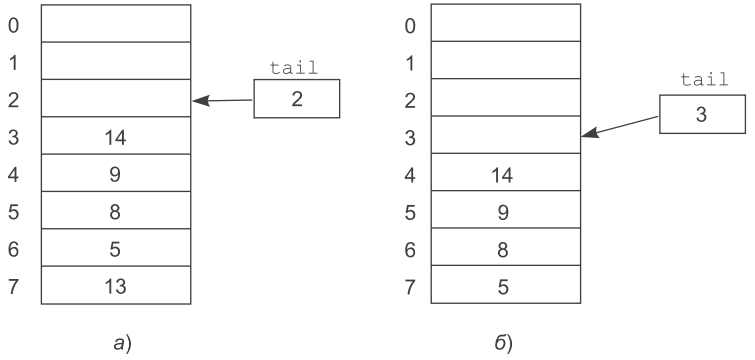


Рис. 4.5. Способ реализации очереди, при котором чтение элементов всегда производится из последней ячейки массива

`head` = **0** (индекс первого элемента очереди) и `tail` = **1** (индекс свободной ячейки, в которую будет помещен следующий добавляемый к очереди элемент). Разумеется, при этом необходимо модифицировать процедуры чтения и записи. Исследуйте этот путь решения проблемы.

5. Проблему проверки пустоты и полноты очереди можно решить, «пожертвовав» одной ячейкой массива, т. е. когда для чтения и записи доступна `n - 1` ячейка. Запись же в ячейку с индексом `n` говорит о переполнении очереди. Например, если массив из 100 элементов объявлен как очередь, то такая очередь может содержать до 99 элементов. Попытка же разместить в очереди 100-й элемент приведет к ее переполнению. Для каких способов реализации очереди через массив применим данный прием?

6. Существует способ организации очереди через циклический массив, в котором фиксируется место первого элемента и количество элементов в очереди. Реализуйте операции работы с очередью для данного способа ее организации. Как в этом случае решить проблему проверки пустоты и полноты очереди?

7*. *Очередь с двусторонним доступом* (или *дек*) — это структура данных, в которой добавлять и удалять эле-

менты можно с обоих концов. Разработайте реализации такой структуры данных с использованием двусвязных списков и массивов. Сравните эти реализации и дайте рекомендации по их использованию.

8. За один просмотр файла `f`, элементами которого являются целые числа, требуется без использования дополнительных файлов переписать его элементы в другой файл так, чтобы сначала в него были записаны все числа, меньшие заданного числа `a`, затем все числа из отрезка [`a`, `b`], а затем — все остальные. Взаимный порядок чисел в каждой из групп должен быть сохранен.

Примечание. В решении этой задачи числа последовательно считываются из файла. Если очередное число меньше `a`, то оно сразу записывается в новый файл; если оно принадлежит отрезку [`a`, `b`], то оно заносится в первую очередь; иначе — во вторую очередь. После завершения чтения в выходной файл записываются числа из первой, а затем — из второй очереди.

9. Содержимое текстового файла `f`, разделенного на строки, требуется переписать в текстовый файл `g`, перенося при этом в конец каждой строки все входящие в нее цифры с сохранением их взаимного исходного порядка.

10*. Выведите в порядке возрастания первые `n` натуральных чисел, в разложение которых на простые множители входят только числа 2, 3 и 5.

Примечание. Для этого вводятся три дека: `dec1`, `dec2` и `dec3`, в которых хранятся числа, кратные 2, 3, 5 и вывод которых еще не выполнен. Из этих деков считываются первые элементы. Минимальный элемент (`t`) выводится в результирующий файл; не минимальные — возвращаются в соответствующие деки и с другого конца этих деков добавляются числа `2*t`, `3*t`, `5*t`. Логику изменения содержимого деков можно проиллюстрировать при помощи табл. 4.1.

Таблица 4.1

Вывод элемента	dec1	dec2	dec3
–	2	3	5
2	4	3, 6	5, 10
3	4, 6	6, 9	5, 10, 15
4	6, 8	6, 9, 12	5, 10, 15, 20
5	6, 8, 10	6, 9, 12, 15	10, 15, 20, 25
6	8, 10, 12	9, 12, 15, 18	10, 15, 20, 25, 30
8	10, 12, 16	9, 12, 15, 18, 24	10, 15, 20, 25, 40

Методические комментарии

Этот раздел обязателен для любой книги по данной проблематике, однако обычно такой материал дается слишком «бегло», не «под занятие» или самостоятельное изучение. При этом практически всегда имеется ссылка на книгу Д. Кнута, трудно читаемую и не очень удобную для реальной работы как со школьником, так и студентом. Автор позволил себе не сохранять эту традицию…

Глава 5

Деревья

> И тут-то она заметила, что в одном дереве есть дверь и эта дверь открывается прямо в дерево. «Как интересно! — подумала Алиса. — А если войти, наверно, будет еще интересней. Пожалуй, войду!»
>
> *Льюис Кэрролл*

5.1. Основные понятия

> Что значит имя? Роза пахнет розой,
> Хоть розой назови ее, хоть нет.
>
> *Вильям Шекспир*

В раздел 3.5 было введено понятие графа. Продолжим знакомство с ним, ибо деревья являются графами с определенными свойствами.

Граф называется *связным*, если для любой пары его вершин u и v существует путь от u к v. Под *циклом* понимается простой путь положительной длины, который начинается и заканчивается в одной и той же вершине. Граф, не содержащий циклов, называют *ациклическим*.

Тогда следующие определения *дерева* эквивалентны. Граф $G = (V, E)$, где $|V| = n$ и $|E| = m$, является деревом (обозначим его как T), если:

1) G — связный граф и $m = n - 1$;
2) G — ациклический граф и $m = n - 1$;
3) любые две не совпадающие вершины графа G соединяет единственная простая цепь;
4) G — ациклический граф, обладающий свойством, что если какую-либо пару несмежных вершин соединить ребром, то полученный граф будет содержать ровно один цикл.

Третье из приведенных выше определений дерева («единственная простая цепь») позволяет выбрать произ-

вольную вершину дерева и указать, что она является его *корнем*, а дерево является *корневым*. При изображении корневого дерева корень обычно рисуют вверху (нулевой уровень дерева), а затем (под ним) идут смежные с корнем вершины (первый уровень) и т. д. Пример такого преобразования приведен на рис. 5.1.

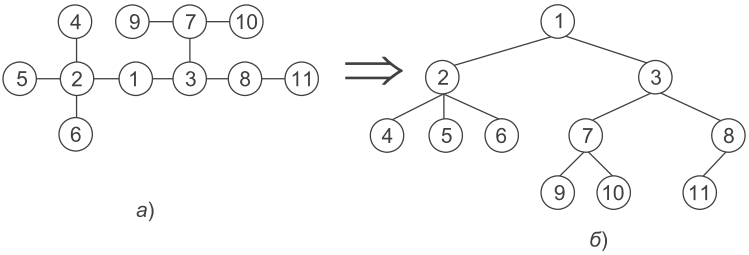


Рис. 5.1. Преобразование дерева (а) в корневое дерево (б)

В дереве T для любой вершины v существует простой путь от корня дерева до нее. Все вершины этого пути называются *предками* вершины v, а последняя вершина этого пути u (та, для которой есть ребро (u,v)) — *родителем*. Для вершины же u вершина v является *потомком*. Вершина, у которой нет потомков, определяется как *лист*.

Под *глубиной* вершины v понимается длина простого пути от корня до v. *Высота дерева* — это длина наибольшего простого пути от его корня до одного из листьев.

Пример. Для дерева T на рис. 5.1*б*:

- у вершины с меткой 10 предками являются вершины 7, 3, 1, а вершина 7 является ее родителем;
- вершины с метками 4, 5, 6, 9, 10, 11 — листья;
- вершина с меткой 11 является потомком вершины 8;
- вершина с меткой 5 имеет глубину 2, а высота дерева равна 3.

Упражнения

1. Приведите пример графа. Выпишите все простые пути между всеми его парами вершин.
2. Приведите пример несвязного графа. Найдите минимальное количество ребер, которые следует добавить, чтобы этот граф стал связным.

3. Приведите пример графа и найдите все его циклы.

4. Приведите пример ациклического графа. Экспериментально убедитесь в эквивалентности приведенных в данном разделе определений дерева.

5. Приведите пример ациклического графа. Преобразуйте его в корневое дерево. Для каждой вершины выпишите ее предков и определите ее глубину. Найдите высоту дерева.

5.2. Двоичные деревья поиска

Человек образованный — тот, кто
знает, где найти то, чего он не знает.

Георг Зиммель

Количество потомков вершины дерева называется ее *степенью*. Максимальное значение этих степеней есть *степень дерева*. Например, степень дерева на рис. 5.1*б* равна 3.

Суть *двоичных* (*бинарных*) *деревьев*, широко распространенных в программировании, следует из их названия: степень такого дерева равна 2. Вершина (узел) дерева может иметь не более двух потомков — их называют *левыми* и *правыми*. *Двоичные деревья поиска* характеризуются тем, что значение информационного поля (мы далее будем использовать целые числа и указывать их в вершинах), связанного с вершиной дерева, больше любого соответствующего значения из левого поддерева и меньше, чем содержимое любого узла его правого поддерева.

Эффективность алгоритмов, использующих двоичные деревья поиска, зависит от высоты дерева. Для высоты дерева h, содержащего n вершин, справедлива оценка $\lfloor \log_2 n \rfloor \leqslant h \leqslant n - 1$.

На рис. 5.2 приведены двоичные деревья для одного и того же набора данных из семи чисел ($n = 7$). Как видим, во втором случае (5.2*б*) дерево является обычным линейным списком. Для повышения эффективности алгоритмов необходимо стремиться к тому, чтобы дерево поиска имело «правильный» вид — тот, который показан на рис. 5.2*а*, а не «вырожденный» — см. рис. 5.2*б*.

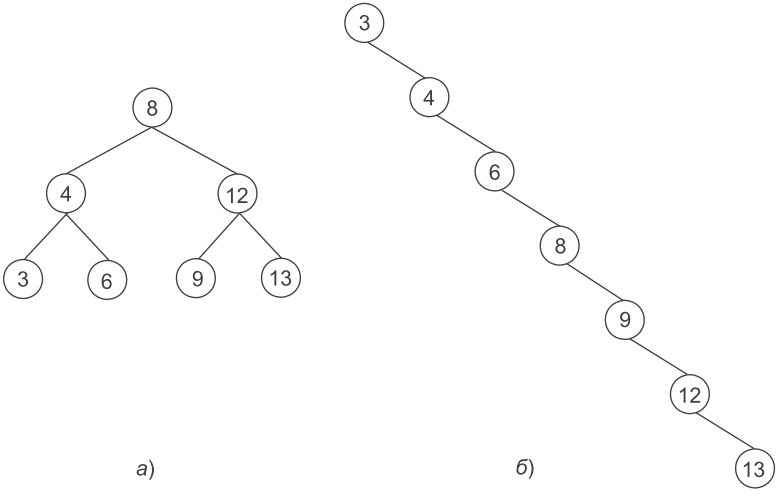


Рис. 5.2. Различные виды двоичного дерева поиска для одного и того же набора данных

Дерево поиска традиционно описывается в системах программирования с использованием указателей. Описание вершины дерева при этом имеет вид:

```
Type pt=^node;
     node=Record
       data:Integer; {Ключ}
       left,right:pt;
       {Ссылки на левого и правого потомков}
     End;
Var root:pt; {Указатель на корень дерева}
```

Таким образом, минимальное описание обязательно содержит *ключ поиска* и ссылки на левого и правого потомков. Информационная часть определяется решаемой задачей, но в ее составе обязательно наличие *ключа* — данных, на которых есть отношение порядка и по которым, собственно, и строятся связи двоичного дерева поиска. Стандартная реализация двоичного дерева поиска приведена на рис. 5.3.

Основными операциями при работе с двоичным деревом поиска являются:

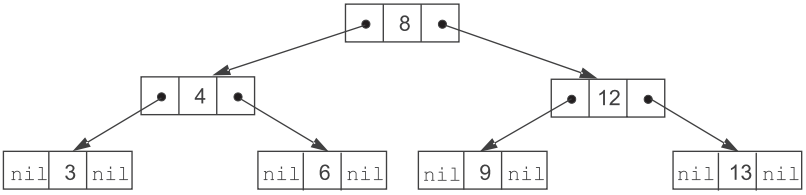


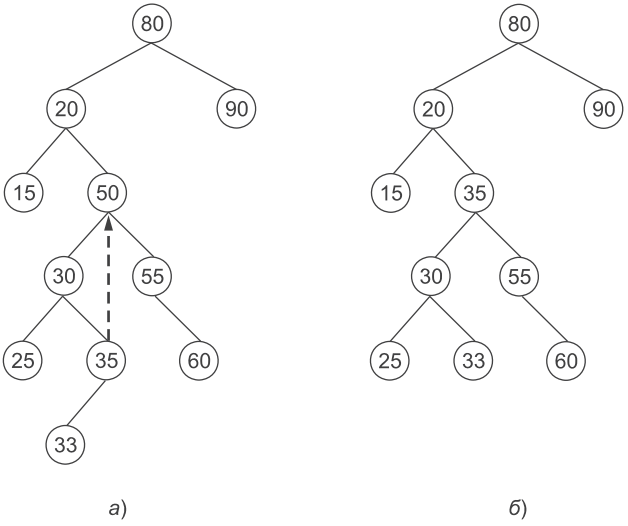
Рис. 5.3. Стандартная реализация двоичного дерева поиска, приведенного на рис. 5.2*а*

- вставка элемента в дерево;
- удаление элемента из дерева;
- обход дерева.

Вставка элемента в двоичное дерево поиска реализуется с помощью следующей рекурсивной процедуры (первоначальный ее вызов — `Ins_Tree(root,number)`).

```
Procedure Ins_Tree(Var t:pt;x:Integer);
{x - значение вставляемого элемента}
  Begin
    If t=nil Then Begin
      New(t);
      With t^ Do Begin
        left:=nil;
        right:=nil;
        data:=x;
      End;
    End
    Else If x<=t^.data Then Ins_Tree(t^.left,x)
                       Else Ins_Tree(t^.right,x);
  End;
```

Реализация *удаления элемента из дерева* выглядит чуть сложнее. Если узел имеет одного потомка, то в поле ссылки родителя удаляемого элемента записывается ссылка, не равная `nil`, и на этом все заканчивается. В случае же, когда у удаляемого элемента два потомка, для сохранения структуры дерева поиска на место этого элемента необходимо записать или самый правый элемент левого поддерева, или самый левый элемент правого поддерева. Один из этих двух вариантов (а третьего не дано!) можно выбрать по своему желанию. На рис. 5.4 приведена схема удаления элемен-



а) *б*)

Рис. 5.4. Двоичное дерево поиска до (*а*) и после (*б*) удаления элемента 50

та для первого варианта (самый правый элемент левого поддерева удаляемого элемента).

```
Procedure Del_Tree(Var t:pt; x:Integer);
  Var q:pt;
  Procedure Del(Var w:pt);
  {Поиск самого правого элемента в дереве}
  Begin
    If w^.right<>nil Then Del(w^.right)
    Else Begin
      q:=w; {Запоминаем адрес, чтобы
      освободить место в "куче"}
      t^.data:=w^.data;
      w:=w^.left;
    End;
  End;
  Begin
    If t<>nil Then
      If x<t^.data Then Del_Tree(t^.left,x)
      Else If x>t^.data Then Del_Tree(t^.right,x)
           Else Begin {Элемент найден.
           Приступаем к его удалению}
```

```
            q:=t;
            If t^.right=nil Then t:=t^.left
            {Правого поддерева нет}
            Else If t^.left=nil Then t:=t^.right
            {Левого поддерева нет}
                 Else Del(t^.left); {Находим самый
                 правый элемент в левом поддереве}
            Dispose(q);
          End;
  End;
```

Первоначальный вызов этой процедуры — `Del_Tree(root,numb)`, где `numb` — значение удаляемого элемента.

Для вывода значений элементов двоичного дерева необходимо выполнить *полный обход дерева*, когда его отдельные вершины посещаются в определенном порядке. В процедуре `Print_Tree`, с помощью которой осуществляется обход дерева, можно шестью способами переставить операторы 1, 2 и 3, и каждый способ такой перестановки дает вывод элементов дерева в определенной последовательности.

```
Procedure Print_Tree(t:pt);
  Begin
    If t<>nil Then Begin
      Print_Tree(t^.left);  {Оператор (1)}
      Write(t^.data:3);     {Оператор (2)}
      Print_Tree(t^.right); {Оператор (3)}
    End;
  End;
```

В табл. 5.1 перечислены все логически возможные очередности просмотра вершин элементарного двоичного дерева (условно обозначенных буквами A, B, C), приведенного на рис. 5.5, которые могут быть получены с помощью перестановки операторов в процедуре `Print_Tree`.

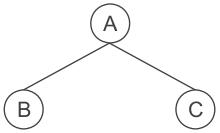


Рис. 5.5. Элементарное двоичное дерево

Таблица 5.1

Способ	Очередность вывода
1, 2, 3	*B A C*
1, 3, 2	*B C A*
2, 1, 3	*A B C*
2, 3, 1	*A C A*
3, 1, 2	*C B A*
3, 2, 1	*C A B*

Наиболее интересны для нас первые три варианта обхода. Их называют, соответственно, «слева направо» (или «симметричный обход» — корень посещается после левого поддерева, но перед посещением правого поддерева); «снизу вверх» («обход в обратном порядке» — корень посещается после левого и правого поддеревьев) и «сверху вниз» («обход в прямом порядке» — посещается корень, а затем левое и правое поддеревья).

Упражнения

1. Приведите пример двоичного дерева поиска. Определите по нему, в какой последовательности поступали данные на обработку, если считать, что вставка элементов в дерево осуществлялась с использованием процедуры `Ins_Tree`.

2. Приведите пример последовательности чисел. Выполните «ручную» трассировку логики процедуры вставки в двоичное дерево поиска чисел из этой последовательности.

3. Приведите пример двоичного дерева поиска. Выполните «ручную» трассировку логики процедуры удаления из этого двоичного дерева поиска для нескольких чисел.

4. Приведите пример двоичного дерева поиска. Обойдите его слева направо, снизу вверх и сверху вниз. Выпишите результаты обхода в виде последовательностей чисел.

5. Определите, что вычисляется с помощью функции:

```
Function Height(t:pt):Integer;
  Begin
    If t=nil Then Height:=0
    Else Height:=Max(Height(t^.left),
    Height(t^.right))+1;  {Max - функция
    вычисления максимального из двух чисел}
  End;
```

6. Определите, корректна ли следующая функция вычисления количества листьев в двоичном дереве поиска:

```
Function Counter(t:pt):Integer;
  Begin
    If t<>nil Then
      If (t^.left=nil) And (t^.right=nil)
      Then Counter:=1
      Else Counter:=Counter(t^.left)+
           Counter(t^.right)
    Else Counter:=0;
  End;
```

Если да, то обоснуйте свой ответ, а если нет, то внесите в нее требуемые изменения.

7. Определите, что ищется с помощью функции:

```
Function Tree_Search(t:pt;k:Integer):Integer;
 Begin
  While (t<>nil) And (t^.data<>k) Do
    If k<t^.data Then t:=t^.left
      Else t:=t^.right;
  If t<>nil Then Tree_Search:=t^.data
   Else Tree_Search:=0; {Предполагаем, что
   в дереве нет нулевого ключа}
 End;
```

8. Определите, что ищется с помощью функции:

```
Function Tree_Min(t:pt):Integer;
{t<>nil при первом вызове}
  Begin
    While t^.left<>nil Do t:=t^.left;
    Tree_Min:=t^.data;
  End;
```

9. Определите, что ищется с помощью функции:

```
Function Tree_Max(t:pt):Integer;
{t<>nil при первом вызове}
  Begin
    While t^.right<>nil Do t:=t^.right;
    Tree_Max:=t^.data;
  End;
```

10. Предположим, что описание вершины дерева содержит еще один указатель parent на родителя вершины:

```
node=Record
  data:Integer; {Ключ}
  parent,left,right:pt;
  {Ссылки на левого и правого потомков}
End;
```

На рис. 5.6 приведен пример двоичного дерева поиска. Пусть значение указателя t определяет положение элемента 55.

Требуется решить задачу поиска следующего элемента дерева. Так, для элемента 55 в нашем примере (см. рис. 5.6) он равен 80, а для элемента 80 — равен 85. Определите, корректна ли следующая функция; если да, то обоснуйте свой ответ, а если нет, то внесите в нее требуемые изменения.

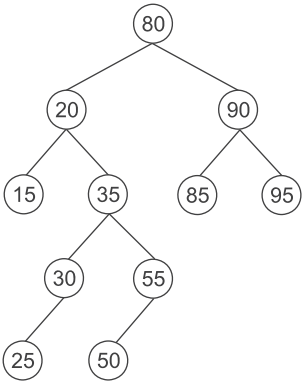


Рис. 5.6. Пример двоичного дерева

```
Function Tree_Succ(t:pt):Integer;
  Var q:pt;
  Begin
    If t^.right<>nil Then
    Tree_Succ:=Tree_Min(t^.right)
    {Функция из упражнения 8}
    Else Begin
      q:=t^.parent;
      While (q<>nil) And (t=q^.right) Do Begin
        t:=q;
        q:=q^.parent;
      End;
    End;
    If q=nil Then Tree_Succ:=0
    {t указывает на последний элемент в дереве}
    Else Tree_Succ:=q^.data;
  End;
```

11. Разработайте функцию поиска предыдущего элемента в двоичном дереве (`Tree_Pred`).

12. Разработайте функцию, возвращающую значение указателя на второй минимальный элемент в двоичном дереве поиска.

13. Двоичное дерево поиска считается *идеально сбалансированным*, если для каждой его вершины количество вершин в левом и правом поддеревьях различается не более чем на 1. Напишите функцию проверки идеальной сбалансированности двоичного дерева.

14. Пусть из двоичного дерева поиска удаляются две вершины с ключами x и y. В первом случае они удаляются в последовательности: x, а затем y; во втором — в обратном порядке. Совпадают ли результаты этих операций? Другими словами, будут ли получаемые деревья идентичными?

5.3. Способы описания деревьев

> И деревья были такие
> необыкновенные — легкие,
> сквозящие, будто сиреневые,
> и полны такой внутренней тишиной
> и покоем...
>
> *С. Козлов*

Описание с использованием массивов

Подобное представление двоичного дерева поиска требует наличия массива для хранения значений ключей (назовем его `data`), левых (`left`) и правых ссылок (`right`). В ячейках этих массивов с одинаковыми индексами хранится описание вершины дерева. Кроме того, требуется отдельная переменная (`root`), значением которой является указатель на корень дерева. Чтобы не искать потом свободную ячейку, адрес первой же свободной ячейки записывается в переменную `headfree`, а сами ячейки связаны адресами в отдельном массиве `next`. В начальный момент времени (до работы с деревом) элементы массива размечаются: в `headfree` записывается адрес первой свободной ячейки, а `next[i]` указывает на следующую свободную ячейку. Фрагмент этой логики имеет вид:

```
For i:=1 To n-1 Do next[i]:=i+1;
{n - количество элементов массива}
```

Пример. Рассмотрим дерево, показанное на рис. 5.4*а*. Его описание (в какой-то момент времени после выполнения операций по вставке и удалению элементов) с использованием массивов (`n` = 20, выбрано произвольно) показано в табл. 5.2. Значение переменной `root` равно 18, а переменной `headfree` — равно 12.

Рассмотрим *вставку элемента в дерево поиска*. Практически это не что иное, как нерекурсивная реализация процедуры `Ins_Tree`. При поиске элемента в дереве поиска нам приходится (из-за отсутствия ссылки на родителя) хранить адрес предыдущего элемента в дереве (переменная `q`).

```
Procedure Inset_Tree(x:Integer);
  Var t,q,w:Integer;
  Begin
    If root=0 Then Begin
```

Таблица 5.2

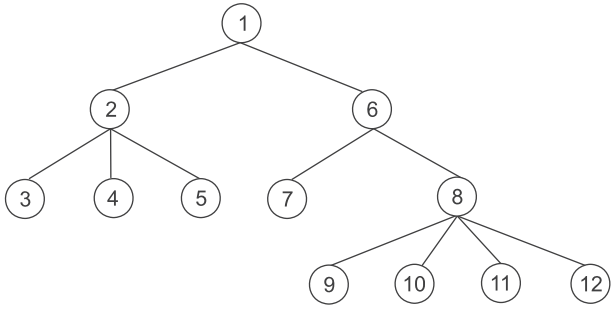
Номер элемента массива	data	left	right	next
1	0	0	0	3
2	90	0	0	0
3	0	0	0	6
4	15	0	0	0
5	50	19	7	0
6	0	0	0	9
7	55	0	10	0
8	25	0	0	0
9	0	0	0	17
10	60	0	0	0
11	33	0	0	0
12	0	0	0	13
13	0	0	0	1
14	0	0	0	15
15	0	0	0	
16	20	4	5	
17	0	0	0	14
18	80	16	2	
19	30	8	20	
20	35	11	0	0

```
{Вставка первого элемента корня в дерево}
  root:=headfree;
  headfree:=next[headfree];
  data[root]:=x;
  next[root]:=0;
End
Else Begin
  t:=root;
  While t<>0 Do Begin
  {Находим место элемента в дереве}
    q:=t; {Номер предыдущего элемента}
```

```
        If x<data[t] Then t:=left[t]
          Else t:=right[t];
          {Идем по правой или по левой ссылке}
      End;
      w:=headfree;
      headfree:=next[headfree];
      next[w]:=0; {Берем ячейку из списка свободных}
      data[w]:=x;
      If x<data[q] Then left[q]:=w
      {Изменяем ссылку у предыдущего элемента}
        Else right[q]:=w;
   End;
End;
```

Описание произвольного дерева в виде одного массива

Пусть у дерева `T` вершины имеют метки 1, 2, ..., `n`. Тогда возможно описание дерева с помощью одного одномерного массива (например, `A`) так, как это показано на рис. 5.7.



а) *Дерево*

1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	2	2	1	6	6	8	8	8	8

б) *Массив указателей на вершины родителей*

Рис. 5.7. Дерево и массив указателей на вершины родителей

Элемент `A[i]` дерева является указателем на родителя вершины с номером `i`, причем корень дерева имеет нулевой

указатель. Такое представление основано на том факте, что каждая вершина дерева имеет только одного родителя. Прохождение по цепочке от одного родителя к другому родителю при подобном описании удобно, оно пропорционально количеству вершин в пути. Однако информация о сыновьях вершин отсутствует, и это затрудняет (без введения дополнительных структур данных) реализацию операций просмотра дерева и операций, которые приведены в упражнениях к разделу 5.2.

Представление деревьев с помощью упорядоченных списков связи

Одним из способов описания произвольного дерева (не обязательно двоичного) является создание упорядоченных по значению ключей списков связи сыновей для каждой вершины. Список начинается с самого левого сына вершины и заканчивается самым правым сыном. Указатели на начала таких списков хранятся в отдельном массиве, например `A`. Индексом же для обращения к элементам массива `A` является значение ключа. Указатель на корень дерева при этом хранится в отдельной переменной (`root`).

Пример. На рис. 5.8 приведено представление дерева, показанного на рис. 5.7*а* в виде упорядоченных списков связи. Так, элементы списка `A[6]` являются сыновьями вершины 6: это вершины 7 и 8.

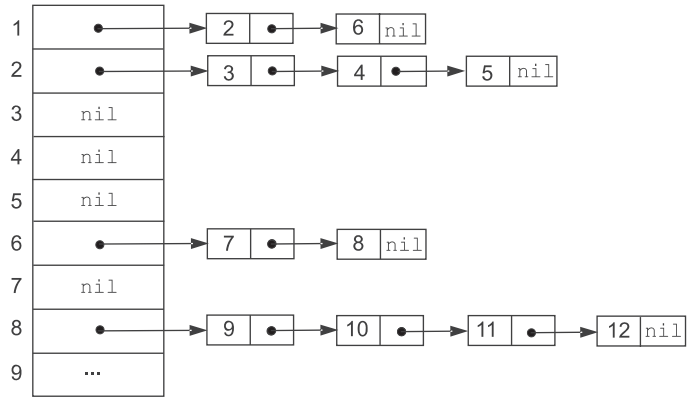


Рис. 5.8. Представление дерева (см. рис. 5.7*а*) с помощью упорядоченных списков связи

Пусть необходимо реализовать операцию вставки элемента в такое дерево. Мы сообщаем процедуре Ins_Tree ключ вставляемого элемента x (его номер) и ключ родителя вставляемого элемента y. Предположим, что у нас уже есть процедура вставки элемента в упорядоченный список — Ins_List, параметрами которой являются значение элемента и указатель на начало списка. Тогда процедура Ins_Tree сводится к вызову Ins_List(x,A[y]).

А вот функция определения родителя вершины с ключом x:

```
Function Parent(x:Integer):Integer;
{x – ключ вершины, родитель которой ищется. Функция
возвращает ключ вершины родителя или 0, если
родитель не найден}
  Var pp:Boolean;
  {Фиксация признака нахождения элемента}
      i:Integer;
      t:pt; {Предполагаем, что описание элемента
            списка имеет вид:
            pt=^node;
            node=Record
            data:Integer; - ключ
            next:pt; - ссылки на следующий
            элемент списка
            End;}
  Begin
    pp:=False;
    i:=1;
    While (i<=maxtree) And Not pp Do Begin
    {maxtree – максимальное количество элементов
    в дереве; размерность массива A}
      t:=A[i]; {Просмотр списка связи вершины i}
      While (t<>nil) And (t^.data<>x) Do t:=t^.next;
      If t^.data=x Then pp:=True; {Вершина найдена}
    End;
    If pp Then Parent:=i {вершина i – родитель x}
    Else Parent:=0;
  End;
```

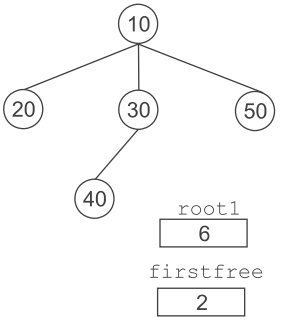
Описание дерева по принципу «левый сын и правый брат»

Предположим, что решаемая нами задача требует интенсивного использования операции объединения деревьев. При всех предыдущих способах описания деревьев это действие будет заключаться в последовательной вставке элементов одного дерева в другое, т. е. время выполнения такой операции (количество вставок) окажется пропорционально количеству вершин одного из деревьев — $O(n)$, где n — количество вершин дерева. При этом второе дерево как бы «переписывается в память» первого дерева. Возможен другой вариант, когда оба дерева переписываются в память вновь создаваемого третьего дерева, являющегося их объединением.

А можно ли операцию объединения деревьев выполнять за константное время? Решение такой задачи возможно, но для этого, как минимум, необходимо хранить объединяемые деревья в общем поле памяти.

Итак, пусть необходимо выполнить операцию `Create(v,root1,root2)`, где объединяются два дерева: T_1 (указатель на корень — `root1`) и T_2 (указатель на корень — `root2`) и создается вершина с меткой `v` и двумя сыновьями, которыми являются корни деревьев T_1 и T_2. Рассмотрим эту операцию на конкретном примере.

Пример. На рис. 5.9 показано дерево и его описание по принципу «левый сын, правый брат». В переменной `root1` записан адрес корня дерева. Массив (либо поле записи) `data`



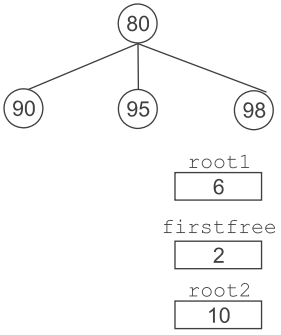
	leftchild	data	rightsibling	free
1	/	40	/	
2				5
3	/	50	/	
4	/	20	8	
5				7
6	4	10	/	
7				9
8	1	30	3	
9				10
10				11
11				12
12				13
13				/

Рис. 5.9. Дерево и его описание в памяти

предназначен для хранения меток вершин, массив `leftchild` — для указателей на левого сына вершины, а массив `rightsibling` — для указателей на правого брата вершины. Массив `free` обеспечивает манипуляции со свободной памятью (конечно, информацию о свободной памяти можно интегрировать с массивом `rightsibling`, но нами для простоты изложения выбран именно такой способ, с отдельным массивом). Ячейки связаны адресами связи, где адрес первой свободной ячейки фиксируется в указателе `firstfree`. Символом «/» в данном представлении дерева отмечены нулевые значения указателей.

Итак, корень дерева (метка 10) содержит указатель на левого сына — адрес 4 (правого брата у этой вершины нет). Вершина с меткой 20 имеет правого брата — это вершина 30, данные по которой размещены по адресу 8. Вершина же с меткой 30 имеет и левого сына (адрес 1), и правого брата (адрес 3).

Предположим, что в этой же памяти размещена информация и по второму дереву — рис. 5.10.

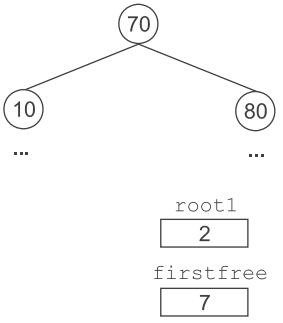


	leftchild	data	rightsibling	free
1	/	40	/	
2				7
3	/	50	/	
4	/	20	8	
5	/	95	12	
6	4	10	/	
7				9
8	1	30	3	
9				11
10	13	80	/	
11				/
12	/	98	/	
13	/	90	5	

Рис. 5.10. Второе дерево и его совместное описание в памяти с деревом, показанным на рис. 5.9

На рис. 5.11 показан результат выполнения операции `Create(70, root1, root2)`. Из списка свободных ячеек берется первая. В ней размещается информация о вершине с меткой 70; ее левым сыном становится корень поддерева с меткой 10, а правым братом этого корня (левого сына) будет корень поддерева с меткой 80.

Приведем возможный вариант реализации функции `Create`, которая возвращает указатель на корень вновь созданного дерева.



	leftchild	data	rightsibling	free
1	/	40	/	
2	6	70		
3	/	50	/	
4	/	20	8	
5	/	95	12	
6	4	10	10	
7				9
8	1	30	3	
9				11
10	13	80	/	
11				/
12	/	98	/	
13	/	90	5	

Рис. 5.11. Результат выполнения операции `Create`

```
Function Create(v,root1,root2:Integer):Integer;
  Var t:Integer;
  Begin
    t:=firstfree;
    {Берем первую ячейку из списка свободных}
    firstfree:=free[t];
    {Первой свободной делаем следующую ячейку}
    data[t]:=v;
    {Формируем корень дерева, присваиваем метку}
    leftchild[t]:=root1; {Адрес левого сына}
    righsibling[t]:=0;
    rightsibling[root1]:=root2;
    {Левый сын имеет правого брата}
    Create:=t;
  End;
```

Очевидно, что операция объединения деревьев при таком способе описания выполняется за константное время $O(1)$.

Упражнения

1. Предположим, что дано описание двоичного дерева поиска в виде массивов (как в табл. 5.2). Предположим также, что в каждые ячейки, описывающие вершину дерева, запись осуществлялась один раз. Сколько операций вставки и удаления при этом было выполнено?

2. Пусть в описании вершины дерева дается ссылка на родителя. Модифицируйте процедуру `Ins_Tree` для этого случая.

3. Для первого из рассмотренных описаний двоичного дерева поиска с использованием массивов разработайте процедуры вывода значений его элементов:
- слева направо (симметричный обход — корень посещается после левого поддерева, но перед посещением правого поддерева);
- снизу вверх (обход в обратном порядке — корень посещается после левого и правого поддеревьев);
- сверху вниз (обход в прямом порядке — посещается корень, а затем левое и правое поддеревья).

4. Разработайте процедуру удаления элемента двоичного дерева поиска. При этом рекомендуется использовать отдельную процедуру возврата ячейки с номером `t` в список свободных, когда возвращаемая ячейка становится первой в списке свободных.

```
Procedure Return(t:Integer);
{Возврат ячейки в список свободных}
  Begin
    If t<>root Then Begin
      left[t]:=0;
      right[t]:=0;
      data[t]:=0;
      next[t]:=headfree;
      headfree:=t;
    End;
  End;
```

5. В упражнениях 5–10 к разделу 5.2 предлагалось по приведенному тексту программы определить исходную задачу или проанализировать корректность задачи. Разработайте программную реализацию для решения этих же задач при использовании описания двоичного дерева поиска с использованием массивов.

6. Разработайте процедуру вставки в произвольное дерево для случая, когда его описание задается с помощью одного массива указателей на родителя вершины. При встав-

ке используйте заданные значения: i — номер вставляемой вершины и j — номер вершины «родителя».

7. В некоторых случаях сведения о вершинах дерева (например, значения ключей) удобно выводить не в виде строки, когда структура дерева не просматривается, а размещая их на экране с учетом взаимных связей в дереве. Следующая процедура выполняет эту операцию для двоичного дерева:

```
Procedure Step_Tree(t:pt;h:Integer);
{h - количество выводимых пробелов, вставляемых
перед выводом значения информационного поля узла}
  Var i:Integer;
  Begin
    If t<>nil Then
      With t^ Do Begin
        Step_Tree(left,h+5);
        {Отступаем от предыдущего уровня на пять
        позиций (пробелов)}
        For i:=1 To h Do Write(' ');
        WriteLn(data:3);
        {Выводим h пробелов и значение ключа}
        Step_Tree(right,h+5);
      End;
  End;
```

Модифицируйте эту процедуру для решения той же задачи для случая, когда описание произвольного дерева (не двоичного) задано с помощью единственного одномерного массива указателей на родителей вершин.

8. Приведите пример произвольного дерева. Изобразите его представление с использованием упорядоченных списков связи.

9. Напишите процедуру удаления элемента из дерева при его представлении с помощью упорядоченных списков связи.

10. Напишите рекурсивную процедуру вывода элементов дерева при его описании упорядоченными списками связи.

11. Нарисуйте три произвольных дерева, описанных по принципу «левый сын и правый брат». Последовательно выполните две операции объединения этих деревьев, задавая значения ключей для новых вершин (двух) результирующего дерева.

12. Напишите процедуру вставки элемента в дерево, представленное в памяти по принципу «левый сын и правый брат».

13. Напишите процедуру удаления элемента из дерева, представленного в памяти по принципу «левый сын и правый брат».

14. Напишите функцию поиска родителя вершины с ключом x (дерево задается значением указателей на левого сына и правого брата).

15. Напишите процедуру определения всех сыновей вершины дерева с ключом x (дерево задается значением указателей на левого сына и правого брата).

16. Предположим, что в объединяемых деревьях есть вершины с совпадающими ключами. Предложите стратегию борьбы с этим явлением.

5.4. Оптимальные двоичные деревья поиска

> Часто приходится слышать: «Он еще не нашел себя». Но найти себя невозможно — себя можно только создать.
>
> *Томас Сас*

Предположим, что нам известны вероятности обращения к вершинам двоичного дерева поиска. Тогда возникает закономерный вопрос: как организовать дерево, чтобы среднее количество сравнений при поиске было минимальным? Такое дерево называется *оптимальным*. На рис. 5.12 представлены все бинарные деревья с тремя вершинами — их будет пять. Для четырех вершин их будет уже четырнадцать. В общем же случае для двоичного дерева поиска с n верши-

нами их количество равно n-му числу Каталана[1] $c_n = \frac{C_{2n}^n}{n+1}$ при $n > 0$, $c_0 = 1$. Среди этих деревьев требуется выбрать одно, удовлетворяющее нашему критерию.

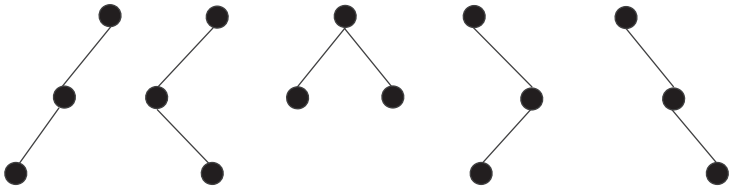


Рис. 5.12. Все возможные бинарные деревья с тремя вершинами

Пример. Пусть есть четыре ключа: 2, 5, 9, 13. Их поиск осуществляется с вероятностями 0.2, 0.3, 0.4, 0.1 соответственно. На рис. 5.13 изображены два различных дерева с этими ключами. Для первого дерева среднее количество сравнений равно $0.1 \cdot 1 + 0.2 \cdot 2 + 0.4 \cdot 2 + 0.3 \cdot 3 = 2.2$, а для второго — $0.4 \cdot 1 + 0.2 \cdot 2 + 0.1 \cdot 2 + 0.3 \cdot 3 = 1.9$. Очевидно, для поиска оптимального дерева мы должны перебрать все четырнадцать возможных деревьев — эти два дерева не обязательно являются оптимальными.

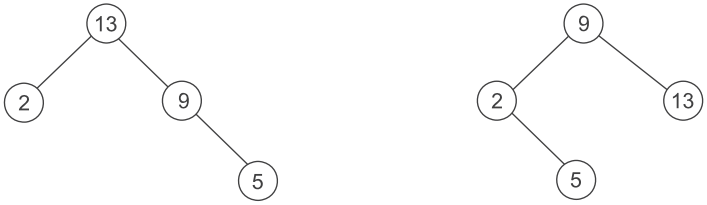


Рис. 5.13. Два различных дерева, построенных на одном и том же наборе данных

Примечание. В данном случае уровни дерева отсчитываются начиная с единицы. Тогда количество сравнений совпадает с номером уровня.

[1] Подробнее о числах Э. Каталана можно почитать в книге: *Окулов С. М.* Дискретная математика: теория и практика решения задач по информатике. М.: БИНОМ. Лаборатория знаний, 2007.

Сложность проблемы заключается в том, что поиск оптимального дерева путем перебора всех возможных вариантов не подходит нам по временны́м ограничениям: мы имеем экспоненциальный рост количества вариантов с увеличением значения n[1].

Прежде чем перейти к обсуждению вопроса, сделаем небольшое математическое отступление.

Математическое отступление. Два корневых ориентированных двоичных дерева T_1 и T_2 называются *изоморфными*, если существует изоморфизм φ графов T_1 и T_2, такой, что:

1) корень дерева T_1 отображается в корень дерева T_2;
2) v_i является левым сыном вершины v_j тогда и только тогда, когда $\varphi(v_i)$ — левый сын вершины $\varphi(v_j)$;
3) v_i является правым сыном вершины v_j тогда и только тогда, когда $\varphi(v_i)$ — правый сын вершины $\varphi(v_j)$.

Примечание. Ориентация в корневых деревьях идет от корня сверху вниз по ребрам и обычно не изображается в виде отрезков со стрелками.

Теорема. Количество неизоморфных корневых двоичных ориентированных деревьев с n вершинами равно числу Каталана c_n.

Доказательство. Пусть t_n — количество неизоморфных корневых деревьев с n вершинами. Для пустого дерева и дерева из одной вершины считаем, что $t_0 = 1$ и $t_1 = 1$ соответственно. Осталось рассмотреть деревья с $n \geqslant 2$ вершинами. Для каждого k ($1 \leqslant k \leqslant n$) существуют деревья T_k с n вершинами, в которых k-й корень есть левое поддерево с $k - 1$ вершинами и правое поддерево с $n - k$ вершинами. Количество неизоморфных левых поддеревьев равно t_{k-1}, а правых — равно t_{n-k}. Исходя из комбинаторного принципа умножения, для каждого фиксированного значения k имеем $T_k = t_{k-1} \cdot t_{n-k}$. Суммируя эти произведения (комбинаторный принцип сложения) по k от 1 до n, получаем: $t_n = \sum_{k=1}^{n} t_{k-1} \cdot t_{n-k}$, а это не что иное, как рекуррентное соотно-

[1] *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 252–253.

шение, решение которого дает числа Каталана. При этом совпадают и начальные условия. Отсюда следует, что $t_n = c_n$.

Рассмотренное выше математическое отступление наталкивает нас на идею метода поиска оптимального дерева, отличающуюся от прямого перебора, а именно на идею *динамической схемы поиска оптимального дерева*. Такая задача обладает свойством аддитивности — если мы нашли оптимальное поддерево для некоторого подмножества ключей, то этот результат как неизменный (не изменяемый) мы можем использовать и при построении следующего оптимального поддерева для расширенного подмножества ключей.

Пусть $k_1, k_2, \ldots, k_n$ — ключи, упорядоченные по возрастанию (причем совпадающих ключей нет), а $p_1, p_2, \ldots, p_n$ — вероятности их поиска. Считаем, что $A[i,j]$ — наименьшее среднее количество сравнений при успешном поиске в двоичном дереве $T_{i,j}$, построенном для ключей $k_i, \ldots, k_j$, где $1 \leqslant i \leqslant j \leqslant n$. Если в качестве корня взята вершина со значением ключа k_t, выбранного из ключей $k_1, k_2, \ldots, k_n$, то дерево $T_{i,j}$ разбивается на поддеревья $T_{i,t-1}$ и $T_{t+1,j}$, а для этих подмножеств ключей $k_1, \ldots, k_{t-1}$ и $k_{t+1}, \ldots, k_j$, соответственно, уже найдены оптимальные поддеревья. Нам остается только «пробежаться» по всем значениям t от i до j и выбрать значение, на котором достигается минимальная оценка времени поиска.

Формальная запись:

$$A[i,j] = \min_{i \le t \le j} \left\{ p_t \cdot 1 + \sum_{l=1}^{t-1} p_l \cdot (\text{уровень } k_l \text{ в } T_{i,t-1} + 1) + \right.$$

$$\left. + \sum_{l=t+1}^{j} p_l \cdot (\text{уровень } k_l \text{ в } T_{t+1,j} + 1) \right\} =$$

$$= \min_{i \le t \le j} \{A[i,t-1] + A[t+1,j]\} + \sum_{l=i}^{j} p_l .$$

Значение $A[1,n]$ дает среднее количество сравнений при успешном поиске в оптимальном двоичном дереве. Очевидно, что это значение получается путем последовательного заполнения матрицы A по диагоналям начиная с первой над главной диагональю (вверх). Для построения самого опти-

мального дерева на каждом шаге требуется фиксировать номер ключа (корня), на котором достигается минимум; для этого необходимо ввести еще одну матрицу (например, R) и формировать ее значения одновременно со значениями матрицы A.

Пример. Вернемся к деревьям, показанным на рис. 5.13, и найдем оптимальное двоичное дерево поиска.

Начальный вид матриц A и R — рис. 5.14.

A:

	0	1	2	3	4
1	0	0.2			
2		0	0.3		
3			0	0.4	
4				0	0.1
5					0

R:

	0	1	2	3	4
1		1			
2			2		
3				3	
4					4
5					

Рис. 5.14. Начальный вид матриц A и R

Вычисляем $A[1,2]$. При $t = 1$ имеем: $A[1,0] + A[2,2] + \sum_{l=1}^{2} p_l =$ $= 0 + 0.3 + 0.5 = 0.8$, а при $t = 2$ — $A[1,1] + A[3,2] + \sum_{l=1}^{2} p_l =$ $= 0.2 + 0 + 0.5 = 0.7$. Минимальное значение равно 0.7; корень поддерева на данном подмножестве ключей — вершина с ключом 5 (ее номер 2). После аналогичных вычислений $A[2,3]$ и $A[2,4]$ имеем ситуацию, показанную на рис. 5.15.

A:

	0	1	2	3	4
1	0	0.2	0.7		
2		0	0.3	1.0	
3			0	0.4	0.6
4				0	0.1
5					0

R:

	0	1	2	3	4
1		1	2		
2			2	3	
3				3	3
4					4
5					

Рис. 5.15. Вид матриц A и R после первой итерации

При вычислении $A[1,3]$ мы делаем оценку при $t = 1$: она равна 1.9; затем делаем такую оценку при $t = 2$ она равна 1.5

и при $t = 3$ она равна 1.6. Минимальное значение здесь равно 1.5. Окончательный вид матриц приведен на рис. 5.16.

A:

	0	1	2	3	4
1	0	0.2	0.7	1.5	1.8
2		0	0.3	1.0	1.4
3			0	0.4	0.6
4				0	0.1
5					0

R:

	0	1	2	3	4
1		1	2	2	2
2			2	3	2
3				3	3
4					4
5					

Рис. 5.16. Окончательный вид матриц A и R

Построение оптимального двоичного дерева поиска на основе данных матрицы R сводится к рекурсивной логике. Выбирается корень дерева — вершина с номером $R[1, n] = t$ (в нашем примере это вершина с номером 2), а затем из R выбираются корни поддеревьев $T_{1,t-1}$ и $T_{t+1,n}$, соответственно, это элементы $R[1,t-1]$ и $R[t+1,n]$. Такой процесс продолжается до тех пор, пока в поддереве не останется одна вершина. Для нашего примера: левое поддерево состоит из одной вершины с номером 1, правое — из двух; корень поддерева — вершина с номером 3 ($R[3,4]$).

Временна́я сложность такого алгоритма не может быть меньше размерности формируемого результата, а это квадратная матрица, — значит, временна́я сложность равна $O(n^2)$. Однако при получении каждого элемента матрицы необходимо использовать еще одну циклическую конструкцию; итого получаем — $O(n^3)$.

Упражнения

1. Всегда ли логика построения оптимального двоичного дерева поиска приводит к однозначному результату? Проверьте это на примере, выполнив вручную все подсчеты.

2. Задайте множество ключей и их вероятности ($n = 6$). Выполните ручной подсчет значений матриц A и R. Нарисуйте полученное оптимальное двоичное дерево поиска.

3. Реализуйте в виде программы логику формирования A и R.

4. Напишите рекурсивную процедуру построения оптимального двоичного дерева поиска по сформированным матрицам A и R. Оцените временну́ю сложность решения.
Примечание. Используйте процедуру `Ins_Tree`.

5. *Код Хаффмена*[1]. Поясним принцип кодирования на следующем примере. Пусть у нас есть сообщения A_1, A_2, A_3, A_4 и A_5, имеющие вероятности p_1, p_2, p_3, p_4 и p_5 ($p_1 \geqslant p_2 \geqslant p_3 \geqslant p_4 \geqslant p_5$). Их требуется закодировать двоичными словами a_1, a_2, a_3, a_4 и a_5 с длинами l_1, l_2, l_3, l_4 и l_5 так, чтобы средняя длина $\bar{l} = \sum_{i=1}^{5} l_i \cdot p_i$ кодовых слов была минимальной (такой код называется оптимальным; код Хаффмена обладает этим свойством), а код однозначно декодировался.
Принцип кодирования по Хаффмену показан в табл. 5.3.

Таблица 5.3

Сообщения	Вероятности и кодовые обозначения					
	Исходные сообщения и их кодировка		Сжатые множества			
			$A^{(1)}$		$A^{(2)}$	
A_1	0.4	1	0.4	1	0.4	1
A_2	0.25	01	0.25	01	0.35	00
A_3	0.15	001	0.2	000	0.25	01
A_4	0.12	0000	0.15	001		
A_5	0.8	0001				

Сообщения	Вероятности и кодовые обозначения	
	Сжатые множества	
	$A^{(3)}$	
A_1	0,6	0
A_2	0.4	1
A_3		
A_4		
A_5		

Здесь выполняются два последовательных процесса: сжатие и расщепление. Первый процесс отражен в табл. 5.3 линиями со стрелками: сообщения с наименьшими вероятностями суммируются по значениям вероятностей и становятся одним сообщением. Процесс сжатия $A^{(1)}$, $A^{(2)}$, $A^{(3)}$ продолжается до тех пор, пока не

[1] Этот метод кодирования был предложен в 1952 г. американским математиком Д. А. Хаффменом.

останется два сообщения; им приписываются кодовые обозначения 0 и 1. Затем начинается обратный процесс (по стрелкам) — объединенное сообщение расщепляется путем приписывания 0 и 1 к тому кодовому обозначению, которое оно имело.

Примечание. Код Хаффмена является *префиксным кодом*. Префиксные коды обладают таким свойством, что никакое кодовое слово не является началом (префиксом) другого кодового слова. Если, например, у нас есть кодовое слово 110, то кодовые слова 1 или 11 уже недопустимы. В этом случае любая закодированная последовательность сообщений (например, 10100000001011001) декодируется однозначно. Проверьте это.

Построенный код можно представить в виде двоичного дерева Хаффмена — рис. 5.17.

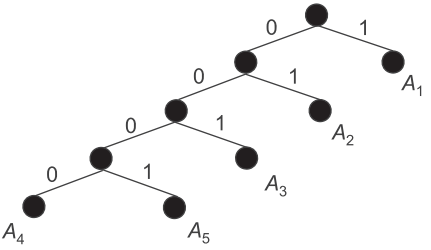


Рис. 5.17. Двоичное дерево Хаффмена

Кодовым словам соответствуют листья дерева, а процесс построения дерева можно рассматривать как последовательное слияние деревьев. То есть используется «лес» — совокупность деревьев, листья которых помечены кодируемыми сообщениями, а корни помечаются суммами вероятностей всех сообщений, соответствующих листьям дерева. Первоначально каждому сообщению соответствует дерево из одной вершины.

Первые три итерации слияния деревьев для нашего примера показаны на рис. 5.18. Заметим, что некая регулярность слияний на рис. 5.18 (на каждой итерации дерево, полученное на предыдущем шаге, участвует в слиянии) не должна вводить нас в заблуждение. На первой итерации объединяются деревья с сообщениями A_4

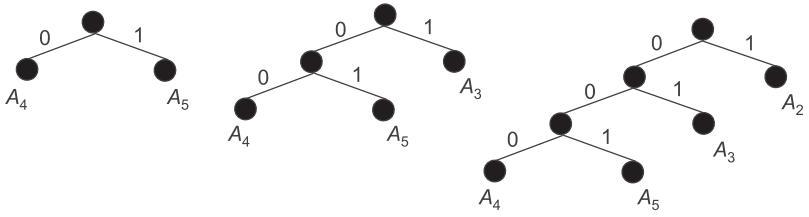


Рис. 5.18. Процесс слияния деревьев

и A_5, как имеющие наименьшие вероятности (5.18*а*). На второй итерации полученное дерево объединяется с деревом, соответствующим сообщению A_3 (рис. 5.18*б*). Аналогично — и на третьей итерации — рис. 5.18*в*.

Требуется выбрать структуры данных для описания дерева Хаффмена и разработать программу его построения.

Рекомендации. Введите в описание вершины дерева указатель на ее родителя.

6. Задайте множество сообщений и их вероятности ($n \geqslant 6$). Найдите оптимальный код Хаффмена и нарисуйте соответствующее ему дерево.

7*. Разработайте алгоритм построения оптимального двоичного дерева поиска с временно́й сложностью $O(n^2)$.

Методические комментарии

Двоичные деревья, видимо, были независимо предложены многими специалистами по информатике примерно в 1950–1960 гг. Указать конкретное имя «первооткрывателя» здесь не представляется возможным. Д. Кнут пишет о Г. Хоппер и ее языке компилятора A-1 (1951 г.), разработанного для манипуляций с алгебраическими формулами, в котором использовались арифметические выражения, записанные в трехадресном коде, эквивалентном `data`, `left` и `right` в представлении двоичных деревьев[1)].

Материал данного раздела достаточно полно представлен в учебной литературе. Первыми широко известными публикациями на эту тему, вероятно, следует считать следующие:

- *Вирт Н.* Алгоритмы + структуры данных = программы. — М.: Мир, 1985. С. 219–248.

[1)] *Кнут Д.* Искусство программирования для ЭВМ. Т.1. Основные алгоритмы. М.: Мир, 1976. С. 561.

- *Лэнгсам Й., Огенстайн М., Тененбаум А.* Структуры данных для персональных ЭВМ. М.: Мир, 1989. С. 301–377.
- *Райли Д.* Абстракция и структуры данных: Вводный курс. М.: Мир, 1993. С. 503–569.

Первая из этих книг — «классика», но «конструировать» материал занятий по ней достаточно сложно. Во второй книге дается подробное изложение темы, но оно излишне отягощено программными конструкциями на Бейсике. Третья же книга, написанная с учетом рекомендаций ACM 1984 г., содержит скорее технологию разработки программ на языке Модула-2, чем действительно детальное описание работы со структурами данных, хотя отдельные методические находки (в частности, по материалу данного раздела) она все же содержит.

Позже этот раздел стал практически обязательным в фундаментальных учебниках по информатике — начиная с книги Т. Кормена, Ч. Лейзерсона и Р. Ривеста[1)] и заканчивая книгой В. Е. Алексеева и В. А. Таланова[2)]. Указанные учебники имеют свои достоинства и недостатки, как и приведенный здесь вариант, основанный на собственном опыте преподавания темы, — но автор надеется, что приведенный здесь материал более прост и доступен для преподавателей и учащихся.

Добавим также, что материал раздела 5.3 основан на одной из лучших книг[3)] по данной проблематике.

---

1) *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 236–253.

2) *Алексеев В. Е., Таланов В. А.* Графы и алгоритмы. Структуры данных. Модели вычислений. М.: ИНТУИТ; БИНОМ. Лаборатория знаний, 2006. С. 239–254.

3) *Ахо А. В., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. М.: Издательский дом «Вильямс», 2001. С. 89–91

Глава 6

Множества

Одному деревенскому брадобрею приказали брить всякого, кто не бреется сам, и не брить того, кто сам бреется. Бреется ли сам брадобрей?

Одна из популярных формулировок парадокса Б. Рассела о множествах

6.1. Основные понятия

Линия есть длина без ширины.

Евклид

Формально понятия множества (*Set*) и элемента множества неопределимы — так же как, например, понятия «точка» или «прямая». Можно сказать, что под множеством M понимается совокупность некоторых объектов, которые называются элементами множества M. Запись $x \in M$ означает, что x является элементом M. Множество обычно изображается в виде последовательности его элементов, заключенных в фигурные скобки. Например, $\{2, 5\}$ обозначает множество, состоящее из двух элементов — чисел 2 и 5.

Говорят, что множество A является подмножеством B ($A \subset B$), если все элементы A являются также и элементами B.

Множества A и B равны ($A = B$), если они состоят из одних и тех же элементов, т. е. если $A \subset B$ и $B \subset A$.

Если A — подмножество B, не равное всему B, то A называют собственным подмножеством B.

Пустое множество $\varnothing$ не содержит ни одного элемента и является подмножеством любого множества.

Пересечение AB двух множеств A и B состоит из элементов, которые принадлежат обоим множествам A и B:

$$A \cap B = \{x \mid x \in A \text{ и } x \in B\}.$$

Объединение множеств $A \cup B$ состоит из элементов, которые принадлежат хотя бы одному из множеств A и B:

$$A \cup B = \{x \mid x \in A \text{ или } x \in B\}.$$

Разность множеств $A \backslash B$ состоит из элементов, которые принадлежат A, но не принадлежат B:

$$A \backslash B = \{x \mid x \in A \text{ и } x \notin B\}.$$

Симметрическая разность множеств $A \Delta B$ состоит из элементов, которые принадлежат только одному из множеств A и B:

$$A \Delta B = (A \backslash B) \cup (B \backslash A) = (A \cup B) \backslash (A \cap B).$$

Приведем перечень операций, выполняемых над множествами, которые часто включаются в реализацию различных абстрактных типов данных.

1. Операции `Union(A,B,C)`, `Intersection(A,B,C)` и `Difference(A,B,C)` имеют в качестве входных параметров множества A и B, а в качестве выходного параметра — множество C, равное, соответственно, $A \cup B$, $A \cap B$, $A \backslash B$.
2. Операция `Merge(A,B,C)` — слияние (или объединение) непересекающихся множеств A и B в множество C. Эта операция не определена, если A и B имеют общие элементы.
3. Операция `Member(x,A)` в качестве входных параметров имеет множество A и значение x — объект того же типа, что и элементы A, а возвращает логическое значение `True`, если $x \in A$, или `False` — в противном случае.
4. Операция `Insert(x,A)` включает элемент x в множество A: $A \cup \{x\} \Rightarrow A$.
5. Операция `Delete(x,A)` удаляет элемент x из множества A: $A \backslash \{x\} \Rightarrow A$.
6. Операции `Min(A)` и `Max(A)` определяют, соответственно, наименьший и наибольший элементы множества A. На элементах множества A при этом должно быть определено отношение порядка.
7. Операция `Find(x)` для набора непересекающихся множеств возвращает имя (единственное) множества, в котором находится элемент x.

6.2. Стандартные способы реализации множества

Множество, по моему представлению, подобно бездне.

Георг Кантор

Выбор реализации такого абстрактного типа данных как множества осуществляется на основе выполняемых операций и мощности (размера) множества. Для небольших множеств, между элементами которых и элементами множества {1, 2, 3, ..., n} можно установить взаимно однозначное соответствие (при фиксированном значении n), наилучшей реализацией является представление в виде двоичного (булева) вектора, в котором i-й бит равен 1 (`True`), если i является элементом множества. Операции `Member`, `Insert` и `Delete` выполняются в этом случае за константное время (прямая адресация к соответствующему биту), а операции `Union`, `Intersection` и `Difference` — за время, пропорциональное n.

Рассмотрим в качестве примера язык программирования Паскаль, в котором для работы с небольшими множествами предусмотрен встроенный тип данных `Set`. Максимально допустимое значение n при этом зависит от конкретной реализации языка программирования (точнее, от используемого компилятора). Если величина n сопоставима с размером машинного слова, то операции `Union`, `Intersection` и `Difference` можно выполнить с помощью простых логических операций.

Пусть A, B и C — множества одного типа. В табл. 6.1 приведены реализации основных операций над ними.

Таблица 6.1

Операция	Обозначение	Реализация
`Union`	$C = A \cup B$	`C:=A Or B`
`Intersection`	$C = A \cap B$	`C:=A And B`
`Difference`	$C = A \backslash B$	`C:=A And Not B`

Другим способом представления множеств является использование линейного списка, элементы которого описывают элементы множества. В этом случае снимается ограничение на величину n.

Пусть списки L_1, L_2 и L_3 используются для представления множеств A, B и C (которые состоят из элементов одного типа). Тогда операция `Union` сводится к перезаписи элементов списков L_1 и L_2 в список L_3; при этом повторяющиеся в обоих исходных списках элементы должны быть записаны в результирующий только один раз. Время выполнения этой операции пропорционально $O(n \cdot m)$, где n и m — количества элементов в списках L_1 и L_2. Аналогична и реализация других операций (как по логике, так и по времени выполнения): `Intersection` требует поиска общих элементов в списках L_1 и L_2, а `Difference` — поиска элементов списка L_1, не принадлежащих списку L_2.

Ситуация несколько улучшается по временно́й характеристике при использовании упорядоченных списков. В этом случае временна́я оценка равна $O(n + m)$. Например, при реализации операции `Union` для пары упорядоченных списков можно одновременно просматривать оба этих списка (в одном цикле), выполняя сравнение очередных элементов. В результирующий список при этом записывается меньший элемент, а в случае совпадения запись осуществляется однократно. После того как один из списков исчерпан, в результирующий список просто дописываются все элементы из второго списка.

Теоретически можно рассматривать и реализацию множеств с помощью двоичных деревьев поиска. Пусть деревья T_1, T_2 и T_3 представляют множества A, B и C соответственно. Тогда, например, логика выполнения операции `Union` сводится к следующим шагам:

- просмотреть T_1 и исключить из T_2 элементы, встречающиеся в T_1 (множество A и полученное множество B' не пересекаются);
- последовательно переписать элементы T_1 и оставшиеся элементы T_2 в T_3.

Очевидно, что и этот способ реализации не улучшает временны́е оценки выполнения операций.

Упражнения

1. Пусть $n = 2^{12} = 4096$. Используя двоичные векторы для представления множеств, реализуйте для них основные операции `Union`, `Intersection` и `Difference`.

2. Разработайте реализацию операций `Union`, `Intersection` и `Difference` для представления множеств в виде линейных упорядоченных списков.

3. Разработайте реализацию операций `Union`, `Intersection` и `Difference` для представления множеств в виде двоичных деревьев поиска.

6.3. Объединение непересекающихся множеств

> Люди на земле должны дружить. Не думаю, что можно заставить всех людей любить друг друга, но я желал бы уничтожить ненависть между людьми.
>
> *Айзек Азимов*

Существует целый класс задач, в которых основными операциями над множествами являются `Merge(A,B,C)` и `Find(i)`. Пусть есть некая последовательность из этих операций, и ее требуется эффективно обработать. Без потери общности предположим, что множества образуются из целых чисел от 1 до n и они не пересекаются, а также что множества имеют имена, которые являются целыми числами от 1 до n. Разумеется, два множества не могут иметь одинаковые имена.

Будем различать внутренние имена (`in_name`) и внешние имена (`out_name`) множеств, где и те и другие — это числа от 1 до n. В операции `Merge(A,B,C)` используются внешние имена, где каждому внешнему имени соответствует внутреннее и наоборот. Элементы, принадлежащие одному множеству, помечаются одним и тем же значением внутреннего имени. Будем считать, что элементы каждого множества образуют свой список, который реализуется при помощи массива (см. раздел 2.3). Адрес первого элемента списка, а также количество элементов в списке пр этом хранятся в отдельных массивах, индексом для обращения к элементам которых является значение внутреннего имени. Первоначально предполагаем, что `in_name[i] = i` для всех значений `i` от 1 до n.

Пример. Пусть $n = 12$ и в некоторый момент времени мы имеем множества {5, 6, 8}, {1, 4, 7, 10, 12}, {3} и {2, 9, 11} (они перечислены в соответствии со значениями их внутренних имен). Внешние имена этих множеств — 3, 1, 4, 2. Состояние структур данных в этом случае отражено на рис. 6.1.

	in_name	next
1	2	4
2	4	9
3	3	0
4	2	7
5	1	6
6	1	8
7	2	10
8	1	0
9	4	11
10	2	12
11	4	0
12	2	0

	out_name	in_name
1	2	
2	4	
3	1	
4	3	
5	0	

in_name	head	size	out_name
1	5	3	3
2	1	5	1
3	3	1	4
4	2	3	2
5	0	0	0

Рис. 6.1. Состояние структур данных для подмножеств {5, 6, 8}, {1, 4, 7, 10, 12}, {3} и {2, 9, 11}

Пусть выполняется операция `Merge(1,2,5)`. Внешнему имени (`out_name`) 1 соответствует внутреннее имя (`in_name`) 2, а внешнему имени 2 — внутреннее имя 4. Список с внутренним именем 4 короче, поэтому мы его подключаем к списку с именем 2 путем изменения значений элементов `in_name`, а в поле `next` последнего элемента записываем адрес первого элемента списка, имеющего на момент начала вставки больший размер, т. е. вставка более короткого списка осуществляется в начало более длинного списка («вливаем меньшее множество в большее»). После этого останется откорректировать поля `head` и `size` нового списка, а также соответствие внешних и внутренних имен. Результат этих действий для рассматриваемого примера отражен на рис. 6.2.

	in_name	next
1	2	4
2	2	9
3	3	0
4	2	7
5	1	6
6	1	8
7	2	10
8	1	0
9	2	11
10	2	12
11	2	0
12	2	0

	out_name	in_name
1		0
2		0
3		1
4		3
5		2

	in_name	head	size	out_name
1		5	3	3
2		2	8	5
3		3	1	4
4		0	0	0
5		0	0	0

Рис. 6.2. Результат выполнения операции `Merge(1,2,5)`

Формализованная запись этих действий имеет вид:

```
Procedure Merge(a,b,k:Word);
  Var i, j, q, w, t, last:Word;
  Begin
    i:=in_name[a]; {Внутреннее имя первого множества}
    j:=in_name[b]; {Внутреннее имя второго множества}
    {Определяем множество меньшей мощности}
    If size[i]<size[j] Then Begin
        q:=head[i];
        w:=i;
        t:=j;
      End
    Else Begin
        q:=head[j];
        w:=j;
        t:=i;
      End;
    While q<>0 Do Begin
    {Просматриваем элементы меньшего множества}
      name[q]:=t;
      {Изменяем внутренние имена}
      last:=q;
      {Определяем адрес последнего элемента списка}
```

```
    q:=next[q];
  End;
  next[last]:=head[t];
  {Связываем списки; меньший по размеру
  стал началом большего списка}
  head[t]:=head[w];
  {Изменяем указатель на начало списка}
  size[t]:=size[t]+size[w];
  {Новый размер списка}
  {Корректируем соответствие внешних и внутренних
  имен}
  in_name[k]:=t;
  in_name[a]=0;
  in_name[b]:=0;
  out_name[t]:=k;
  head[w]:=0;
  size[w]:=0;
  out_name[w]:=0;
End;
```

Максимальное количество операций `Merge`, которые можно выполнить с помощью данного алгоритма, равно $n - 1$. Всякий раз, когда элемент перемещается из одного списка в другой, он оказывается в списке, который по крайней мере в два раза длиннее прежнего, — значит, общее количество перемещений имеет порядок $O(\log_2 n)$, а время выполнения $n - 1$ операций `Merge` — $O(n \cdot \log_2 n)$.

Операция `Find(i)` выполняется за константное время. По значению `i` определяется `in_name[i]`, по которому в свою очередь находится `out_name[in_name[i]]`. Последовательность из m операций `Find(i)` будет выполнена за время $O(m)$, а последовательность из $n - 1$ операций `Merge(a,b,k)` и m операций `Find(i)` имеет временну́ю оценку $O(\max(m, n \cdot \log_2 n))$.

Упражнения

1. Дано некое разбиение множества $\{1, 2, \ldots, n\}$ на подмножества, представленное описанием, аналогичным показанному на рис. 6.1. Найдите последовательность операций `Merge(a,b,k)`, приводящих к данному разбиению, и проследите весь процесс изменения состояния структур данных, описывающих представление множеств.

2. Пусть $n = 16$ и первоначально множество разбито на 16 подмножеств типа $\{i\}$, где $i = 1, 2, \ldots, 16$. Найдите множество, которое получается в результате работы следующей процедуры:

```
Procedure Solve;
  Var i:Word;
  Begin
    i:=1;
    While i<=15 Do Begin
      Merge(i,i+1,i);
      i:=i+2;
    End;
    i:=1;
    While i<=13 Do Begin
      Merge(i,i+2,i);
      i:=i+4;
    End;
    Merge(1,5,1);
    Merge(9,13,9);
    Merge(1,9,1);
  End;
```

Графически представьте изменения состояния структур данных, описывающих представление множеств, в процессе работы приведенной выше процедуры `Solve`.

3. Пусть для представления списков используются не массивы, а их организация с помощью указателей (ссылочного типа данных). Изобразите графически структуры данных, соответствующие этому представлению, и модифицируйте соответствующим образом процедуру `Merge`.

4. Напишите завершенную программу выполнения последовательности операций `Merge` и `Find`, выполняемых на наборе множеств, элементы которых представлены целыми числами от 1 до n, когда основной структурой данных, используемой для описания множеств в памяти компьютера, является массив. Выходные данные в этом случае представляют собой последовательность ответов на операции `Find` из заданной последовательности.

6.4. Использование древовидных структур данных в задаче объединения непересекающихся множеств

> Всегда старайтесь работать настолько близко к границе ваших способностей, насколько это возможно. Поступайте так потому, что это единственный способ эту границу определить и отодвинуть.
>
> *Эдсгер Дейкстра*

Рассмотрим более эффективный, чем описано в предыдущем разделе, метод реализации операций `Merge(A,B,C)` и `Find(i)` с непересекающимися множествами. По-прежнему будем считать, что существует некая последовательность этих операций, и предполагаем, что множества образованы из целых чисел от 1 до n.

Допустим, что каждое множество A представлено корневым деревом T_A, вершинам которого поставлены в соответствие элементы из A, а корню этого дерева, кроме элемента, поставлено в соответствие имя этого множества. Тогда операция `Merge(A,B,C)` сводится к преобразованию корня дерева T_A в сына корня T_B (а можно и наоборот) и к замене имени в корне дерева T_B на C. Операция `Find(i)` же реализуется путем определения положения вершины, представляющей элемент i в каком-то дереве T, и прохождения пути из этой вершины к корню T. Имя, приписанное корню, указывает множество, которому принадлежит элемент i.

При описанной логике обработки операция `Merge(A,B,C)` реализуется за константное время, а временна́я сложность операции `Find(i)` пропорциональна длине пути от вершины i до корня. В результате выполнения $n - 1$ операции `Merge(A,B,C)` может получиться дерево, представленное на рис. 6.3. Это могут быть операции: `Merge(1,2,2)`, `Merge(2,3,3)`, ..., `Merge(n-1,n,n)`. Тогда временна́я сложность n последовательных операций `Find(i)` равна $O(n^2)$ — $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$. Эту оценку можно уменьшить за счет использования приема (эвристики), рассмотренного в предыдущем разделе: в каждом дереве под-

Рис. 6.3. Дерево, получающееся в результате выполнения $n - 1$ операции `Merge`

считывается количество вершин и при слиянии меньшее дерево присоединяется к корню большего.

В этом случае высота дерева может достичь значения h, только если оно имеет не менее 2^h вершин. Действительно, это верно при $h = 0$ и $h = 1$. Предположим, что оно верно при всех значениях h до некоторого значения k включительно. Пусть T — дерево высоты $k + 1$ с наименьшим количеством вершин, которое получается в результате слияния двух деревьев T_1 и T_2. При этом T_1 является деревом с высотой k и имеет не больше вершин, чем T_2. По предположению, каждое из деревьев T_1 и T_2 с высотой k имеет не менее 2^k вершин, тогда у T будет не менее 2^{k+1} вершин.

Описанное выше правило слияния деревьев улучшает временну́ю оценку выполнения n операций `Merge(A,B,C)` и `Find(i)`. Действительно, никакое дерево не может иметь высоту больше $\log_2 n$. Следовательно, временна́я сложность $O(n)$ операций `Merge(A,B,C)` и `Find(i)` не превосходит $O(n \cdot \log_2 n)$.

Вторую эвристику (правило), используемую в рассматриваемой задаче, называют *сжатием путей*. При выполнении операции `Find(i)` мы проходим некий путь от вершины i до корня дерева r. Выпишем все вершины на этом пути: $i, v_1, v_2, \ldots, v_{n-1}, v_n, r$ и сделаем вершины $i, v_1, v_2, \ldots, v_{n-1}$ сы-

новьями (детьми) корня *r*. Это и есть сжатие путей (рис. 6.4). Очевидно, что следующая операция `Find(i)` для какой-либо вершины из этого пути будет выполняться за меньшее время, чем то, которое было бы затрачено без сжатия путей, и в целом это приведет к уменьшению времени обработки *n* операций `Find(i)`.

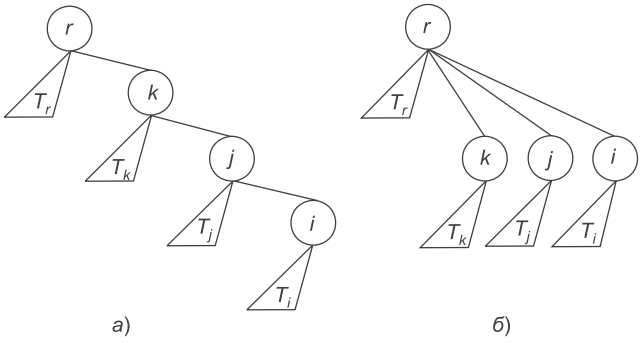


Рис. 6.4. Правило сжатия путей:
а) — до сжатия; *б*) — после сжатия

Покажем описанную выше организацию данных для задачи выполнения последовательности операций `Merge(A,B,C)` и `Find(i)` на конкретном примере. Пусть $n = 12$. В начальный момент времени каждый элемент *i* образует одноэлементное множество $\{i\}$, т. е. мы имеем «лес» из двенадцати деревьев (рис. 6.5), и каждая вершина является корнем дерева (массив `root`). Внешние и внутренние имена множеств совпадают. Внутренние имена в данном случае — это номера элементов, но по аналогии с предыдущим разделом будем называть их «именами». Количество вершин (элементов) в каждом дереве (массив `size`) равно единице, а указатель на отца вершины (массив `father`), естественно, равен нулю.

Пусть выполняется последовательность операций: `Merge(1,3,1);Merge(5,7,5);Merge(9,11,9);Merge(2,4,2);Merge(6,8,6);Merge(10,12,10)`. Состояние структур данных после ее завершения приведено на рис. 6.6. Так, отцом вершины с `in_name`, равным 3, является вершина 1. Количество вершин в дереве равно 2, внешнее имя — 1. Аналогично — и для остальных пяти деревьев.

out_name	root
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12

in_name	size	out_name	father
1	1	1	0
2	1	2	0
3	1	3	0
4	1	4	0
5	1	5	0
6	1	6	0
7	1	7	0
8	1	8	0
9	1	9	0
10	1	10	0
11	1	11	0
12	1	12	0

Рис. 6.5. Начальная организация структур данных

out_name	root
1	1
2	2
3	0
4	0
5	5
6	6
7	0
8	0
9	9
10	10
11	0
12	0

in_name	size	out_name	father
1	2	1	0
2	2	2	0
3	0	0	1
4	0	0	2
5	2	5	0
6	2	6	0
7	0	0	5
8	0	0	6
9	2	9	0
10	2	10	0
11	0	0	9
12	0	0	10

Рис. 6.6. Состояние структур данных после серии операций `Merge`: `Merge(1,3,1); Merge(5,7,5); Merge(9,11,9); Merge(2,4,2); Merge(6,8,6); Merge(10,12,10)`

Зададим себе вопрос: можно ли не присваивать значение «нуль» в массиве `root` для вершин, которые перестали быть корнями деревьев? Предположим, что после этой серии опе-

раций идет, например, операция `Merge(8,9,8)`, а элемент `root[8]` ≠ 0. С точки зрения логики выполнения операции `Merge`, она будет допустима, но приведет к необратимым последствиям. Остается или возложить вопрос о корректности вызова операций на пользователя, или предусмотреть эту ситуацию, которая требует разделения внешних и внутренних имен и поддержки соответствия между ними. Итак, подчеркнем еще раз: параметрами операции `Merge` являются внешние имена множеств, а параметром операции `Find` — номер элемента или его внутреннее имя.

Итак, выполнение операции `Merge` сводится к нахождению корней деревьев с помощью массива `root`, а затем к действиям, приводящим к тому, что корень меньшего дерева становится сыном большего дерева. Программная реализация такой операции имеет вид:

```
Procedure Merge(i,j,k:Word);
{i, j, k – имена множеств, внешние имена}
  Var w,q,t:Word;
  Begin
    If (root[i]=0) Or (root[j]=0) Then Exit;
    {Вызов операции не является корректным}
  {Определяем меньшее по количеству вершин дерево}
    If size[root[i]]<size[root[j]]
      Then Begin
        w:=root[i];
        {Внутреннее имя корня меньшего дерева}
        q:=root[j];
        {Внутреннее имя корня большего дерева}
      End
      Else Begin
        w:=root[j];
        q:=root[i];
      End;
    father[w]:=q;
    {Корень меньшего дерева становится
    сыном корня большего дерева}
    size[q]:=size[q]+size[w];
    {Изменяем количество вершин в дереве}
    size[w]:=0;
    {Убираем данные о старом дереве}
    out_name[w]=0;
```

```
    out_name[q]:=k;
    {Корректируем внешнее имя полученного дерева}
    t:=q;
    root[i]:=0;
    root[j]:=0;
    root[k]:=t;
    {Внутренним именем нового дерева является имя
    корня большего дерева}
  End;
```

При выполнении операции `Find` осуществляется проход от вершины `i` к корню дерева с помощью адреса предка вершины из массива `father`; при этом путь запоминается в рабочем массиве `A`. Когда же корень дерева найден и путь известен, мы корректируем адреса предков вершин, принадлежащих этому пути.

```
Procedure Find(i:Word);
{i - элемент множества, он же - внутреннее имя}
  Var cnt, v, j:Word;
      A:Array[1..n] Of Word;
  Begin
    cnt:=0;
    v:=i;
    While father[v]<>0 Do Begin
    {Идем от вершины до корня}
      cnt:=cnt+1;
      A[cnt]:=v; {Запоминаем вершину}
      v:=father[v];
    End;
    <вывод out_name[v]>;
    {Вывод имени множества, к которому принадлежит
    элемент i}
    For j:=1 To cnt-1 Do father[A[j]]:=v;
    {У всех вершин пути, кроме последней, изменяем
    ссылку на родителя}
  End;
```

Математическое отступление[1]. Определим функцию F следующим образом: $F(0) = 1$ и $F(i) = 2^{F(i-1)}$ для $i > 0$. Для

[1] См. книгу *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. С. 155.

примера вычислим несколько значений: $F(0) = 1$, $F(1) = 2$, $F(2) = 4$, $F(3) = 16$, $F(4) = 2^{16} = 65536$, $F(5) = 2^{65536}$. Как видим, функция F растет сверхбыстро. Пусть функция $G(i)$ определяется как наименьшее k такое, что $F(k) > i$. При этом функция G растет очень медленно — $G(i) \leqslant 5$ практически для всех реальных значений i, а именно для $i \leqslant 2^{65536}$.

Известно, что алгоритм, использующий описанные две эвристики, выполняет последовательность из $c \cdot n$ операций `Merge` и `Find` за время, не большее $c' \cdot n \cdot G(n)$, где c и c' — константы, причем c' зависит от c. Эта оценка лучше, чем $O(n \cdot \log_2 n)$. С практической точки зрения, мы имеем более эффективный алгоритм, хотя временна́я сложность при этом и не является линейной функцией.

Упражнения

1. Приведите пример последовательности операций `Merge` для n-элементного множества и проследите (графически изобразите) процесс изменения структур данных при выполнении этой последовательности операций.

2. Дано описание разбиения n-элементного множества на непересекающиеся подмножества в виде конечного состояния структур данных. Найдите последовательность операций `Merge`, в результате выполнения которых получено данное множество.

3. Предположим, что вторая эвристика (сжатие путей) не используется. Приведите пример из n операций `Merge` и `Find` на n-элементном множестве, для которых временна́я оценка равна $O(n \cdot \log_2 n)$.

4. Напишите рекурсивный вариант реализации операции `Find`.

5. Напишите завершенную программу выполнения последовательности операций `Merge` и `Find`, выполняемых на наборе множеств, элементы которых представлены целыми числами от 1 до n, где основой для описания множеств в памяти компьютера являются древовидные структуры. Выходным параметром при этом является последовательность ответов на операции `Find`.

6.5. Словари и хеширование

> Подобно тому, как все искусства
> тяготеют к музыке, все науки
> стремятся к математике.
>
> *Джордж Сантаяна*

Множества, на которых определены три операции — `Member(x,A)`, `Insert(x,A)` и `Delete(x,A)`, называют *словарями*.

Возможны различные способы реализации словарей.

Первый из них — это использование упорядоченного или неупорядоченного линейного списка. В этом случае указанные операции сводятся к обычным операциям вставки, удаления и поиска элемента в списке.

Если элементам множества можно поставить в соответствие целые числа от 1 до n, то множество можно описать двоичным (булевым) вектором; в этом случае выполнение операций сводится к изменению соответствующего бита и проверке его значения.

Третий способ сводится к использованию массива с указателем на последнюю используемую ячейку. Этот вариант допустим, если известно, что мощность множества не превысит некоторую величину. Операция `Insert(x,A)` при этом сводится к записи элемента в конец массива по значению указателя на последний элемент; операция `Delete(x,A)` — к поиску элемента в массиве и его удалению путем сдвига или записи на его место последнего элемента массива; операция `Member(x,A)` — это опять-таки поиск элемента в массиве.

Итак, второй способ более эффективен по временны́м характеристикам операций, но применим только для небольших множеств, а первый и третий таковы, что время выполнения отдельных операций пропорционально n.

Известен также и применяемый во многих приложениях еще один способ реализации словарей, называемый *хешированием*. В этом случае снимается ограничение на размер множества и требуется константное (в среднем) время на выполнение операций, хотя в худшем случае оно попрежнему пропорционально размеру множества.

Ключевой идеей метода хеширования является прямая адресация к ячейкам памяти, в которых хранится информация об элементах множества. Если проводить аналогию, то это — развитие идеи прямой адресации к элементам массива по индексу или по номеру элемента массива.

Пусть каждый элемент множества характеризуется уникальным ключом (обозначим его как x), причем количество различных значений этого ключа ограничено сверху значением n. Суть метода сводится к тому, чтобы с помощью некой функции h (она называется *хеш-функцией*) вычислить значение $h(x)$ (*хеш-значение* x) и трактовать его как адрес элемента в таблице (*хеш-таблице*) размера t. Отсюда следует, что

- значения h находятся в интервале от 0 до $t - 1$;
- вычисление h должно осуществляться быстро;
- количество совпадений (их называют *коллизиями*) $h(x_1) = h(x_2)$ при $x_1 \neq x_2$ (а они неизбежны при $n \gg t$) должно быть минимальным.

Традиционная схема организации данных при хешировании приведена на рис. 6.7. Хеш-таблица здесь — это массив `M[0..t-1]`. Адрес элемента таблицы вычисляется с помощью хеш-функции h. Все элементы множества, имеющие

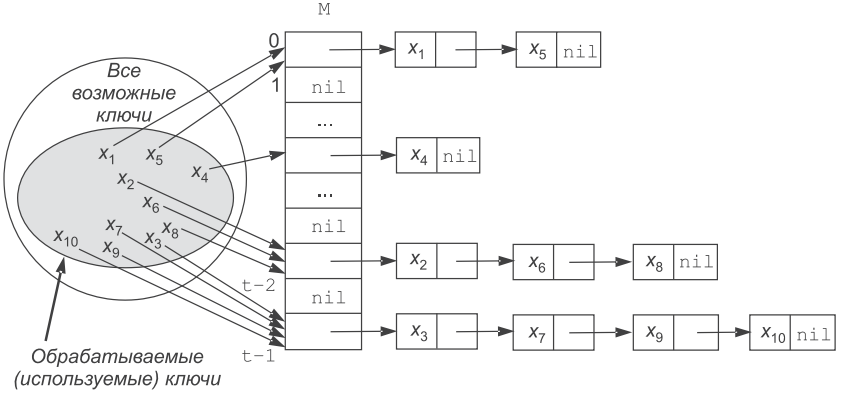


Рис. 6.7. Схема организации данных при хешировании. Коллизии разрешаются с помощью списков: в позиции `M[i]` хранится указатель на список элементов множества с хеш-значением `i`

одно значение h, увязываются в список с указателем на первый элемент этого списка, хранящимся в M.

При такой организации данных операции Member(x,A), Insert(x,A) и Delete(x,A) сводятся к вычислению значения хеш-функции и выполнению соответствующих операций со списком элементов. Вставка элемента в множество — это добавление элемента к началу списка; поиск — просмотр элементов списка; удаление элемента из множества — изъятие элемента из списка.

Выбор хеш-функции

Пример. Предположим, что нужно анализировать текст из специфической страны «Д», в которой все слова начинаются с буквы «П», а в качестве h выбран цифровой аналог первой буквы. В этом случае получаем одно значение h, т. е. один адрес в M для всех слов, — следовательно, все слова страны «Д» будут организованы в один список. От чего мы уходили, к тому и пришли, но в стране «Д» все возможно. ☺

Хеш-функция, кроме того что она вычисляется достаточно быстро, должна работать так, чтобы количество коллизий для множества элементов было минимальным, другими словами, она должна равномерно «разбрасывать» данные по таблице (английский глагол *to hash* как раз и означает «мелко порубить, перемешивая»). Теоретически для любой хеш-функции можно подобрать исходные данные, для которых возможен вырожденный случай, приведенный в нашем примере про страну «Д», но обычно на практике этот метод достаточно эффективен, т. е. в среднем время выполнения операций Member(x,A), Insert(x,A) и Delete(x,A) составляет $O(1)$.

Рассмотрим различные примеры хеш-функций.

1. Пусть ключами элементов множества являются строки из восьми символов. В Паскале есть встроенная функция Ord(a), возвращающая целочисленный код символа a. Тогда достаточно хорошей (но, вероятно, не отличной) будет следующая хеш-функция:

```
Function h(x:String): 0..t-1;
  Var i, sum:Integer;
  Begin
    sum:=0;
    For i:=1 To 8 Do sum:=sum+Ord(x[i]);
```

```
    h:=sum Mod t;
  End;
```

2. Если область определения хеш-функции есть множество целых неотрицательных чисел, то каждому значению ключа x можно поставить в соответствие остаток от его деления на t: $h(x) = x \texttt{ Mod } t$. Обычно в качестве t выбирается простое число, достаточно далеко отстоящее от чисел, равных степени двойки. Действительно, если $t = 2^q$, то $h(x)$ — это q младших битов числа. При выборе в качестве t числа, близкого к 2^q (например, $2^q - 1$), значением h является младшая цифра числа в 2^q-ичной системе счисления, тогда при перестановке цифр числа (кроме младшей) для этой совокупности ключей получается одно значение хеш-функции.

3. Построение хеш-функции путем умножения выглядит следующим образом: $h(x) = \lfloor t \cdot ((x \cdot A) \texttt{ Mod } 1) \rfloor$, где A — константа в интервале $0 < A < 1$. В этом случае значение t незначительно влияет на результат и может выбираться как число, являющееся степенью двойки. Значение $(x \cdot A) \texttt{ Mod } 1$ — дробная часть произведения. Выбор константы A достаточно произволен и обычно определяется хешируемыми данными. Т. Кормен[1] и другие со ссылкой на Д. Кнута утверждают, что значение $A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$ является достаточно хорошим.

Пример. Пусть $x = 9876543$ и $t = 100000$, тогда адрес в хеш-таблице вычисляется как

$$h(x) = \lfloor 100000 \cdot (9876543 \cdot 0.610339887 \texttt{ Mod } 1 \rfloor = \\ = \lfloor 100000 \cdot (6028048.138570641 \texttt{ Mod } 1 \rfloor = \\ = \lfloor 100000 \cdot 0.13857\ldots \rfloor = 13857.$$

4. При построении хеш-функции обычно стремятся как можно сильнее «перемешивать» разряды в двоичном представлении ключа — это, как правило, приводит к более равномерному заполнению хеш-таблицы. Один из способов заключается в нахождении произведения чисел, полученных из младших и старших битов ключа, и выборке из результата q каких-либо разрядов (лучше — средних). Значение q выбирается так, чтобы 2^q было равно t.

[1] *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 223.

```
Function h(x:LongInt):Integer;
{Считаем, что хеш-таблица состоит из 512 элементов
(t=512), а ячейка памяти - из 32 разрядов
Для переменных типа LongInt выделяется 32 разряда,
а типа Integer - 16 разрядов}
  Var w,v:LongInt;
  Begin
    v:=(x And $FFFF0000) Shr 16;
    {Выделяем старшие разряды числа и сдвигаем
    на 16 разрядов. Символ $ - признак
    шестнадцатеричной константы}
    w:=(x And $FFFF);
    {Выделяем младшие разряды числа}
    h:=(Integer(v*w) And $7FC0) Shr 6;
    {Функция Integer преобразует представление
    результата из типа LongInt в тип Integer.
    Выделяем 9 битов из произведения чисел (q=9)}
  End;
```

5. Рассмотрим еще один традиционный способ «перемешивания» разрядов. Предположения о размере таблицы и количестве разрядов в двоичном представлении ключа здесь те же, что и в предыдущем варианте построения хеш-функции. В данном случае разряды в двоичном представлении ключа x разбиваются на группы по q разрядов в каждой (последняя группа может быть неполной). Например, выполняется логическое сложение этих групп и его результатом является значение хеш-функции.

```
Function h(x:LongInt):Integer;
  Var w,v:Integer;
  Begin
    w:=Integer(x And $1FF);
    {Выделение младших 9 битов числа}
    v:=Integer(x And $3FE00 Shr 9);
    {Выделение следующих 9 битов числа}
    w:=w Xor v;
    {Первый шаг формирования адреса элемента
    таблицы}
    v:=Integer(x And $7FC0000 Shr 18);
    {Выделение следующей группы разрядов числа}
```

```
    w:=w Xor v;
    v:=Integer(x And $F8000000 Shr 27);
    {Выделение оставшихся разрядов}
    w:=w Xor v;
    h:=w;
  End;
```

В практике использования хеширования есть и другой способ разрешения коллизий. В одних источниках он называется закрытым хешированием[1], в других — открытым[2], но суть его в любом случае заключается в том, что ключи хранятся в самой хеш-таблице, и при коллизиях (повторим, что они неизбежны) производится сравнение ключей. В этом случае указатели не используются.

Пример. Пусть $t = 10$ и с помощью хеш-функции обработаны ключи a, b, c, d, e, f, g, k (именно в этой последовательности!), для которых хеш-значения равны $h(a) = 2$, $h(b) = 3$, $h(c) = 7$, $h(d) = 8$, $h(e) = 2$, $h(f) = 3$, $h(g) = 7$, $h(k) = 8$, соответственно. Состояние такой хеш-таблицы приведено на рис. 6.8.

0	*k*
1	
2	*a*
3	*b*
4	*e*
5	*f*
6	
7	*c*
8	*d*
9	*g*

Рис. 6.8. Состояние хеш-таблицы

[1] *Ахо А. В., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. М.: Издательский дом «Вильямс», 2001. С. 120.

[2] *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 226.

При возникновении коллизии мы «идем вниз» по таблице в поиске свободного места (когда, например, начальные значения элементов таблицы равны нулю). Так, при вставке элемента «*e*» мы получаем адрес 2, но эта ячейка занята. Тогда мы берем следующую ячейку — и она занята. Наконец, находим свободную ячейку (4) и записываем в нее значение ключа. Аналогично поступаем и для ключей *f*, *g*, *k*. Операция вычисления адреса следующей ячейки таблицы при этом осуществляется по модулю *t*.

Описанный в вышеприведенном примере способ разрешения коллизий называют *линейным хешированием*. Если через *i* обозначить номер попытки, то $h_i(x) = (h(x)+i)$ `Mod` t, где значение *i* изменяется от 0 до $t-1$. Если же свободное место не найдено, то хеш-таблица считается заполненной.

Мы рассмотрели вставку элемента в множество, т. е. операцию `Insert`. Аналогично выполняется и операция `Member` — вычисление адреса элемента осуществляется по тому же принципу. С операцией же `Delete` дело обстоит чуть сложнее. Если удалить найденный элемент путем записи на его место, например, значения «нуль» (предполагаем, что его нет среди значений ключа), то станет невозможной работа с элементами, которые были записаны в таблицу после удаляемого элемента, так что при их записи было прохождение через ячейку, где находится удаляемый ключ. Различные же схемы сжатия приводят к потере эффективности. Возможным вариантом решения является запись на это место специального признака (например, «$»), который воспринимается при поиске как занятый элемент таблицы, а при записи — как свободный.

Эта идея разрешения коллизий может быть использована и с другими вариантами вычисления адреса следующей ячейки. Так, использование функции $h_i(x) = (h(x) + c_1 \cdot i + c_2 \cdot i^2)$ `Mod` t называют *квадратичным хешированием* (константы c_1 и c_2 здесь выбираются не случайно, а под конкретную задачу), но наилучшим способом является *двойное хеширование*: $h_1^i(x) = (h_1(x) + i \cdot h_2(x))$ `Mod` t, где h_1 и h_2 — различные хеш-функции. В этом случае последовательность адресов хеш-таблицы является арифметической прогрессией с первым членом $h_1(x)$ и шагом $h_2(x)$.

Упражнения

1. Пусть используется хеш-функция $h(x) = x$ `Mod` t, где $t = 11$, а x — целое неотрицательное число. Как будут выглядеть хеш-таблица и списки элементов, имеющих одно хеш-значение, для последовательности чисел: 5, 13, 25, 47, 16, 24, 36, 90, 2, 35, 40, 3, 7?

2. Дана хеш-таблица со списками элементов и используемая хеш-функция. Восстановите одну из возможных последовательностей ключей, в результате обработки которых может быть получена эта хеш-таблица.

3. Считаем, что средняя длина списков равна сумме длин списков, деленной на количество списков. Значение t известно. Напишите программу, оценивающую среднюю длину списков для различных хеш-функций, рассмотренных в данном разделе. Оценка должна осуществляться на одних и тех же исходных данных; количество ключей (элементов множества) n значительно больше значения t.

4. Предположим, что $t = 5$ и используется хеш-функция $h(x) = x$ `Mod` 5 с линейным способом разрешения коллизий. Покажите результат вставки элементов 13, 28, 48, 35, 4 в первоначально пустую таблицу.

5. Определим α как степень заполнения хеш-таблицы. Для пустой таблицы $\alpha = 0$, для заполненной $\alpha = t/(t + 1)$, где t — размер таблицы. Например, значение $\alpha = 0.1$ говорит о том, что таблица заполнена на 10%. Пусть имеется хеш-таблица с известной степенью заполнения и случайно выбранный ключ x. Количество попыток записи ключа в хеш-таблицу обозначим как m. Если процесс записи выполнить n раз со случайными ключами, то можно оценить среднее значение m (обозначим его как $m_{ср}$). Напишите программу для оценки $m_{ср}$ при линейном хешировании. Насколько результаты вашей оценки будут отличаться от сведений, приведенных в табл. 6.2 (из книги Н. Вирта[1])?

---

[1] *Вирт Н.* Алгоритмы+Структуры данных=Программы. М.: Мир, 1985. С. 313.

Таблица 6.2

α	$m_{ср}$
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.99	10.50

Примечание. При выполнении упражнения первоначально следует по значению α заполнить хеш-таблицу случайным образом, но с данной степенью заполнения.

6. Выполните упражнение 5 для квадратичного хеширования с $c_1 = 1$ и $c_2 = 3$.

7. Пусть $t = 13$, $h_1(x) = x$ Mod 13, а $h_2(x) = 1 + (x$ Mod $11)$. Изобразите графически результат работы двойного хеширования для некоторой последовательности ключей.

8. Пусть $h_1(x) = x$ Mod t, а $h_2(x) = 1 + (x$ Mod $(t - 1))$. Выполните упражнение 5 для случая двойного хеширования с этими хеш-функциями.
Примечание. Оцените результат при различных значениях t (простое число, степень двойки и т. д.).

Методические комментарии

Полное, а не фрагментарное описание задачи «объединить и найти» можно отыскать, вероятно, только в двух книгах: Ф. Ахо, Дж. Хопкрофта, Дж. Ульмана[1)] и Т. Кормена, Ч. Лайзерсона, Р. Ривеста[2)].

Вопросы хеширования как метода поиска данных рассматриваются в том или ином объеме в любой книге по фундаментальным алгоритмам информатики. Упомянем здесь только «классику», а именно книги Н. Вирта[3)] и Д. Кнута[4)].

Следует отметить, что в числе первых, кто рассмотрел и исследовал метод хеширования, был и наш соотечественник — академик А. П. Ершов (1957 г.).

---

1) *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. С. 146–168.

2) *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 414–433.

3) *Вирт Н.* Алгоритмы+Структуры данных=Программы. М.: Мир, 1985. С. 303–314.

4) *Кнут Д.* Искусство программирования. Т. 3. М.: Мир, 1978. С. 601–675.

Глава 7

Очереди с приоритетами

Вкусы у меня простые. Я предпочитаю все самое лучшее.

Уинстон Черчилль

7.1. Двоичная куча и пирамидальная сортировка

Создается впечатление, что на примерах сортировок можно построить целый курс программирования.

Никлаус Вирт

Двоичную кучу, или *пирамиду*[1)], можно определить как двоичное дерево, в котором значение ключей у потомков (непосредственных, т. е. «детей») не больше значения ключа вершины, и это свойство выполняется для всех вершин дерева, не являющихся листьями. В дереве, являющемся кучей, все уровни (кроме, может быть, последнего) заполнены полностью. При представлении двоичной кучи в виде массива (A) вершине с номером i соответствует элемент $A[i]$ (значение ключа), а детям вершины — элементы $A[2 \cdot i]$ и $A[2 \cdot i+1]$. Отсюда следует, что должны выполняться неравенства $A[i] \geqslant A[2 \cdot i]$ и $A[i] \geqslant A[2 \cdot i+1]$. Таким образом, следует говорить, что элементы массива A образуют двоичную кучу (являются ею), если удовлетворяют указанным требованиям для всех значений i.

---

1) Эти понятия следует рассматривать как синонимы. Обычно использовалось понятие «пирамида», но после выхода книги Т. Кормена, Ч. Лейзерсона и Р. Ривеста первое понятие («двоичная куча») применяется более регулярно, и это несмотря на то, что понятие «куча» используется в информатике для обозначения области памяти, выделяемой в процессе выполнения программы для динамических данных.

Пример. На рис. 7.1 приведено изображение двоичной кучи в виде дерева и массива.

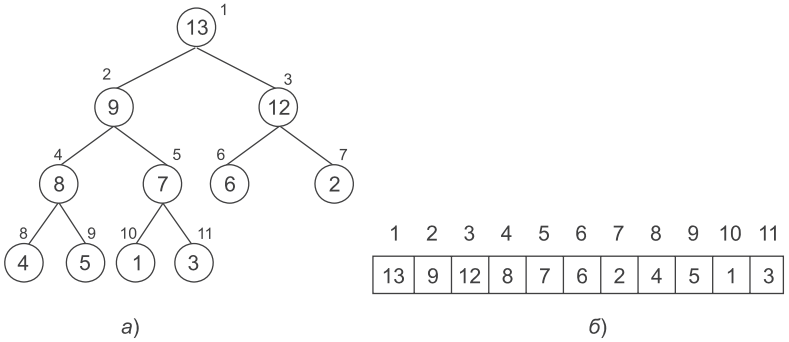


Рис. 7.1. Двоичная куча: *а*) представление в виде дерева; *б*) представление в виде массива. В вершинах дерева указаны значения ключей, а рядом с ними — индексы элементов массива

Сформулированное выше свойство кучи задает некое отношение порядка на множестве элементов. Однако куча должна обладать еще одним свойством, назовем его «*формой*». Идея формы лучше всего разъясняется графически — см. рис. 7.2.

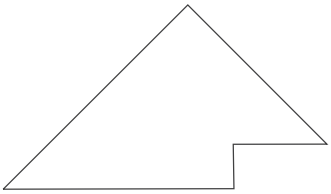


Рис. 7.2. Форма кучи

Таким образом, двоичное дерево, являющееся кучей, имеет свойство формы, если листья дерева находятся не более чем на двух уровнях, причем находящиеся на нижнем уровне вершины (листья) группируются слева. Скажем, двоичное дерево, удовлетворяющее первому свойству (наличие отношения порядка), но имеющее форму, графическое изображение которой приведено на рис. 7.3, не является кучей. В куче нет незанятых вершин, и если она содержит n вершин, то ни одна из них не может отстоять более чем на $\log_2 n$ от корня.

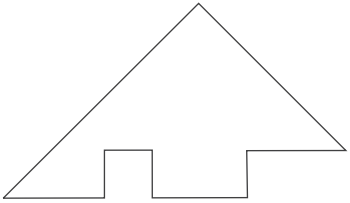


Рис. 7.3. Двоичное дерево, не являющееся кучей

Рассмотрим представление двоичной кучи в виде массива. Пусть дан произвольный массив и из него требуется сформировать кучу. Элементы массива с индексами от n `Div` $2 + 1$ до n образуют кучу по той простой причине, что удвоение индекса выводит нас за пределы массива. А затем, начиная с элемента $A[n$ `Div` $2]$ и заканчивая элементом $A[1]$, мы последовательно находим в построенной части кучи место для очередного элемента так, чтобы выполнялось свойство кучи и он (элемент) был в составе кучи. Если процедура `Push(i,n)` ответственна за нахождение места для элемента $A[i]$ в куче, то ее формализованная запись логики может выглядеть так:

```
Procedure BuildHeap;
  Var i:Integer;
  Begin
    For i:=n Div 2 DownTo 1 Do Push(i,n);
  End;
```

Примечание. Параметр n (количество элементов массива в данном случае, или «верхняя» граница кучи) пока в нашем обсуждении не требуется.

Заметим, что здесь выполнение второго свойства кучи (формы) осуществляется автоматически в силу логики размещения очередного элемента.

Пример. Пусть дан массив $A = \{2, 4, 13, 9, 5, 12, 8, 7, 3, 1, 6\}$ и необходимо сформировать из него двоичную кучу. Процесс такого преобразования представлен в табл. 7.1. Элементы $A[6...11]$ являются кучей (жирным шрифтом в табл. 7.1 выделена уже построенная часть кучи). На первой итерации $A[5]$ сравнивается с наибольшим из элементов $A[10]$ и $A[11]$. По результатам сравнения $A[5]$ записывается на 11-е место, а $A[11]$ — на 5-е. На второй и третьей итераци-

ях состояние массива не меняется, ибо в первом случае $A[4]$ больше и $A[8]$, и $A[9]$, а во втором случае $A[3]$ превосходит $A[6]$ и $A[7]$. На четвертой итерации на место $A[2]$ записывается $A[4]$; далее процесс «проталкивания» $A[2]$ на свое место продолжается, и на 4-е место попадает элемент $A[8]$, а $A[2]$ записывается на 8-е место. Аналогичные действия выполняются и с первым элементом массива на пятой итерации.

Таблица 7.1

Номер итерации	*A*										
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Начальное состояние	2	4	13	9	5	**12**	**8**	**7**	**3**	**1**	**6**
1-я итерация	2	4	13	9	**6**	**12**	**8**	**7**	**3**	**1**	**5**
2-я итерация	2	4	13	**9**	**6**	**12**	**8**	**7**	**3**	**1**	**5**
3-я итерация	2	4	**13**	**9**	**6**	**12**	**8**	**7**	**3**	**1**	**5**
4-я итерация	2	**9**	**13**	**7**	**6**	**12**	**8**	**4**	**3**	**1**	**5**
5-я итерация	**13**	**9**	**12**	**7**	**6**	**2**	**8**	**4**	**3**	**1**	**5**

Формализованная запись процесса «проталкивания» элемента по построенной части кучи имеет вид:

```
Procedure Push(r,t:Integer);
  Var i,j,v:Integer;
      pp:Boolean;
  Begin
    v:=A[r];
    {Запоминаем элемент}
    i:=r;
    {Индекс рассматриваемого элемента}
    j:=2*i;
    {Индекс элемента, с которым производится
    сравнение}
    pp:=False;
    {Считаем, что для элемента не найдено
    места в куче}
    While (j<=t) And Not pp Do Begin
      If (j<t) And (A[j]<A[j+1]) Then j:=j+1;
      {Сравниваем с большим элементом}
```

```
    If v>=A[j] Then pp:=True
    {Место для A[r] найдено}
      Else Begin
      {Переставляем элемент с номером j на место i
      и идем дальше по массиву}
        A[i]:=A[j];
        i:=j;
        j:=2*i;
      End;
  End;
  A[i]:=v;
  {Записываем A[r] на найденное для него место
  в куче}
End;
```

Очевидно, что временна́я сложность логики «проталкивания» элемента равна $O(\log_2 n)$, так как на каждом шаге индекс увеличивается вдвое, а количество увеличений не превосходит $\log_2 n$. В целом время построения кучи составляет $O(n \cdot \log_2 n)$, но фактически оно равно $O(n)$, что мы и покажем при обсуждении представления кучи в виде двоичного дерева.

Как реализуется сортировка элементов массива A с использованием двоичной кучи? По традиции мы назовем ее не «методом сортировки с помощью кучи», а «пирамидальной сортировкой» (*heapsort*). Пусть нам требуется отсортировать элементы массива A в порядке неубывания. Двоичная куча, или пирамида, строится за время $O(n \cdot \log_2 n)$, при этом максимальный элемент является первым в массиве. Поменяем его местами с последним элементом массива, и он окажется на своем месте, а затем «протолкнем» первый элемент по куче, — но она теперь уже состоит не из n, а из $n - 1$ элементов. Снова поменяем местами первый и $(n - 1)$-й элементы и вновь «протолкнем» первый элемент, но уже в куче из $n - 2$ элементов, и т. д. В результате получим массив A, отсортированный по неубыванию. Время «проталкивания» — $O(\log_2 n)$, количество итераций — $O(n)$, а общая оценка — $O(n \cdot \log_2 n)$, причем этот метод не требует дополнительной памяти.

Формализованная запись метода пирамидальной сортировки — следующая:

```
Procedure Sort;
  Var t,w,i:Integer;
  Begin
    t:=n Div 2+1;
    {Эта часть массива является кучей}
    For i:=t-1 DownTo 1 Do Push(i,n);
    {Формирование кучи (только один раз)}
    For i:=n DownTo 2 Do Begin
      w:=A[1];
      A[1]:=A[i];
      A[i]:=w;
      {Меняем местами первый и i-й элементы}
      Push(1,i-1);
      {"Проталкиваем" в куче первый элемент}
    End;
  End;
```

Следующей нашей задачей является рассмотрение представления кучи в виде двоичного дерева (рис. 7.1*а*). Пусть имеется двоичное дерево, и его требуется преобразовать в двоичную кучу. Описание вершины дерева при этом имеет традиционный вид:

```
Type pt=^el;
     el=Record
       data:Integer;
       left,right:pt;
     End;
Var root:pt; {Указатель на корень дерева}
```

Прежде чем работать с этой задачей, отвлечемся немного и обсудим функцию построения «правильного» дерева. Она возвращает в основную программу указатель на корень дерева и является, конечно же, рекурсивной. Параметр этой функции (`t`) определяет количество элементов, из которых строится дерево. При этом создается новая вершина, а затем рекурсивно строятся левое и правое поддеревья с количеством вершин, в два раза меньшим, чем значение `t` (для правого поддерева уменьшаем количество вершин на единицу – вычитается создаваемая вершина):

```
Function Tree(t:Integer):pt;
  Var x:Integer;
      w:pt;
  Begin
    If t=0 Then Tree:=nil
    {Нулевое количество вершин определяет ссылку
    типа nil}
    Else Begin
      Read(x);
      {Читаем значение очередного ключа}
      New(w);
      With w^ Do Begin
        data:=x;
        left:=Tree(t Div 2);
        {Формируем значения левой и правой
        ссылок вершины}
        right:=Tree(t-t Div 2-1);
      End;
      Tree:=w;
    End;
  End;
```

Итак, у нас есть двоичное дерево, для которого выполнено второе свойство (форма), и его требуется преобразовать в двоичную кучу, т. е. надо добиться, чтобы для вершин дерева выполнялось первое свойство — отношение порядка (определяется по ключам, соответствующим каждой вершине). При представлении кучи с помощью массива задача была нами решена. А как быть в этом случае? Решает ли задачу обход дерева с проверкой для каждой родительской вершины выполнения условия, что значение ключа родительской вершины не меньше, чем значения ключей ее детей. Другими словами, работает ли следующая логика (она является простой модификацией обхода дерева `Print_Tree` из раздела. 5.2)?

```
Procedure Change(t:pt);
  Begin
    If t<>nil Then Begin
      Change(t^.left);
      {Идем в левое поддерево}
      Change(t^.right);
      {Затем идем в правое поддерево}
```

```
    If (t^.left<>nil) And (t^.data<t^.left^.data)
      Then Swap(t^.data,t^.left^.data);
      {Записываем максимальное значение в корень
      поддерева}
    If (t^.right<>nil) And (t^.data<t^.right^.data)
      Then Swap(t^.data,t^.right^.data);
  End;
End;
```

Здесь процедура `Swap` выполняет обмен значений ключей у вершины-родителя и его потомка. На рис. 7.4 приведен пример дерева и показано его изменение в процессе работы процедуры `Change`.

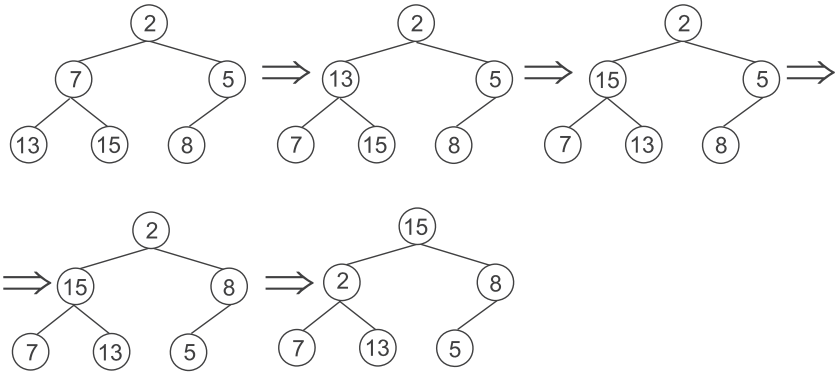


Рис. 7.4. Обход двоичного дерева с изменением значения ключей по принципу кучи

Результат очевиден — однократное выполнение процедуры `Change` не решает задачу. Но если она будет выполнена h раз (высота дерева $h = \lfloor \log_2 n \rfloor$), т. е. `For i:=1 To h Do Change(root);`, то куча будет сформирована. Первое выполнение процедуры при этом гарантирует нам выполнение условий кучи для корня (нулевого уровня), второе — для вершин первого уровня и т. д., а для листьев требования выполняются автоматически. Временна́я сложность такого преобразования — $O(n \cdot log_2 n)$. На рис. 7.5 приведен пример преобразования двоичного дерева ($h = 3$) в кучу с помощью описанной логики. Второе дерево — это результат работы `Change` на первой итерации ($h = 1$), третье дерево получается после второй итерации ($h = 2$), и, наконец, четвертое дерево — после третьей итерации ($h = 3$).

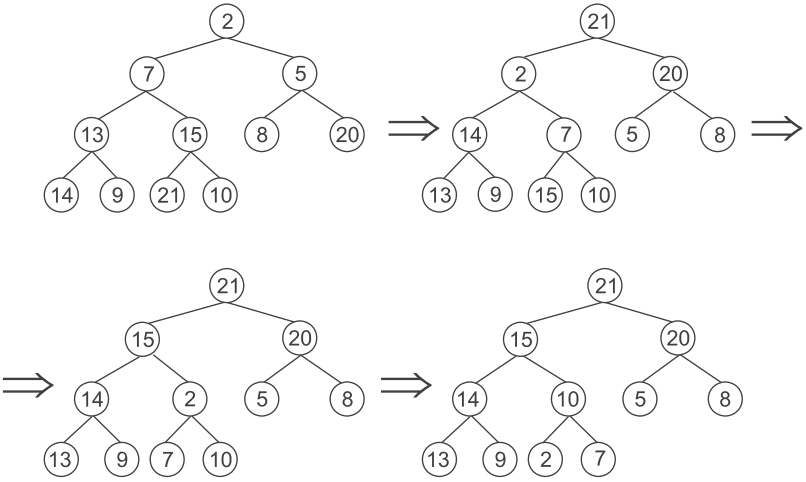


Рис. 7.5. Пример формирования кучи

В чем состоит основное различие между построением кучи в случае ее представления в виде массива и ситуацией, когда она представлена двоичным деревом? В первом случае n `Div` 2 раз элементы «проталкивается» по частично построенной куче, а во втором — h раз обходится все дерево. И, хотя оба указанных варианта реализации имеют одну и ту же временну́ю сложность, уже из примера на рис. 7.5 просматривается некоторая избыточность второго варианта — в нем явно совершаются лишние обмены.

Вернемся к задаче сортировки в предположении о том, что данные представлены в виде кучи, для реализации которой использовано двоичное дерево. Идея проста: в корне дерева у нас находится максимальный элемент. Выбираем этот элемент, записываем в корень минимально допустимое значение и «проталкиваем» его по куче (`Push`). После «проталкивания» в корень записывается следующий максимальный элемент; с ним мы выполняем аналогичные действия, и так n раз. Время «проталкивания» пропорционально $O(\log_2 n)$, общая временна́я сложность составляет $O(n \cdot \log_2 n)$. Логика «проталкивания» при этом заключается в следующем: мы выбираем у текущей вершины (если она не лист) потомка с максимальным значением ключа, сравниваем с выбранным значением ключ родителя; если ключ родительской вершины меньше, то обмениваем значе-

ния ключей и продолжаем процесс с соответствующим потомком. Тогда запись процедуры Push может иметь вид:

```
Procedure Push(t:pt);
  Var v,w,q:pt;
  {Вспомогательные переменные v и w введены для
  наглядности (обозримости) текста процедуры}
  Begin
    If t<>nil Then Begin
      v:=t^.left;
      {Запоминаем адреса корневых элементов левого
      и правого поддеревьев}
      w:=t^.right;
      If (v<>nil) And (w<>nil) Then
      {Если оба поддерева есть, то выбираем
      поддерево с наибольшим значением ключа
      в корне}
        If v^.data>w^.data Then q:=v
          Else q:=w
          {Обработка случаев, когда вершина имеет
          одно поддерево или является листом}
        Else If v<>nil Then q:=v
          Else If w<>nil Then q:=w
            Else Exit;
        If t^.data<q^.data Then Begin
          Swap(t^.data,q^.data);
        Push(q);
      End;
    End;
  End;
```

Математическое отступление[1]. Оценим время работы процедуры BuildHeap. Время работы процедуры Push определяется высотой вершины, для которой она вызвана. Так как количество вершин высоты h в куче из n элементов не превышает $\left\lceil \frac{n}{2^{h+1}} \right\rceil$, а высота кучи — не больше $\lfloor \log_2 n \rfloor$, то время работы BuildHeap оценивается как $\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) =$

[1] *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 143.

$= O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right)$. Из курса математического анализа известно, что $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ (дифференцируем сумму бесконечной убывающей геометрической прогрессии при $|x| < 1$). Подставляя $x = 1/2$ в формулу, получим: $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$. Тогда $O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$, следовательно, временна́я оценка работы процедуры `BuildHeap` равна $O(n)$.

Упражнения

1. На рис. 7.1*б* показан массив A, элементы которого образуют двоичную кучу. Для построения кучи использован рассмотренный выше алгоритм `BuildHeap`. Найдите начальное состояние массива A.

2. Двоичная куча строилась по принципу: $A[i] \geqslant A[2 \cdot i]$ и $A[i] \geqslant A[2 \cdot i+1]$. Измените процедуру `Shift` так, чтобы элементы кучи удовлетворяли условию $A[i] \leqslant A[2 \cdot i]$ и $A[i] \leqslant A[2 \cdot i+1]$.

3. Является ли двоичной кучей массив {35, 18, 13, 9, 8, 10, 12, 11, 7, 6, 4}?

4. Двоичная куча имеет высоту h. Определите максимальное и минимальное количество элементов в этой куче.

5. Докажите, что двоичная куча из n элементов имеет высоту $\lfloor \log_2 n \rfloor$.

6. Выполните трассировку работы процедуры `BuildHeap` для массива {3, 1, 2, 9, 8, 10, 12, 11, 7, 6, 4}.

7. Будет ли процедура `BuildHeap` выполнять свои функции, если изменить параметры цикла на `For i:=1 To n Div 2 Do Push(i,n)` (или, другими словами, вставлять элементы в кучу, начиная с первого)?

8. Докажите, что в куче из n элементов содержится не более $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ вершин высоты h.

9. Выполните трассировку работы функции `Tree` для ключей 3, 1, 2, 9, 8, 10, 12, 11, 7, 6, 4. Преобразуйте полученное правильное двоичное дерево в кучу, используя h раз процедуру `Change`.

10. Является ли двоичное дерево, показанное на рис. 7.6, кучей? Обоснуйте свой ответ.

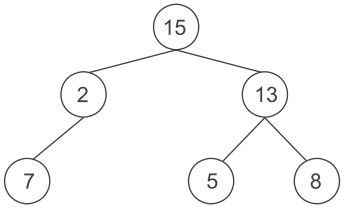


Рис. 7.6. Пример двоичного дерева

7.2. Очередь с приоритетом на базе двоичной кучи

> Сила, мало-помалу сливающаяся с небом, — вот что такое дерево... Таков и ты, человек. Осуществление — вот что такое ты.
>
> *Антуан де Сент-Экзюпери*

Понятие «очередь с приоритетом» предполагает существование некоего отношения порядка на множестве элементов этой абстрактной структуры данных. В обычной очереди предполагается, что входные данные обслуживаются по принципу: «первый пришел — первый получил обслуживание», т. е. отношение порядка задается временем прихода. Слово же «приоритет» говорит о том, что обслуживание осуществляется не в соответствии с временем поступления, а согласно заданному приоритету (значению ключа) для каждого элемента данных, который поступает на обработку (в очередь).

Очередью с приоритетами называют множество элементов, на котором задана функция приоритета, т. е. для каждого элемента a имеющегося множества вычисляется его приоритет $f(a)$ (обычно — действительное значение) и выполняются следующие операции:

- поиск элемента с наивысшим (наибольшим) приоритетом (`Search`);
- удаление (извлечение) элемента с наибольшим приоритетом (`Extract`);
- добавление (включение) нового элемента в множество (`Insert`).

Значение приоритета определяется как ключ. Сопутствующую информацию обычно не рассматривают (в реальных задачах она перемещается в структуре данных вместе с ключом).

Для реализации очереди с приоритетом можно использовать линейный список, в котором элементы упорядочены по невозрастанию. В этом случае операция `Search` сводится к чтению первого элемента списка (с временем работы $O(1)$), операция `Extract` — к удалению первого элемента списка (время работы — $O(1)$), а включение нового элемента в очередь (`Insert`) требует последовательного просмотра элементов списка и времени $O(n)$.

Наиболее приемлемой с точки зрения времени выполнения всех операций структурой данных для реализации очереди с приоритетом является двоичная куча.

Пусть для представления кучи используется массив. Операция `Search` выполняется путем чтения первого элемента кучи: он имеет наибольший приоритет или значение ключа (время выполнения — $O(1)$). При выполнении операции `Extract` первый элемент кучи удаляется (извлекается), на его место записывается последний элемент кучи, размер кучи уменьшается на единицу, а новый первый элемент проталкивается по куче с помощью операции `Push` (см. раздел 7.1). Время выполнения этой операции — $O(\log_2 n)$, оно определяется временем выполнения процедуры `Push`.

```
Function Extract:Integer;
  Begin
    Extract:=A[1];
    A[1]:=A[n];
```

```
    n:=n-1;
    Push(1,n);
  End;
```

Осталось разобрать логику выполнения операции `Insert`. Поясним ее на примере. Пусть у нас есть куча (массив A), начальное состояние которой приведено в первой строке табл. 7.2. Требуется вставить в нее элемент с ключом 15. Размер кучи при этом увеличивается на один элемент — это место отмечено в табл. 7.2 символом «$». А затем значение нового ключа сравнивается с ключом родителя ($A[6]$). Если он больше, то ключ родителя переписывается на место, отмеченное $, а само это место «перемещается вверх по куче» в позицию родителя. Эти действия продолжаются до тех пор, пока не будет найдено правильное положение элемента в куче, т. е. свойство последней должно выполняться. В рассматриваемом примере элемент вставляется на первое место.

Таблица 7.2

Номер шага	A											
	1	2	3	4	5	6	7	8	9	10	11	12
Начальное состояние	13	9	12	7	6	2	8	4	3	1	5	$
После 1-го шага	13	9	12	7	6	$	8	4	3	1	5	2
После 2-го шага	13	9	$	7	6	12	8	4	3	1	5	2
После 3-го шага	$	9	13	7	6	12	8	4	3	1	5	2
Результат	15	9	13	7	6	12	8	4	3	1	5	2

Графическое изображение этого процесса приведено на рис. 7.7, где символ $ изображен пустым кружком.

Формализованная запись рассмотренных действий имеет вид:

```
Procedure Insert(x:Integer);
  Var i:Integer;
  Begin
    n:=n+1;
    i:=n;
    While (i>1) And (A[i Div 2]<x) Do Begin
      A[i]:=A[i Div 2];
```

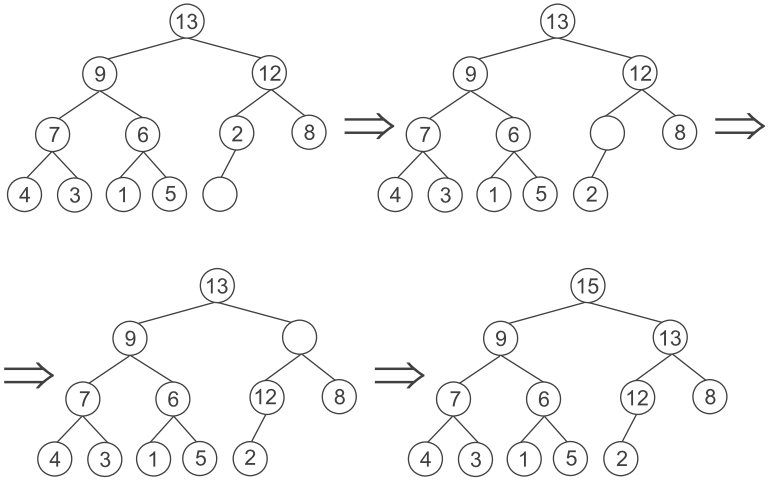


Рис. 7.7. Вставка элемента с ключом 15 в кучу

```
    i:=i Div 2;
  End;
  A[i]:=x;
End;
```

Очевидно, что временна́я сложность процедуры `Insert` равна $O(\log_2 n)$. Таким образом, при реализации очереди с приоритетом с помощью двоичной кучи время выполнения всех операций имеет временну́ю оценку $O(\log_2 n)$.

Упражнения

1. Изобразите графически очередь с приоритетом, полученную в результате последовательной вставки в пустую очередь следующих элементов: 5, 6, 4, 9, 3, 13, 1, 7, 15, 8, 12.

2. Изобразите графически работу процедуры `Extract` для очереди с приоритетом, полученной в предыдущем упражнении.

3. Реализуйте операцию увеличения элемента $A[i]$ двоичной кучи на значение k с восстановлением основного свойства кучи. Время работы процедуры — $O(\log_2 n)$.

4. Реализуйте операцию удаления элемента $A[i]$ из двоичной кучи. Время работы — $O(\log_2 n)$.

5. Пусть для представления кучи используется двоичное дерево. Можно ли (и если да, то как) в этом случае реализовать операции `Search` и `Extract` для работы с приоритетной очередью?

6. Пусть для представления кучи используется двоичное дерево. Как реализовать процедуру вставки элемента в дерево (`Insert`) так, чтобы ее последовательное применение для ключей, например, равных 3, 1, 2, 9, 8, 10, 12, 11, 7, 6, 4, приводило к образованию двоичной кучи?
   Примечание. Сложность этой задачи заключается в том, чтобы выполнить второе свойство кучи. При использовании массива для представления кучи место вставки известно (оно определяется требованием соблюдать форму), и остается только пройти снизу вверх по куче для восстановления первого свойства.

7.3. Биномиальная куча

$$(x+y)^n = C_n^n x^n y^0 + C_n^{n-1} x^{n-1} y^1 + \ldots + C_n^1 x^1 y^{n-1} + C_n^0 x^0 y^n$$

Исаак Ньютон[1)]

Биномиальная куча (H) — это очередь с приоритетом, в которой, кроме эффективного выполнения операций `Insert(H,x)` (вставка элемента в очередь) и `Extract_Min(H)` (извлечение минимального элемента из очереди), эффективно выполняется операция `Merge(H1,H2,H3)` (слияние двух куч в одну).

Биномиальная куча строится из *биномиальных деревьев* (B). На рис. 7.8 показаны биномиальные деревья B_0, B_1, B_2, B_3, B_4 и B_5.

---

1) Говорят, что сэр Исаак Ньютон в свободное от научных размышлений время любил что-нибудь мастерить по дому. Однажды он выпилил во входной двери отверстие для кошки, чтобы она могла свободно выходить во двор, когда ей вздумается. А когда кошка родила шестерых котят, сэр Ньютон выпилил в двери еще шесть маленьких отверстий...

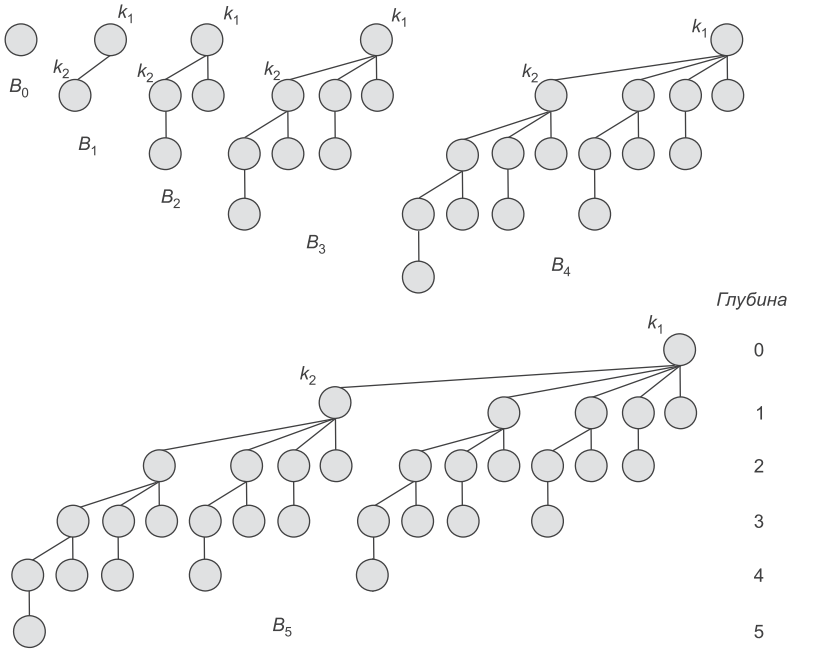


Рис. 7.8. Биномиальные деревья B_0, B_1, B_2, B_3, B_4 и B_5

Дерево B_0 — это одна вершина. Каждое дерево B_k строится из двух деревьев $B_{k\text{-}1}$ по принципу: корень одного дерева (k_2) становится самым левым сыном корня (k_1) другого дерева (рис. 7.8).

Свойства биномиального дерева

Дерево B_k имеет: 2^k вершин; высоту k; C_k^i (число сочетания из k по i, или *биномиальный коэффициент*) вершин с глубиной i; корень со степенью k. У дерева B_k сыновья корня есть поддеревья B_{k-1}, B_{k-2}, ..., B_1, B_0.

Биномиальная куча (H) — это некая совокупность биномиальных деревьев, в вершинах которых записаны ключи, причем в корне каждого дерева записан наименьший ключ, а ключ любой вершины-родителя всегда меньше ключей сыновей. Основополагающим для биномиальной кучи является условие, что в ней нет двух одинаковых биномиальных деревьев (с одной степенью корня).

Пусть в куче n элементов. Представим n в двоичной системе счисления (такое представление однозначно):

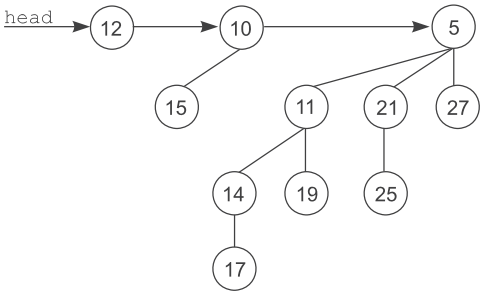


Рис. 7.9. Пример биномиальной кучи

$n = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_k}$, где все i_1, i_2, ..., i_k различны. Тогда в кучу будут входить деревья $B_{i_1}, B_{i_2}, \ldots, B_{i_k}$, каждое из которых содержит соответствующее количество элементов, составляющих кучу.

Пример (см. рис. 7.9). Пусть $n = 11$ — куча с 11 вершинами. В двоичной системе счисления $11_{10} = 1011_2$, а значит, H состоит из B_0, B_1 и B_3.

В разделе 5.3 для представления деревьев использовался принцип «левый сын и правый брат», который разумно использовать (из-за необходимости выполнения операции `Merge`) и при работе с биномиальными деревьями. Кроме того, корни биномиальных деревьев связаны в корневой список в порядке возрастания степеней. Структура элемента кучи при этом показана на рис. 7.10, а ее формализованная запись имеет вид:

```
Type pt=^el;
     el=Record
       parent:pt;
       key:Integer;
       degree:Integer;
       child,sibl:pt;
     End;
Var head:pt; {Указатель на первый корень кучи}
```

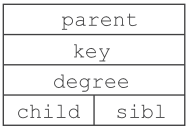


Рис. 7.10. Структура элемента кучи

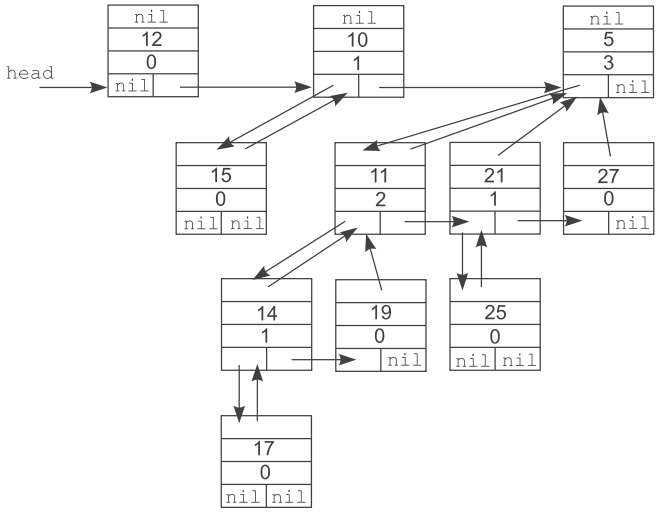


Рис. 7.11. Представление кучи в памяти компьютера

Здесь `parent` — указатель на вершину родителя, `key` — значение ключа, `degree` — степень вершины, `child` — указатель на самого левого сына вершины, `sibl` — указатель на правого брата вершины или, в случае корневого списка, — на следующий элемент списка. Для примера, показанного на рис. 7.9, его описание в памяти представлено на рис. 7.11.

Рассмотрим операции с биномиальной кучей и начнем с процедуры `Merge(head1,head2,head)`. Для ее реализации потребуются две вспомогательные процедуры. Первая из них — слияние двух биномиальных деревьев одного размера B_{k-1} в дерево B_k. Пример этой операции для деревьев второго порядка проиллюстрирован на рис. 7.12 (цифрами обозначены соответствующие действия в процедуре `Link`), а формализованная запись процедуры имеет вид:

```
Procedure Link(x,y:pt);
{x, y - указатели на корни деревьев}
  Begin
    x^.parent:=y;       {1}
    x^.sibl:=y^.child; {2}
    y^.child:=x;        {3}
    y^.degree:=y^.degree+1;
  End;
```

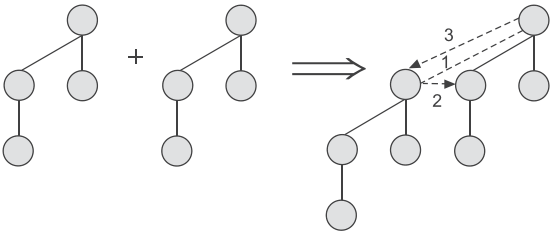


Рис. 7.12. Слияние двух биномиальных деревьев

Вторая вспомогательная процедура (назовем ее, например, `Coufl`) соединяет два корневых списка куч H_1 и H_2, отсортированных по значению `degree`, в один список, отсортированный в порядке неубывания степеней вершин. Пример слияния двух списков приведен на рис. 7.13, где в кружках указаны степени корневых вершин, а пунктирными линиями показаны связи в результирующем списке.

```
Procedure Coufl(x,y:pt; Var z:pt);
  Begin
    If (x<>nil) And (y<>nil) Then Begin
      If (x^.degree<y^.degree) Then Begin
        z:=x;
        Coufl(x^.sibl,y,z^.sibl);
      End
      Else Begin
        z:=y;
        Coufl(x,y^.sibl,z^.sibl);
      End;
    End
    Else
      If (x=nil) Then z:=y Else z:=x;
End;
```

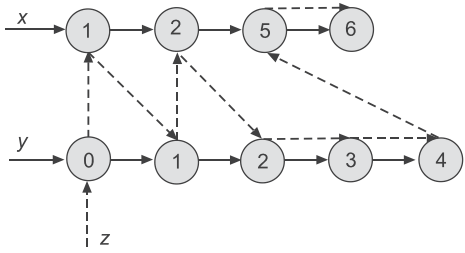


Рис. 7.13. Пример слияния двух списков

```
 7 6 5 4 3 2 1 0
 1 1 0 0 1 1 0
+     1 1 1 1 1
 1 0 0 0 0 1 0 1
```

Рис. 7.14. Сложение двоичных чисел, соответствующих биномиальным кучам на рис. 7.13. В первой строке указаны степени двойки в разложении чисел (порядки биномиальных деревьев)

Первая биномиальная куча на рис. 7.13 состоит из биномиальных деревьев B_1, B_2, B_5 и B_6, а вторая — из B_0, B_1, B_2, B_3 и B_4. В результате сложения соответствующих двоичных чисел (см. рис. 7.14) мы получаем число 10000101. Другими словами, в результате слияния куч (операция `Merge`) мы должны получить кучу, состоящую из биномиальных деревьев B_0, B_2, и B_7 (рис. 7.15; в кружках по-прежнему указаны степени корней биномиальных деревьев).

Рис. 7.15. Результат слияния биномиальных куч, представленных на рис. 7.13

Как получается этот результат? На рис. 7.16 вначале приведен корневой список для примера, показанного на рис. 7.13. Нижний индекс здесь указывает на кучу, из которой взято соответствующее биномиальное дерево. Мы видим, что имеется два биномиальных дерева B_1, а затем идет биномиальное дерево B_2. Деревья B_1 сливаются в одно; в результате в куче оказывается три дерева B_2. Очевидно, что в куче не может оказаться более трех деревьев одного порядка, так как исходные кучи содержали не более одного дерева, а в результате слияния добавляется еще одно. Последние два дерева B_2 (именно последние!) преобразуются в B_3. Процесс преобразования кучи продолжается (см. рис. 7.16) до тех пор, пока в ней все биномиальные деревья не будут различными. (Сравните это преобразование кучи со сложением соответствующих двоичных чисел на рис. 7.14.)

Уточним некоторые детали выполненного преобразования. Введем указатель `x` на текущую вершину корневого списка. При этом `prev` будет храниться адрес предыдущего элемента списка, а в `next` — следующего. Когда биномиаль-

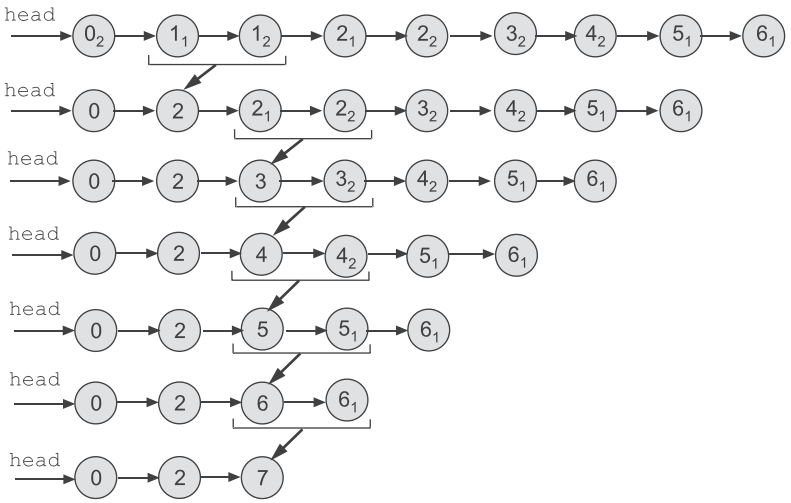


Рис. 7.16. Последовательное слияние двух биномиальных куч

ные деревья, соответствующие `x` и `next`, — разного порядка, осуществляется простая переадресация по списку (вариант 1 на рис. 7.17). Аналогичный переход осуществляется и если `x`, `next` и `next^.sibl` указывают на корневые вершины одинаковых биномиальных деревьев (вариант 2 на рис. 7.17).

Таким образом, слияние биномиальных деревьев выполняется (что явно просматривается на рис. 7.16), только когда указателям `x` и `next` соответствуют биномиальные деревья одного порядка, а `next^.sibl` — более высокого порядка.

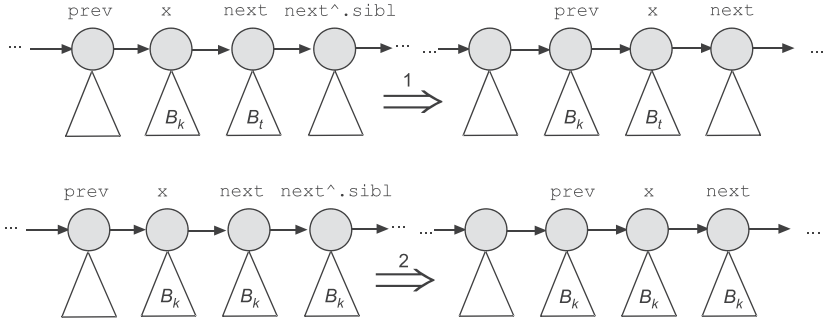


Рис. 7.17. Варианты простого перехода по корневому списку

Далее, помня, что корневой список кучи должен содержать вершины, упорядоченные по неубыванию по параметру *degree*, следует проанализировать `x^.key` и `next^.key`. В зависимости от результата сравнения здесь также возможны два варианта — рис. 7.18.

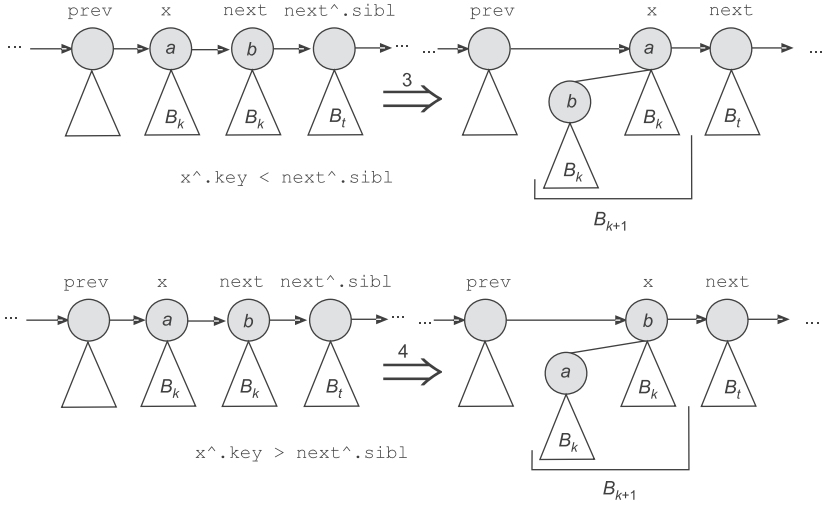


Рис. 7.18. Варианты слияния биномиальных деревьев

Формализованная запись процедуры `Merge` может иметь следующий вид:

```
Procedure Merge(head1,head2:pt;Var head:pt);
  Var prev, x, next:pt;
  Begin
    Coufl(head1,head2,head);
    {Сливаем корневые списки}
    prev:=nil;
    x:=head;
    next:=x^.sibl;
    While next<>nil Do Begin
      If (x^.degree<>next^.degree) Or
        ((next^.sibl<>nil) And
        (next^.sibl^.degree=x^.degree))
        Then Begin
        {Варианты показаны на рис. 7.17}
          prev:=x;
```

```
        x:=next;
      End
      Else
        If x^.key<=next^.key Then Begin
        {Рис. 7.18 - вариант 3}
          x^.sibl:=next^.sibl;
          Link(next,x);
        End
        Else Begin {Рис. 7.18 - вариант 4}
          If prev=nil Then head:=next
            Else prev^.sibl:=next;
          Link(x,next);
          x:=next;
        End;
      next:=x^.sibl;
    End;
  End;
```

Оценим время выполнения этой операции. Процедура `Link` требует времени $O(1)$. В куче из n элементов количество биномиальных деревьев не превышает $\lfloor \log_2 n \rfloor + 1$, поскольку n является суммой количества элементов в биномиальных деревьях, а эти количества есть степени двойки. Или, иначе, для представления числа n необходимо $\lfloor \log_2 n \rfloor + 1$ двоичных разрядов. Таким образом, и процедура `Coufl`, и процедура `Merge` выполняются за время, пропорциональное $O(\log_2 n)$.

После проделанной нами работы процедура добавления элемента в кучу (`Insert`) достаточно очевидна и не требует комментариев:

```
Procedure Insert(Var head:pt;x:Integer)
  Var h:pt;
  Begin
    New(h);
    With h^ Do Begin
      parent:=nil;
      child:=nil;
      sibl:=nil;
      degree:=0;
      key:=x;
    End;
```

```
    Merge(h,head,head);
    {Сливаем кучу из одного элемента и старую кучу}
  End;
```

С операцией же `Extract_min` дела обстоят чуть сложнее (см. рис. 7.19). Так как минимальный элемент в каждом биномиальном дереве кучи находится в корне, то необходимо просмотреть корневой список. Пусть дана куча из B_2, B_4, B_5 и B_7, а минимальный элемент находится в B_5 (рис. 7.19; система обозначений та же, что и на предыдущих рисунках). В этом случае куча распадается на две. В первой будут содержаться биномиальные деревья, которые были в исходной куче до дерева с минимальным элементом, а во второй — поддеревья с корнями, являющимися сыновьями удаленного корня дерева, и элементы кучи, которые следовали в исходной куче за удаленным деревом (можно сделать и иначе — этот «остаток» поместить в первую кучу). На рис. 7.19 для примера приведена структура первой и второй куч.

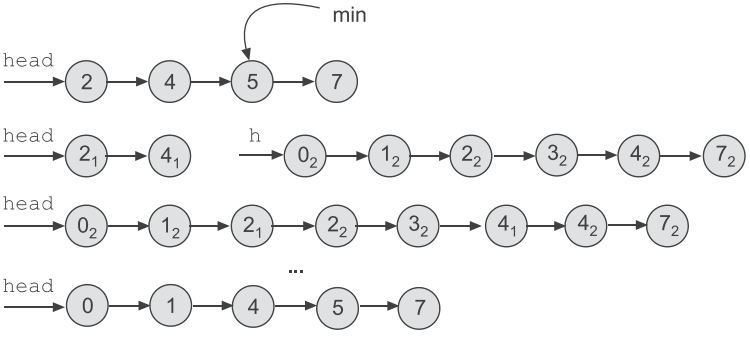


Рис. 7.19. Иллюстрация операции `Extract_Min`

Дальнейшее очевидно — выполняем операцию `Merge` для полученных куч. Формализованная запись логики имеет вид:

```
Function Extract_Min(Var head:pt):Integer;
  Var x:Integer;
      h,prev,p,next:pt;
```

```
Begin
  x:=maxint;
  {Имитация "бесконечно большого" значения}
  If (head<>nil) Then Begin
  {Ищем элемент с минимальным ключом}
    p:=head;
    prev:=nil;
    While (p<>nil) Do Begin
      If (p^.key<x) Then Begin
        x:=p^.key;
        h:=prev;
      End;
      prev:=p;
      p:=p^.sibl;
    End;
    {Разъединяем кучу на две части}
    If (h=nil) Then Begin
      p:=head;
      head:=nil;
    End
    Else Begin
      p:=h^.sibl;
      h^.sibl:=nil;
    End;
    h:=p^.sibl;
    next:=p^.child;
    Dispose(p);
    {Удаление минимального элемента}
    {Преобразование остатка дерева в начало кучи,
    при этом изменяется порядок элементов}
    While (next<>nil) Do Begin
      p:=next^.sibl;
      next^.sibl:=h;
      h:=next;
      next:=p;
    End;
    Merge(h,head,head);
  End;
  Extract_min:=x;
End;
```

Упражнения

1. Изобразите графически биномиальные кучи, которые получаются при последовательном добавлении к куче, показанной на рис. 7.9, ключей 13, 16, 18 и 20.

2. Нарисуйте биномиальную кучу из 15 элементов. Последовательно добавьте к ней два элемента. Изобразите графически процесс изменения кучи.

3. Уменьшение конкретного ключа элемента биномиальной кучи требует знания указателя на соответствующую вершину и может быть реализовано так:

```
Procedure Decrease(head, w: pt; x:Integer);
{w – значение указателя на элемент кучи,
x – новое значение ключа; x < w^.key}
  Var y,z:pt;
      t:Integer;
  Begin
    w^.key:=x;
    y;=w;
    z:=y^.parent
    While (z<>nil) And (y^.key<z^.key) Do Begin
      t:=y^.key;
      {Обмен значениями ключей}
      y^.key:=z^.key;
      z^.key:=t;
      y:=z;
      {Переадресация вверх по дереву}
      z:=y^.parent;
    End;
  End;
```

   В биномиальной куче (см. рис. 7.9) с помощью приведенной выше логики ключ 17 изменили на 3. Изобразите получившуюся кучу.

4. Удаление конкретного элемента из кучи осуществляется путем замены его ключа на значение, заведомо меньшее (например, на $-\infty$) всех ключей (см. упражнение 3), а затем — извлечения минимального элемента из кучи. В результате первого действия минимальный элемент «всплывает» вверх по биномиальному дереву, а второе

действие — это не что иное, как выполнение процедуры `Extract_min`. Реализуйте процедуру удаления элемента из кучи (`Delete`).

5. В биномиальной куче нельзя эффективно находить элемент с заданным ключом (процедура `Search`). Разработайте процедуру `Search` и оцените время ее работы, чтобы убедиться в правильности этого утверждения.

6. Процедура вставки элемента в кучу (`Insert`) использует процедуру `Merge`. Можно ли, используя аналогию со сложением двоичных чисел, реализовать `Insert` без вызова `Merge`?

Методические комментарии

Алгоритм пирамидальной сортировки разработан Дж. У. Дж. Уильямсоном (1964 г.) на основе описания двоичной кучи, данного Р. У. Флойдом (1964 г.). Наиболее детальное и лаконичное описание (из числа известных автору) дано в книге Д. Бентли[1)], а также в книге Т. Кормена, Ч. Лейзерсона и Р. Ривеста[2)].

Достаточно полное описание очереди с приоритетом можно найти в книге А. Ахо, Дж. Хопкрофта и Дж. Ульмана[3)], правда, в несколько другой терминологии — для реализации очереди с приоритетом используется понятие «частично упорядоченного дерева» вместо «двоичного дерева».

Биномиальная куча как структура данных предложена Д. Виллемином в 1978 г. Лучшее и, пожалуй, единственное ее описание на русском языке дано в книге Т. Кормена, Ч. Лейзерсона и Р. Ривеста[4)]. Другие авторы ограничиваются или упоминанием, или сжатыми определениями основных положений.

---

1) *Бентли Дж.* Жемчужины программирования. СПб.: Питер, 2002. С. 177–190.

2) *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 138–150.

3) *Ахо А., Хопкрофт Дж., Ульман Дж.* Структуры данных и алгоритмы. М.: Издательский дом «Вильямс», 2001. С. 129–137.

4) *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 376–394.

Глава 8

Сбалансированные деревья

> Всякое движение — это видимое нами стремление к недостающему равновесию. Все живое движется в поисках его, в поисках утраченной гармонии, в стремлении к совершенству, когда покой не есть отсутствие движения, но равнодействующая всех движений.
>
> *Делия Стейнберг Гусман*

8.1. АВЛ-деревья

> На свете нет кружева тоньше,— негромко сказал отец. И показал рукой вверх, где листва деревьев вплеталась в небо — или, может быть, небо вплеталось в листву?
>
> *Рей Брэдбери*

Основные понятия

Двоичное дерево считается *идеально сбалансированным*, если для каждой его вершины количества вершин в левом и правом поддеревьях различаются не более чем на 1.

Для одних и тех же данных, например для целых чисел от 1 до n (в зависимости от порядка их поступления на обработку), структура двоичного дерева поиска будет различной. Возможны варианты как идеально сбалансированного дерева, так и простого линейного списка, когда или все левые, или все правые ссылки вершин равны `nil`. В этом случае время вставки и удаления имеет оценку $O(n)$, а в случае идеальной сбалансированности — $O(\log_2 n)$.

Реализация операции восстановления идеальной сбалансированности при случайной вставке или удалении элемента из двоичного дерева поиска, разумеется, возможна, однако она достаточно сложна. Поэтому в информатике в свое время были предприняты попытки (и они продолжаются по сей день) создания и использования структур, для которых операции по восстановлению балансировки двоичного дерева были бы простыми и выполнялись достаточно быстро.

Изменим немного определение сбалансированности и дадим ему следующую формулировку: *дерево является сбалансированным тогда и только тогда, когда для каждой вершины высота ее двух поддеревьев различается не более чем на единицу*. Деревья, удовлетворяющие этому условию, называют *АВЛ-деревьями*. Эта структура данных была предложена Г. М. Адельсоном-Вельским и Е. М. Ландисом в 1962 г. (отсюда и название — «АВЛ-дерево»), и эти их исследования, вероятно, были первыми, посвященными данной проблематике.

Предположим, что у каждой вершины двоичного дерева поиска есть поле `bal` (баланс вершины), значение которого определяет разность высот ее поддеревьев (из высоты правого поддерева вычитается высота левого поддерева). Очевидно, что значение `bal` для сбалансированного дерева лежит в диапазоне от –1 до 1. На рис. 8.1 приведены примеры деревьев, где дерево, показанное на рис. 8.1*а*, не является АВЛ-деревом, а на рис. 8.1*б* — является им. Заметим, что оба этих дерева не являются идеально сбалансированными.

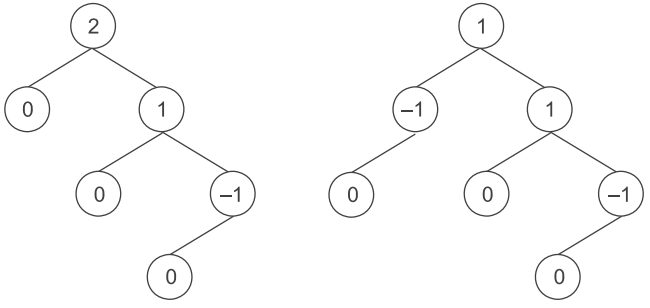


Рис. 8.1. Деревья: *а*) не АВЛ-дерево; *б*) АВЛ-дерево. В вершинах указаны разности высот поддеревьев

Работа с АВЛ-деревом (основная часть логики по вставке и удалению элементов) практически совпадает с операциями для двоичного дерева поиска, за исключением восстановления балансировки дерева, поэтому подробно мы рассмотрим здесь именно последнюю. Но прежде для большей конкретности изложения давайте введем описание дерева и опишем вспомогательные процедуры.

Описание вершины дерева, по аналогии с двоичным деревом поиска (см. раздел 5.2), имеет вид:

```
Type pt=^node;
         node = Record
           height:Word;
           data:Integer;
           left,right:pt ;
         End;
Var first:pt;
```

Как видим, здесь в описание вершины введено новое поле — высота дерева (`height`). Его назначение мы раскроем позднее, а сейчас лишь отметим, что именно введение этого поля (а не поля `bal`) позволяет упростить логику как изложения материала, так и программного кода.

Вычисление высоты дерева реализуется процедурой `MakeNewHeight`, которой при вызове необходимо передать значение указателя на вершину, являющуюся корнем дерева (поддерева). В этой процедуре берется значение высоты левого поддерева вершины, затем значение высоты правого поддерева, определяется максимальное из этих двух чисел, а затем к нему прибавляется единица.

```
Procedure MakeNewHeight(q:pt);
{Вычисление высоты дерева (поддерева)}
  Var a, b:Word;
  Begin
    If (q^.left<>nil) Then a:=q^.left^.height
      Else a:=0;
    If (q^.right<>nil) Then b:=q^.right^.height
      Else b:=0;
    If a>b Then q^.height:=a+1
      Else q^.height:=b+1;
  End;
```

Тогда для вычисления баланса вершины достаточно найти разность значений высот ее правого и левого потомков:

```
Function GetBallance(q:pt):Integer;
{Вычисление разности высот левого и правого потомков
вершины}
  Var a,b:Word;
```

```
Begin
  If (q^.left<>nil) Then a:=q^.left^.height
    Else a:=0;
  If (q^.right<>nil) Then b:=q^.right^.height
    Else b:=0;
  GetBallance:=b-a;
End;
```

Рассмотрим теперь операции восстановления баланса. Пусть у нас есть фрагмент дерева, показанный на рис. 8.2*а*, после вставки в него элемента с ключом 60. Баланс вершины (ее адрес — значение указателя q) становится равным −2, и необходимо восстановить балансировку. Мы как бы берем дерево за его правый конец и тянем его вниз, причем гвоздик вбит под вершиной с ключом 100 — этот фрагмент дерева поворачивается, и такой поворот мы назовем *малым левым поворотом*. Сложность здесь заключается в том, что у вершины с ключом 80 появится три потомка (на рис. 8.2 это не показано), и необходимо правого потомка вершины с ключом 80 сделать левым потомком вершины с ключом 100. Естественно, что после корректировки адресов связи (значений указателей в вершинах деревьев) необходимо заново подсчитать значения height для вершин, участвующих в операции.

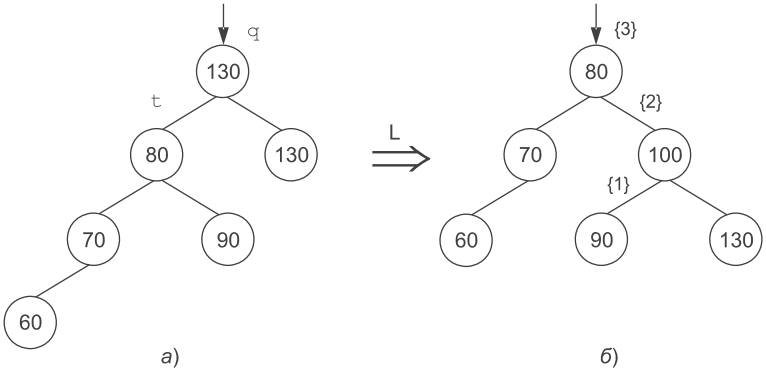


Рис. 8.2. Малый левый поворот

Процедура малого левого поворота на входе должна получить адрес вершины q, относительно которой нарушено условие баланса. Тогда мы запоминаем значение левой

ссылки вершины (t), а на ее место записываем (первое действие) адрес правого поддерева вершины, переходящей в корень поддерева. На освобожденное место записываем (второе действие) значение q, пересчитываем высоты вершин с адресами q и t и в качестве выходного значения передаем вызывающей программе информацию, что новым корнем поддерева является вершина, которая ранее имела адрес t (третье действие).

```
Procedure TurnL(Var q:pt);
{Малый левый поворот: осуществляется для левого
поддерева вершины q}
  Var t:pt;
  Begin
    t:=q^.left;
    {Запоминаем значение левой ссылки}
    q^.left:=t^.right; {1}
    t^.right:=q;       {2}
    MakeNewHeight(q);
    MaketNewHeight(t);
    q:=t;              {3}
  End;
```

Аналогично осуществляется и поворот в другую сторону — *малый правый поворот* (рис. 8.3).

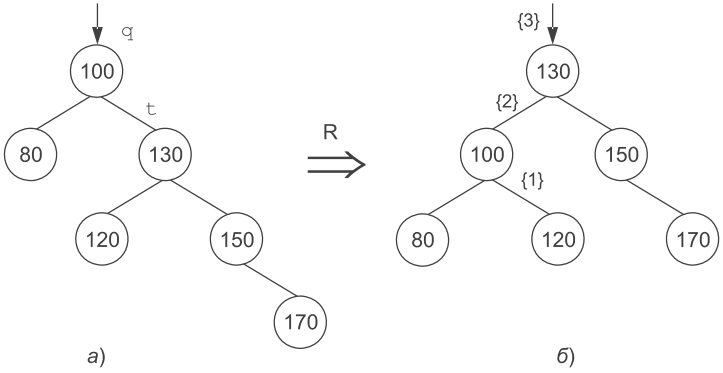


Рис. 8.3. Малый правый поворот

Процедура, реализующая это действие, повторяет процедуру TurnL с точностью до замены ссылок left на ссылки right и наоборот, что вполне естественно.

```
Procedure TurnR(Var q:pt);
{Малый правый поворот}
  Var t:pt;
  Begin
    t:=q^.right;
    q^.right:=t^.left; {1}
    t^.left:=q;        {2}
    MakeNewHeight(q);
    MakeNewHeight(t);
    q:=t;              {3}
  End;
```

Рассмотрим фрагмент дерева, показанный на рис. 8.4. Здесь нарушена балансировка в левом поддереве вершины с указателем q (например, если была выполнена вставка вершины с ключом 70; баланс вершины с ключом 100 равен −2). Выполняя малый левый поворот, мы не исправим эту ситуацию: в повернутом поддереве баланс вершины с ключом 60 равен 2. Вывод: левый поворот (опустим в его названии слово «малый», ибо «большого» поворота, как это сделано в работе Н. Вирта[1)], в нашем изложении не будет) относительно вершины с указателем q приводит к результату, только если у левого потомка вершины левое поддерево имеет бóльшую высоту, чем правое.

На рис. 8.5 приведена схема разрешения возникшего затруднения. Относительно левого потомка вершины с указа-

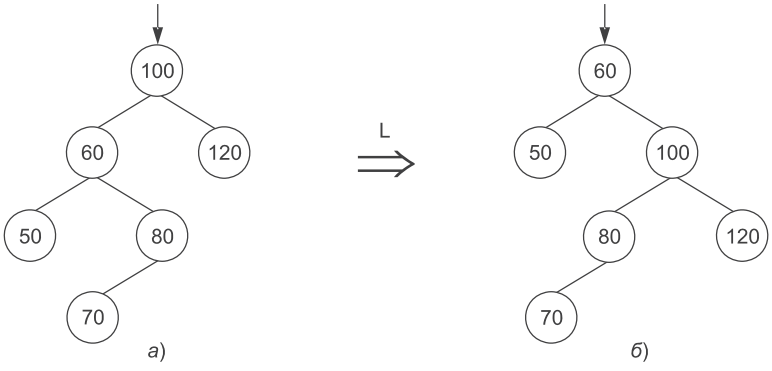


Рис. 8.4. Пример дерева, для которого малый левый поворот не дает правильного результата

1) *Вирт Н.* Алгоритмы+структуры данных=программы. М.: Мир, 1985. С. 248–260.

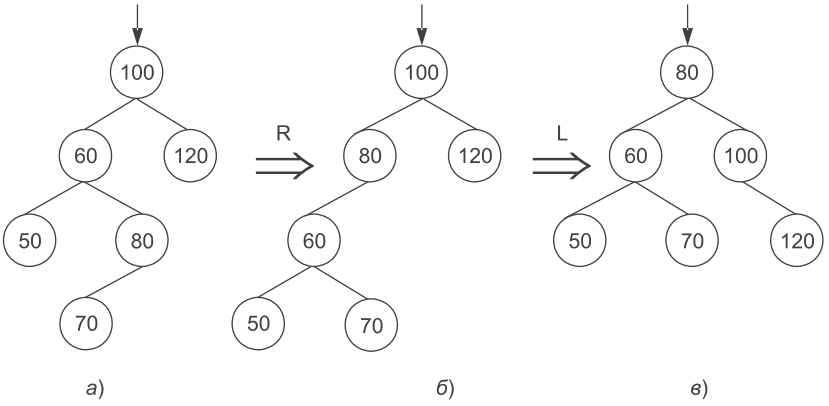


Рис. 8.5. Выполнение двух поворотов — правого и левого — восстанавливает балансировку дерева

телем q мы, несмотря на то что эта часть дерева уже сбалансирована, выполняем правый поворот (8.5*б*). Дерево останется несбалансированным, но теперь у левого потомка вершины q левое поддерево будет иметь бóльшую высоту, и появится возможность выполнить левый поворот (8.5*в*), который приведет к нужному нам результату: балансировка дерева будет восстановлена.

Аналогично разрешается ситуация и когда правое поддерево вершины с указателем q увеличивается по высоте — значение bal равно 2 (или когда левое поддерево уменьшается по высоте на единицу при bal = 1).

Перечисленными выше вариантами исчерпываются все случаи нарушения балансировки, и теперь у нас есть возможность реализовать полный текст логики с необходимыми и достаточными комментариями. Отметим, что она одна и та же как при вставке элемента в АВЛ-дерево, так и при удалении из него элемента (конечно, если при этом нарушается балансировка).

```
Procedure Ballance(Var q:pt);
  Var bal, old_height:Integer;
  Begin
    old_height:=q^.height;
    {Сохранение высоты поддерева}
    MakeNewHeight(q);
    {Вычисление новой высоты с учетом изменений}
```

```
    bal:=GetBallance(q);
    {Вычисление разности высот}
    If (bal>1) Then Begin
    {Правое поддерево выше допустимого уровня}
      If (GetBallance(q^.right)<0) Then
        TurnL(q^.right);
        {Если у правого потомка q, как корня
        поддерева, левое  поддерево имеет
        большую высоту, то необходимо выполнить
        левый поворот относительно этой вершины,
        хотя эта часть дерева уже сбалансирована.
        После этого становится возможным правый
        поворот относительно q}
      TurnR(q);
      {Выполняется правый поворот}
      If (q^.height=old_height) Then h:=False;
      {Сбрасывание флага. h – глобальная переменная
      для фиксации факта вставки или удаления
      элемента, когда, возможно, требуется
      балансировка дерева. Если новая высота q
      совпала со старой, а последняя соответствовала
      сбалансированному дереву, то дальнейшая
      балансировка не требуется}
    End
    Else
      If (bal<-1)Then Begin
      {Левое поддерево выше допустимого}
        If (GetBallance(q^.left)>0) Then
          TurnR(q^.left);
          {Если у левого потомка q, как корня
          поддерева, правое  поддерево имеет
          большую высоту, то выполнение сразу левого
          поворота относительно q не приводит
          к нужному результату. Относительно этого
          потомка, как корня поддерева, несмотря
          на сбалансированность этой части дерева,
          необходимо выполнить правый поворот}
        TurnL(q); {Выполняется левый поворот}
        If (q^.height=old_height) Then h:=False;
      {Сброс флага}
    End;
  End;
```

Хотя логика процедур вставки и удаления элементов АВЛ-дерева практически совпадает с описанной в разделе 5.2 для двоичного дерева поиска, приведем текст процедуры Insert, чтобы указать место и время вызова процедуры Balance. Восстановление балансировки дерева осуществляется на выходе из рекурсии, т. е. мы как бы идем снизу вверх по дереву, последовательно проверяя нарушение баланса вершин и восстанавливая балансировку, пока не достигнем корня дерева.

```
Procedure Insert(x:Integer; Var q:pt);
  Begin
    If (q=nil) Then Begin
    {Место для вставки вершины найдено}
      New(q);
      h:=True;
      {Признак: элемент вставлен и необходима
      проверка того, что дерево осталось
      сбалансированным}
      With q^ Do Begin
        data:=x;
        left:=nil;
        right:=nil;
        height:=1;
      End;
    End
    Else
      If x<q^.data Then Begin
      {Новая вершина должна принадлежать левому
      поддереву данной вершины}
        Insert(x, q^.left);
        If h Then Ballance(q);
        {h - глобальная переменная, ее инициализация
        значением False осуществляется в основной
        программе}
      End
      Else Begin
      {Или новая вершина должна принадлежать левому
      поддереву данной вершины}
        Insert(x, q^.right);
        If h Then Ballance(q);
      End;
  End;
```

Математическое отступление. Пусть n_h — минимальное кочичество узлов в АВЛ-дереве высоты h. Тогда $n_0 = 1$, $n_1 = 2$, $n_2 = 4$, $n_h = n_{h-1} + n_{h-2} + 1$ при $h \geqslant 2$.

Теорема. Для любого $h \geqslant 3$ выполняется неравенство $n_h \geqslant \alpha^{h+1}$, где $\alpha = (1+\sqrt{5})/2$ — положительный корень уравнения $x^2 - x - 1 = 0$.

Доказательство этой теоремы выполняется по индукции. Проверяется базис n_3 и n_4. Предполагается, что неравенство верно для интервала значений до некоторого значения t включительно, и выводится его верность для значения $t + 1$.

Из теоремы следует, что для любого АВЛ-дерева высоты h с n узлами выполняется соотношение $h + 1 < \log_\alpha n = \log_\alpha 2 \cdot \log_2 n \approx 1.44 \cdot \log_2 n$. Таким образом, время выполнения операций вставки и удаления имеет порядок $O(\log_2 n)$.

Упражнения

1. Можно ли изменить последовательность вызовов процедуры `MakeNewHeight` в процедурах `TurnL` и `TurnR`? Обоснуйте свой ответ.

2. Приведите пример дерева, для которого требуется выполнить последовательность из левого и правого поворотов для восстановления балансировки. Объясните, почему один только правый поворот не восстанавливает баланс этого дерева.

3. Приведите пример дерева, в котором после удаления элемента необходимо выполнить левый (правый) поворот.

4. Приведите пример дерева, в котором после удаления элемента необходимо выполнить последовательность из левого и правого поворотов.

5. Приведите пример дерева, в котором после удаления элемента необходимо выполнить последовательность из правого и левого поворотов.

6. Напишите процедуру удаления элемента из АВЛ-дерева. Выполните ее трассировку для различных типов поворотов.

7. Выполните трассировку процедуры `PrintStep` для дерева на рис. 8.2*б*. Первый вызов — `PrintStep(first,1)`. Какая информация и в какой последовательности (в каком виде) будет выведена в результате работы процедуры?

```
Procedure PrintStep(q:pt; r:Integer);
  Begin
    If (q<>nil) Then Begin
      PrintStep(q^.right,r+8);
      WriteLn(q^.data : r,',',q^.height : 2, ' ');
      PrintStep(q^.left,r+8);
     End;
  End;
```

8. Напишите программу для работы с АВЛ-деревом. Вставка, удаление элементов должны осуществляться случайным образом. Обеспечьте наглядный вывод АВЛ-дерева, отражающий его структуру.

9. Найдите АВЛ-дерево с 12 вершинами, имеющее максимальную высоту среди всех АВЛ-деревьев с 12 вершинами. В какой последовательности необходимо вставлять вершины (с помощью процедуры `Insert`), чтобы было получено такое дерево?

10*. Найдите такую последовательность из n включаемых в АВЛ-дерево элементов, чтобы повороты: левый, правый, левый —правый и правый — левый выполнялись, по крайней мере, один раз. Какова минимальная длина n такой последовательности?

11*. Найдите АВЛ-дерево с ключами 1, 2, ..., n и такую перестановку этих ключей, чтобы при удалении элементов из этого дерева выполнялись повороты: левый, правый, левый — правый и правый — левый, по крайней мере, один раз. Какова будет последовательность с минимальной длиной n?

8.2. «2–3»-деревья

Физику, биологу и математику предлагают объяснить, как могло случиться, что в пустой дом вошли *два* человека, а через некоторое время вышли *три*.

Физик: «Это ошибка наблюдения, такого быть не может».

Биолог: «Это естественный процесс размножения: у двоих родился третий».

Математик: «Нет ничего проще! Определим пустой дом как дом, в котором не более одного человека».

Анекдот

Определение

«2–3»-деревом называется дерево, в котором каждая вершина, не являющаяся листом, имеет двух или трех сыновей, а длины всех путей из корня в листья одинаковы.

Вершины в «2–3»-дереве могут быть двух видов: имеющие двух сыновей и имеющие трех сыновей. В вершине дерева хранятся два ключа — k_1 и k_2. В первом случае, как и в обычном бинарном дереве, левое поддерево содержит вершины с ключами, меньшими или равными значению k_1, а правое поддерево — с ключами, меньшими или равными значению k_2. Во втором случае левое поддерево имеет вершины с ключами, меньшими или равными значению k_1, среднее поддерево — вершины с ключами, большими k_1 и меньшими или равными k_2, а правое поддерево — с ключами, большими значения k_2 (рис. 8.6). Таким образом, k_1 и k_2 — это наибольшие элементы множеств в соответствующих поддеревьях.

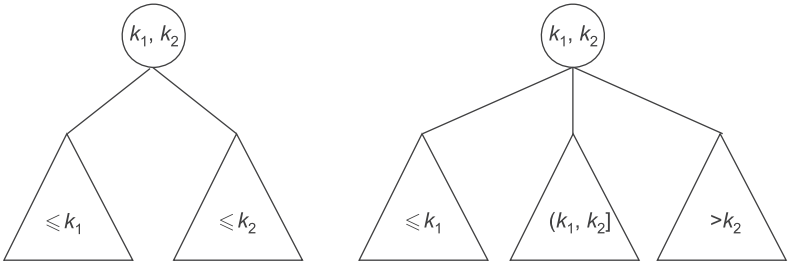


Рис. 8.6. Два вида вершин в «2–3»-дереве

«2–3»-дерево — это сбалансированное дерево. Другими словами, все его листья должны находиться на одной высоте (на одном уровне), так что длина пути от корня дерева до любого листа должна быть одинаковой. Эффективность операций поиска, вставки и удаления зависит от высоты h «2–3»-дерева. Элементы исходного множества приписаны листьям по порядку слева направо.

Дерево с минимальным количеством ключей — полное дерево, в котором все вершины (не листья) имеют двух сыновей. Следовательно, для «2–3»-дерева высотой h с n вершинами получаем:

$$n \geqslant 1 + 2 + \ldots + 2^h = 2^{h+1} - 1 \quad \text{и} \quad h \leqslant \log_2(n+1) - 1.$$

Аналогично, для «2–3»-дерева, все вершины (не листья) которого имеют трех сыновей:

$$n \leqslant 2 \cdot 1 + 2 \cdot 3 + \ldots + 2 \cdot 3^h = 2(1 + 3 + \ldots + 3^h) = 3^{h+1} - 1$$

и $h \geqslant \log_3(n+1) - 1$.

Из полученных оценок следует, что

$$\log_3(n+1) - 1 \leqslant h \leqslant \log_2(n+1) - 1.$$

Таким образом, операции поиска, вставки и удаления имеют временну́ю сложность $O(\log n)$.

Следующим утверждением (доказываемым классической индукцией по высоте дерева h) по-другому фиксируется связь между количеством вершин и числом листьев «2–3»-дерева с его высотой.

Утверждение. Пусть T — «2–3»-дерево высоты h. Тогда количество вершин этого дерева заключено между $2^{h+1} - 1$ и $(3^{h+1} - 1)/2$, а количество листьев — между 2^h и 3^h.

Рассмотрим вставку нового элемента с ключом k в «2–3»-дерево, т. е. пусть нам необходимо найти место для нового листа l, который будет содержать k. Предполагаем при этом, что дерево содержит более двух элементов. В любом случае поиск k окончится в вершине t, имеющей двух или трех сыновей, являющихся листьями.

Для содержательности обсуждения введем описание вершины дерева и ряд вспомогательных процедур:

```
Type pt=^elem;
     elem=Record
       leaf:Boolean;
       {True - лист; False - внутренняя вершина}
       key1,key2:Integer; {key1 - максимальное
       значение ключа в левой ветви дерева; key2 -
       максимальное значение в средней ветви дерева;
       для листа - значение в key1}
       left,mid,right:pt;
       {Указатели на левую,среднюю и правую
       ветви дерева}
     End;
Var root:pt; {Корень дерева}
```

В функции Cr создается новая внутренняя вершина и ссылка на нее передается вызывающей логике. Параметрами этой функции являются указатели на поддеревья создаваемой вершины и значения ключей.

```
Function Cr(l,m:pt; key1,key2:Integer):pt;
{Создание внутренней вершины с заданными параметрами}
  Var uk:pt;
  Begin
    New(uk);
    uk^.leaf:=False;
    uk^.left:=l;
    uk^.mid:=m;
    uk^.right:=nil;
    uk^.key1:=key1;
    uk^.key2:=key2;
    Cr:=uk;
  End;
```

Процедура Swap меняет значения указателей на поддеревья вершины.

```
Procedure Swap(Var t1,t2:pt);
  Var q:pt;
  Begin
    q:=t1;
    t1:=t2;
    t2:=q;
  End;
```

Функция Max предназначена для поиска максимального элемента в дереве.

```
Function Max(tree:pt):Integer;
{Выполняет поиск максимального элемента в поддереве}
  Begin
    While Not tree^.leaf Do
    {Переход на самый правый элемент в поддереве}
      If tree^.right<>nil Then tree:=tree^.right
      Else tree:=tree^.mid;
    Max:=tree^.key1;
  End;
```

Функция Max введена здесь только для наглядности изложения (максимальное значение можно передавать вверх по дереву при его модификации, а не искать его на каждом уровне). Ее использование изменяет временны́е характеристики операций вставки (Ins) и удаления (Del), поэтому в упражнении 4 вам будет предложено модифицировать эти процедуры, исключив из них функцию Max.

Пример. Пусть имеется дерево, изображенное на рис. 8.7*а*, и в него вставляется элемент 2. Спускаемся по дереву влево. У вершины t (она имеет ключи 1 и 4) только два сына, поэтому есть возможность добавить третьего, и в результате мы получаем дерево, представленное на рис. 8.7*б*.

Далее пусть необходимо вставить элемент 6 в дерево, представленное на рис. 8.7*б*. Находим вершину t. Она имеет трех сыновей и ключи 5 и 7. Поэтому мы создаем новую вершину $1'$ с двумя сыновьями и ключами 7 и 8. У старой (найденной) вершины оставляем двух сыновей и ключи 5 и 6. У отца вершины t (в данном случае это — корень) уже три сына. Если бы их было два, то путем добавления третьего сына обработка дерева бы заканчивалась. Мы же вынуждены создать новую вершину $2'$ с двумя сыновьями, где один сын — вновь созданная вершина $1'$, а второй — самый правый сын отца вершины t. У отца вершины t мы оставляем двух сыновей и, наконец, образуем новый корень $3'$ с двумя сыновьями — отцом вершины t и вершиной $2'$. Отметим, что процесс расщепления (разделения) вершин может продолжаться по всей цепочке предков вставленного листа, но так как само расщепление вершины требует константного вре-

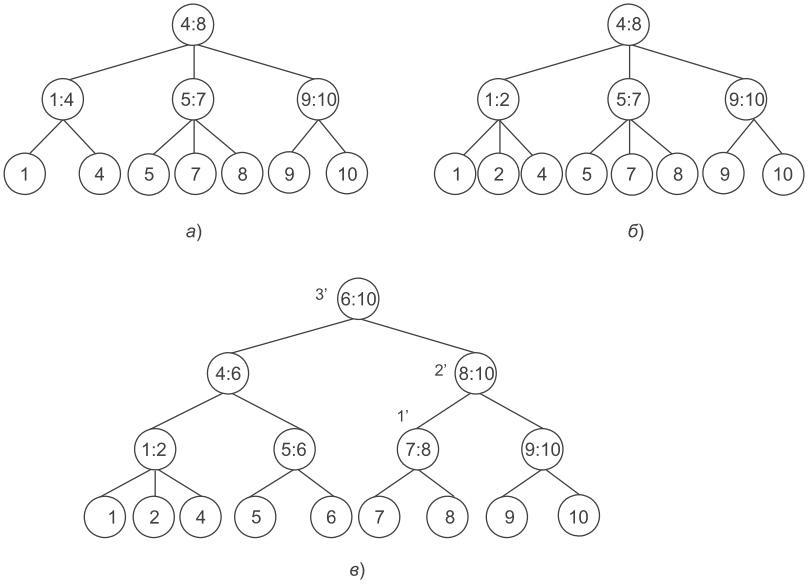


Рис. 8.7. Вставка элемента в «2–3»-дерево

мени, а количество вершин имеет порядок $O(\log_2 n)$, то время вставки имеет тот же порядок.

Приведем теперь формализованную запись логики вставки элемента в «2–3»-дерево, а так как она достаточно объемная, проиллюстрируем ее по ходу изложения дополнительными рисунками (рис. 8.8 и рис. 8.9[1]).

```
Procedure Ins(Var t:pt; k:Integer; Var bal:pt);
{Вставка элемента в дерево}
  Var uk:pt; {Новый лист}
  Begin
    If t^.left^.leaf Then Begin
    {Нашли вершину, сыновьями которой являются
    листья}
      New(uk); {Создаем лист}
      uk^.leaf:=True;
      uk^.key1:=k;
```

[1] Автор не претендует на то, что данную запись логики нельзя улучшить. Конечно, это сделать можно и нужно, — но в данном случае нам важнее «прозрачность» для лучшего объяснения материала.

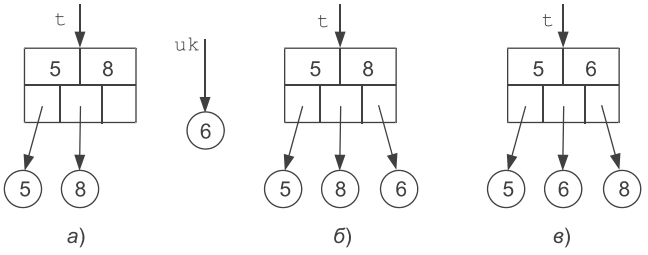


Рис. 8.8. Добавление третьего сына к вершине: *а*) первый шаг — создаем новый лист; *б*) второй шаг — формально подключаем его к вершине; *в*) третий шаг — наводим порядок в старшинстве сыновей

```
If t^.right=nil Then Begin
{Вершина – отец созданного листа - содержит
двух сыновей (см. пример на рис. 8.8)}
  bal:=nil;
  {Вершина не создается, балансировка
  не требуется}
  t^.right:=uk; {Созданный лист временно
  делаем самым правым сыном вершины t}
{Сортировка и формирование ключей вершины t}
  If t^.right^.key1<t^.mid^.key1
    Then Swap(t^.right, t^.mid);
  If t^.left^.key1>t^.mid^.key1
   Then Swap(t^.left,t^.mid);
  t^.key2:=t^.mid^.key1;
  t^.key1:=t^.left^.key1;
End
Else Begin
{Вершина – отец созданного листа - имеет
трех сыновей. Требуется создание новой
вершины bal с двумя листьями (см. пример
на рис. 8.9)}
  New(bal);
  bal^.leaf:=False;
  bal^.left:=t^.right;
  bal^.mid:=uk;
  bal^.right:=nil;
{Сортировка листьев в узлах t и bal
и заполнений ключей этих узлов}
```

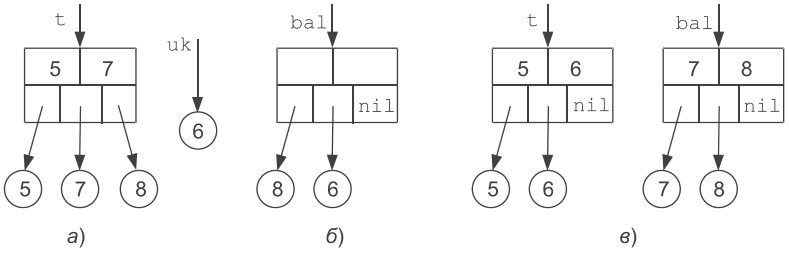


Рис. 8.9. Создание новой вершины: *а*) первый шаг — создаем новый лист; *б*) второй шаг — формально создаем новую вершину; *в*) третий шаг — наводим порядок в сыновьях новой и старой вершин

```
      If bal^.left^.key1>bal^.mid^.key1
        Then Swap(bal^.left,bal^.mid);
      If bal^.left^.key1<t^.mid^.key1
        Then Swap(bal^.left,t^.mid);
      If t^.left^.key1>t^.mid^.key1
        Then Swap(t^.left,t^.mid);
      bal^.key1:=bal^.left^.key1;
      bal^.key2:=bal^.mid^.key1;
      t^.key1:=t^.left^.key1;
      t^.key2:=t^.mid^.key1;
      t^.right:=nil;
    End;
  End
  {Обработка вершины, сыновьями которой являются
  листья, завершена}
  Else
  {Сыновья вершины не являются листьями}
    If k<t^.key1 Then Begin
    {Поиск места вставки нового элемента в левом
    поддереве}
      Ins(t^.left,k,bal);
      {Вызов вставки}
      If bal<>nil Then Begin {При вставке была
      создана новая вершина bal}
        If t^.right=nil Then Begin
        {Создание дополнительной вершины
        не требуется}
```

```
        t^.right:=t^.mid;
        t^.mid:=bal;
        t^.key1:=Max(t^.left);
        t^.key2:=Max(bal);
        bal:=nil;
      End
      Else Begin
      {Создание дополнительной вершины bal
      на этом уровне}
        uk:=Cr(t^.mid,t^.right,Max(t^.mid),
               Max(t^.right));
        t^.mid:=bal;
        t^.right:=nil;
        t^.key1:=Max(t^.left);
        t^.key2:=Max(bal);
        bal:=uk;
      End;
    End;
    {Завершение обработки bal <> nil}
End
{Завершение обработки в левом поддереве;
поиск места вставки нового элемента в среднем
поддереве}
Else
  If (k<t^.key2) Or (t^.right=nil) Then Begin
    Ins(t^.mid,k,bal);
    {Вызов вставки}
    If bal<>nil Then Begin
    {При вставке создана новая вершина bal}
      If t^.right=nil Then Begin
      {Создание дополнительной вершины
      не требуется}
        t^.right:=bal;
        bal:=nil
      End
      Else Begin
      {Создание дополнительной вершины
      bal на этом уровне}
        uk:=Cr(bal,t^.right,Max(bal),
               Max(t^.right));
```

```
            t^.right:=nil;
            bal:=uk;
          End;
          t^.key2:=Max(t^.mid);
        End;
      End
      {Завершение обработки в среднем поддереве}
    Else Begin
    {Поиск места вставки нового элемента
    в правом поддереве вершины}
      Ins(t^.right,k,bal);
      If bal<>nil Then Begin
      {Была создана новая вершина bal}
        uk :=Cr(t^.right,bal,Max(t^.right),
                Max(bal));
        t^.right:=nil;
        bal:=uk;
      End;
    End;
End;
```

В начальный момент работы с «2–3»-деревом необходимо создать вершину с двумя листьями, а затем в процессе работы с процедурой `Ins` предусмотреть «расщепление» корня, например, с помощью следующего фрагмента:

```
...
Read(k);
bal:=nil;
Ins(root,k,bal);
If bal<>nil Then
{Если была создана новая вершина bal на нижнем
уровне, то корни root и bal становятся сыновьями
нового корня root}
  root:=Cr(root,bal,Max(root),Max(bal));
...
```

Перейдем теперь к удалению элемента k из «2–3»-дерева. В принципе, ее действие — обратное вставке.

В начале приведем очевидные вспомогательные процедуры.

С помощью DelNode освобождается место в области памяти, предназначенной для переменных, определяемых с помощью указателей:

```
Procedure DelNode(Var tree:pt); {Удаление вершины}
  Begin
    Dispose(tree);
    tree:=nil;
  End;
```

Если у вершины удалено какое-либо поддерево (например, самое левое), то указатели на остальные поддеревья следует сдвинуть в описании вершины — это делает процедура Shift, чтобы сохранить структуру вершины:

```
Procedure Shift(uk:pt);
{Сдвиг ветвей в левую сторону на место пустых
(удаленных) ветвей}
  Begin
    If uk^.left=nil Then Begin
      uk^.left:=uk^.mid;
      uk^.mid:=uk^.right;
      uk^.right:=nil;
    End;
    If uk^.mid=nil Then Begin
      uk^.mid:=uk^.right;
      uk^.right:=nil;
    End;
  End;
```

В результате процесса удаления (а он заключается не только в удалении листа, так как у очередной вершины может остаться одно поддерево) требуется проверка того, что у вершины осталась одна ветвь, — это делает процедура Check:

```
Procedure Check(uk:pt;Var bal:pt);
{Проверка количества ветвей в вершине. Если
у вершины есть есть единственный сын, то значением
переменной bal является указатель на это поддерево}
  Begin
    bal:=nil;
    If uk^.mid=nil Then
      bal:=uk^.left;
  End;
```

Следующая процедура NewKeys предназначена для обновления значений ключей у вершины, являющейся корнем поддерева. Она введена для повышения наглядности изложения:

```
Procedure NewKeys(tree:pt);
{В поддереве tree - вычисление ключей key1 и key2}
  Begin
    tree^.key1:=Max(tree^.left);
    tree^.key2:=Max(tree^.mid);
  End;
```

Рассмотрим теперь особенности удаления элементов из «2–3»-дерева. Пусть имеется дерево, представленное на рис. 8.10*а*, и из него удаляется элемент 7.

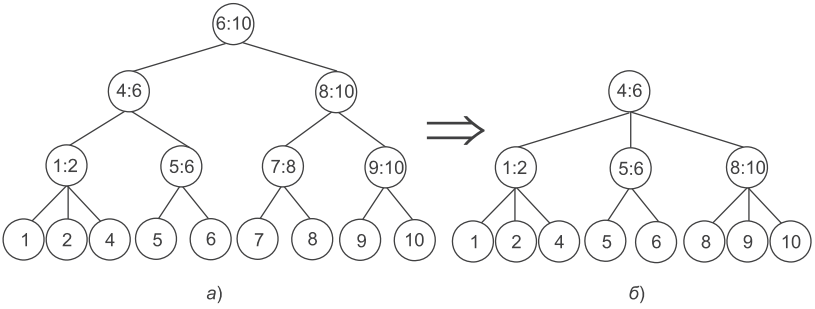
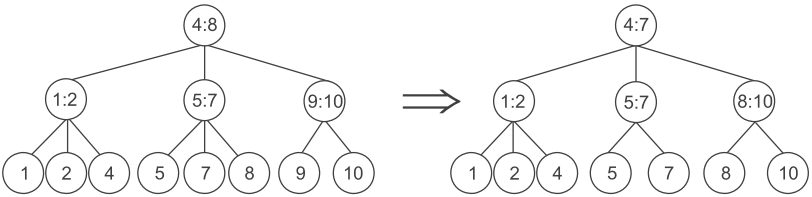


Рис. 8.10. Пример удаления элемента 7 из «2–3»-дерева

После того как этот элемент найден и удален у вершины, которая является его отцом, у этой вершины остается один сын. Ветвь, содержащая этого сына, подключается как самая левая к брату отца, а так как он имеет только двух сыновей, есть возможность сделать эту ветвь крайней левой ветвью брата отца. Однако после этого родитель братьев — вершина с ключами 8 и 9 — имеет одного потомка. Значит, необходимо обратиться уже к левому брату этой вершины и проанализировать количество сыновей у него. В данном примере левый брат имеет только двух сыновей, и эта ветвь дерева подключается как самая правая в качестве третьей ветви. Результат приведен на рис. 8.10*б*. В данном примере мы видим необходимость присоединения ветви дерева (как в качестве самой левой, так и в качестве самой правой), к некоторой (другой) вершине дерева.

Рассмотрим еще один пример дерева, показанный на рис. 8.11*а*. В нем удаляется элемент 9.



а) *б*)

Рис. 8.11. Пример удаления элемента 9 из «2–3»-дерева

Подключение к левому брату вершины с ключами 9 и 10 невозможно по той причине, что он уже имеет трех сыновей. Поэтому необходимо создать новую вершину и взять в качестве ее сыновей самого правого сына вершины с ключами 5 и 7 и оставшегося сына той вершины, у которой был удален сын. Разумеется, в процессе этих изменений, как в первом примере, так и во втором, должны корректироваться значения ключей у вершин.

Итак, в процессе удаления необходимо подключать ветви дерева к вершинам как в качестве самых левых, так и в качестве самых правых. Оформим эти действия в виде отдельной процедуры InsBranch, поясняя ее логику не только комментариями в листинге, но и на рис. 8.12 и 8.13.

```
Procedure InsBranch(Var bal:pt;Var t:pt;as:Boolean);
{Вставка ветви bal в ветвь t; параметр as
определяет левое или правое подключение}
  Var uk:pt;
  Begin
    uk:=nil;
    If as Then Begin
    {bal вставляется как самая левая ветвь
    (рис. 8.12)}
      If t^.right<>nil Then Begin
      {bal и t^.left образуют новую вершину, так как
      у вершины t три сына (рис. 8.12б)}
        uk:=Cr(t^.mid,t^.right,Max(t^.mid),
               Max(t^.right));
```

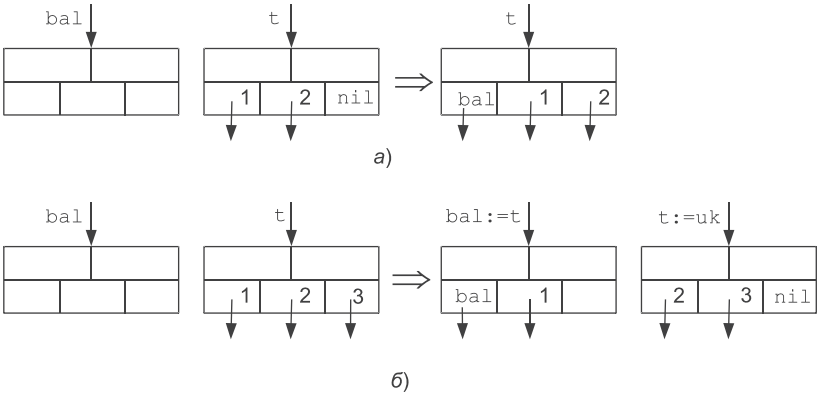


Рис. 8.12. Поддерево с указателем `bal` вставляется как самая левая ветвь поддерева с корнем `t`: ***а***) `t` имеет двух сыновей; ***б***) `t` имеет трех сыновей

```
  {Процедура создания новой вершины
  рассмотрена при обсуждении вставки элемента
  в дерево}
  t^.right:=nil;
End
Else t^.right:=t^.mid;
{bal сдвигает все ветви вправо; у вершины t
два сына (рис. 8.12а)}
  t^.mid:=t^.left;
  t^.left:=bal;
  t^.key2:=t^.key1;
  t^.key1:=Max(t^.left);
  bal:=t;
  t:=uk;
End
Else Begin
{bal вставляется как самая правая ветвь}
  If t^.right<>nil Then Begin
  {bal и t^.right образуют новую вершину;
  у вершины t три сына (рис. 8.13б)}
    uk:=Cr(t^.right,bal,Max(t^.right),
           Max(bal));
    t^.right:=nil;
    bal:=uk;
  End
```

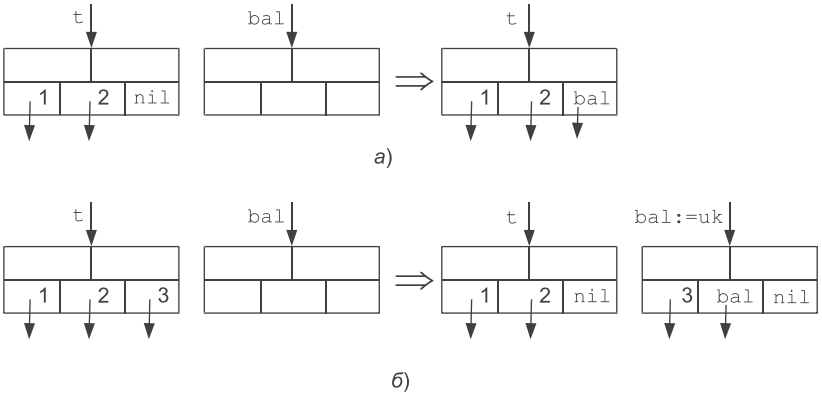


Рис. 8.13. Поддерево с указателем bal вставляется как самая правая ветвь поддерева с корнем t: а) t имеет двух сыновей; б) t имеет трех сыновей

```
      Else t^.right:=bal;
      {bal записывается в свободную ветвь
      t^.right (рис. 8.13а)}
  End;
End;
```

Теперь остается только «сложить» из описанных выше «кирпичиков» процедуру удаления — Del. «Костяк» этой процедуры составляет поиск удаляемого элемента (рекурсивный спуск), а на выходе из рекурсии (рекурсивный подъем) необходимо добавить логику балансировки дерева, которая требует знаний о том, в какой ветви дерева (левой, средней или правой) относительно каждой вершины t осуществлялся поиск. Другими словами, при возврате в вершину t необходимо проверить, требуется ли выполнять балансировку дерева, и если да, то необходимо знать, из какой ветви произошел возврат, так как только это знание позволяет корректно выполнить балансировку. Описанная ситуация разъясняется на рис. 8.14, на который имеются ссылки и в формализованной записи процедуры:

```
Procedure Del(Var t:pt;k:Integer;Var bal:pt);
{Удаление вершины и балансировка дерева}
  Var q:pt;
  Begin
    bal:=nil;
```

```
If t^.left^.leaf Then Begin
{Сыновьями вершины являются листья}
{Поиск удаляемого элемента}
  If t^.left^.key1=k Then DelNode(t^.left);
  If t^.mid^.key1=k Then DelNode(t^.mid);
  If t^.right<>nil Then
    If t^.right^.key1=k Then DelNode(t^.right);
    Shift(t);
    {Сдвиг листьев влево после удаления}
    Check(t,bal);
    {Проверка на единственность ветви}
    If bal=nil Then NewKeys(t);
    {В поддереве t не одна ветвь (bal = nil),
    поэтому корректируем значения ключей key1
    и key2}
    End
  Else
    If k<=t^.key1 Then Begin
    {Поиск удаляемого элемента в левой ветви}
      Del(t^.left,k,bal);
      If bal<>nil Then Begin
      {Возвращена единственная ветвь для вставки
      в другую ветвь}
        InsBranch(bal,t^.mid,true);
        {Вставка ветви bal в ветвь t^.mid
        (рис. 8.14а)}
        t^.left:=bal;
        Shift(t);
        {Сдвиг ветвей влево после
        удаления/вставки}
        Check(t,bal);
        {Проверка на единственность ветви}
      End;
      If bal=nil Then NewKeys(t);
      {В поддереве t вычисление ключей key1
      и key2}
    End
    Else
      If k<=t^.key2 Then Begin
      {Поиск удаляемого элемента в средней
      ветви}
```

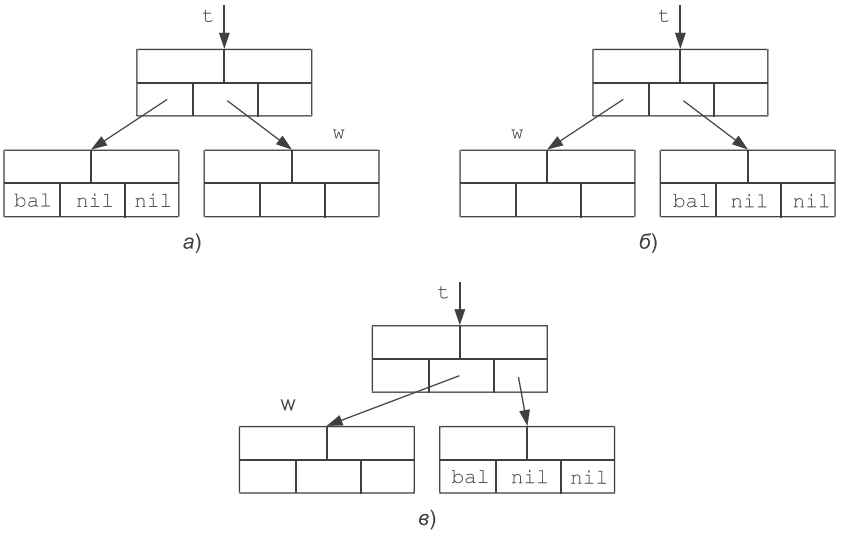


Рис. 8.14. Три случая присоединения ветви `bal` к вершине `w`:
***а*) в качестве самой левой к средней ветви вершины `t`;**
***б*) в качестве самой правой к левой ветви вершины `t`;**
***в*) в качестве самой правой к средней ветви вершины `t`**

```
  Del(t^.mid,k,bal);
  If bal<>nil Then Begin
  {Возвращена единственная ветвь
  для вставки в другую ветвь}
    InsBranch(bal,t^.left,false);
    {Вставка ветви bal в ветвь t^.left
    (рис. 8.14б)}
    t^.mid:=bal;
    Shift(t);
    {Сдвиг ветвей влево после
    удаления/вставки}
    Check(t,bal);
    {Проверка на единственность ветви}
  End;
  If bal=nil Then NewKeys(t);
  {В поддереве tree - вычисление ключей
  key1 и key2}
End
```

```
        Else
          If t^.right<>nil Then Begin
          {Поиск удаляемого элемента в правой
          ветви}
            Del(t^.right,k,bal);
            If bal<>nil Then Begin
            {Возвращена единственная ветвь
            для вставки в другую ветвь}
              InsBranch(bal,t^.mid,false);
              {Вставка ветви bal в ветвь t^.mid
              (рис. 8.14в)}
              t^.right:=bal;
              Shift(t);
              {Сдвиг ветвей влево после
              удаления/вставки}
            End;
            t^.key2:=Max(t^.mid);
          End;
  End;
```

Остался нерассмотренным лишь случай изменения значения указателя на корень дерева, действия с которым осуществляются в вызывающей логике. Следующий фрагмент описывает эти действия:

```
Begin
  Read(k);
  Del(root,k,bal);
  {Вызов процедуры удаления элемента из дерева}
  If bal<>nil Then Begin
  {Одна из ветвей была удалена}
    Shift(root);
    If root^.mid=nil Then
    {Если в корне одна ветвь, то изменение корня}
    root:=root^.left;
  End;
End;
```

Итак, мы завершили достаточно детальное описание операций вставки, поиска и удаления в «2–3»-дереве. Каждая операция выполняется за время $O(\log_2 n)$ для дерева с n листьями, а n таких операций — за время $O(n \cdot \log_2 n)$.

Таким образом, «2–3»-дерево может быть использовано для организации словаря с указанной производительностью.

Очевидно, что поиск как максимального, так и минимального элемента в «2–3»-дереве осуществляется за время $O(\log_2 n)$. Поскольку очередь с приоритетами характеризуется набором операций: «вставить», «удалить» и «найти максимальный элемент», «2–3»-дерево может быть использовано как структура данных для реализации очереди с приоритетом с производительностью $O(n \cdot \log_2 n)$ для n операций.

Упражнения

1. Постройте «2–3»-дерево для следующего набора символов: И, Н, Ф, О, Р, М, А, Т, И, К, А. Используйте при этом алфавитное упорядочение букв и их последовательную вставку в пустое дерево. Повторяющиеся символы в дереве не дублируются.
2. Дана последовательность из 12 целых чисел: 6, 4, 9, 1, 13, 5, 7, 12, 8, 3, 11, 10. Пусть числа именно в этом порядке вставляются в первоначально пустое «2–3»-дерево. Изобразите графически процесс вставки каждого элемента. Листья дерева упорядочены по возрастанию слева направо.
3. Дано «2–3»-дерево, имеющее 12 листьев. Значения ключей упорядочены слева направо. Изобразите графически процесс удаления шести самых правых (шести средних, шести левых) элементов дерева. Элементы удаляются в произвольном, но заранее заданном порядке.
4. Исключите функцию `Max` из реализации процедуры `Ins` и, аналогично, процедуру `NewKeys` из реализации процедуры `Del`.
5. Разработайте алгоритм поиска разности между наибольшим и наименьшим значениями ключей «2–3»-дерева. Оцените его эффективность в наихудшем случае.
6. Напишите полную реализацию программы работы с «2–3»-деревом, где множество листьев упорядочено, а вставка и удаление элементов в дерево осуществляются случайным образом.

7. Пусть T_B и $T_{2\text{–}3}$ —это, соответственно, классическое бинарное дерево поиска и «2–3»-дерево, построенные для одного и того же набора ключей, вставляемых в деревья в одном и том же порядке. Истинно ли утверждение, что поиск одного и того же ключа в $T_{2\text{-}3}$ всегда требует меньшего или такого же количества сравнений ключей, что и поиск в T_B?

8.3. *Б*-деревья

> Наблюдайте, обобщайте,
> доказывайте и передоказывайте
> по-новому.
>
> *Д. Пойа*

Другим типом сбалансированных деревьев являются *Б-деревья* (*B-tree*), которые относятся к классу сильно ветвящихся деревьев. Отличие заключается в том, что в вершине дерева хранится не одно значение ключа, а несколько и, соответственно, это количество ключей определяет количество потомков вершины. При этом вершину называют *страницей*. *Б*-дерево также определяется характеристикой, называемой *порядком* и обозначаемой как n (*Б-дерево порядка n*).

Определение *Б*-дерева:

- каждая страница содержит не более $2 \cdot n$ элементов (ключей);
- каждая страница, кроме корневой, содержит не менее n элементов;
- каждая страница является либо листом, т. е. не имеет потомков, либо имеет $m + 1$ потомков, где m — количество находящихся в ней ключей;
- все листья находятся на одном уровне.

Приведем возможное описание страницы, удовлетворяющее этому определению; именно оно определяет организацию данных в единое целое и взаимосвязи между ними, — другими словами — структуру данных:

```
Const n=2;
      max_2n=2*n;
```

```
Type  pt=^Page;
      Cell=Record
        value:LongInt;
        child:pt;
      End;
      Page=Record
        m:LongInt;
        p0:pt;
        E:Array[1..max_2n] Of Cell;
      End;
```

Страница (вершина) *Б*-дерева содержит количество ключей (`m`), указатель на самого левого сына (`p0`) и массив записей (`Cell`), в структуру каждой из которых входит значение ключа (`value`) и указатель на соответствующего сына (`child`).

Рассмотрим вставку элемента в *Б*-дерево. Для последовательности чисел 45, 55, 65, 75, **85**, 10, 20, 70, 90, **100**, 68, 80, 95, 105, **110**, **72** (на обработку числа поступают именно в такой очередности!) *Б*-дерево порядка 2 имеет вид, представленный на рис. 8.15 (у листьев позиции для указателей не показаны). При этом жирным шрифтом в последовательности чисел выделены элементы, при вставке которых формируются новые страницы. По своему виду это — дерево поиска, только более сложное по структуре вершины.

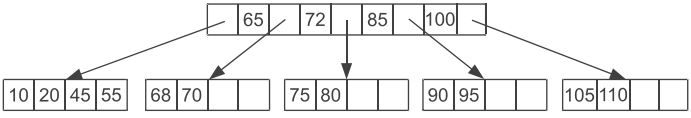


Рис. 8.15. Пример *Б*-дерева

Как получено это *Б*-дерево?

Первые четыре элемента последовательности (при $n = 2$) располагаются в корне (см. рис. 8.16*а*, левая часть).

Теперь необходимо вставить число 85. Выделим первый процесс — это «расщепление» вершины на две и передача среднего элемента (ключа) наверх. В данном случае «передача наверх» сводится к созданию новой вершины — корня с одним значением ключа, в результате чего мы получаем *Б*-дерево, изображенное на рис. 8.16*а* (правая часть рисунка).

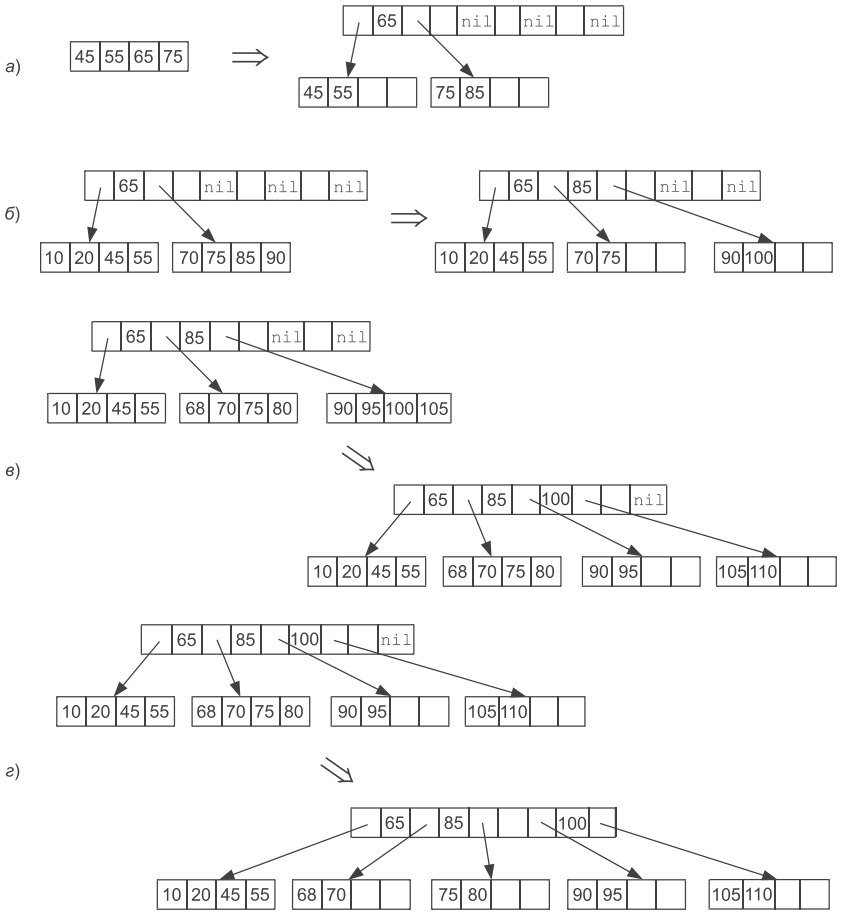


Рис. 8.16. Процесс вставки элементов в *Б*–дерево: *а*) элемента 85; *б*) элемента 100; *в*) элемента 110; *г*) элемента 72

Оформим теперь все сказанное в формализованном виде:

```
Procedure Insert(value:LongInt; Var root:pt);
  Var w:Cell;
      h:Boolean;
      t:pt;
  Begin
    h:=False;
```

```
    {Логическая переменная, значение которой
    определяет, надо ли создавать вершины на
    данном уровне дерева}
    InsertInside(value,root,h,w);
    {Вставка осуществляется внутри дерева. Мы пока
    не знаем как, но это и не страшно, – главное,
    что если в результате вставки внутри дерева
    оказалось, что необходимо изменить корень, то
    h = True, а переменная w содержит указатель на
    страницу, являющуюся правым сыном корня. В этот
    момент времени корень всегда содержит один ключ}
    If h Then Begin
      t:=root;
      New(root);
      {Создаем новую страницу – корень Б-дерева}
      With root^ Do Begin
      {Пример, показанный на рис. 8.16а:
      указатель w определяет новую вершину,
      созданную в InsertInside с ключами 75 и 85;
      по старому значению root находится ключ 65
      с указателем p0 на вершину с ключами 45 и 55}
        m:=1;
        p0:=t;
        E[1]:=w;
      End
    End
  End;
```

Элементы последовательности 10, 20, 70, 90 записываются в существующие страницы *Б*-дерева (листья). Пока нам еще непонятно, как это делается, но выделим некоторые принципиальные моменты. Вставка элемента всегда осуществляется на уровне страниц-листьев путем создания записи типа `Cell`, а прежде чем вставлять элемент, требуется пройти по *Б*-дереву для поиска нужной страницы-листа. Такой поиск аналогичен обычному поиску в двоичном дереве; отличие только в том, что массивы ключей в каждой странице упорядочены по возрастанию, и с помощью двоичного поиска за время $O(\log_2 m)$ определяется ветвь дерева, по которой следует идти при поиске места для вставки элемента. На последнем уровне поиска (значение указателя равно `nil`) создается элемент массива `E` (запись типа `Cell`), а пере-

менной логического типа, фиксирующей этот факт, присваивается значение True:

```
Procedure InsertInside(val:LongInt; a:pt; Var
          h:Boolean; Var v:Cell);
  Var l,r,k:LongInt;
      u:Cell;
  Begin
    If a=nil Then
    {Создаем запись типа Cell. Мы находимся на самом
    нижнем уровне - уровне листьев Б-дерева}
      With v Do Begin
        h:=True;
        value:=val;
        child:=nil
      End
    Else Begin
    {Мы находимся во внутренней (не лист) вершине
    Б-дерева. Выполняем двоичный поиск
    для определения - куда идти дальше}
      With a^ Do Begin
        l:=1; {Двоичный поиск}
        r:=m;
        Repeat
          k:=(l+r) Shr 1;
          If val<=E[k].value Then r:=k-1;
          If val>=E[k].value Then l:=k+1;
        Until r<l;
        If r=0 Then InsertInside(val,p0,h,u)
          Else InsertInside(val,E[r].child,h,u);
        If h Then InsertLocal;
        {Действия на выходе из рекурсии -
        осуществляется вставка элемента на уровне
        конкретной вершины. Последняя может быть
        как листом, так и внутренней вершиной,
        в частности - корнем Б-дерева. Процедура
        InsertLocal включена в InsertInside -
        переменные InsertInside "видимы" и
        в InsertLocal}
      End
    End
  End;
```

Пусть в *Б*-дерево на рис. 8.16*г* вставляется элемент 15. В результате поиска находим вершину (указатель — значение переменной a) с ключами 10, 20, 45, 55, но она содержит максимальное количество ключей, и требуется ее «расщепление». Создается новая вершина (указатель — значение переменной b) с ключами 40 и 55 (рис. 8.17); в старой мы оставляем ключи 10, 15, а ключ 20 и указатель на новую вершину передаем на следующий по пути к корню уровень.

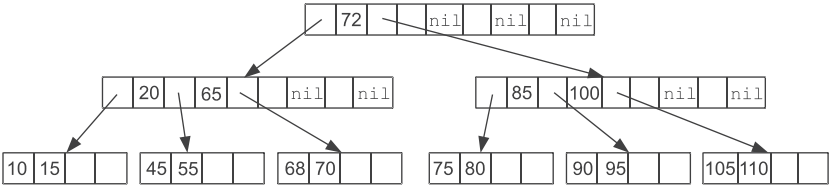


Рис. 8.17. Вставка элемента 15 в результирующее *Б*-дерево, изображенное на рис. 8.16*г*

На этом уровне вершина (указатель — значение переменной a) содержит максимальное количество ключей: 65, 72, 85, 100. Следовательно, вновь требуется ее «расщепление». Создается новая вершина (указатель — значение переменной b) с ключами 85, 100, а в старой вершине ключи становятся равными 20 и 65 (рис. 8.17), так как средний ключ 72 (а также указатель на новую вершину) опять передается вверх по дереву. Этот уровень в нашем примере — уровень корня, и действия на нем осуществляются уже в вызывающей логике.

Действия по вставке элемента в *Б*-дерево на конкретном уровне в формализованном виде оформим как процедуру InsertLocal:

```
Procedure InsertLocal;
  Var i:LongInt;
      b:pt;
  Begin
    With a^ Do Begin
    {a - значение указателя вершины, на уровне
    которой осуществляется вставка}
      If m<max_2n Then Begin
      {Простой случай - вершина содержит
      не максимальное количество ключей
```

```
В этой ветви логики вставлялись элементы 10,
20, 70, 90, 68, 80, 95, 105 (рис. 8.16б, в)}
  m:=m+1;
  {Увеличиваем количество ключей на странице}
  h:=False;
  {Структура Б-дерева не изменяется}
  For i:=m DownTo r+2 Do E[i]:=E[i-1];
  {Значение r получено в результате двоичного
  поиска именно на этой вершине-странице
  Б-дерева. Выполняем сдвиг ячеек массив E
  вправо, освобождая тем самым место для
  новой ячейки, содержащей ключ и ссылку
  на вершину-сына}
  E[r+1]:=u;
  {Переменная u – это ячейка (Cell),
  содержащая ключ и указатель на вершину
  следующего уровня в Б-дереве. Она создана
  при следующем вызове InsertInside.
  Этот шаг уже сделан, так как InsertLocal
  работает на выходе из рекурсивного обхода}
End
Else Begin
{Количество ключей вершины, с учетом
вставляемого, превышает максимальное значение}
  New(b);
  {Требуется создание новой вершины. Пример
  показан на рис. 8.16б, в, г – вставка
  элементов 100, 110, 72}
  If r<=n Then Begin
  {Место (r), найденное с помощью двоичного
  поиска на этом уровне, меньше или равно
  половине максимального количества элементов
  массива E. Вставка элемента 72 (рис. 8.16г)}
    If r=n Then v:=u
    {Ячейка, передаваемая на верхний уровень, -
    это ячейка, созданная на данном уровне}
    Else Begin
      v:=E[n];
      {Ячейка, передаваемая на верхний уровень
      (предку текущей вершины), - это средняя
      ячейка массива ключей}
```

```
      For i:=n DownTo r+2 Do E[i]:=E[i-1];
      {Сдвиг – освобождаем место для ячейки}
      E[r+1]:=u;
      {Вставляем созданную ячейку в найденное
      место массива E вершины-страницы
      текущего уровня}
    End;
    For i:=1 To n Do b^.E[i]:=a^.E[i+n];
    {Вновь создаваемой вершине (указатель b)
    присваиваем ячейки из старшей (правой)
    части вершины текущего уровня
    (указатель a)}
  End
  {Конец ветви Then сравнения r<=n}
  Else Begin
  {Место r, найденное с помощью  двоичного
  поиска на этом уровне, больше половины
  максимального количества элементов
  массива E. Пример - рис. 8.16б, в}
    r:=r-n;
    {Работаем с правой частью массива E}
    v:=E[n+1];
    {Передаем вверх по дереву первый
    элемент правой части массива E}
 {Формирование массива E для вновь создаваемой
 вершины}
    For i:=1 To r-1 Do b^.E[i]:=a^.E[i+n+1];
    b^.E[r]:=u;
    {Вставляем на найденное место}
    For i:=r+1 To n Do b^.E[i]:=a^.E[i+n]
  End;
  m:=n;
  {У вершины этого уровня (указатель a)
  и вновь созданной вершины (указатель b)
  количество ключей равно n}
  b^.m:=n;
  b^.p0:=v.child;
  v.child:=b;
End
{Конец ветви Else – количество ключей вершины,
с учетом вставляемого, превышает
максимальное значение}
```

```
    End {Конец With }
  End;
```

Перейдем теперь к обсуждению логики удаления элемента из *Б*-дерева. Основная процедура получает значение удаляемого элемента и указатель на корень дерева. Удаление элемента из *Б*-дерева может повлечь за собой изменение всей структуры дерева вплоть до удаления корня. Необходимость последнего действия фиксируется значением `True` логической переменной `h`, а все действия по удалению внутренних вершин возлагаются на процедуру `DelInside`.

```
Procedure Delete(value:LongInt;Var root:pt);
  Var h:Boolean;
      t:pt;
  Begin
    h:=False;
    DelInside(value,root,h);
    If h And (root^.m=0) Then Begin
      t:=root;
      root:=t^.p0;
      Dispose(t);
    End
  End;
```

Рассмотрим теперь логику процедуры `DelInside`. Ее основное предназначение — найти удаляемый элемент в *Б*-дереве. При этом мы получаем значение указателя на текущую вершину *Б*-дерева. С помощью двоичного поиска определяем, есть ли соответствующий ключ. Если нет, то двоичный поиск дает нам указатель на поддерево, в котором может быть ключ, и следует перейти по этой ссылке для продолжения поиска. Если да, то элемент найден. Смотрим, где он находится, — лист это или внутренняя вершина. В первом случае осуществляется простое удаление и формирование значения переменной `h`. Во втором случае необходимо найти самый правый элемент поддерева, находящегося слева от удаляемого элемента (возможен также вариант поиска самого левого элемента, но в правом поддереве относительно удаляемого элемента). Эти действия мы вынесем в отдельную процедуру `Del`. Осталось заметить, что в результате рекурсивного спуска по дереву и выполняемых при этом действий (удаление элемента из листа, замена на

самый правый элемент поддерева) может нарушиться балансировка *Б*-дерева. Необходимость балансировки (если в какой-то вершине количество ключей стало меньше значения n) фиксируется в переменной h. На каждом шаге выхода из рекурсии значение h проверяется, и если требуется балансировка, то осуществляется обращение к соответствующей логике, оформленной как процедура MergeSplit («слияние — расщепление»).

Теперь у нас появилась возможность дать формализованную запись DelInside с необходимыми комментариями:

```
Procedure DelInside(val:LongInt; a:pt; Var
          h:Boolean);
  Var r,l,k:LongInt;
      q:pt;
  Begin
    If a=nil Then Exit
    {Элемент не найден в Б-дереве}
      Else
        With a^ Do Begin
          l:=1; {Двоичный поиск}
          r:=m;
          Repeat
            k:=(l+r) Shr 1;
            If x<=E[k].value Then r:=k-1;
            If x>=E[k].value Then l:=k+1;
          Until l>r;
          If r=0 Then q:=p0 Else q:=E[r].child;
          If l-r>1 Then Begin
          {Элемент найден: l-r=2}
            If q=nil Then Begin
            {Если это лист, то уменьшаем количество
            элементов и сдвигаем влево ключи,
            которые расположены справа от удаляемого
            элемента}
              m:=m-1;
              h:=m<n;
              {Если количество элементов в вершине
              меньше минимально допустимого, то
              требуется перестройка дерева. h=True}
```

```
            For l:=k To m Do E[l]:=E[l+1];
          End
        Else Begin
          Del(q,h);
          {Находим самый правый элемент этого
          поддерева. Сравните листинг с процедурой
          Del (см. раздел 5.20}
          If h Then MergeSplit(a,q,r,h);
          {Восстановление балансировки Б-дерева
          путем слияния и расщепления вершин}
        End
      End
      Else Begin {Элемент не найден}
        DelInside(x,q,h);
        {Продолжаем поиск. Идем дальше по Б-дереву
        в поддерево, определяемое значением
        указателя q}
        If h Then MergeSplit(a,q,r,h);
      End
    End
End;
```

Поясним процедуру Del на конкретном примере (рис. 8.18). Пусть удаляется элемент 100, который найден в вершине (указатель a) на месте k (рис. 8.18). Находим самый правый элемент в левом поддереве (это элемент 99): указателем на вершину, содержащую 99, является p. Действиями, приведенными на рис. 8.18, дерево подправляется в вершине a^, но в вершине p^ остается один элемент, и значением h

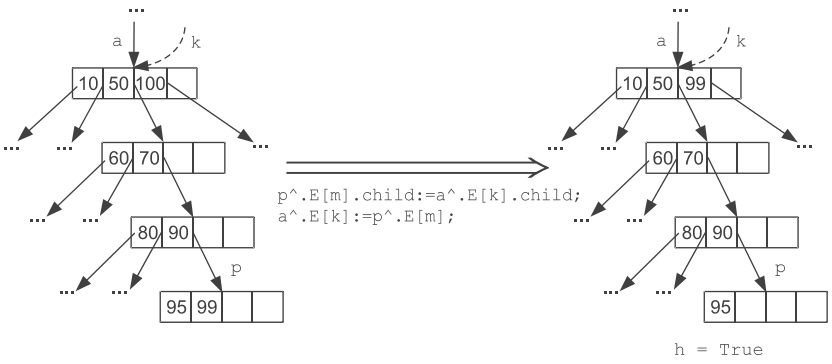


Рис. 8.18. Пример поиска самого правого элемента поддерева

становится True: следовательно, требуется балансировка. (Если бы в вершине p^ осталось не менее n элементов, то на этом процесс удаления элемента был бы завершен.)

```
Procedure Del(p:pt; Var h:Boolean);
{Поиск самого правого элемента поддерева. Процедура
Del - локальная по отношению к DelInside, поэтому
из нее есть доступ к переменным последней}
  Var t:pt;
  Begin
    With p^ Do Begin
      t:=E[m].child;
      If t<>nil Then Begin
        Del(t,h);
        {Идем вправо по поддереву}
        If h Then MergeSplit(p,t,m,h);
        {Мы обязаны проверить: требуется ли
        балансировка дерева}
      End
      Else Begin
      {Самый правый элемент поддерева найден.
      Выполняем корректировку вершины, из которой
      удаляется элемент (указатель a)}
        p^.E[m].child:=a^.E[k].child;
        {Рис. 8.18 - иллюстрация этой ситуации}
        a^.E[k]:=p^.E[m];
        m:=m-1;
        h:=m<n;
      End;
    End;
  End;
```

Остается проанализировать восстановление балансировки *Б*-дерева, которая может оказаться нарушенной в результате удаления элемента.

Начнем с простых примеров. Пусть удален элемент 23 (см. рис. 8.19; напомним, что n здесь равно 2). В вершине (w) тогда остается один элемент (т. е. не выполняется требование по количеству ключей), и у нее есть правый брат (t). В этой ситуации все зависит от количества ключей у брата. Если оно равно n, то единственной возможностью является объединение вершин (слияние — рис. 8.19*а*), а если это ко-

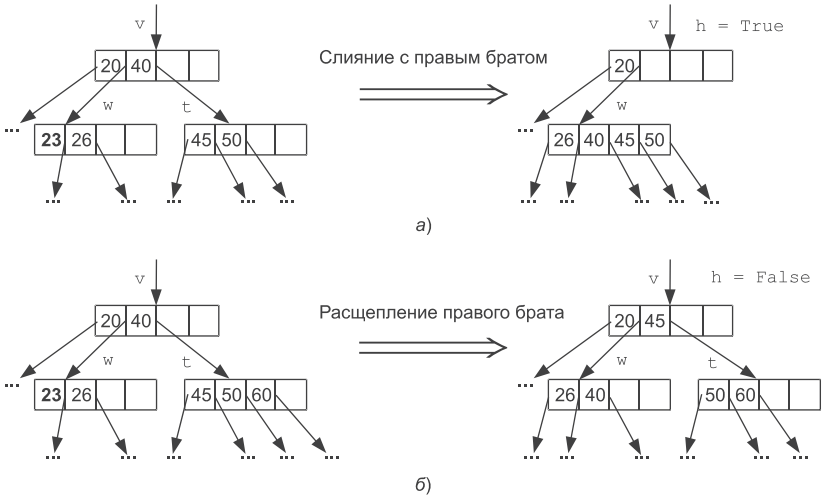


Рис. 8.19. Балансировка *Б*-дерева: *а*) путем слияния вершины с правым братом; *б*) путем расщепления правого брата. Жирным шрифтом выделен удаляемый элемент (23)

личество ключей больше `n`, то у брата можно позаимствовать ключи (расщепить его с точки зрения вершины `w` — рис. 8.19*б*). В последнем случае структура *Б*-дерева сохраняется в том смысле, что дальнейшая балансировка не требуется. (На рис. 8.19 и 8.20 обозначены указатели `v`, `w` и `t` на соответствующие вершины; иногда для краткости мы будем говорить, например, просто «вершина `v`», подразумевая при этом вершину, адрес которой определяется значением указателя `v`. Заметим также, что на рис. 8.19 и 8.20 не записаны места для указателей, так как это было сделано на предыдущих рисунках.)

Особым является случай, когда вершина с удаленным элементом оказывается самым правым сыном вершины-предка. В этом случае приходится обращаться к ее левому брату — см. рис. 8.20. В остальном же логика как слияния, так и расщепления здесь аналогична с (точностью до деталей) той, что представлена на рис. 8.19.

Формализованная запись восстановления балансировки *Б*-дерева оформлена ниже как процедура `MergeSplit` и сводится к подробной записи всех разобранных действий:

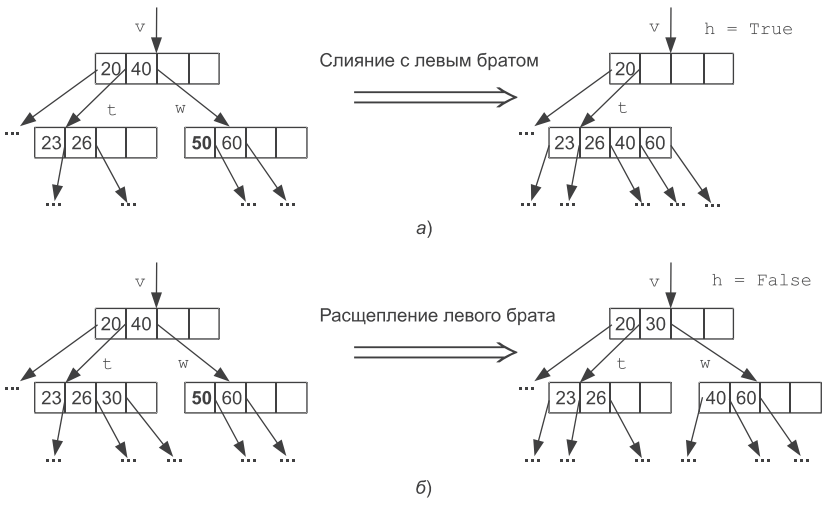


Рис. 8.20. Балансировка *Б*-дерева: *а*) путем слияния с левым братом; *б*) путем расщепления левого брата. Жирным шрифтом выделен удаляемый элемент (50)

```
Procedure MergeSplit(v,w:pt; s:LongInt;
                     Var h:Boolean);
{w – страница, на которой был удален элемент; v –
страница-родитель w; s – место ссылки (индекс
в массиве E) у вершины v на потомка w; h=True;
w^.m=n-1}
  Var t:pt;
      i,k,mt,mv:LongInt;
  Begin
    mv:=v^.m;
    If s<mv Then Begin
    {У вершины w есть правый брат - см. рис. 8.19}
      s:=s+1;
      t:=v^.E[s].child;
      {Указатель на правого брата}
      mt:=t^.m;
      {Количество сыновей у правого брата}
      k:=(mt-(n-1)) Shr 1;
      w^.E[n]:=v^.E[s];
      {Один ключ от родителя (v) отдаем сыну (w),
      тогда самое левое поддерево t становится
      правым поддеревом у отданного ключа}
```

```
w^.E[n].child:=t^.p0;
If (k>0) Then Begin
{Расщепление правого брата – см. рис. 8.19б}
  For i:=1 To k-1 Do w^.E[n+i]:=t^.E[i];
  {Забираем k-1 ключей у вершины t и передаем
  их вершине w}
  v^.E[s]:=t^.E[k];
  {Один ключ отдается родителю – вершине v,
  на то место, где был ключ, отданный вершине w}
  v^.E[s].child:=t;
  t^.p0:=t^.E[k].child;
  {Корректировка данных в вершине t, у которой
  были заимствованы ключи и которая
  подвергалась расщеплению}
  mt:=mt-k; {Оставшееся количество ключей}
  For i:=1 To mt Do t^.E[i]:=t^.E[i+k];
  {Сдвигаем элементы массива E влево на k
  освободившихся мест}
  t^.m:=mt;
  w^.m:=n-1+k;
  {Новое количество ключей у вершины w -
  той, у которой была их нехватка}
  h:=False;
  {Дальнейшая балансировка не требуется}
End
Else Begin
{Слияние с правым братом - см. рис. 8.19а}
  For i:=1 To n Do w^.E[n+i]:=t^.E[i];
  {Отдаем n ключей (все ключи) вершины t
  вершине w}
  For i:=s To mv-1 Do v^.E[i]:=v^.E[i+1];
  {Сдвигаем на одну позицию ключи вершины v;
  один ключ был передан сыну w вершины v}
  w^.m:=max_2n;
  {У вершины w становится максимальное
  количество ключей}
  v^.m:=mv-1;
  {У вершины v - отца w - остается на один
  ключ меньше}
  Dispose(t);
  h:=v^.m<n;
```

```
    {Продолжение процесса балансировки
    зависит от количества ключей в вершине v}
  End
End
Else Begin
{Правого брата нет, тогда приходится
обращаться к левому брату - см. рис. 8.20}
  If s=1 Then t:=v^.p0
    Else t:=v^.E[s-1].child;
  mt:=t^.m+1;
  k:=(mt-n) Shr 1;
  If k>0 Then Begin
  {Расщепление вершины t - см. рис. 8.20б}
    For i:=n-1 DownTo 1 Do w^.E[i+k]:=w^.E[i];
    {Освобождаем место для ключей, заимствуемых
    у вершины t}
    w^.E[k]:=v^.E[s];
    {Один ключ берем у родителя}
    w^.E[k].child:=w^.p0;
    mt:=mt-k;
    For i:=k-1 DownTo 1 Do w^.E[i]:=t^.E[i+mt];
    {Передаем вершине с нехваткой ключей
    ключи от расщепляемой вершины}
    w^.p0:=t^.E[mt].child;
    v^.E[s]:=t^.E[mt];
    {Родителю отдаем один ключ от расщепляемой
    вершины}
    v^.E[s].child:=w;
    t^.m:=mt-1;
    {Корректируем количество ключей
    у расщепляемой вершины}
    w^.m:=n-1+k;
    {Вершине с нехваткой ключей добавлено
    k ключей}
    h:=False;
  End
  Else Begin
  {Слияние с левым братом - см. рис. 8.20а}
    t^.E[mt]:=v^.E[s];
    {Берем ключ у родителя и самое левое
    поддерево у удаляемой вершины}
    t^.E[mt].child:=w^.p0;
```

```
        For i:=1 To n-1 Do t^.E[i+mt]:=w^.E[i];
        {Передаем оставшиеся ключи вершины  w
        вершине t}
        t^.m:=max_2n;
        {У вершины t - максимальное количество ключей}
        v^.m:=mv-1;
        {А у вершины-родителя становится на один
        ключ меньше}
        Dispose(w);
        h:=v^.m<n;
        {Процесс дальнейшей балансировки зависит
        от количества оставшихся ключей у вершины v}
      End
    End
  End;
```

Оценим эффективность использования *Б*-деревьев, основываясь на рассуждении Д. Кнута[1]. Очевидно, что временны́е характеристики операций работы с *Б*-деревом определяются высотой этого дерева. Для оценки высоты найдем, чему равно наименьшее количество ключей в *Б*-дереве порядка n и высотой h. Корень дерева содержит как минимум один ключ. На первом уровне должно быть, как минимум, две вершины с, как минимум, n ключами в каждой. На втором уровне имеется не менее $2(n + 1)$ вершин с, как минимум, n ключами в каждой, т. е. минимально возможное общее количество ключей на втором уровне равно $2n(n + 1)$. На третьем уровне имеется не менее $2(n + 1)(n + 1)$ вершин, а ключей — $2n(n + 1)(n + 1)$. В общем случае вершины на i-м уровне содержат, как минимум, $2n(n+1)^{i-1}$ ключей. Таким образом, для любого *Б*-дерева порядка n с t ключами и высотой $h > 0$ справедливо неравенство:

$$t \geqslant 1 + 2n\sum_{i=1}^{h}(n+1)^{i-1} = 2(1+n)^h - 1.$$

Отсюда

$$h \leqslant \frac{\ln(t+1)}{\ln(2\cdot(n+1))}.$$

Например, при $t = 1999999$ и $n = 199$ получаем, что $h \leqslant 3$.

---

[1] *Кнут Д.* Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. М.: Мир, 1978. С. 566.

Упражнения

1. Приведите пример последовательности чисел из 30 элементов. Изобразите графически процесс вставки этих элементов в *Б*-дерево второго порядка.

2. Вставка элемента 15 в *Б*-дерево, показанное на рис. 8.16*г*, может быть осуществлена так, что вместо дерева, вид которого представлен на рис. 8.17, получится дерево, показанное на рис. 8.21.

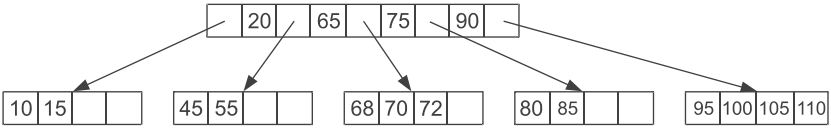


Рис. 8.21. Вид *Б*-дерева после вставки элемента 15

   Разработайте алгоритм вставки в *Б*-дерево, приводящий к такому результату.

3. Пусть ключи 1, 2, 3, ... вставляются в пустое *Б*-дерево порядка 2. Какие ключи вызывают расщепление страниц? Какие ключи вызывают увеличение высоты дерева?

4. В пустое *Б*-дерево порядка 100 последовательно вставляются ключи 1, 2, 3, 4, ... Какие ключи впервые приведут к появлению листьев на 3 и 4 уровнях?

5. Для *Б*-дерева, показанного на рис. 8.21, приведите пример последовательности удаляемых ключей, приводящих это дерево к одной корневой вершине. Изобразите графически процесс удаления каждого элемента.

6. Разработайте программу для работы с *Б*-деревом. Вставка и удаление элементов осуществляются в произвольном порядке.
   Примечание. В этой программе необходимо предусмотреть наглядный вывод исходного и результирующего *Б*-деревьев.

7*. В некоей файловой системе каталог файлов организован в виде сбалансированного *Б*-дерева, где каждая ячейка (`Cell`) соответствует одному файлу. В каждой ячейке содержатся имя и атрибуты соответствующего файла, в том

числе дата последнего обращения к этому файлу, являющаяся ключом. Напишите программу, которая обходит дерево и удаляет все файлы, последнее обращение к которым происходило до определенной даты; при этом сбалансированность дерева должна сохраняться.

8.4. Красно-черные деревья

> Главное — идти. Дорога не кончается, а цель — всегда обман зрения странника: он поднялся на вершину, но ему уже видится другая...
>
> *Антуан де Сент-Экзюпери*

> Истина сложна; нам лишь дано постичь приближение к ней.
>
> *Джон фон Нейман*

Красно-черные деревья — это сбалансированные деревья поиска, высота которых не превосходит $O(\log n)$, что гарантирует эффективность операций поиска, вставки и удаления.

Двоичное дерево поиска называется красно-черным деревом, если оно имеет следующие свойства:

- каждая вершина либо красная, либо черная («цвет» вершины — это дополнительная информация (1 бит), связанная с каждой вершиной дерева);
- каждый лист — черный;
- если вершина красная, то оба ее сына — черные;
- все пути, идущие вниз от корня к листьям, содержат одинаковое количество черных вершин.

Если у вершины отсутствует сын или родитель, то соответствующее поле равно значению `nil`. Для удобства работы с такими вершинами вводятся дополнительные фиктивные вершины-листья дерева, в этом случае вершина становится внутренней. Пример красно-черного дерева приведен на рис. 8.22.

Если у обычного двоичного дерева поиска окрасить все вершины в черный цвет, то оно станет красно-черным деревом, только если оно идеально сбалансировано. В этом слу-

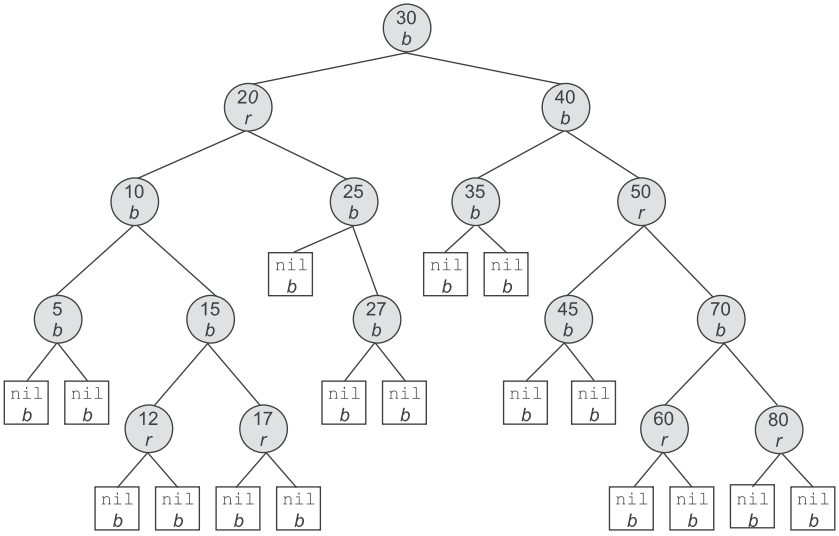


Рис. 8.22. Пример красно-черного дерева. Черные вершины отмечены буквой *b* (*black*), красные — *r* (*red*). Фиктивные вершины обозначены прямоугольниками. Все они, а также корень дерева всегда имеют черный цвет

чае все пути от корня содержат одинаковое количество черных вершин — т. е. выполняется четвертое свойство. Третье же свойство снимает условие жесткой идеальной сбалансированности, но не позволяет (в сочетании с четвертым свойством) дереву «выродиться» в линейный список.

Для любой внутренней вершины `a` ее черной высотой (`bh(a)`) мы назовем количество черных вершин на пути от нее до листа (при этом сама вершина `a` не учитывается). Согласно четвертому свойству, эта высота не зависит от выбора пути. Черная высота всего дерева определяется черной высотой его корня.

Утверждение. Красно-черное дерево с n внутренними вершинами имеет высоту не более $2\log_2(n+1)$. При этом поддерево с корнем `a` содержит по крайней мере $2^{\mathtt{bh(a)}}-1$ внутренних вершин.

Для листьев это утверждение выполняется. Сделаем индуктивное предположение и рассмотрим вершину с черной высотой `k`. Оба ее сына имеют черную высоту не менее

k–1 — красный k, а черный — k–1. По предположению, поддеревья с корнями в этих сыновьях имеют не менее $2^{k-1}-1$ вершин. Тогда общее количество внутренних вершин равно $2^{k-1}- 1 + 2^{k-1} - 1 + 1 = 2^k - 1$.

Пусть h — это высота дерева. Согласно третьему свойству, по меньшей мере половина всех вершин на пути от корня к листу являются черными (сам корень не считается), значит, $bh \geqslant h/2$. Отсюда следует, что $n \leqslant 2^{h/2} - 1$ или, после логарифмирования, — $h \leqslant 2log_2(n + 1)$.

Определим следующее описание вершины красно-черного дерева:

```
Const red=True;
      black=False;
Type pt=^node;
     node=Record
       color: Boolean;
       {Цвет вершины, True - красный, False - черный}
       key:Integer;
       {Значение ключа}
       left,right,parent:pt;
       {parent - ссылка на родителя}
     End;
Var root:pt;
    {Указатель на корень дерева}
    nul:pt;
    {Указатель на фиктивную вершину. Одна
    фиктивная вершина заменяет все листья дерева.
    Другими словами, все указатели nil в красно-
    черном дереве заменяются одним значением
    указателя на фиктивную вершину}
```

Инициализация пустого красно-черного дерева производится с помощью процедуры Init:

```
Procedure Init(Var root,nul:pt);
  Begin
    New(nul);
    With nul^ Do Begin
      left:=nil;
      right:=nil;
```

```
      parent:=nil;
      key:=0;
      color:=black;
    End;
    root:=nul;
End;
```

Балансировка красно-черного дерева, т. е. восстановление его свойств при выполнении операций вставки (`Insert`) и удаления (`Delete`) элемента требует, как мы увидим чуть позже, выполнения двух типов поворотов (`Turn`) — левого и правого. Эти повороты сохраняют свойство упорядоченности и изменяют только значения указателей для ряда вершин.

На рис. 8.23*а* показан левый поворот относительно вершины с указателем `a`. Правый сын вершины (указатель `b`) не является листом и в результате такого поворота занимает место отца. Левый сын `b` становится в результате поворота правым сыном `a`, и свойство упорядоченности сохраняется.

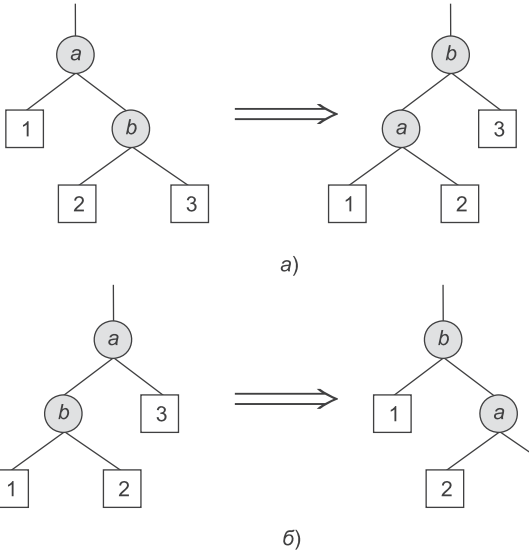


Рис. 8.23. Два типа поворотов: а) левый; б) правый. Буквами `a` и `b` обозначены не значения ключей, а указатели на соответствующие вершины. Прямоугольники с цифрами обозначают поддеревья

```
Procedure LeftTurn(Var root:pt; a:pt);
  Var b:pt;
  Begin
    b:=a^.right;
    a^.right:=b^.left;
    {Левый сын b становится правым сыном a}
    If b^.left<>nul Then b^.left^.parent:=a;
    b^.parent:=a^.parent;
    {Корректируем ссылку на родителя в поддереве,
    обозначенном на рис. 8.23а цифрой 2}
    If a^.parent=nul Then root:=b
    {Если вершина с указателем a является корнем
    дерева, то в результате поворота правый сын a
    должен стать корнем дерева}
      Else {"Подцепляем" b к родителю a}
        If a=a^.parent^.left Then a^.parent^.left:=b
          Else a^.parent^.right:=b;
    b^.left:=a;
    {Вершина a становится левым сыном b}
    a^.parent:=b;
  End;
```

Аналогично, только в обратном направлении выполняется и правый поворот — см. рис. 8.23*б*.

```
Procedure RightTurn(Var root:pt; a:pt);
  Var b:pt;
  Begin
    b:=a^.left;
    a^.left:=b^.right;
    {Правый сын b становится левым сыном a}
    If b^.right<>nul Then b^.right^.parent:=a;
    b^.parent:=a^.parent;
    {Корректируем ссылку на родителя в поддереве,
    обозначенном на рис. 8.23б цифрой 2}
    If a^.parent=nul Then root:=b
    {Если вершина с указателем a является корнем
    дерева, то в результате поворота левый сын a
    должен стать корнем дерева}
      Else {"Подцепляем" b к родителю a}
        If a=a^.parent^.right Then
           a^.parent^.right:=b
          Else a^.parent^.left:=b;
```

```
    b^.right:=a;
    {Вершина a становится правым сыном b}
    a^.parent:=b;
  End;
```

Перейдем теперь к обсуждению вставки элемента в красно-черное дерево. Логически она разбивается на два шага. На первом из них элемент обычным способом вставляется в двоичное дерево поиска (см. раздел 5.2) и вершина окрашивается в красный цвет — процедура TreeInsert. На втором же шаге восстанавливаются (в случае их нарушения) свойства красно-черного дерева.

Напомним суть процедуры TreeInsert, тем более что у нее есть некоторые отличия от процедуры, приведенной в разделе 5.2. Входными параметрами здесь являются указатели на корень дерева и на вставляемый элемент; второй указатель формируется с помощью функции NewEl:

```
Function NewEl(el:Integer; nul:pt; p:Boolean):pt;
  Var t:pt;
  Begin
    New(t);
    With t^ Do Begin
      left:=nul;
      right:=nul;
      key:=el;
      parent:=nul;
      color:=p;
    End;
    NewEl:=t;
  End;
```

В обсуждаемой процедуре вставки отыскивается место для вставляемого элемента в иерархии дерева и осуществляется стандартная работа с указателями вершин по увязыванию дерева в единое целое:

```
Procedure TreeInsert(Var root:pt; c:pt);
  Var a,b:pt;
  Begin
    b:=nul;
    a:=root;
```

```
  While a<>nul Do Begin
  {Поиск места для новой вершины в иерархии
  дерева (нерекурсивный вариант реализации)}
    b:=a;
    {В вершине b после изменения ее значения
    окажется отец вершины a}
    If c^.key<a^.key Then a:=a^.left
      Else a:=a^.right;
  End;
  c^.parent:=b;
  If b=nul Then root:=c
  {Вставка первого элемента в дерево}
    Else
      If c^.key<b^.key Then b^.left:=c
        Else b^.right:=c;
End;
```

Вставка элемента с помощью процедуры `TreeInsert` не влияет ни на какие свойства красно-черного дерева, кроме третьего: у новой красной вершины может оказаться красный родитель (отец). В этой ситуации (а их может быть несколько в зависимости от цвета других соседних вершин) требуется «перекраска» вершин и выполнение поворотов.

Первый возможный случай представлен на рис. 8.24. Новая (вставленная) вершина здесь имеет указатель `x`, и оказывается, что не только отец вершины (`x^.parent^`), но и брат отца («дядя») имеет красный цвет. Отец вершины может быть как правым сыном своего отца, тогда «дядя» вставленной вершины — его левый брат (см. рис. 8.24*а*), так и левым сыном своего отца («дядя» — правый брат — см. рис. 8.24*б*). Это различие фиксируется значением логической переменной `t` и, соответственно, определяет значение указателя на дядю вершины (`y`).

Что следует в этом случае предпринять? Надо изменить цвет вершин (отца и «дяди») на черный, а цвет вершины-«деда» — на красный, и перейти вверх по дереву к вершине-«деду». Далее этот процесс нужно продолжать до тех пор, пока мы не достигнем корня или пока балансировка дерева не будет восстановлена. Такая «перекраска» не нарушает четвертого свойства дерева (количество черных вершин на всех путях от корня до листьев — одно и то же).

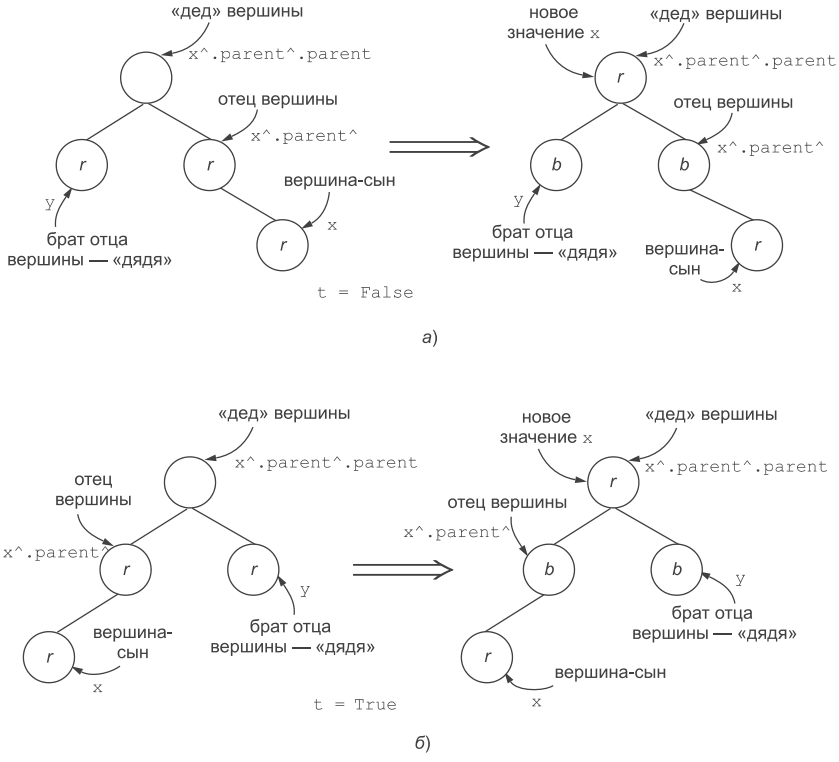


Рис. 8.24. Отец и «дядя» вставленной вершины имеют красный цвет

Во-первых, хотя это не показано на рис. 8.24, вершина-«дед» имела черный цвет, а во-вторых, мы предполагаем, что корень дерева всегда имеет черный цвет.

Второй возможный случай заключается в том, что цвет вершины-«дяди» — черный. На рис. 8.25 показан вариант, когда «дядя» приходится правым братом отца, а на рис. 8.26 — когда он приходится левым братом отца. (На обоих рисунках мы специально сохраняем названия вершин: сын, отец, «дед», «дядя», а не меняем их в соответствии с логикой выполнения поворотов. Прямоугольниками здесь обозначены поддеревья и их положение после поворотов.)

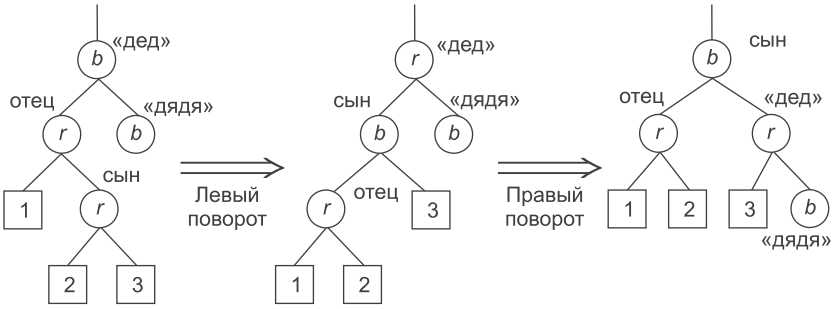


Рис. 8.25. «Дядя» — правый брат отца

Если вершина (указатель x) — это правый сын (рис. 8.25), то после выполнения левого поворота он (сын) становится левым сыном своего отца. Этот поворот не делается, если вершина уже была левым сыном. Затем вершины — сын и «дед» — перекрашиваются и выполняется правый поворот относительно вершины-«деда». Тогда третье свойство красно-черного дерева становится истинным, а четвертое свойство (количество черных вершин на путях от корня до листьев) сохраняется. Таким образом, балансировка дерева восстановлена, и дальнейший просмотр вершин вверх по дереву уже не нужен.

Ситуация, когда «дядя» приходится левым братом отца, аналогична показанной ранее (рис. 8.26). Правым поворотом (если он требуется — когда вершина x — это левый сын отца), мы подготавливаем участок (фрагмент) дерева к левому повороту, до выполнения которого следует «перекрасить» вершины — сына и «деда».

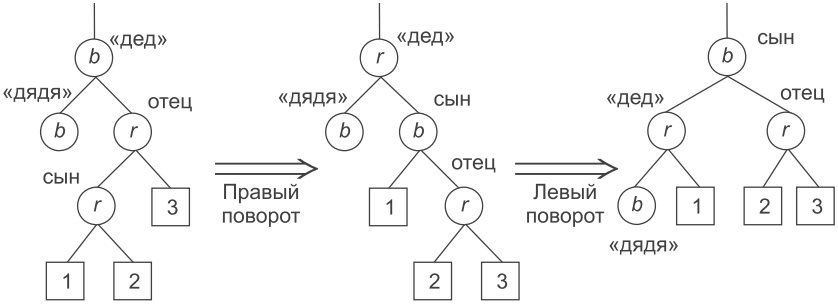


Рис. 8.26. «Дядя» — левый брат отца

Формализованная запись этих действий имеет вид:

```
Procedure Insert(Var root:pt; x:pt);
{x – указатель на вставляемый элемент}
  Var y:pt;
      t:Boolean;
  Begin
    TreeInsert(root,x);
    {Вставка элемента в двоичное дерево поиска}
    x^.color=red;
    {Вставляем только красные вершины}
  {Восстановление свойств красно-черного дерева}
    While (x<>root) And (x^.parent^.color=red)
          Do Begin
    {Идем вверх по дереву, пока не достигнем
    корня; цвет отца вершины - красный}
      t:=(x^.parent=x^.parent^.parent^.left);
      {Переменная для различения симметричных
      случаев. У красно-черных деревьев корень
      всегда черный, а так как, по условию цикла,
      x^.parent^.color=red, то x^.parent - не корень
      и x^.parent^.parent^.left существует
      (см. рис. 8.24)}
      If t Then y:=x^.parent^.parent^.right
      {"Дядя" – правый брат отца}
        Else y:=x^.parent^.parent^.left;
        {"Дядя" – левый брат отца}
      If (y<>nul) And (y^.color=red) Then Begin
      {1-й случай: "дядя" x имеет красный цвет}
        x^.parent^.color:=black;
        {"Перекрашиваем" вершины}
        y^.color:=black;
        x^.parent^.parent^.color:=red;
        x:=x^.parent^.parent;
        {Переадресация по x}
      End
      Else Begin
      {2-й и 3-й случаи: "дядя" вершины x имеет
      черный цвет}
        If t Then Begin
        {"Дядя" – правый или левый брат отца?}
```

```
        If x=x^.parent^.right Then Begin
        {"Дядя" - правый брат отца, а вершина
        x - правый сын отца - см. рис. 8.25}
          x:=x^.parent;
          {Выполняем левый поворот относительно
          отца вершины - см. рис. 8.25}
          LeftTurn(root,x)
        End;
      End
    Else Begin
    {"Дядя" - левый брат отца}
      If x=x^.parent^.left Then Begin
      {Вершина x - левый сын отца -
      см. рис. 8.26}
        x:=x^.parent;
        RightTurn(root,x);
        {Выполняем правый поворот относительно
        отца вершины - см. рис. 8.26}
      End;
    End;
    x^.parent^.color:=black;
    {Изменяем цвет у отца. Если были выполнены
    повороты, то сын стал отцом, а отец -
    сыном. Значения указателей изменены
    в процессе поворотов}
    x^.parent^.parent^.color:=red;
    {Изменяем цвет у "деда"}
    If t Then RightTurn(root,x^.parent^.parent)
    {Второй поворот - см. рис. 8.25}
      Else LeftTurn(root,x^.parent^.parent);
      {Второй поворот - см. рис. 8.26}
    End;
  End; {While}
  {Корень, по соглашению - черный}
  root^.color:=black;
End;
```

Если в дереве n вершин, то его высота равна $O(\log_2 n)$, поэтому время выполнения вставки элемента в красно-черное дерево составляет $O(\log_2 n)$, так как цикл повторяется только в первом случае. При этом x сдвигается вверх по дереву, а эта операция выполняется не более $O(\log_2 n)$ раз.

Перейдем к анализу операции удаления элемента из красно-черного дерева. Вначале приведем достаточно очевидные факты. В вызывающей логике вводится удаляемый элемент, отыскивается его место в дереве, выполняются необходимые действия по удалению с сохранением балансировки, а затем если элемент есть в дереве, то происходит освобождение памяти, выделенной ранее для его хранения. Это можно записать так:

```
...
ReadLn(el);
t:=Delete(root,Find(root,el));
If t<>nul Then Dispose(t);
...
```

Отыскание элемента в дереве и его удаление с сохранением балансировки у нас разнесены в отдельные функции.

Первая — очевидна и не нуждается в разъяснениях:

```
Function Find(t:pt; el:Integer):pt;
  Begin
    While (t<>nul) And (t^.key<>el) Do
      If el<t^.key Then t:=t^.left
        Else t:=t^.right;
    Find:=t;
  End;
```

Со второй функцией дело обстоит несколько сложнее. Прежде чем перейти к ней, рассмотрим вспомогательную функцию поиска следующего элемента в дереве (аналогичная задача для двоичного дерева поиска была нами рассмотрена в упражнении 10 раздела 5.2, но в данном случае она более простая), когда точно известно, что у данной вершины есть правое поддерево. Эта функция получает указатель на вершину дерева (a), т. е. известно значение ключа (key), и возвращает значение указателя на вершину дерева (b), имеющую следующее по порядку значение ключа. Все действия сводятся к нахождению самого левого элемента (не имеющего левого сына) в правом поддереве вершины.

Формализованная запись логики:

```
Function TreeSuc(a:pt):pt;
{Находим следующий за a^.key элемент дерева. Функция
используется, только если у вершины a есть правое
поддерево. Это - неполный аналог упражнения 10 из
раздела 5.2, так как находить предка с наиболее
близким значением ключа в данном случае не
требуется}
  Var b:pt;
  Begin
    b:=a^.right;
    While b^.left<>nul Do b:=b^.left;
    TreeSuc:=b;
  End;
```

Пусть удаляемый элемент найден в дереве (z). Тогда возможны два случая (см. рис. 8.27): когда у вершины имеется полный комплект сыновей (два — см. рис. 8.27*а*) или когда у вершины имеется неполный комплект сыновей (только один или они вообще отсутствуют — см. рис. 8.27*б*). Обозначим указатель на фактически удаляемую вершину дерева как y. В первой ситуации — это самая левая вершина в правом под-

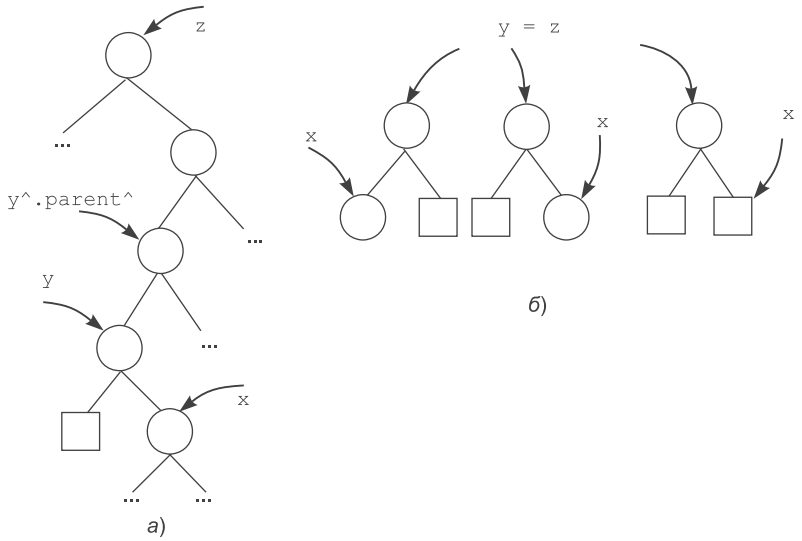


Рис. 8.27. Удаление элемента из дерева

дереве (рис. 8.27*а*), найденная с помощью функции TreeSuc, а во второй ситуации (рис. 8.27*б*) — y совпадает с z.

Обозначим указатель на сына вершины y как x. Эта вершина (x) может быть в том числе и фиктивной (они обозначены на рис. 8.27 прямоугольниками). После того как выяснен тип ситуации и определен сын, нам остается только выполнить корректировку значений указателей в задействованных вершинах и переписать данные из вершины y в вершину z. А как быть с балансировкой дерева? Вынесем ее, как и раньше, в отдельную процедуру, которая требуется только при удалении черной вершины. При удалении же красной вершины свойства дерева не нарушаются (красные вершины не могут стать соседними, так что значение черных высот вершин дерева сохраняется).

```
Function Delete(Var root:pt; z:pt):pt;
  Var y,x:pt;
  Begin
    If z<>nul Then Begin
    {y - указатель на фактически удаляемую вершину}
      If (z^.left=nul) Or (z^.right=nul) Then y:=z
      {У вершины есть хотя бы один сын или оба -
      фиктивные вершины (см. рис. 8.27б)}
        Else y:=TreeSuc(z);
        {Находим вершину со следующим значением
        ключа (рис. 8.27а). У вершины z - два сына.
        Вершина y имеет не более одного сына}
      If y^.left<>nul Then x:=y^.left
        Else x:=y^.right;
        {x - сын удаляемой вершины y (рис. 8.27)}
      x^.parent:=y^.parent;
      {Отцом сына становится отец удаляемой
      вершины - "дед"}
      If y^.parent=nul Then root:=x
      {Удаляемая вершина - корень дерева}
        Else Begin
        {Левым или правым сыном является удаляемая
        вершина?}
          If y=y^.parent^.left Then
                                  y^.parent^.left:=x
            Else y^.parent^.right:=x;
        End;
```

```
    If y<>z Then z^.key:=y^.key;
    {Переписываем данные из вершины y в вершину z}
    If y^.color=black Then Balance(root,x);
    {Балансировку дерева следует проверить,
    только если удаляемая вершина имеет
    черный цвет}
    Delete:=y;
  End
  Else Delete:=z;
End;
```

Таким образом, самая сложная часть логики удаления элемента из красно-черного дерева — это восстановление балансировки. Если удалена черная вершина, вследствие чего любой путь, проходящий через нее, содержит на одну черную вершину меньше, то нарушено четвертое свойство в определении красно-черного дерева. Тогда нашей задачей является поиск такого способа добавления одной черной вершины в поддерево, содержащее сына удаленной вершины (x), чтобы все пути в этом поддереве имели одинаковое количество черных вершин.

Если единственный сын удаленной вершины (x) имеет красный цвет, то все просто: цвет вершины x изменить на черный, и свойства дерева будут восстановлены.

Однако если цвет вершины x — черный, то единственный путь восстановления балансировки — это заимствование черной вершины в поддереве, корнем которого является брат (w) вершины x.

Первый возможный случай заключается в том, что цвет брата — красный — см. рис. 8.28. С помощью поворотов относительно отца (x^.parent) у вершины x появляется новый брат черного цвета. Балансировка дерева при этом не восстанавливается (так как поддерево с корнем в x по-прежнему имеет меньшую черную высоту), но сводится к ситуации, когда братья (x и w) имеют один черный цвет.

Для дальнейшего анализа нам недостаточно знания, что цвет брата вершины — черный. Приходится привлекать сыновей вершины w.

На рис. 8.29 представлен случай, когда сыновья брата имеют черный цвет. Если «перекрасить» w в красный цвет, то черные высоты поддеревьев с корнями x и w совпадут

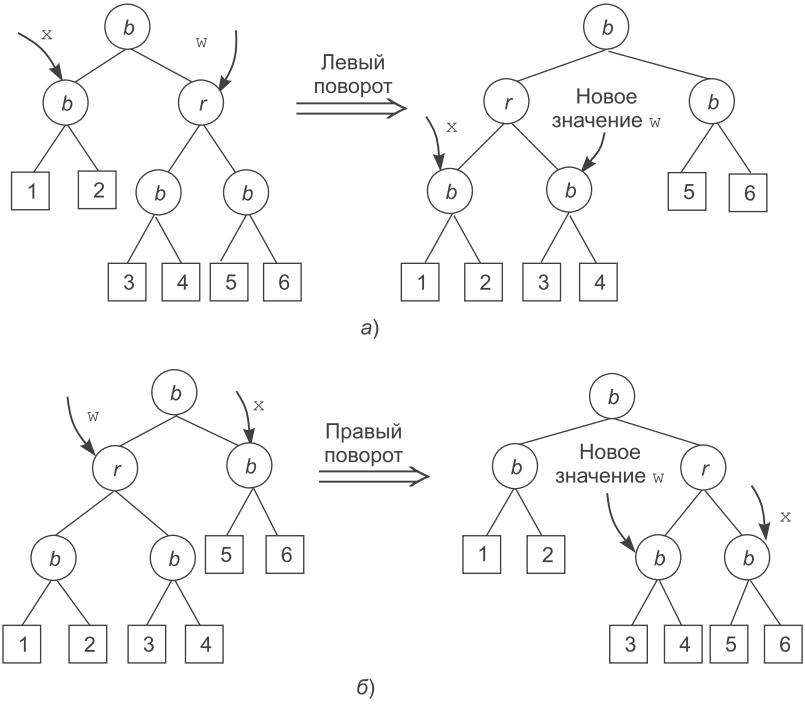


Рис. 8.28. Брат вершины x имеет красный цвет

(у x высота была на единицу меньше, а мы убрали единицу у w). Тогда проблема балансировки «продвигается вверх» по дереву — к отцу вершин x и w, так как черная высота отца уменьшилась на единицу.

Далее рассмотрим ситуацию, когда один из сыновей (или оба) w имеет красный цвет.

Выделим следующие случаи: 1) брат x является правым сыном отца, а его (брата) левый сын имеет заведомо красный цвет; 2) брат x является левым сыном отца, а его правый сын имеет красный цвет.

С помощью соответствующих поворотов оба выделенных случая сводятся к ситуации, когда заведомо известно, что правый сын брата или его левый сын имеют красный цвет; при этом, конечно же, вершина x получает нового

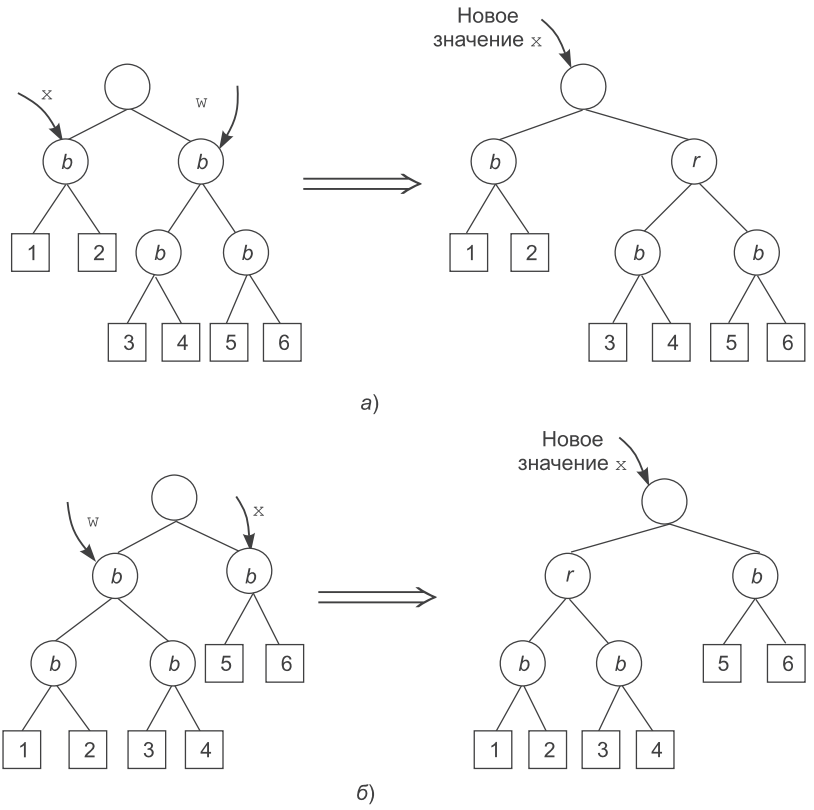


Рис. 8.29. Сыновья вершины w имеют черный цвет. Вершина без меток *b* или *r* может иметь любой цвет

брата. После такого преобразования мы получаем стандартную ситуацию: братья x и w — черного цвета, а правый (или, соответственно, левый) сын w имеет красный цвет — см. рис. 8.31.

Оказывается, что в этом случае, выполнив один поворот и изменив цвет ряда вершин, можно восстановить балансировку дерева. На рис. 8.31*а* показан поворот и «перекраска» вершин для ситуации, когда w является правым братом x, а на рис. 8.31*б* — когда w является левым братом x. На рис. 8.31 введены условные метки вершин. Так, в первом

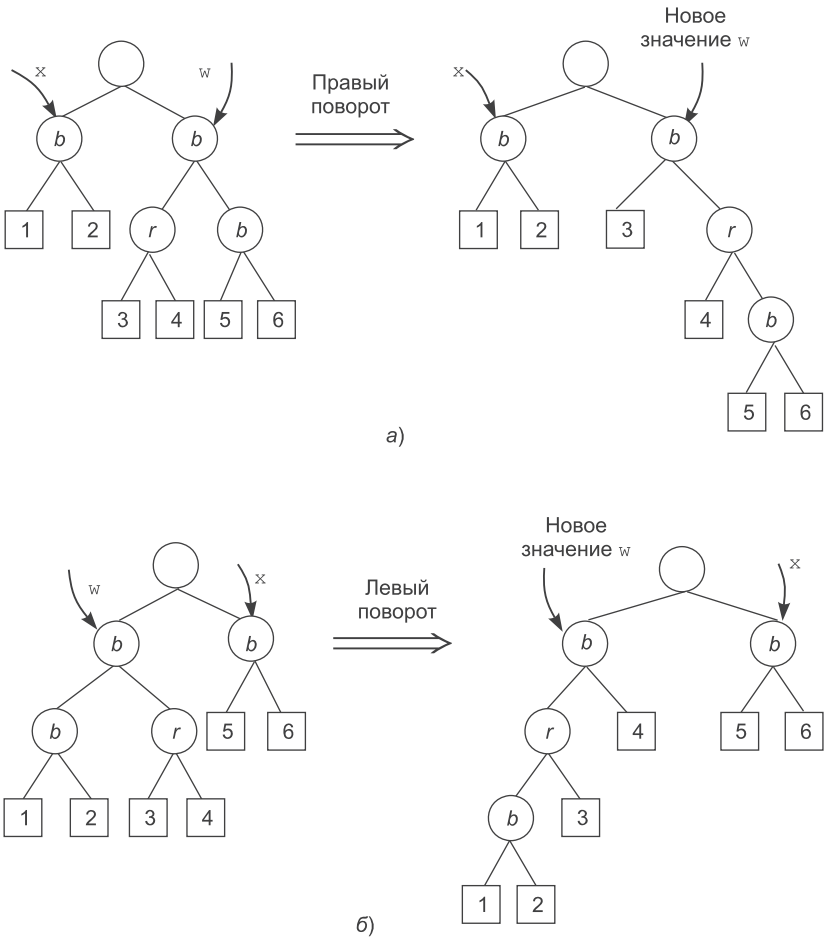
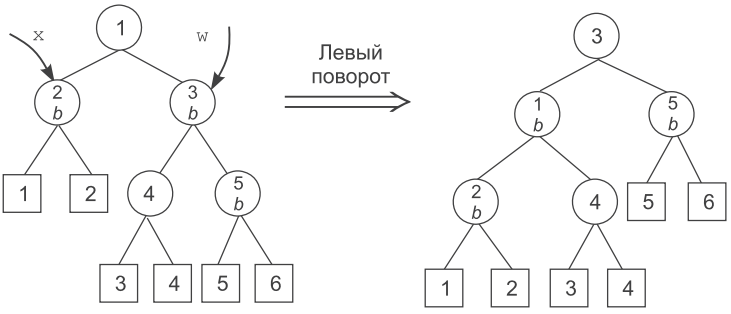
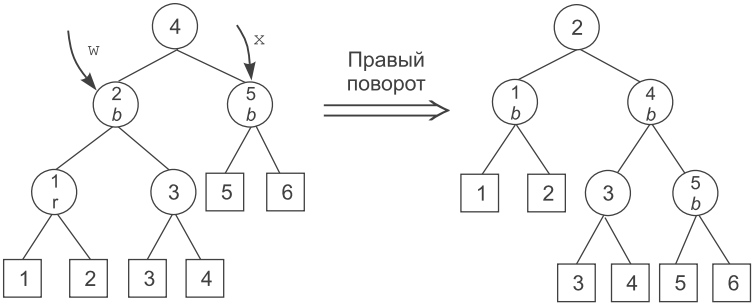


Рис. 8.30. Один из сыновей вершины w имеет черный цвет: а) правый сын; б) левый сын

варианте корнем рассматриваемого поддерева становится вершина с меткой 3, но главное в том, что в левое поддерево (там, где находится x) добавляется одна черная вершина; при этом черная высота правого поддерева остается прежней. Балансировка дерева восстановлена, так что двигаться дальше вверх по дереву не имеет смысла.



а)



б)

Рис. 8.31. У брата `w` вершины `x` один из сыновей имеет красный цвет — значит, есть возможность восстановить балансировку. Вершинам присвоены условные метки 1, 2, 3, 4 и 5, чтобы показать их положение в дереве и цвет после поворотов. Цвет вершин без меток *b* и *r* сохраняется. Так, вершина с меткой 3 (*а*) имеет цвет вершины с меткой 1

Формализованная запись восстановления балансировки красно-черного дерева после удаления элемента имеет вид:

```
Procedure Balance(Var root:pt; x:pt);
  Var w:pt;
  Begin
    While (x<>root) And (x^.color=black) Do Begin
    {Идем вверх по дереву}
      If x=x^.parent^.left Then Begin
```

```
{Вершина x является левым сыном -
см. рис. 8.28a}
  w:=x^.parent^.right;
  {Указатель на правого брата вершины}
  If w^.color=red Then Begin
  {Брат красного цвета - тогда отец вершин
  x и w имеет черный цвет}
    w^.color:=black;
    {Изменяем цвета}
    x^.parent^.color:=red;
    LeftTurn(root,x^.parent);
    {Левый поворот относительно отца вершин}
    w:=x^.parent^.right;
    {Мы пришли к случаю, когда пара вершин
    с указателями x и w имеют черный цвет}
  End;
  If (w^.left^.color=black) And
     (w^.right^.color=black) Then Begin
     {Оба сына вершины w имеют черный цвет -
     см. рис. 8.29a}
    w^.color:=red; {Перекрашиваем вершину w}
    x:=x^.parent; {Переходим вверх по дереву}
  End
Else Begin
  If w^.right^.color=black Then Begin
  {Правый сын вершины w имеет черный цвет -
  см. рис. 8.30a}
    w^.left^.color:=black;
    {"Перекрашиваем" левого сына вершины w}
    w^.color:=red;
    {"Перекрашиваем" w}
    RightTurn(root,w);
    {Правый поворот относительно вершины w}
    w:=x^.parent^.right;
    {Правым братом x стала другая вершина}
  End;
{На рис. 8.31a проиллюстрированы производимые
далее действия}
  w^.color:=x^.parent^.color;
  {Брату присваиваем цвет отца}
  x^.parent^.color:=black;
```

```
    {Вершина-отец получает черный цвет}
    w^.right^.color:=black;
    {Правый сын брата x получает черный
    цвет}
    LeftTurn(root,x^.parent);
    {Левый поворот относительно отца вершины x}
    x:=root;
    {Выходим из цикла (искусственный выход) -
    балансировка восстановлена}
    End;
  End
  Else Begin
    w:=x^.parent^.left;
    {x - правый сын своего отца, а w - его
    левый брат - рис. 8.28б}
    If w^.color=red Then Begin
    {У брата красный цвет}
      w^.color:=black;
      {"Перекрашиваем" брата в черный цвет,
      а отца - в красный цвет}
      x^.parent^.color:=red;
      RightTurn(root,x^.parent);
      {Выполняем правый поворот относительно
      отца вершины x}
      w:=x^.parent^.left;
      {Пришли к случаю, когда пара вершин
      с указателями x и w имеет черный цвет}
    End;
    If (w^.left^.color=black) And
       (w^.right^.color=black) Then Begin
       {Оба сына вершины w имеют черный цвет -
       см. рис. 8.29б}
       w^.color:=red;
       {"Перекрашиваем" вершину w}
       x:=x^.parent;
       {Переходим вверх по дереву}
    End
  Else Begin
    If w^.left^.color=black Then Begin
    {Левый сын w имеет черный цвет -
    см. рис. 8.30б}
```

```
          w^.right^.color:=black;
          {Изменяем цвета вершин}
          w^.color:=red;
          LeftTurn(root,w);
          {Выполняем левый поворот относительно
          вершины w}
          w:=x^.parent^.left;
          {Левым братом вершины x стала другая
          вершина}
        End;
      {На рис. 8.31б проиллюстрированы производимые
      далее действия}
        w^.color:=x^.parent^.color;
        {Брат получает цвет отца}
        x^.parent^.color:=black;
        {Отец получает черный цвет}
        w^.left^.color:=black;
        {Левый сын брата получает черный цвет}
        RightTurn(root,x^.parent);
        {Правый поворот относительно отца вершины}
        x:=root;
        {Выходим из цикла - балансировка дерева
        восстановлена}
        End;
      End;
    End; {While}
    x^.color:=black;
End;
```

Высота красно-черного дерева с n вершинами есть $O(\log_2 n)$. Время выполнения операции удаления без учета процедуры `Balance` пропорционально $O(\log_2 n)$. В процедуре `Balance` есть цикл. В случае, представленном на рис. 8.28, мы переходим к ситуации, показанной на рис. 8.29, но отец вершин `x` и `w` имеет красный цвет, и цикл заканчивается. Случаи, показанные на рис. 8.30 и 8.31, приводят к завершению цикла и выполняются за константное время (выполняется, самое большее, три поворота). Если братья `x` и `w` имеют черный цвет, а сыновья `w` также черного цвета (рис. 8.29), то поворотов не производится, а осуществляется изменение цвета брата `w` и переход вверх по дереву. Таких переходов —

конечное количество, так что общее время работы процедуры удаления пропорционально $O(\log_2 n)$.

Упражнения

1. Изобразите графически полное двоичное дерево поиска с ключами 1, 2, 3, ..., 31. Его высота равна 4. Добавьте фиктивные вершины. «Покрасьте» вершины различными способами так, чтобы у получающихся красно-черных деревьев черная высота была равна 2, 3, 4 или 5. Убедитесь, что первое значение (2) возможно, только если корень дерева имеет красный цвет.

2. Для красно-черного дерева с n ключами можно определить отношение количества красных внутренних вершин к количеству черных внутренних вершин. Вычислите это отношение для каждого из красно-черных деревьев из упражнения 1. Приведите примеры красно-черных деревьев с минимальным и максимальным значениями этого отношения.

3. Изобразите графически результат выполнения левого поворота дерева, показанного на рис. 8.22, относительно вершины с ключом 40.

4. Изобразите графически результат выполнения правого поворота дерева, показаноого на рис. 8.22, относительно вершины с ключом 20.

5. Пусть `v`, `w` и `t` — произвольные вершины в поддеревьях, отмеченных на рис. 8.23 цифрами 1, 2 и 3, соответственно. Как изменится высота (количество вершин до корня) вершин `v`, `w` и `t` в результате выполнения левого (правого) поворота?

6. Приведите пример произвольного двоичного дерева поиска с n вершинами. Вычислите минимальное количество поворотов, требуемых для его преобразования в линейный список.

7. Продемонстрируйте на примере, что произвольное двоичное дерево поиска с n ключами можно преобразовать в любое другое дерево с тем же количеством вершин за $O(n)$ поворотов.

8. Обязательными операциями при вставке элемента в красно-черное дерево является «покраска» корня в черный цвет, а вставляемой вершины — в красный цвет. Почему? Можно ли сделать наоборот?

9. Приведите пример произвольной последовательности из 15 целых чисел. Изобразите графически красно-черные деревья, получающиеся при их последовательной вставке в пустое дерево.

10. Задайте черную высоту поддеревьев 1, 2 и 3, показанных на рис. 8.25 и 8.26. Определите черную высоту всех вершин, задействованных в поворотах на рис. 8.25 и 8.26.

11. В упражнении 9 было построено красно-черное дерево. Изобразите графически (при каждом удалении) красно-черные деревья, получающиеся в процессе удаления из него восьми элементов.

12. Какой цвет остается у корня красно-черного дерева после удаления из него элемента?

13. На рис. 8.28–8.31 поддеревья обозначены прямоугольниками с метками. Подсчитайте количества черных вершин от корня поддеревьев, изображенных на рисунках, до каждого из поддеревьев. Проверьте, что эти значения не изменяются при преобразованиях.

14. В красно-черное дерево вставлена вершина, а затем она же удалена. Совпадает ли полученное дерево с исходным?

15*. Разработайте программы для работы с красно-черным деревом. Вставка и удаление элементов осуществляются в случайном порядке.
Примечание. Предусмотрите наглядный вывод красно-черного дерева после каждой операции.

Методические комментарии

Балансировка двоичного дерева поиска как идея сохранения его структуры, обеспечивающей эффективный поиск, вставку и удаление элементов, впервые предложена в 1962 г. советскими математиками Г. М. Адельсоном-Вельским и Е. М. Ландисом.

Наиболее детальное изложение работы с АВЛ-деревом можно найти, вероятно, только в книге Н. Вирта[1]. Остальные авторы ограничиваются только описанием идеи, не опускаясь до «прорисовки» особенностей реализации и указывая только на необходимость выполнения четырех типов поворотов. Однако изложение этой темы у Н. Вирта хотя и полное, но достаточно сложное. Для каждого из четырех типов поворотов (малый левый, большой левый, малый правый и большой правый) предлагается свой программный код, причем он различен при вставках и удалениях элементов. Логика работы с полем `bal`, фиксирующим разность высот поддеревьев, при этом скрыта, и ее приходится извлекать из программного кода. В одной из книг[2] автором была сделана попытка другого изложения этой темы, но и оно с трудом объясняется в аудитории. В данной книге излагается новый вариант изложения темы, более приемлемый как для самостоятельного изучения, так и для проведения занятий.

Развитием идеи балансировки являются «2–3»-деревья, предложенные Дж. Хопкрофтом (J. E. Hopcroft) в 1970 г. Балансировка «2–3»-дерева поддерживается за счет изменения степеней его вершин. Наиболее полное описание этого материала можно найти в работе А. Ахо, Дж. Хопкрофта и Дж. Ульмана[3].

Б-деревья как обобщение «2–3»-деревьев разработаны Р. Байером (R. Bayer) и Э. Мак-Грейтом (E. McGreight) в 1972 г. Материал по этой теме можно найти в книге Н. Вирта[4] (но в ней, как обычно, особенности выполнения операций приходится извлекать из программного кода) или Д. Кнута[5].

---

1) *Вирт Н.* Алгоритмы+структуры данных=программы. М.: Мир, 1985. С. 248–260.

2) *Окулов С. М.* Основы программирования. 2-е изд. М.: БИНОМ. Лаборатория знаний, 2005. С. 413–427.

3) *Ахо А., Хопкрофт Дж., Ульман Дж.* Постоение и анализ алгоритмов. М.: Мир, 1979. С. 168–175.

4) *Вирт Н.* Алгоритмы+структуры данных=программы. М.: Мир, 1985. С. 278–295.

5) *Кнут Д.* Искусство программирования. Т. 3. Сортировка и поиск. М.: Мир, 1978. С. 563–571.

Правда, рекомендация обратиться к Д. Кнуту — скорее дань традиции, чем действительная необходимость.

Что же касается красно-черных деревьев, то лучшее и, вероятно, единственное изложение такого материала на русском языке приведено в книге Т. Кормена, Ч. Лейзерсона и Р. Ривеста[1)].

[1)] *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 254–270.

Минимальные системные требования определяются соответствующими требованиями программ Adobe Reader версии не ниже 11-й либо Adobe Digital Editions версии не ниже 4.5 для платформ Windows, Mac OS, Android и iOS; экран 10"

Учебное электронное издание

Серия: «Развитие интеллекта школьников»

Окулов Станислав Михайлович

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

Ведущий редактор *Д. Усенков*
Художник *С. Инфантэ*
Технический редактор *Е. Денюкова*
Корректор *Е. Клитина*
Компьютерная верстка: *В. Носенко*

Подписано к использованию 24.03.20.
Формат 125×200 мм

Издательство «Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499)157-5272
e-mail: info@pilotLZ.ru, http://www.pilotLZ.ru

РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ

Абстракция, абстрагирование – одна из составляющих мыслительного процесса творческой личности. Для развития этого компонента мышления в процессе обучения информатике есть дополнительные возможности, так как знание абстрактных типов данных, умение оперировать ими – это необходимый элемент профессиональной культуры специалиста, связанного с разработкой программных комплексов.

Книга предназначена для школьников, преподавателей информатики и студентов младших курсов университетов. Она может быть использована как в обычных школах при проведении факультативных занятий, так и в образовательных учреждениях с углубленным изучением информатики.

ОКУЛОВ Станислав Михайлович

Декан факультета информатики Вятского государственного гуманитарного университета, кандидат технических наук, доктор педагогических наук, профессор. Автор 9 изобретений и автор (соавтор) 14 книг по информатике для школьников и студентов.

Область интересов: развитие интеллектуальных способностей школьника при активном изучении информатики, исследование ассоциативных систем обработки информации.

С 1993 по 2003 год деятельность в вузе совмещал с работой учителя информатики. За это время его ученики отмечены 33 дипломами (1-й и 2-й степени) на Российских олимпиадах школьников по информатике; трое из них представляли Россию на международных олимпиадах.