© www.phys.nsu.ru

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Физический факультет
Кафедра автоматизации физико-технических исследований

О. Ю. Дашевский

Основы программного конструирования

Учебное пособие

Новосибирск
2009

© www.phys.nsu.ru

УДК ???.? (???):?? (?/?)
Д ????????

Дашевский О. Ю. Основы программного конструирования: Учеб. пособие / Новосибирский гос. ун-т. Новосибирск, 2009. 131 с.

ISBN 978-?-?????-???-?

Пособие посвящено двум «священным коровам» программирования: алгоритмам и структурам данных.

Подробно рассматриваются динамические структуры данных: массивы, списки, деревья, хеш-таблицы. Обсуждаются различные варианты их реализации, приводится готовый код на языке C.

Отдельная глава посвящена алгоритмам сортировки. Описываются общеизвестные алгоритмы, обсуждаются их свойства и нюансы, также приводится реализация на C.

Читатель найдет в пособии и детали машинного представления данных. На примерах объясняется, как хранятся в памяти целые и действительные числа, строки, символы, массивы.

Наконец, пособие содержит введение в общий контекст программирования, описывающее его историю и развитие. Приведены сведения о различных парадигмах программирования.

Предназначено для студентов и магистрантов высших учебных заведений, начинающих программистов, а также всех интересующихся программированием.

Рецензент
доц., зам. зав. кафедрой АФТИ ФФ НГУ М. Ю. Шадрин

Пособие подготовлено в рамках реализации Программы развития НИУ-НГУ на 2009–2018 гг.

ISBN 978-?-?????-???-?

©Новосибирский государственный университет, 2009
©Дашевский О. Ю., 2009

© www.phys.nsu.ru

Оглавление

© www.phys.nsu.ru

© www.phys.nsu.ru

Предисловие

Данное учебное пособие построено на основе одноименного курса, который в течение ряда лет читался автором на кафедре АФТИ ФФ НГУ для студентов первого года обучения.

Курс является базовым и отвечает нескольким основным задачам. Первая из них — ввести студентов в контекст грамотного, культурного современного программирования. Это достигается через освоение базовых структур данных (списки, деревья, хеш-таблицы) и базовых алгоритмов (поиск, сортировка и проч.). Студенты обучаются мыслить не операторами, а модулями, функциями и алгоритмами; не переменными, а структурами данных.

Вторая задача состоит в достижении понимания «как все работает». Для этого часть второго семестра традиционно отводилась под изучение и написание программ на ассемблере для MIX-машины Дональда Кнута [5]. Кроме того, изучалась низкоуровневая работа с памятью в C и некоторые другие темы, близкие к «железу».

Третья задача состоит в приобретении практического опыта по написанию реально работающих программ: освоение синтаксиса и семантики языка, базовых типов данных, функций стандартных библиотек, обучение практической реализации алгоритмов с обработкой ошибок (некорректного ввода и т. п.).

Книга, которую вы держите в руках, отчасти является отражением этих задач. Первые две главы задают контекст; в них рассказывается об истории компьютеров и программирования и о современном состоянии, в том числе об их роли в жизни людей. Третья глава целиком посвящена машинному представлению

© www.phys.nsu.ru

статических данных различных типов. В главах 4–7 описываются различные динамические структуры данных: списки, деревья и хеш-таблицы. Глава 8 содержит описание большинства известных алгоритмов сортировки.

Большинство алгоритмов, описываемых в книге, содержат реализацию на C. Язык C был выбран как своего рода lingua franca достаточно низкого уровня, чтобы можно было продемонстрировать работу с указателями и динамической памятью. Книга ожидает от читателя умеренного владения языком C и ни в коей мере не является введением в C — для этого есть прекрасные пособия [3, 4].

Автор благодарен коллегам-преподавателям, проводившим совместно с ним занятия по курсу «Основы программного конструирования»: Дмитрию Анипко, Владиславу Цырюку и Вячеславу Рудаковскому. Отдельная благодарность Владимиру Парфиненко за помощь в проведении занятий и активное ценное сотрудничество помимо учебного процесса, а также секретарю кафедры АФТИ Ирине Черношвец — за *долготерпение*.

Приведенный в книге исходный код доступен для скачивания по адресу `http://github.com/be9/opkbook`. Связаться с автором можно по электропочте `oleg.dashevskii@gmail.com`.

Олег Дашевский,
24 ноября 2009 г.,
Новосибирск.

© www.phys.nsu.ru

Глава 1

Основы основ

1.1. О программировании

Программирование — молодая профессия. Первые компьютеры появились не так давно, около 60 лет назад, а вместе с ними появились и первые программисты. За этот небольшой по историческим меркам срок программирование из удела нескольких людей стало массовой востребованной профессией. Количество компьютеров в мире выросло в миллионы раз (ведь когда-то был всего один компьютер!), их производительность выросла на много порядков. Поначалу компьютеры занимали целые здания, теперь же они запросто умещаются в спичечном коробке. Сейчас каждый мобильный телефон, карманный плеер (холодильник, СВЧ-печь...) — это специализированный компьютер.

Что роднит все эти компьютеры? Каждый из них нуждается в программном обеспечении (ПО). Без программы компьютер — просто груда железа и пластика. А для написания и изменения программ нужны профессионалы — программисты.

Кому служат программисты? Бездушной ли машине, от которой они пытаются добиться милости с помощью заклятий из букв, цифр и «странных» символов? Ни в коем случае! Настоящий кумир хорошего программиста — это *пользователь*.

Общество готово щедро оплачивать услуги программистов, но что оно получает взамен? Программы, двоичные коды? Будем откровенны: в действительности программы никого, кроме самих

© www.phys.nsu.ru

программистов, не интересуют. Они всего лишь средство для того, чтобы компьютер помогал пользователю решить его задачу (а еще лучше — решал задачу за него!).

1.2. Задачи, решаемые компьютером

В чем может помочь компьютер? Оказывается, что во многом, хотя и не во всем. Спектр решаемых задач вытекает, в частности, из устройства компьютера. Не отвлекаясь на подробное описание (его можно почерпнуть, например, в [12]), выделим две важнейших составных части: *память* и *процессор*.

Память (оперативная, дисковая...) хранит данные в двоичном представлении[1], процессор выполняет различные операции над этими данными.

Соответственно, если мы хотим решить задачу с помощью компьютера, мы должны представить ее входные и выходные данные в двоичном виде (ибо ни с чем другим компьютер работать не может), а также записать способ решения этой задачи как набор операций над данными (получив тем самым *алгоритм*).

В более компактной форме сказанное выше отражает формула Н. Вирта «Алгоритмы + Структуры данных = Программы», положенная в основу книги [2].

С данными дело, как правило, обстоит легко: числа и символы легко представимы в двоичном виде (см. гл. 3), изображения и звук могут быть закодированы в числах. Все остальное также может быть обработано, если будет оцифровано, в чем человечество довольно преуспело.[2]

[1] Память состоит из большого количества отдельных ячеек, каждая из которых может принимать значение 0 или 1

[2] Существуют многочисленные датчики, переводящие различные параметры физических процессов и среды в цифровой вид, в зависимости от задачи. Остается одно: могут ли быть оцифрованы переживания человеческой души? Исследователи долгое время пытаются измерять электрическую активность коры головного мозга для раскрытия данного феномена, но пока им это не очень удается...

© www.phys.nsu.ru

С алгоритмами ситуация другая. Некоторые задачи легко представляются в виде последовательности операций (алгоритмизуются), некоторые — нет. Например, задача решения системы линейных уравнений легко записывается в форме алгоритма (носящего имя Гаусса), а вся сложность состоит в реализации этого алгоритма с учетом особенностей представления чисел в компьютере (см. разд. 3.6). С другой стороны, задача надежного распознавания текста на изображении алгоритмизуется значительно хуже.[3]

Но если алгоритм найден и реализован в виде программы, то — эврика! Компьютер усердно, последовательно, без устали и отвлечений будет ее выполнять. Он не наделен психологическими особенностями, присущими человеку: ему не надоест, он не попросит прибавки к зарплате и не ляпнет грубую ошибку, потому что был не в духе, поссорившись с соседом. То есть компьютер силен там, где человек слаб: в монотонных, повторяющихся операциях. И это прекрасно, ибо он может стать человеку хорошим и полезным помощником, пусть и не другом.[4]

1.3. Работа программиста

Итак, программист автоматизирует решение задач пользователей. А как он это делает, в чем состоит его работа?

Ниже приведен список различных деятельностей, с которыми приходится сталкиваться программисту.[5]

[3] Есть целый класс подобных задач. Для их решения используются различные ухищрения: нейронные сети, нечеткая логика, генетические алгоритмы и др.

[4] Японские инженеры со своими собачками AIBO готовы оспорить последнее утверждение.

[5] Часто, особенно в больших проектах, каждый программист занимается чем-то одним. Существуют специализации: архитектор, кодер, тестировщик, технический писатель и др. Однако и один в поле воин — с помощью современных технологий можно выполнить нетривиальный проект в одиночку.

© www.phys.nsu.ru

Проектирование. Всегда есть момент, когда пользователи уже высказали, чего они примерно хотят[6], но программы еще нет. Здесь программист должен подобрать адекватные структуры данных, используемые алгоритмы, а в сложных задачах также осуществить *декомпозицию*, разбив задачу на множество более простых.

Кодирование. После работ по проектированию становится ясно, что и как делать, остается сделать, т. е. закодировать алгоритм на выбранном языке программирования.

Тестирование. Минимально работающая программа уже выдает какие-то результаты, поэтому возникает вопрос корректности этих результатов, соответствия ожидаемым для разных наборов входных данных.

Отладка. Когда тестирование показывает, что программа выдает некорректные результаты (или вообще дает сбой), необходимо искать и устранять ошибки в ней.

Документирование. Разработчики программы, конечно, знают, как ее запускать, как задавать входные данные, режимы работы и как получать результат в требуемой форме. К сожалению, это знание не передается магическим образом пользователям, поэтому для них пишется документация, т. е. тексты, описывающие программную систему и взаимодействие с ней с разных сторон.

Чем сложнее программный проект, тем больше может потребоваться документации: не только для пользователей, но и для разработчиков (хорошее описание структуры и особенностей реализации программы помогает новым разработчикам быстрее

[6]Полезная аксиома: *пользователи* ***никогда*** *не знают, чего они хотят.* Впрочем, взаимодействие с пользователями — это отдельная большая тема, которой мы не будем касаться в данном пособии.

© www.phys.nsu.ru

включаться в проект), для тех, кто будет сопровождать программу и т. д.

Внедрение. Когда программа готова, нужно сделать так, чтобы она заработала на оборудовании заказчика.[7]

Сопровождение. У пользователей программы возникают вопросы и затруднения, порой даже нештатные ситуации. И, конечно, со всем этим они обращаются к тому, кто разработал программу!

1.4. Хорошие и плохие программы

Какими плодами своего труда программисты гордятся, а что заставляет их не спать по ночам и краснеть, отводя глаза, когда внуки спрашивают: «Дед, а расскажи про свою работу?!» Оценивать качество программы можно двояким способом.

Пользователи обычно оценивают программу как «черный ящик» (не видя, как она написана). Им важно, чтобы программа:

- работала, т. е. выдавала требуемый результат без сбоев.
- была удобной в работе.
- была функциональной (расширяемой).
- работала как можно быстрее.
- тратила как можно меньше ресурсов.
- была «дружественной», т. е. прощала, исправляла ошибки и по максимуму «догадывалась» обо всем сама.

[7] Обычно это легко, но если оборудование заказчика — космический аппарат, летящий в просторах Вселенной, могут быть сложности, которые нужно учитывать заранее. История американских космических аппаратов серии Voyager содержит удивительные примеры удаленной отладки и внедрения.

© www.phys.nsu.ru

- была предсказуемой (соответствовала ожиданиям пользователя).

- была надежной (по возможности не теряла данные и не переставала запускаться после нештатных ситуаций).

- стоила недорого или вовсе обходилась бесплатно.

Список не является исчерпывающим. Однако далеко не все критерии из перечисленных важны для всех пользователей, да и эти критерии во многом субъективны. То, что удобно для опытных пользователей, может не оказаться таковым для начинающих, и наоборот. Из этого следует простой вывод: *знай своего пользователя!* Как правило, всем угодить невозможно.

Программисты, как правило, оценивают программы как «белый ящик», видя исходный код. Их внимание при этом обращено на следующее:

- Оптимальность выбора структур данных и алгоритмов для решаемой задачи.

- Отсутствие повторений (вынос повторяющегося кода в процедуры и функции).

- Модульность программы, оптимальность декомпозиции на модули.

- Красота кода (стиль форматирования, выбор имен, удачное использование выразительных средств языка).

- Документированность (комментарии в исходном коде, сопроводительная документация).

- Наличие автоматических тестов.

- Стремление автора программы к «изобретению велосипеда», т. е. повторной реализации процедур, уже реализованных в библиотеках используемой среды программирования,

© www.phys.nsu.ru

или придумыванию нового алгоритма там, где можно использовать типовой.

Как видно, требования в обоих списках даже не пересекаются. Можно написать такую программу, что пользователи будут счастливы, но «свои» (программисты) будут показывать пальцем — хотя обычно случается наоборот.

По-настоящему хорошей программой, как правило, довольны все: и пользователи, и программисты. Лучше всего нацеливаться именно на такой вариант.

1.5. Культура программирования

Почему программисты оценивают программы именно по таким критериям, а не другим (см. предыдущий раздел)? У начинающих часто возникает непонимание, зачем «наводить лоск» (красиво форматировать код с отступами, выделять функции, когда можно все писать в одной функции и т. д.), если разрабатываемая ими программа является учебной и пойдет «на выброс».[8]

Эти критерии берутся из общей *культуры программирования*. Дело в том, что человечеством за 60 лет накоплен огромный опыт по всем аспектам разработки и использования программ, взаимодействия внутри команды программистов, общения с заказчиком и т. п. Пространство культуры содержит в себе этот опыт в превращенной форме, отлитый в виде рекомендаций, как что следует делать. И культурный программист должен быть как минимум знаком с этими рекомендациями!

Помимо общей культуры и общей традиции существуют также частные, связанные с конкретными языками программирования, средами, операционными системами (ОС) и др. Примером может служить традиция разработки программ под ОС на базе UNIX [10], имеющая исторически сложившиеся отличия от тра-

[8] Если сомнений в необходимости «красоты» нет, можно порекомендовать книгу [9], в которой подробно описывается, как эту «красоту» наводить.

© www.phys.nsu.ru

диции ОС семейства Windows.[9] Другой пример: языки Ruby и Python являются близкими друг к другу по области применения и предоставляемым возможностям, но традиции различаются довольно сильно.[10]

Остается заметить, что нет правильных и неправильных культур (хотя приверженцы различных традиций без устали ломают копья на просторах Интернета). Начиная использовать новый язык или среду программирования («в чужой монастырь»), после вводного знакомства (синтаксис и семантика) следует ознакомиться с традицией: как принято оформлять код, как решать типовые задачи... В дальнейшем это окупится сторицей, ибо среда станет играть на руку: будет легче понять уже существующий код, найти взаимопонимание с коллегами и др.[11]

Призыв автора: будьте культурными программистами!

[9] Один из принципов UNIX — наличие множества маленьких программ, каждая из которых выполняет отдельную функцию. Необходимый функционал достигается с помощью их совместного увязывания. Для Windows более характерны монолитные программы.

[10] «Любимым коньком» сообщества Ruby-программистов является эстетика кода: не считается зазорным изобрести и реализовать новый способ решения старой задачи, если он будет красивее предыдущего. С другой стороны, сообщество Python придерживается принципа «для каждой задачи существует один правильный, канонический способ ее решения».

[11] При этом, конечно, традиция не должна становиться догмой. Она содержит общие рекомендации, а конкретная ситуация может потребовать иного.

© www.phys.nsu.ru

Глава 2

От машинных кодов к языкам программирования

2.1. Машинный код

Вы написали программу, но каким образом она «работает»? Ответом на этот вопрос является устройство компьютера.

Все современные компьютеры построены на принципах фон Неймана [12]. Процессор, сердце компьютера, ничего не знает о программах, а умеет лишь выполнять команды из своей *системы команд*. Как правило, любая система команд включает в себя арифметические и логические операции, работу с памятью (запись и чтение), операции ввода/вывода, условного и безусловного перехода. Каждая команда закодирована в виде набора бит (в современных архитектурах команда занимает один или несколько байт). Упрощенно работу процессора можно представить в виде замкнутого цикла:

Выборка команды → Исполнение команды → Переход к следующей команде → Выборка...

Команды лежат в последовательных ячейках памяти, и переход к следующей команде осуществляется путем увеличения *указателя команд*, содержащего в себе адрес следующей исполняемой команды. Для организации циклов и ветвлений используются команды условного и безусловного перехода, которые перемещают указатель команд по указанному адресу.

© www.phys.nsu.ru

Таким образом, программа для процессора выглядит в виде набора бит. Когда появились первые компьютеры, программы в такой форме и создавались (в первом компьютере ENIAC биты вводились с помощью тумблеров). Очевидно, это было непросто, ибо ошибка даже в одном бите могла повлечь за собой некорректную работу всей программы! От программиста требовалось помнить наизусть все коды команд и вручную вычислять адреса переходов. Внесение изменений в программу требовало величайшей аккуратности, много времени уходило на проверку программы и поиск ошибок.

Поэтому довольно скоро программисты создали способ более эффективной разработки программ.

2.2. Язык ассемблера

Язык ассемблера по классификации относится ко второму поколению языков программирования, хотя он стал первым настоящим языком.[1] В его основе лежит простая идея: дать всем машинным командам мнемонические имена, аргументы команд (операнды) также записывать в понятной человеку форме и отказаться от ручного вычисления адресов, используя вместо них метки.

Очевидно, возникает необходимость в отдельной программе-трансляторе (она также называется *ассемблером*), которой на вход поступает исходный код, записанный в мнемониках. На выходе ассемблер выдает машинный код, годный для исполнения процессором.

На листинге 2.1 приведена программа на ассемблере для архитектуры AMD64, выводящая под ОС Linux строчку «Hello world!». В ней использованы команды «mov», «lea», «xor» и «syscall». Кроме того, «msg» является меткой, соответствующей адресу, по которому лежит выводимая строка. Этот адрес будет автоматически вычислен ассемблером и подставлен в адресное

[1] К первому поколению относится прямой ввод машинного кода в компьютер.

© www.phys.nsu.ru

Листинг 2.1. Пример программы на ассемблере

```
format ELF64 executable at 0000000100000000h
segment readable executable
entry $
        mov     edx, msgsize
        lea     rsi, [msg]
        mov     edi, 1                  ; STDOUT
        mov     eax, 1                  ; sys_write
        syscall
        xor     edi, edi                ; exit code 0
        mov     eax, 60                 ; sys_exit
        syscall
segment readable writeable
msg     db 'Hello world!', 0xA
msgsize = $-msg
```

поле команды mov. Символ «;» означает начало комментария (игнорируемого транслятором), который продолжается до конца строки.

Удобство, которое дает язык ассемблера, можно оценить, сравнив исходный код программы с машинным кодом, в который она транслируется (машинный код записан в шестнадцатеричном виде).

```
BA 0D 00 00 00        ; mov edx, 13
48 8D 35 15 10 00 00  ; lea rsi, [rip+1015h]
BF 01 00 00 00        ; mov edi, 1
B8 01 00 00 00        ; mov eax, 1
0F 05                 ; syscall
31 FF                 ; xor edi, edi
B8 3C 00 00 00        ; mov eax, 60
0F 05                 ; syscall
48 65 6C 6C 6F 20     ; "Hello "
77 6F 72 6C 64 21 0A  ; "world!"
```

Как видно, сравнение не в пользу машинного кода. Поэтому

© www.phys.nsu.ru

ассемблеры с момента появления получили широкое распространение.

Но и на этом программисты не остановились...

2.3. Языки высокого уровня

При всем удобстве языка ассемблера по сравнению с чистым машинным кодом его сложно назвать венцом творения.

Во-первых, язык ассемблера на 100% привязан к архитектуре компьютера, и ассемблерную программу, написанную для одной архитектуры, нельзя выполнить на компьютере с другой архитектурой, ее нужно полностью переписывать. Во-вторых, язык ассемблера все же далек от человеческого мышления.

Возьмем для примера школьную формулу корней квадратного уравнения $ax^2 + bx + c = 0$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{2.1}$$

При одном взгляде на нее становится ясно, о чем речь и как вычислить корень. Однако реализация программы вычисления корней на языке ассемблера может занимать до 100 строк кода! При этом программисту необходимо помнить ассемблерные команды, какие регистры[2] можно использовать и т. д. При сложных вычислениях легко допустить ошибку.[3]

Поэтому возникло стремление перейти от машинных команд к более компактной записи, которая была бы одновременно «ближе к человеку». Это стремление воплотилось в языках высокого уровня (ЯВУ).

[2] Регистры — особый вид быстродействующей памяти, расположенной на самом процессоре. В листинге 2.1 это edx, rsi, edi, eax. Некоторые машинные команды работают только с определенными регистрами.

[3] Автору памятна ассемблерная программа для архитектуры PDP-11, в которой он смог найти и исправить ошибку только через полгода после ее первого проявления. Все это время приходилось использовать обходные пути *(workaround)*, чтобы ошибка не мешала работе.

© www.phys.nsu.ru

Листинг 2.2. Программа на Фортране, вычисляющая площадь треугольника

```
PROGRAM area
REAL base, height, area
PRINT *,'Base and height of a triangle? '
READ *, base, height
area = (1.0/2.0) * base * height
PRINT *,'The area of a triangle with base ', base
PRINT *,'and height ', height,' is ', area
STOP
END
```

Первым ЯВУ был Фортран, разработанный компанией IBM в середине 1950-х. Название языка является сокращением (**Фор**мульный **тран**слятор); язык хорошо подходил для научных и численных расчетов.[4]

На листинге 2.2 приведена программа на современном диалекте Фортрана, вычисляющая площадь треугольника с заданным основанием и высотой. Видно, что формула $S = \frac{1}{2}bh$ в исходном коде записывается почти «как есть».

На данный момент насчитывается несколько тысяч ЯВУ, созданных различными людьми для различных целей. Впрочем, широкое распространение получили не более 10–20.

Обозначим некоторые общие свойства ЯВУ.

В отличие от ассемблерных программ, которые преобразуются в машинный код по принципу «одна строка кода — одна машинная команда»[5], программы на ЯВУ транслируются более сложным образом. Нельзя предсказать, в какие машинные ко-

[4] В 1979 г. создатель Фортрана Джон Бэкус признался в интервью: «Значительная часть моей работы выросла из моей лени. Мне не нравилось писать программы, и, разрабатывая на компьютере IBM 701 программы для расчета траекторий ракет, я занялся созданием системы программирования, которая бы облегчила написание программ».

[5] За исключением комментариев и псевдокоманд, указаний самому ассемблеру. Примером псевдокоманды может служить «segment» в листинге 2.1.

© www.phys.nsu.ru

манды преобразуется каждая строка кода — это зависит от транслятора.

С одной стороны, это делает программы на ЯВУ *переносимыми*. Программа на Фортране, написанная 50 лет назад, скорее всего, будет работать и сейчас, на современном компьютере с современным транслятором.

С другой стороны, «удаление» от ассемблера лишает программиста возможности использовать возможности процессора на 100% в каждом конкретном случае. Поэтому программа на ЯВУ с большой вероятностью будет работать медленнее и использовать больше памяти, чем программа на ассемблере. Тем не менее:

- Современные оптимизирующие трансляторы с каждым годом генерируют все более хороший код. Современные же процессоры становятся все сложнее. Зачастую хороший генератор кода, встроенный в транслятор, может выполнить работу лучше программиста, знающего ассемблер «в принципе», но не знакомого с тонкостями исполнения различных команд.[6]

- Программа должна быть не быстрой, а лишь «достаточно быстрой». Зачастую в системе «компьютер — человек» самым медленным элементом является пользователь.[7]

- Большинство трансляторов ЯВУ предоставляют возможность подключения к программам внешних модулей, разработанных на других языках, в том числе на ассемблере. Это позволяет реализовывать на ассемблере наиболее кри-

[6] Руководства по оптимизации кода для современных процессоров AMD и Intel занимают 300–600 страниц.

[7] Восприятие скорости работы программ пользователями является субъективным. Известно, что средняя скорость реакции неподготовленного человека на событие составляет около 200 мс. Если программа успевает за это время выдать ответ, ее работа воспринимается как «мгновенная» и не важно, было это 10, 50 или 150 мс.

© www.phys.nsu.ru

тичные участки, а остальные модули программы разрабатывать на выбранном ЯВУ.

- Проблемы с быстродействием и расходом памяти часто решаются покупкой более быстродействующего аппаратного обеспечения («железа»). С позиции заказчика может оказаться выгоднее (т. е. дешевле!) купить более совершенное «железо», чем оплачивать труд программистов по оптимизации программы.

Таким образом, развитие аппаратного обеспечения и инструментальных средств (трансляторы, отладчики, среды программирования, делающие разработку программ более удобной) привело к повсеместному доминированию ЯВУ. Ассемблер сохранил за собой небольшую экологическую нишу, связанную с реализацией типовых алгоритмов в случаях, где действительно требуется высокое быстродействие, а также в системном программировании, тесно завязанном на аппаратное обеспечение (как правило, небольшая часть ядра любой операционной системы написана на ассемблере для соответствующей компьютерной архитектуры).

2.4. Парадигмы программирования

Появление первых трансляторов с ЯВУ привело к возникновению интересного вопроса: какими вообще могут быть языки программирования? Часть требований можно почерпнуть из принципа функциональности:[8]

- *Транслируемость.* Должна существовать возможность разработки транслятора, который позволит перевести программу в машинный код и, следовательно, исполнить ее.

[8] Язык программирования, не будучи художественным произведением, все-таки должен приносить конкретную пользу. Впрочем, некоторые языки создавались в основном для удовлетворения особых эстетических потребностей авторов и примкнувших к ним (см. языки Brainfuck и Befunge).

© www.phys.nsu.ru

- *Удобство.* Язык должен быть удобным для решения определенного круга задач.

Эти требования задают рамку, но ничего не говорят о содержании языка, о том, что в нем «есть». Требуется еще что-то.

И здесь рождается представление о *парадигмах программирования.*[9] Парадигма задает термины, в которых программист будет описывать логику разрабатываемой им программы. Именно эти термины отражены в синтаксических конструкциях языка.

Если программист оперирует *действиями*, это императивная парадигма. Фокус на *определениях* выдает функциональную парадигму, а если программа представляет из себя *набор логических предикатов и правил вывода*, можно говорить о логической парадигме.

Парадигмы не являются принципиально противоречащими друг другу и могут одновременно реализовываться в рамках одного языка программирования или использоваться внутри одной программы.[10]

2.5. Императивная парадигма

Из всех парадигм первой появилась императивная. Она предполагает, что программа в каждый момент времени имеет определенное *состояние* — значения переменных, текущий оператор и др. При этом сама программа является набором операторов по изменению собственного состояния.

Очевидным образом императивная парадигма соответствует принципу работы компьютера, который также имеет состояние (память, регистры процессора, состояния устройств ввода/вывода) и программу в виде набора машинных команд, каждая из которых изменяет состояние компьютера (путем записи в ячейки

[9] Парадигма — термин, имеющий историко-философское происхождение и приблизительно означающий «устоявшийся способ видеть вещи».

[10] С возможным добавлением других парадигм: объектно-ориентированной, аспектно-ориентированной, обобщенной и др.

© www.phys.nsu.ru

памяти, регистры и др.)

Как первый ЯВУ, Фортран, так и множество других языков основаны на императивной парадигме. В настоящее время она является наиболее распространенной.

В 1970-е гг. в рамках данной парадигмы Э. Дейкстрой и Н. Виртом была разработана методология *структурного* программирования. В соответствии с ней любая программа — это иерархическая структура, состоящая из трех типов базовых блоков:

1) *последовательное исполнение* — однократное выполнение операций в том порядке, в котором они записаны в тексте программы;

2) *ветвление* — однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия;

3) *цикл* — многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла).

Никаких других способов управления последовательностью исполнения операций, кроме ветвления и цикла, не допускается.[11]

Кроме того, предполагается, что повторяющиеся участки кода, а также логически и смыслово связанные последовательности блоков по возможности оформляются в виде подпрограмм (процедур и функций).

Структурное программирование было признано «самой сильной формализацией 1970-х гг.» и в настоящее время является плотью и кровью всех языков, реализующих императивную парадигму.

[11] Структурное программирование объявило настоящую войну оператору безусловного перехода GO TO, широко используемому во многих ЯВУ на тот момент. По мнению Вирта и Дейкстры, этот оператор значительно снижает читаемость программ, затрудняя их сопровождение.

© www.phys.nsu.ru

2.6. Функциональная парадигма

Как сказано выше, функциональная парадигма фокусируется на *определениях*. В данной парадигме результат программы — это функция от входных данных, причем эта функция выражается через другие функции. Те, в свою очередь, могут выражаться через третьи функции, или через сами себя с другими аргументами, и т. д.

Чисто функциональный стиль предполагает *отсутствие* явных состояний у программы. Значение функций зависит только от входных параметров, и, следовательно, функция, вызванная с теми же параметрами, обязательно вернет тот же результат (в отличие от императивного стиля, где результат может зависеть, например, от глобальных переменных). Иначе говоря, функции не имеют «побочных эффектов».

Отсутствие состояния[12] дает весьма существенные преимущества.

Во-первых, у транслятора «развязываются руки» — в силу независимости поведения функций от глобального состояния появляется возможность более глубокой оптимизации их работы. Кроме того, программа может естественным образом работать на многих процессорах, если это позволяют среда исполнения и транслятор — распараллеливание произойдет автоматически. В противовес этому, императивные программы требуют ручного распараллеливания, т. е. реализации параллелизма внутри самой программы.[13]

Во-вторых, отсутствие побочных эффектов при исполнении функций существенно упрощает их отладку и тестирование.

[12] В действительности состояние, конечно, присутствует, но оно «спрятано» внутри среды исполнения программы.

[13] В настоящее время параллелизм весьма актуален. Разработчики процессоров уперлись в предел повышения линейной производительности систем; предполагается, что дальнейшее развитие пойдет в сторону увеличения количества процессорных ядер на одном кристалле. Но задействование многих ядер требует поддержки со стороны программного обеспечения. Программы в чисто функциональном стиле уже готовы к этому.

© www.phys.nsu.ru

Листинг 2.3. Программа на Лиспе, вычисляющая факториал

```
(defun factorial (n &optional (acc 1))
  (if (<= n 1)
      acc
      (factorial (- n 1) (* acc n))))
```

В императивных же программах ошибка, прячущаяся внутри функции, порой проявляется только при специфическом общем состоянии программы, т. е. в зависимости от предыстории работы программы.

Первым функциональным языком программирования был Лисп[14], разработанный в конце 1950-х гг. Джоном Маккарти и ставший вторым по счету ЯВУ в мире (после Фортрана). Лисп совсем не похож на другие языки программирования, имея весьма специфический синтаксис. Программа представляет из себя список вложенных друг в друга списков, с особой нотацией: привычное $f(x, y)$ в Лиспе записывается как $(f\ x\ y)$. Выражение $a+b+c$ будет записано как $(+\ a\ b\ c)$, а $a+b*c$ — как $(+\ a\ (*\ b\ c))$. На листинге 2.3 приведен пример программы на Лиспе.

Функция factorial является рекурсивной, но мало-мальски продвинутый транслятор Лиспа произведет оптимизацию «устранение хвостовой рекурсии» и организует вычисления в виде цикла.

Кроме Лиспа (который, к слову, поддерживает все возможные парадигмы программирования) в настоящее время также распространены функциональные языки Haskell и Erlang.

Интересным примером реализации функциональной парадигмы являются, как ни странно, электронные таблицы Microsoft Excel! В Excel можно реализовать достаточно сложные расчеты, при этом в ячейках таблицы указывается либо константное значение, либо формула, по которой рассчитывается значение ячейки. Формула содержит в себе вызовы встроенных функций (ма-

[14] Lisp (англ.), от **Lis**t **P**rocessor (обработчик списков).

© www.phys.nsu.ru

тематических, финансовых и др.) и ссылки на значения других ячеек. Алгоритм и последовательность расчета ячеек определяет сам Excel — ну чем не функциональный подход?!

Остается отметить, что некоторые концепции функциональной парадигмы в настоящее время находят применение и вне чисто функциональных языков — например, концепция *замыканий (closure)*.

2.7. Логическая парадигма

Отдельной областью применения компьютеров являются так называемые *базы знаний*, или *экспертные системы*. Рассмотрим простой пример из жизни, принадлежащий к данной области.

Клиент приезжает на автомобиле в автосервис и сообщает мастеру приемки о том, что его машина неисправна и ее нужно починить. Мастер начинает расспрашивать о неисправности и, собрав достаточно информации (при каких условиях и как именно проявляется неисправность), выдает предположение о причине и способах ее устранения. Остается лишь произвести необходимые операции по ремонту, и замене деталей, и счастливый клиент, оплатив счет, уезжает.

Как у мастера получается находить причину? Он знает, как проявляются типовые неисправности, и пытается путем расспросов и наблюдений найти факт, который сможет указать на причину. К примеру, для бензиновых двигателей есть простая истина: «если двигатель не работает, то либо нечему гореть, либо нечему поджигать». Соответственно, в таком случае нужно искать причину в системе питания (есть ли бензин в баке? подается ли он в цилиндры?), если там все в порядке, то в системе зажигания, и т. д. Получается раскручивание причинно-следственных связей в обратную сторону: от фактов к причинам. Аналогичным образом действуют доктор Хаус в популярном сериале и Шерлок Холмс в романах Конан Дойля.

Чем хороший мастер отличается от плохого? Тем, что у него в голове гораздо больше причинно-следственных связей между

© www.phys.nsu.ru

неисправностями и их проявлениями и эти связи лучше структурированы. Как следствие, хороший мастер будет обращать внимание на большее количество фактов (на нюансы!) и правильнее их оценивать, быстрее, точнее и фокусированнее находя неисправность.

Как стать хорошим мастером? «Опыт и только опыт» — говорят хорошие мастера, подмигивая нам.

Но автомобиль — не такая уж сложная система. Как быть, например, с атомной электростанцией, где различных узлов и механизмов несоизмеримо больше, а неисправности куда как более чреваты? На приобретение опыта по разным неисправностям может попросту не хватить человеческой жизни, а ведь бывают и очень редкие неисправности, которые за это время могут просто не произойти. Кроме того, возникает вопрос передачи опыта.

Не лучше ли доверить хранение фактов, взаимосвязей и работу с ними? Именно так и возникли первые базы знаний, помогающие не только в поиске неисправностей, но и в принятии решений в сложных ситуациях, доказательстве математических теорем и др.

В 1972 г. был создан язык Пролог, предназначавшийся специально для создания баз знаний и работы с ними, а также для решения аналогичных задач.

Основными понятиями в Прологе являются факты, правила логического вывода и запросы. *Факты* описываются логическими предикатами с конкретными значениями. *Правила* записываются в форме правил логического вывода с логическими заключениями и списком логических условий. На *запросы* система Пролог генерирует логические ответы («истина» и «ложь») или находит удовлетворяющие запросу факты из базы знаний. Для получения ответа используются правила логического вывода, а также механизм бэктрэкинга[15].

[15] Бэктрэкинг — общий алгоритм для поиска решений какой-либо вычислительной задачи, состоящий в постепенном подборе возможных кандидатов решений с их отбраковкой, если становится ясно, что кандидат (частичное решение) не может быть достроен до полного решения.

© www.phys.nsu.ru

Листинг 2.4. Программа на Прологе

```
mother_child(Maria, Elena).

father_child(Ivan, Elena).
father_child(Ivan, Anna).
father_child(Sergey, Ivan).

sibling(X, Y)       :- parent_child(Z, X),
                       parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).

?- sibling(Elena, Anna).
Yes
```

На листинге 2.4 приведена программа на Прологе, реализующая простую базу знаний. Первые четыре строки содержат факты в форме предикатов *mother_child(X,Y)* и *father_child(X,Y)*: Мария — мать Елены; Иван — отец Елены и Анны; Сергей — отец Ивана. Затем определяются два правила вывода: X и Y находятся в братско-сестринских отношениях тогда и только тогда, когда имеют общего родителя (Z). В то же время X является родителем Y, когда является истинным один из предикатов *mother_child(X,Y)* или *father_child(X, Y)*. В конце листинга системе предлагается вычислить предикат, свидетельствующий о том, является ли Елена сестрой Анны. Система Пролог делает логические выводы по заданным правилам, самостоятельно определяет значение Z (Ivan) и отвечает на запрос утвердительно.

2.8. Компиляция и интерпретация

Различаются не только парадигмы программирования, но и подходы к построению трансляторов. Обобщая, можно сказать,

© www.phys.nsu.ru

что существует два противоположных подхода и некоторое пространство вариантов между ними.

Исторически первый подход связан с *компиляцией*. Компилирующий транслятор (компилятор) целиком обрабатывает программу, переводя ее в машинный код. Готовый код может запускаться на исполнение на том же или другом компьютере той же архитектуры, программа-компилятор при этом больше не требуется.

Альтернативный подход, появившийся несколько позже — *интерпретация*. Интерпретирующий транслятор (интерпретатор) не переводит программу в машинный код. Вместо этого он исполняет ее прямо «на ходу», в процессе разбора исходного текста.

У обоих подходов есть преимущества и недостатки. Компиляция позволяет получить готовый машинный код в двоичной форме, в дальнейшем этот код можно передавать пользователям. Для успешной работы с программой им уже не потребуется ни компилятор, ни исходный код.

С другой стороны, интерпретатор убыстряет разработку программы. Вносимые изменения идут в дело мгновенно, компиляции и перезапуска не требуется. Однако за удобство приходится платить меньшей скоростью исполнения по сравнению с компиляцией, необходимостью иметь интерпретатор для запуска программы и предоставлять пользователям исходные тексты программы.[16]

На первых компьютерах, которые были довольно медленными, в основном развивался подход, связанный с компиляцией. Бурный рост интерпретации начался в эпоху микрокомпьютеров, в конце 1970-х гг., и связан в первую очередь с языком BASIC (Бэйсик). Будучи простыми и дружелюбными к пользователю, интерпретаторы Бэйсика позволяли быстро научиться писать программы.[17]

[16] А вдруг они изменят имя автора и тоже начнут продавать программу?

[17] При этом профессиональные программы для микрокомпьютеров писались на ассемблере из-за существенных ограничений по производительности и памяти — «каждый байт на учете».

© www.phys.nsu.ru

В настоящее время компиляторы и интерпретаторы интересным образом сблизились, благодаря изобретению *байт-кода*.

Байт-код очень похож на машинный код, за тем исключением, что исполняет его не процессор, а *виртуальная машина* — специальная программа. В отличие от машинного, байт-код обычно содержит в себе более высокоуровневые команды, которые, например, могут оперировать с объектами.

Использование байт-кода позволяет убыстрить работу классических интерпретаторов. Вместо того чтобы исполнять программу по мере ее трансляции, они могут сгенерировать высокоуровневый байт-код, который затем будет исполняться виртуальной машиной. Таким образом, вместо чистой интерпретации возникает смешанный подход: компиляция (генерация байт-кода) и интерпретация (исполнение байт-кода).

Широкое распространение такой подход получил с созданием языка Java, который был изначально ориентирован на использование виртуальной машины, JVM[18]. Эта машина является стековой (вместо регистров все команды работают со стеком) и, кроме выполнения байт-кода, также обеспечивает «сборку мусора» (автоматическое освобождение неиспользуемой памяти).

На листинге 2.5 приведен пример программы на Java, печатающей простые числа, меньшие 1000, вместе с байт-кодом, в который она транслируется. Видно, что байт-код поддерживает вызовы методов и автоматическое выделение памяти под локальные переменные.

Стоит отметить, что реализация JVM фирмы Sun Microsystems (HotSpot) является одной из наиболее продвинутых виртуальных машин на данный момент (содержит в себе около 250 тыс. строк кода на C++ и ассемблере). Она поддерживает режим динамической компиляции JIT[19], когда байт-код вместо прямого исполнения виртуальной машиной подвергается компиляции теперь уже в машинный код! Это может происходить прямо в

[18] Java Virtual Machine — виртуальная машина Java.

[19] Just-in-time compilation.

© www.phys.nsu.ru

Листинг 2.5. Программа на Java и ее байткод

```
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}

/*
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush      1000
6:   if_icmpge   44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge   31
16:  iload_1
17:  iload_2
18:  irem
19:  ifne        25
22:  goto        38
25:  iinc        2, 1
28:  goto        11
31:  getstatic   #84;
  //Field java/lang/System.out:Ljava/io/PrintStream;
34:  iload_1
35:  invokevirtual #85;
  //Method java/io/PrintStream.println:(I)V
38:  iinc        1, 1
41:  goto        2
44:  return
*/
```

© www.phys.nsu.ru

процессе работы программы, причем динамической компиляции может подвергаться не вся программа, а лишь ее наиболее используемые участки.

Платформа .NET фирмы Microsoft устроена сходным образом, имея в своем составе промежуточный архитектуронезависимый «ассемблер» CIL[20]. Все программы внутри платформы .NET транслируются в CIL, который в дальнейшем преобразуется в машинный код по принципу JIT.

Трансляция с помощью байт-кода также повсеместно применяется в динамических языках[21]: Python, Ruby (с версии 1.9), Perl 6 (виртуальная машина Parrot) и др.

[20] Common Intermediate Language — «общий промежуточный язык».

[21] Динамические языки — класс языков, в которых программа может прямо в процессе исполнения добавлять новый код, изменять поведение существующих объектов, типов и т. д.

© www.phys.nsu.ru

Глава 3

Машинное представление данных

3.1. Беззнаковые целые числа

Беззнаковые (неотрицательные) целые числа имеют простейшее машинное представление. N двоичных бит образуют 2^N различных комбинаций. Воспринимая биты как цифры двоичного числа, получаем возможные значения от 0 до $2^N - 1$.

Память современных компьютеров устроена так, что минимальный адресуемый размер равен 1 байту (8 битам). Максимальный размер памяти, с которым можно работать за один прием (одну машинную команду), соответствует разрядности архитектуры. Сейчас это обычно 32 или 64 бита (4 и 8 байт, соответственно). Таким образом, традиционно используются следующие размерности чисел:

Бит (N)	**Байт**	**Название**	**Макс. число**
8	1	Байт	255
16	2	Короткое слово	65 535
32	4	Слово	4 294 967 295
64	8	Длинное слово	$\approx 1{,}8 \times 10^{19}$

Рассмотрим двоичную запись числа 98 для случая $N = 8$:

№ бита	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
	0	1	1	0	0	0	1	0

© www.phys.nsu.ru

Принято нумеровать биты справа налево, при этом 0-й бит называется младшим, а $(N-1)$-й — старшим. Номер бита соответствует степени 2, на которую умножается соответствующая цифра:

$$M = \sum_{k=0}^{N-1} 2^k b_k, \tag{3.1}$$

где M — целое число в диапазоне от 0 до $(2^N - 1)$, k — номер бита, b_k — значение k-го бита (0 или 1).

У двоичной системы существует связь с 8-ричной и 16-ричной. Число $98_{10} = 01100010_2 = 142_8 = 62_{16}$; 8-ричная и 16-ричная запись получается из двоичной путем группировки бит по 3 и по 4, соответственно.[1]

Числа длиной более 1 байта в памяти располагаются согласно *порядку байтов*, определяемому архитектурой компьютера компьютера. Исторически сложились два варианта порядка: «от младшего к старшему» (little-endian) и «от старшего к младшему» (big-endian).[2] Процессоры Intel и AMD (32- и 64-разрядные), применяемые в современных персональных компьютерах, используют порядок «от младшего к старшему». К примеру, для них 32-битное машинное слово $12345 = 3039_{16}$ в памяти будет представлено как четыре последовательных байта *39 30 00 00*. В порядке «от старшего к младшему» это было бы *00 00 30 39*.

Разрядность процессора (64, 32, 16, 8) соответствует максимальной длине числа, с которым он может выполнять арифметические операции за одну команду. Работу с числами большей длины можно реализовать программным путем (подробнее см. [6, разд. 4.3]).

[1] $142_8 = \langle 001_2\rangle\langle 100_2\rangle\langle 010_2\rangle$, $62_{16} = \langle 0110_2\rangle\langle 0010_2\rangle$, где $\langle\cdot\rangle$ означает отдельно стоящую цифру.

[2] Забавный факт: английские названия происходят из произведения Дж. Свифта «Приключения Гулливера», где два вымышленных государства много лет вели войну из-за разногласий, с какого конца нужно разбивать вареные яйца: с тупого (big-endians, «тупоконечники») или с острого (little-endians, «остроконечники»).

© www.phys.nsu.ru

С N-разрядным числом процессор выполняет арифметические операции по модулю 2^N. К примеру, если поместить максимальное значение ($2^{16} - 1 = 65535$) в 16-битное слово, а затем прибавить к нему единицу, в результате получится 0, а не 2^{16}:

$$65535 + 1 \equiv 0 \pmod{2^{16}}.$$

Как правило, в процессоре есть специальный флаговый регистр, в котором присутствует *флаг переноса (carry flag)* и куда как раз попадает N-й бит результата. Однако ЯВУ обычно не предоставляют доступа к флагам процессора, поэтому ситуацию *переполнения* нужно отслеживать косвенным способом (если $c = (a + b) \bmod 2^{16}$ и $c < a$ или $c < b$, то было переполнение).

3.2. Знаковые целые числа

Представление знаковых чисел в машинном слове требует дополнительных усилий. Существует три основных метода.

1) *Хранение с избытком.*

   Если вместо числа M в N-битной ячейке памяти хранить битовое представление числа $M + E$, тогда само M может находится в диапазоне $[-E, 2^N - (E + 1)]$. Выбирая $E = 2^{N-1}$, получаем диапазон $[-2^{N-1}, 2^{N-1} - 1]$.

   Такое представление является весьма неудобным для выполнения арифметических операций (при сложении двух знаковых чисел нужно вычитать лишний избыток и проч.). Кроме того, беззнаковое и знаковое представление одного и того же числа отличаются, что только добавляет путаницы.

2) *Знаковый бит.*

   Поскольку есть всего два значения знака («+» и «−»), можно просто выделить отдельный бит под знак (удобнее всего — старший). Пусть 0 обозначает положительное число,

© www.phys.nsu.ru

1 — отрицательное. Тогда в случае $N = 8$:

$$\begin{aligned} 1 &\equiv 00000001_2; \\ -1 &\equiv 10000001_2; \\ 127 &\equiv 01111111_2; \\ -127 &\equiv 11111111_2. \end{aligned}$$

Данный подход позволяет в N-битном слове представить все M при условии $|M| < 2^{N-1}$.

Видно, что битовое представление положительных чисел совпадает с их представлением в беззнаковом случае. Однако при таком подходе все равно нужна отдельная арифметика, учитывающая возможное несовпадение знаков двух чисел. Кроме того, возникает двойственность в представлении числа 0, т. е. $+0$ и -0:

$$\begin{aligned} 0 &\equiv 00000000_2 \quad (+0); \\ 0 &\equiv 10000000_2 \quad (-0). \end{aligned}$$

3) *Двоичное дополнение.*

Третий способ основан на уже упоминавшемся факте, что для N-битных слов процессор выполняет арифметические операции по модулю 2^N.

Возьмем произвольное число M и попробуем к нему прибавить число $X = 2^N - 1$:

$$M + X = M + 2^N - 1 \equiv M - 1 \pmod{2^N}. \qquad (3.2)$$

Простое сложение $M + X$, не учитывающее никаких знаков, дало в результате $M - 1$. Это позволяет нам принять число X (вернее, его битовое представление) за минус единицу.

© www.phys.nsu.ru

Подобным образом можно получить число -2 и т. д.:

$$-1 \equiv \underbrace{(111\ldots11)}_{N \text{ единиц}}{}_2;$$
$$-2 \equiv (111\ldots10)_2;$$
$$\ldots$$
$$-2^{N-1} \equiv (100\ldots00)_2.$$

В этом случае старший бит также отражает знак (0 — положительное число, 1 — отрицательное). Диапазон представимых чисел равен $[-2^{N-1}, 2^{N-1}-1]$:

$$2^{N-1} \equiv (011\ldots11)_2.$$

При выполнении вычислений необходимо отслеживать переполнение, ибо слишком большое положительное число становится отрицательным и наоборот!

В современных процессорах для представления знаковых целых чисел повсеместно применяется двоичное дополнение. Сложение и вычитание совпадают с беззнаковым случаем[3], нет проблемы двойственности 0. Ниже приведена таблица диапазонов целых знаковых чисел для машинных слов различной длины:

Бит (N)	**Мин. число**	**Макс. число**
8	-128	127
16	$-32\,768$	32 767
32	$-2\,147\,483\,648$	2 147 483 647
64	$\approx -9{,}2 \times 10^{18}$	$\approx 9{,}2 \times 10^{18}$

[3]Умножение и деление в знаковом и беззнаковом случае выполняются поразному.

© www.phys.nsu.ru

3.3. Двоично-десятичный код

Кроме двоичной системы, существует еще одна возможность машинного представления чисел — когда каждая цифра в десятичной записи представлена отдельно.

Такой принцип положен в основу *двоично-десятичного кода*[4], использующего на каждую десятичную цифру по тетраде (4 бита, полубайт). Один байт содержит, таким образом, две десятичных цифры.

Ниже приведен пример кодирования числа 1980 в 16-битном слове с помощью двоично-десятичного кода:

1	**9**	**8**	**0**
0 0 0 1	1 0 0 1	1 0 0 0	0 0 0 0

Получившееся число

$$1100110000000_2 = 6528_{10} = 1980_{16}.$$

То есть представление десятичного числа в двоично-десятичном коде совпадает с двоичным представлением 16-ричного числа, состоящего из тех же цифр, что и искомое десятичное. При этом значения тетрад 1010, 1011, 1100, 1101, 1110, 1111 являются недопустимыми.

Такой способ представления часто применяется во встроенных системах (например, в калькуляторах) в силу удобства вывода чисел в такой форме.[5] Здесь, в отличие от двоичного кода, не требуется множественного деления на 10 для получения отдельных цифр, достаточно простейших битовых операций.

С числами, закодированными в двоично-десятичном коде, возможно выполнять арифметические операции. Сложение и

[4] По-английски: BCD (binary-coded decimal).

[5] Например, микроконтроллер встроенной системы может быть подключен к 7-сегментным индикаторам, отображающим отдельные цифры. Использование двоично-десятичного кода позволяет в таком случае упростить схемотехнику и управляющую программу.

© www.phys.nsu.ru

Таблица 1. Кодировка ASCII

```
 32:   ! " # $ % & ' ( ) * + , - . /
 48: 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
 64: @ A B C D E F G H I J K L M N O
 80: P Q R S T U V W X Y Z [ \ ] ^ _
 96: ` a b c d e f g h i j k l m n o
112: p q r s t u v w x y z { | } ~
```

вычитание производятся по правилам двоичной арифметики, с последующей коррекцией.[6] Умножение и деление могут быть реализованы «столбиком» и «уголком», соответственно.

3.4. Символы

Машинное представление символов — беззнаковые числа, коды в таблице символов (кодировке).

Первая стандартизованная кодировка появилась в 1963 г. — это кодировка ASCII[7]. Кодировка является 7-битной и включает в себя 95 символов: прописные и строчные латинские буквы, цифры, знаки препинания (см. табл. 1).

Первые 32 кода (от 0 до 31), а также код 127 используются для служебных символов. Основные служебные символы приведены в табл. 2.

Кодировка ASCII могла удовлетворить американских и английских пользователей, но в остальных странах, где используются другие алфавиты или дополнения к латинскому алфавиту,

[6] После сложения двух тетрад, приведшего к тетраде с недопустимым значением или к переносу в старшую тетраду, необходимо добавить к соответствующей тетраде 6. После вычитания во всех случаях, когда происходит заем из старшей тетрады, нужно дополнительно вычесть из нее 6.

[7] ASCII — аббревиатура от American Standard Code for Information Interchange — американский стандартный код для обмена информацией.

© www.phys.nsu.ru

Таблица 2. Некоторые служебные символы в кодировке ASCII

Код	Название	Аббревиатура	В языке C
7	Звонок	BEL	\a
8	Забой (Backspace)	BS	\b
9	Табуляция	TAB	\t
10	Перевод строки	LF	\n
12	Новая страница	FF	\f
13	Возврат каретки	CR	\r
26	Конец файла	SUB	\x1A
127	Delete	DEL	\x7F

пользователи не имели возможности кодировать текст на родном языке. Поэтому стали возникать расширения таблицы ASCII.

Поскольку ASCII является 7-битной, т. е. содержит только 128 кодов, при однобайтном кодировании символов остается еще $256-128=128$ кодов, которые и стали использоваться для символов национальных алфавитов.

К сожалению, это привело к возникновению «ада» кодировок. Только для русского языка было изобретено не менее 6 штук:

- *Кодировка КОИ-8.* Использовалась на Unix-подобных системах и при передаче электронной почты.
- *Основная кодировка.* Определялась ГОСТ 19768-87, практически не использовалась.
- *Альтернативная кодировка (CP866).* Также определялась ГОСТ, использовалась в операционной системе MS-DOS на компьютерах IBM PC. Кодировка сохраняла псевдографические символы (вертикальные и горизонтальные линии, углы и др.) на оригинальных местах, соответствующих базовой кодировке IBM PC (CP437), что позволяло использо-

© www.phys.nsu.ru

вать программы с текстовым интерфейсом. Из-за этого, однако, русские буквы размещены в таблице двумя блоками, с разрывом. Эта кодировка до сих пор используется по умолчанию в текстовой консоли Windows, если выбран русский язык.

- *ISO-8859-5*. Стандарт ISO для кириллических алфавитов, так и не нашедший применения.

- *Кодировка Windows (CP1251)*. Кодировка для кириллических алфавитов, изобретенная Microsoft и ParaGraph для ОС Windows. Широко применяется для русского, украинского, белорусского и болгарского языков.

- *Кодировка MacCyrillic*. Использовалась на компьютерах Apple Macintosh.

Все эти кодировки являются ASCII-совместимыми, т. е. нижняя часть (коды 0–127) у них у всех совпадает с ASCII, а верхняя часть (коды 128–255) значительно разнится.

Несмотря на то, что проблемы пользователей большинства языков решаются с помощью верхней половины таблицы, идеальным это решение назвать нельзя:

1) Множество кодировок для одного языка создает путаницу, необходимость в перекодировании и определении кодировки, если она неизвестна.

2) Некоторые алфавиты не помещаются в 128 кодов (Китай, Япония, Корея и др.).

3) Нет возможности писать тексты одновременно на нескольких языках (русском, английском и арабском, например).

Решением описанных проблем стала кодировка Юникод (Unicode), для которой была поставлена задача закодировать *все* символы — буквы и знаки различных алфавитов, пунктуационные, музыкальные, математические и прочие символы. Первая версия

© www.phys.nsu.ru

Юникода ограничивалась 65 536 символами, в дальнейшем верхний предел был поднят до 1 112 064 символов.

Тексты в Юникоде представляются тремя возможными способами:

1) *UTF-8*. Восьмибитный формат с переменной длиной.

   Совместим с кодировкой ASCII — любой символ с кодом меньше 128 занимает 1 байт. Другие символы могут занимать от 2 до 4 байт (см. табл. 3).

2) *UTF-16*. Шестнадцатибитный формат с переменной длиной.

   Символы с кодом менее 65 536 (шестн. 10000) представляются как есть, остальные (шестн. 10000–10FFFF) — в виде последовательности двух 16-битных слов (так называемых суррогатных пар), первое из которых лежит в диапазоне D800–DBFF, а второе — DC00–DFFF.

   Легко видеть, что всего таким способом можно закодировать $2^{20} + 2^{16} - 2\,048 = 1\,112\,064$ символа (это число и было выбрано в качестве величины кодового пространства Юникода).

   Поскольку каждый символ занимает 2 или 4 байта, возникает вопрос порядка байтов (см. разд. 3.1, стр. 34). Существует две разновидности UTF-16: UTF-16LE (little-endian, «от младшего к старшему») и UTF-16BE (big-endian, «от старшего к младшему»).

3) *UTF-32*. Тридцатидвухбитный формат с фиксированной длиной.

   Каждый символ занимает 32 бита.

   Аналогично предыдущему формату, существуют две разновидности для двух байтовых порядков: UTF-32LE и UTF-32BE.

Наиболее распространенным в настоящее время является

© www.phys.nsu.ru

Таблица 3. Принцип кодирования UTF-8

Код символа (16-ричн.)	**Представление в UTF-8**
0–7F	0xxxxxxx
80–7FF	110xxxxx 10xxxxxx
800–FFFF	1110xxxx 10xxxxxx 10xxxxxx
10000–1FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

формат UTF-8. Единственный его недостаток — переменная длина символа, затрудняющая непосредственную работу со строками в данном формате.[8]

3.5. Нецелые числа

Во многих задачах естественным образом возникают нецелые числа, которые также нуждаются в машинном представлении.

Одним из возможных является *рациональное представление*: $M = p/q$ (p и q — целые числа). Работа с числами сводится к целочисленным операциям, выполняемым по правилам работы с дробями. Однако данное представление используется крайне редко в силу его неудобства в большинстве задач.

Альтернативным вариантом является *представление с фиксированной точкой*:

$$x = Mq, \tag{3.3}$$

где x — представляемое число, M — целое число, q — множитель. Обычно q равно либо 10^{-p}, либо 2^{-p}.

[8] Чтобы выбрать i-й символ строки, нужно, идя с начала строки, отсчитывать символы, пока не встретится i-й по счету. Для кодировок с фиксированной длиной символа достаточно просто взять символ по соответствующему адресу, как в массиве (см. разд. 3.7).

© www.phys.nsu.ru

Например, денежные величины можно хранить в целых копейках (центах и т. п.), умножая вводимые суммы на 100, а при выводе деля на 100 ($q = 10^{-2}$).[9]

Беря $q = 2^{-p}$ и используя N-битное целое число, получаем, что для хранения целой и дробной части числа используется $N-p$ и p бит, соответственно. Сложение и вычитание двух чисел x и y происходит непосредственно:

$$x + y = M_x 2^{-p} + M_y 2^{-p} = [M_x + M_y]2^{-p}; \tag{3.4}$$

$$x - y = M_x 2^{-p} - M_y 2^{-p} = [M_x - M_y]2^{-p}. \tag{3.5}$$

Умножение и деление требует дополнительной корректировки (умножения или деления на 2^p). Эти операции являются приближенными[10], ибо корректировка может привести к потере разрядов.

$$x \otimes y = (M_x 2^{-p}) \cdot (M_y 2^{-p}) = (M_x M_y)2^{-2p} = [M_x M_y 2^{-p}]2^{-p}; \tag{3.6}$$

$$x \oslash y = (M_x 2^{-p})/(M_y 2^{-p}) = M_x/M_y = [M_x M_y 2^{p}]2^{-p}. \tag{3.7}$$

Еще одной возможностью является использование двоично-десятичного кода (см. разд. 3.3), когда p младших значащих цифр числа (тетрад) используются в качестве дробной части — в этом случае $q = 10^{-p}$. [11]

Операции с такими числами также производятся по формулам (3.4)–(3.7); базовые операции с M_x и M_y — по правилам двоично-десятичной арифметики.

Представление с фиксированной точкой демонстрирует весь

[9] Центральный банк России публикует курсы валют по отношению к рублю с точностью до 4 знаков, поэтому для надежности лучше выбрать множителем 10 000 ($q = 10^{-4}$).

[10] Для обозначения приближенных арифметических операций принято использовать специальные знаки: $\oplus$, $\ominus$, $\otimes$, $\oslash$.

[11] Например, представление числа 123,45 при $p = 2$ будет соответствовать двоично-десятичному представлению числа 12345, совпадающему с двоичным представлением $12345_{16} = 0001\,0010\,0011\,0100\,0101_2$.

© www.phys.nsu.ru

спектр проблем, связанных с операциями над действительными числами в компьютере.

Во-первых, сам факт выделения фиксированного количества бит (p) на дробную часть делает представление некоторых чисел приближенным. Например, число $1/5 = 0{,}2$ в двоичной системе ($1/101_2$) является бесконечной периодической дробью (как и многие другие числа!). Следовательно, многие операции по факту также будут приближенными и могут приводить к некорректным результатам (подробнее об этом см. следующий раздел).

Во-вторых, p бит на дробную часть в слове длины N оставляет $N - p$ бит на целую часть. Получается своего рода «тришкин кафтан»: большое p дает большую точность, но оставляет мало бит для представления целой части, возможно переполнение; малое же p спасает от переполнения, но приводит к потере точности. Оптимальное значение для p в каждом конкретном случае требует аккуратного выбора, основанного на знании природы величин, участвующих в вычислениях.

В настоящее время данное представление значительно уступает по популярности представлению с плавающей точкой, рассматриваемому в следующем разделе. Преимуществом фиксированной точки, однако, является большее быстродействие и простота реализации.[12]

3.6. Числа с плавающей точкой

Часть проблем представления с фиксированной точкой может быть решена с помощью усложнения представления (3.3). Запишем

$$x = m \cdot b^e, \tag{3.8}$$

где b — основание (обычно 2 или 10), и будем представлять x в виде пары (m, e). Такая запись называется представлением числа

[12] В особенности это касается тех случаев, когда операции с плавающей точкой не поддерживаются используемым процессором аппаратно (например, в микроконтроллерах).

© www.phys.nsu.ru

с плавающей точкой, поскольку положение десятичной точки по сути задается e. Величины m и e имеют специальные названия: *мантисса (значащая часть числа, significand)* и *экспонента*. При этом мантисса *нормализована*: m и e подбираются так, чтобы $|m| < 1$. Пример такого представления для $b = 10$:

$$\begin{aligned} 1 &= 0{,}1 \times 10^1; \\ 0{,}1 &= 0{,}1 \times 10^0; \\ 314 &= 0{,}314 \times 10^3 \quad \text{и т. д.} \end{aligned}$$

Рассмотрим вычисления с плавающей точкой. Пусть даны два числа (m_1, e_1) и (m_2, e_2). Чтобы вычислить сумму

$$(m_a, e_a) = (m_1, e_1) \oplus (m_2, e_2), \tag{3.9}$$

необходимо вначале провести масштабирование, приведя оба числа к одному порядку. Принимая, что $e_1 \geq e_2$ (в противном случае числа можно просто поменять местами), сначала вычисляется сумма:

$$e_a = e_1; \tag{3.10}$$

$$m_a = m_1 + m_2/b^{e_1 - e_2}, \tag{3.11}$$

а затем результат нормализуется, чтобы выполнялось условие $|m_a| < 1$.

Вычитание производится через сложение с отрицательным знаком:

$$(m_1, e_1) \ominus (m_2, e_2) = (m_1, e_1) \oplus (-m_2, e_2). \tag{3.12}$$

Умножение и деление

$$(m_m, e_m) = (m_1, e_1) \otimes (m_2, e_2); \tag{3.13}$$

$$(m_d, e_d) = (m_1, e_1) \oslash (m_2, e_2) \tag{3.14}$$

реализуются проще, чем сложение:

$$e_m = e_1 + e_2, \qquad m_m = m_1 m_2; \tag{3.15}$$

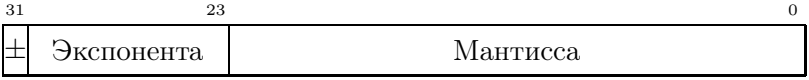
$$e_d = e_1 - e_2 + 1, \qquad m_d = (b^{-1} m_1)/m_2 \tag{3.16}$$

© www.phys.nsu.ru

(после вычислений результат также необходимо нормализовать). Подробнее о вычислениях с плавающей точкой см. [6, разд. 4.2.1].

В настоящее время в большинстве процессоров вычисления с плавающей точкой реализованы аппаратно и удовлетворяют стандарту IEEE 754-2008. Познакомимся с некоторыми вариантами представления чисел, предлагаемыми стандартом.

Типу **float** в C, как правило, соответствует число с одинарной точностью, занимающее 32 бита (в стандарте оно именуется binary32). Биты распределяются следующим образом:



то есть 1 бит на знак (S), 8 бит на экспоненту (E) и 23 бита на мантиссу (M). Определенные значения E (0 и 255) соответствуют специальным числам. Рассмотрим возможные варианты представлений.

Нормализованное число. $1 \le E \le 254$.

Обычное число с плавающей точкой. E представляет собой экспоненту с избытком 127, M — мантиссу с опущенным старшим единичным битом. Хранимое число равно

$$F = (-1)^S \cdot 2^{E-127} \cdot 1.\underbrace{mmm\ldots mm_2}_{\text{23 бита } M}. \tag{3.17}$$

Видно, что сам способ хранения предполагает нормализованное значение мантиссы ($1 \le 1.mmm < 2$).

Нетрудно вычислить минимальное (по модулю) и максимальное нормализованное число:

$$|F|_{min} = 2^{1-127} \cdot 1.000\ldots00_2 = 2^{-126} \approx 1{,}18 \cdot 10^{-38}; \tag{3.18}$$

$$F_{max} = 2^{254-127} \cdot 1.111\ldots11_2 = 2^{127} \cdot (2 - 2^{-23}) \approx 3{,}4 \cdot 10^{38}. \tag{3.19}$$

Из представления нормализованных чисел следует, что формат binary32 позволяет точно представить все *целые числа* от

© www.phys.nsu.ru

-2^{24} до $+2^{24}$. Некоторые целые числа вне этого диапазона — например, большие степени 2 — также могут быть представлены точно, но внутри указанного диапазона точно представимы *все* числа.[13]

Денормализованное число. $E = 0, M \neq 0$.

При $E = 0$ число имеет фиксированный множитель 2^{-126}:

$$D = (-1)^S \cdot 2^{-126} \cdot 0.\underbrace{mmm\ldots mm}_{\text{23 бита } M}{}_2. \tag{3.20}$$

Минимальное по модулю денормализованное число равно

$$|D|_{min} = 2^{-126} \cdot 0.\underbrace{00\ldots00}_{\text{22 нуля}}1_2 = 2^{-126} \cdot 2^{-23} = 2^{-149} \approx 1{,}4 \cdot 10^{-45}. \tag{3.21}$$

Максимальное равно

$$D_{max} = 2^{-126} \cdot 0.\underbrace{111\ldots11}_{\text{23 единицы}}{}_2 = 2^{-126}(1-2^{-23}) \approx 1{,}18 \cdot 10^{-38}. \tag{3.22}$$

Видно, что диапазоны нормализованных и денормализованных чисел граничат между собой ($D_{max} = |F|_{min} - |D|_{min}$).

Из структуры числа binary32 с опущенным старшим битом (как следует из описания, он равен 1 у нормализованных чисел и 0 — у денормализованных) следует, что количество значащих десятичных цифр равно

$$\log_{10}(2^{23+1}) \approx 7{,}225,$$

то есть около 7 цифр.

[13] Это свойство иногда используется: в языке JavaScript *все* числа хранятся в формате с плавающей точкой binary64, описываемом далее.

© www.phys.nsu.ru

Ноль. $E = M = 0$.

В зависимости от значения S, такое двоичное представление дает либо $+0$, либо -0. Следует заметить, что, поскольку представление чисел с плавающей точкой является приближенным, знак нуля имеет значение.[14]

Бесконечность. $E = 255$, $M = 0$.

Этот вариант дает два числа $+\infty$ и $-\infty$. Бесконечность может возникнуть в результате деления на 0 и некоторых других операций:

$$1/0 = \infty; \qquad -1/0 = -\infty;$$
$$\operatorname{arctg} \pi/2 = \infty; \qquad \operatorname{arctg}(-\pi/2) = -\infty.$$

Не-число (NaN, Not a Number). $E = 255$, $M \neq 0$.

NaN возникает в результате вычислений типа $\sqrt{-1}$, ∞/∞, $0/0$ и других, где результат либо не определен, либо не принадлежит множеству действительных чисел.

Значение M не играет роли, но старший бит M (22-й в слове) определяет, является ли NaN «тихим» (бит 0) или «сигнальным» (бит 1). «Тихий» NaN участвует в вычислениях, «сигнальный» приводит к исключению, как только становится аргументом какой-либо операции.

Из всех чисел x, представляемых в формате IEEE 754, только NaN удовлетворяет условию $x \neq x$, то есть NaN не равен сам себе!

Кроме чисел с одинарной точностью, стандарт также определяет числа с двойной точностью (формат binary64, соответствует типу **double** в C). Эти числа во всем похожи на числа с одинарной точностью, но отводят на мантиссу и экспоненту больше бит: 52 и

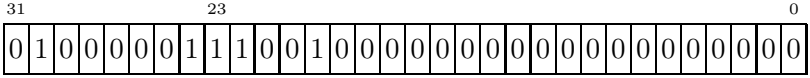
[14] Если взять самое большое число с отрицательным знаком, -2^{-149}, и еще уменьшить его (поделить на 2), мы получим настолько малое по модулю число, что оно непредставимо в формате binary32 и, следовательно, будет округлено до нуля. Но знак результата будет отрицательным: $-2^{-149}/2 \cong -0$.

© www.phys.nsu.ru

11, соответственно. Экспонента нормализованного числа хранится с избытком 1023, множитель для денормализованных чисел равен 10^{-1022}, у бесконечностей и NaN $E = 2047$ (11 единичных бит).

В таблице 4 сведены различные параметры для чисел одинарной и двойной точности.

Рассмотрим, как происходит раскодирование битового представления на примере числа с одинарной точностью, имеющего представление $41C80000_{16}$. Запишем биты:



Выделим знак $S = 0$ и экспоненту $E = 1000\,0011_2 = 131$. Значение E указывает на то, что число является нормализованным.

Дополним мантиссу старшим единичным битом, отбрасывая младшие нули:

$$M' = 1{,}1001_2 = 1 + 2^{-1} + 2^{-4} = 1{,}5625.$$

Вычитая избыток из экспоненты, получаем $E - 127 = 4$. Остается подсчитать

$$X = M' \cdot 2^4 = 1{,}5625 \cdot 16 = 25.$$

То есть битовое представление $41C80000_{16}$ соответствует числу 25_{10}.

Числа с плавающей точкой более удобны для работы, чем числа с фиксированной точкой, они больше «прощают», реже приводя к потере точности. Но и у них есть особенности, которые следует знать. Рассмотрим некоторые из них (более подробный анализ см. в [6, разд. 4.2.2] и в [17]).

1) $a \underbrace{\oplus b \oplus \cdots \oplus b}_{n \text{ раз}} \neq a \oplus nb.$

   Каждое сложение потенциально может приводить к потере точности. Если $a = 0$, $b = 1/3$, $n = 1000$, а вычисления

© www.phys.nsu.ru

Таблица 4. Параметры чисел одинарной (binary32) и двойной (binary64) точности

Параметр	**binary32**	**binary64**
Всего бит	32	64
Бит на мантиссу	23	52
Бит на экспоненту	8	11
Бит на знак	1	1
Количество значащих десятичных разрядов	$N_d = \log_{10}(2^{24}) \approx 7{,}225$	$N_d = \log_{10}(2^{53}) \approx 15{,}955$
Нормализованное число	$(-1)^S \cdot 2^{E-127} \cdot 1.mmm$	$(-1)^S \cdot 2^{E-1023} \cdot 1.mmm$
Денормализованное число	$(-1)^S \cdot 2^{-126} \cdot 0.mmm$	$(-1)^S \cdot 2^{-1022} \cdot 0.mmm$
$\lvert D\rvert_{min}$	$2^{-149} \approx 1{,}4 \cdot 10^{-45}$	$2^{-1074} \approx 5 \cdot 10^{-324}$
D_{max}	$2^{-126}(1 - 2^{-23}) \approx 1{,}18 \cdot 10^{-38}$	$2^{-1022}(1-2^{-52}) \approx 2{,}23 \cdot 10^{-308}$
$\lvert F\rvert_{min}$	$2^{-126} \approx 1{,}18 \cdot 10^{-38}$	$2^{-1022} \approx 2{,}23 \cdot 10^{-308}$
F_{max}	$2^{127} \cdot (2 - 2^{-23}) \approx 3{,}4 \cdot 10^{38}$	$2^{1023} \cdot (2-2^{-52}) \approx 1{,}80 \cdot 10^{308}$
Диапазон точно представимых целых чисел	$-2^{24} \ldots 2^{24} = 16\,777\,216$	$-2^{53} \ldots 2^{53} \approx 9{,}0 \cdot 10^{15}$

© www.phys.nsu.ru

производятся с одинарной точностью, то первые 7 значащих цифр левой части будут равны 333,3341, а у правой — 333,3333. Правая часть соответствует истинному значению 1000/3, а левая отклонилась от него в двух знаках. Чем меньше b и больше n, тем отклонение будет заметнее.

2) $a \otimes a \otimes \cdots \otimes a = a^n \neq \mathrm{e}^{n \ln a}$.

Аналогично п. 1, только здесь накопление ошибки может происходить с каждым умножением. Если a близко к единице, умножения будут приводить к потере младших разрядов. Экспоненциальная формула в правой части в таком случае даст более точный результат.

3) *Сравнение чисел.*

Прямое сравнение двух чисел с плавающей точкой, скорее всего, приведет к нежелательным результатам.

Вместо $x = y$ лучше осуществлять сравнение $|x \ominus y| < \epsilon$, где ϵ — достаточно малое число. Поскольку x и y являются лишь приближенными значениями неких величин x_0 и y_0, равенство последних ($x_0 = y_0$) отнюдь не означает $x = y$.

Например, вычисленные нами числа D_{max} и $|F|_{min}$ по величине отличаются на $|D|_{min}$, то есть на самое малое по модулю число, отличное от 0! С точки зрения пользователя эта разница наверняка будет пренебрежимо малой. И, тем не менее, они отличаются друг от друга, имея разное битовое представление.

Величина ϵ, влияющая на точность результата, обычно определяется из условий решаемой задачи и, как правило, должна соответствовать точности входных данных.

3.7. Массивы

В практических задачах очень часто возникают упорядоченные наборы однотипных данных. Вектор в математике — набор

© www.phys.nsu.ru

чисел (целых, нецелых, комплексных, но — одинаковых по типу). Временной ряд — набор чисел или пар (число, значение). В большинстве ЯВУ для работы с такими объектами используются *массивы*.

В статических языках программирования (C, Паскаль, Фортран и др.) массивы, как правило, имеют фиксированный размер, задающийся при компиляции, и содержат элементы фиксированного типа. Доступ к элементам осуществляется с помощью целочисленных индексов. При этом массивы могут быть одномерными (для доступа необходимо одно значение индекса), так и многомерными — когда требуется несколько индексов.

Язык C поддерживает одно- и многомерные массивы с индексами, начинающимися с нуля:

```
int a[10];
double b[10][20];
```

В приведенном примере a — одномерный массив из 10 целых чисел, к которым можно обращаться a[0], ..., a[9]; b — двумерный массив из 200 чисел с плавающей точкой двойной точности: a[0][0], ..., a[9][19].

Массивы всегда занимают сплошной участок памяти размером $M \times s$, где s — размер одного элемента массива, а $M = N_K \times \cdots \times N_2 \times N_1$; K — количество размерностей, N_k — количество индексов в k-й размерности.

В языке C массивы упорядочиваются в памяти от младших индексов к старшим. Элементы массива **char** b[3][4] разместятся в памяти следующим образом: сначала 4 элемента b[0][0], ..., b[0][3], затем пойдут b[1][0], ..., b[1][3] и, наконец, b[2][0], ..., b[2][3].

Решим задачу размещения элементов в общем виде. Пусть дан массив с K размерностями, каждая содержит N_k индексов, размер одного элемента s. Массив размещен в памяти, начиная с адреса A_0. По какому адресу будет находится элемент массива с индексами $[n_K, n_{K-1}, \ldots, n_1]$, где $0 \le n_k < N_k$?

© www.phys.nsu.ru

Случай $K = 1$ тривиален:

$$A(n_1) = A_0 + n_1 s. \tag{3.23}$$

При $K = 2$, очевидно,

$$A(n_2, n_1) = A(n_2, 0) + n_1 s. \tag{3.24}$$

Перед элементом $(n_2, 0)$ будут лежать элементы с индексами $[n_2 - 1, 0 \dots N_1 - 1]$, перед ними — $[n_2 - 2, 0 \dots N_1 - 1]$ и т. д., вплоть до $[0, 0 \dots N_1 - 1]$. Общее количество элементов, лежащих до $[n_2, 0]$, очевидным образом, равно $n_2 N_1$. Следовательно,

$$A(n_2, n_1) = A_0 + (n_2 N_1 + n_1) s. \tag{3.25}$$

Рассуждая аналогичным образом, получим для $K = 3$

$$A(n_3, n_2, n_1) = A_0 + ((n_3 N_2 + n_2) N_1 + n_1) s \tag{3.26}$$

и в общем случае

$$A(n_K, \dots, n_1) = A_0 + s \sum_{k=1}^{K} n_k \prod_{m=1}^{k-1} N_m. \tag{3.27}$$

Характерно, что количество индексов в самой старшей размерности, N_K, не фигурирует в полученной формуле. В частности, это позволяет писать в языке C конструкции вида

```
extern double arr[][100];
```

Полученная формула полезна в тех случаях, когда требуется работа с динамическими массивами переменной размерности.

3.8. Строки

Строка — это массив символов переменной длины. Чтобы обозначать конец строки, используется два основных приема.

© www.phys.nsu.ru

Первый состоит в том, чтобы выделить один из кодов символов и назначить его маркером конца строки. В языке C для этого используется нулевой байт (`'\0'`).

Второй прием предполагает хранение длины отдельно от самой строки, например так:

```
struct String {
    char *p;
    int len;
};
```

Отдельное хранение длины убыстряет большинство строковых операций, ибо не требуется каждый раз заново вычислять длину строки. Кроме того, нет ограничений на вхождение символов. Однако стандартная библиотека C по сей день ориентирована на строки, заканчивающиеся нулевым байтом.[15]

В 1994 г. был предложен еще один способ хранения строк, в виде *«веревок» (ropes)*.[16] [16] В этом случае строка хранится как двоичное дерево (см. разд. 6), где листьями дерева являются части строк (они являются неизменяемыми), а каждый внутренний узел обозначает операцию конкатенации поддеревьев (объединения строк).

Пример «веревочного» представления приведен на рис. 1. Видно, что вывод полной строки требует обхода дерева «слева направо» (см. разд. 6.3). Конкатенация двух строк осуществляется простым созданием нового корня, ссылающегося на две объединяемые строки.

3.9. Множества

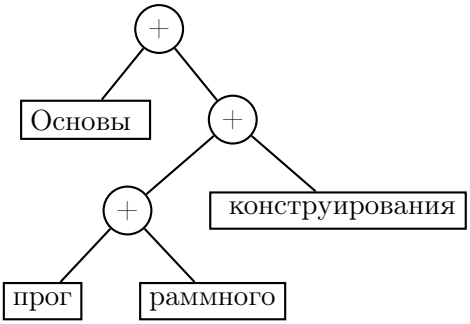
Множества из фиксированного набора элементов легко представляются с помощью набора битов, где каждому элементу поставлен в соответствие один из битов, а значение бита (0 или 1)

[15] В C++ стандартный строковый класс std::string хранит длину отдельно.

[16] Это название, по всей видимости, происходит от принципа связывания между собой кусков веревки для получения более длинной веревки. — *Прим. автора.*

© www.phys.nsu.ru

Рис. 1. Представление строки «Основы программного конструирования» в виде «веревок»



отражает присутствие данного элемента в множестве.

Рассмотрим пример. Пусть необходимо сопоставить некоторым объектам множества признаков из следующего набора: *мягкий*, *круглый*, *вытянутый*, *съедобный*. Выберем для хранения множества тип **int** и сопоставим элементам набора биты с 0-го по 3-й:

```
enum SetElements {
    S_SOFT     = 1,
    S_ROUND    = 2,
    S_EXTENDED = 4,
    S_EDIBLE   = 8
};
```

Значения констант равны 2^n, где n — номер бита.

Теперь множество может задаваться с помощью побитовой операции ИЛИ (|):

```
/* яблоко */
int apple  = S_ROUND | S_EDIBLE;
/* банан */
int banana = S_SOFT | S_EXTENDED | S_EDIBLE;
/* палка */
int stick  = S_EXTENDED | S_ROUND;
```

Добавление элементов в множество осуществляется аналогично:

© www.phys.nsu.ru

```
apple |= S_SOFT;
```

(яблоко поспело и стало мягким)

Проверить наличие элемента в множестве можно с помощью побитового И (&):

```
if (object & S_EDIBLE) {
    /* ... */
}
```

(если объект съедобен, то...)

Удаление элемента из множества происходит с помощью побитового И с инвертированной маской:

```
apple &= ~S_EDIBLE;
```

(яблоко стало несъедобным)

При пополнении списка возможных признаков нужно лишь добавить соответствующие константы в **enum**, назначив им значения следующих степеней двойки: 16, 32 и т. д.

Подробную информацию о побитовых операциях можно найти в [13].

© www.phys.nsu.ru

Глава 4

Динамические структуры данных

4.1. Статическое и динамическое размещение данных

Большинство типов данных, описанных в предыдущем разделе, относятся к *статическим*. Объявляя переменную типа **int**, мы заранее знаем, что она будет занимать 32 бита.[1] Тем самым мы предполагаем, что числа, попадающие в эту переменную, будут заведомо меньше максимального числа, представимого в 32 битах. Объявляя массив, мы указываем его точный размер, также предполагая, что данные уместятся в нем, и т. д.

Статический подход к размещению данных можно кратко сформулировать так: обладая априорной информацией, мы *заранее предполагаем*, сколько места в памяти понадобится для хранения данных. Это может быть точная оценка или оценка сверху. Например, максимальная длина имени файла не может превышать значение, задаваемое константой FILENAME_MAX из заголовочного файла `stdio.h`. Таким образом, хранить строку с именем файла можно в символьном массиве:

```
char filename[FILENAME_MAX+1];
```

Обрабатывая в C ввод с клавиатуры, можно ограничить строку каким-то «разумным» значением (например, 1024) и использовать для ввода функцию fgets, которая не дает строке переполниться:

[1] Это число может варьироваться в зависимости от архитектуры.

© www.phys.nsu.ru

```
char input[1024];

fgets(input, sizeof (input), stdin);
```

Здесь, если пользователь введет строку более 1023 символов, она будет обрезана, чтобы уместиться в массив input, включая нулевой символ (конец строки).

Статический подход позволяет решать многие задачи, в том числе некоторые из тех, что работают с данными, обладающими динамической природой. Например, может показаться, что обработка файлов не может быть выполнена в статическом подходе, ибо входной файл может иметь произвольный размер (от 0 до очень большого числа), следовательно, нельзя в программе завести такой массив, чтобы туда заведомо поместился файл целиком. Однако в большинстве случаев результат обработки одной части файла не зависит от другой части файла! Из этого следует, что файл можно загружать в память (массив фиксированного размера) частями. В частности, большинство программ сжатия данных грузят файл в память блоками размером до мегабайта. Более того, в некоторых задачах достаточно посимвольного чтения файла: для подсчета количества слов в файле (разделенных пробельными символами) одновременно нужно хранить в памяти лишь два символа: текущий и предыдущий.

Тем не менее, существуют задачи, где статический подход пасует и нужно размещать данные динамически, то есть в структуре переменного размера. Простейшей такой структурой является динамический массив, рассматриваемый в следующем разделе. Более сложным структурам посвящены дальнейшие разделы.

4.2. Динамические массивы

Динамический массив в C представляется с помощью указателя на его первый элемент. Память для массива выделяется из «кучи» с помощью функций malloc и realloc, а освобождается с помощью функции free (функции объявлены в заголовочном файле

© www.phys.nsu.ru

stdlib.h).

В примере

```
void f(int n) {
    int *ary = (int *)malloc(sizeof (int) * n), i;

    for (i = 0; i < n; ++i)
        ary[i] = i;

    if (ary) {
        do_something(ary, n);
        free(ary);
    } else {
        /* error: out of memory */
    }
}
```

функция f(**int**) выделяет динамическую память под массив из n чисел типа **int**, заполняет его значениями и передает в функцию do_something, затем освобождает память.

Пример показывает, что по сравнению со статическим размещением данных динамическое порождает следующие проблемы:

1) *Динамическую память необходимо явно выделять.*

   Если не вызвать функцию malloc, это заведомо приведет к ошибкам. Кроме того, необходимо корректно подсчитать размер выделяемой памяти (если в вызове malloc из примера забыть умножить n на **sizeof**(**int**), будет выделено памяти меньше, чем нужно, что опять же приведет к ошибкам).

2) *Выделение памяти может окончиться неудачей.*

   Функция malloc может вернуть нулевой указатель в случае, если памяти недостаточно. Поскольку использование нулевого указателя немедленно приведет к аварийному завершению работы программы, необходимо проверять возвращаемое значение и как-то обрабатывать такую ситуацию (например, сохранить пользовательские данные в файл и завершить исполнение).

© www.phys.nsu.ru

3) *Динамическую память необходимо освобождать.*

Если не вызывать free, программа будет занимать все больше и больше памяти, не используя ее, что также приводит к плохим последствиям (постепенно исчерпание физической памяти и дисковой памяти подкачки). Для программ, работающих в течение долгого времени (сутками, неделями, месяцами...), это может являться проблемой.

Также возможна ситуация, когда уже освобожденная память продолжает использоваться. Это приводит к весьма трудно уловимым ошибкам.[2]

Во многих современных системах программирования реализована концепция «сборки мусора», когда программа лишь выделяет память, а освобождение происходит автоматически (подробнее см. у Кнута в [5, разд. 2.3.5]). Но даже это решает лишь проблему № 3, добавляя вместо нее другие (например, процесс «сборки мусора» может начаться в неподходящий момент). Так или иначе, в языке C освобождение памяти происходит вручную.

Вернемся к динамическим массивам. Выше был рассмотрен случай, когда размер массива известен — хоть и не во время компиляции программы (тогда можно было бы использовать статический массив), но во время ее работы. Однако встречаются ситуации, когда требуется изменение размера прямо в процессе использования массива.

Рассмотрим задачу чтения строки произвольного размера. Чтение происходит посимвольно, и заранее знать, сколько будет символов, невозможно, поэтому память под строку придется увеличивать прямо по ходу чтения, с помощью функции realloc.

На листинге 4.1 приведена функция safe_gets, реализующая такой принцип. Однако она является не очень эффективной, вызывая функцию realloc на каждый символ.[3] Эффективность

[2] Для избежания этого рекомендуется после вызова free явно занулять указатели: free(p), p=0;.

[3] Функция realloc пытается либо изменить размер выделенного блока, ли-

© www.phys.nsu.ru

Листинг 4.1. Чтение строки произвольного размера

```
char *safe_gets()
{
    int sz = 1;
    char *s = (char *)malloc(sz);

    if (!s)
        return 0;

    for (;;) {
        int c = getchar();

        if (c == EOF)
            break;

        s = (char *)realloc(s, ++sz);
        s[sz-2] = c;

        if (c == '\n')
            break;
    }

    s[sz-1] = '\0';
    return s;
}
```

может быть повышена, если выделять память квантами. Листинг 4.2 отражает исправленную функцию safe_fgets. Константа SZ_INITIAL задает начальный размер выделяемой памяти; если строка превышает этот размер, память будет приращиваться каждый раз по SZ_DELTA байт.

После использования строку, возвращаемую safe_gets, необхо-

бо, если такой возможности нет, выделяет новый блок и копирует туда данные из предыдущего, впоследствии освобождая его. При неудачном стечении обстоятельств строка будет копироваться каждый раз при вызове realloc.

© www.phys.nsu.ru

Листинг 4.2. Модификация функции safe_gets с выделением памяти квантами

```
#define SZ_INITIAL 10
#define SZ_DELTA   10

char *safe_gets()
{
    int sz = SZ_INITIAL, l = 1;
    char *s = (char *)malloc(sz);

    if (!s)
        return 0;

    for (;;) {
        int c = getchar();

        if (c == EOF)
            break;

        if (l >= sz)
            s = (char *)realloc(s, sz += SZ_DELTA);

        s[l-1] = c;
        ++l;

        if (c == '\n')
            break;
    }

    s[l-1] = '\0';
    return s;
}
```

© www.phys.nsu.ru

димо освободить с помощью вызова free.

Выше рассматривались лишь одномерные динамические массивы. Для работы с многомерными массивами потребуется вычисление индексов по формуле (3.27).

4.3. Абстрактные типы данных

Как уже упоминалось в разделе 1.2, компьютер может работать только с теми данными, которые представлены в двоичном виде, и задача программиста состоит в том, чтобы выбрать подходящее представление и преобразовать в него входные данные.

С каждым типом данных связаны свои, вполне определенные операции. Например, с числами возможны разнообразные арифметические и побитовые операции. Над булевыми значениями можно производить операции из булевой алгебры: логическое И, ИЛИ и др. Строки можно складывать между собой, вычислять длину, получать символ по номеру. Массивы можно индексировать, читая или записывая элемент по заданному индексу, и т. д.

Следовательно, с одной стороны имеем *машинное представление данных* (гл. 3). С другой стороны, любым данным соответствует *набор возможных операций* над ними, который связан не с машинным представлением[4], а с природой самих данных. Имея операции, реализованные в виде встроенных в язык операторов или отдельных функций с аргументами, мы можем полностью абстрагироваться от машинного представления и работать с *данными*, а не с битами и байтами.

Но и это еще не все. Отделяя структуру данных от содержания, получаем еще один уровень абстракции. Простейшим примером здесь может служить обычный массив. Его структура (фиксированный размер, последовательное расположение элементов,

[4]Наоборот, машинное представление данных получается как следствие требуемого набора операций, с учетом эффективности по памяти и быстродействию.

© www.phys.nsu.ru

обращение к элементам по индексам) никоим образом не зависит от типа данных. Символьные, целочисленные, массивы структур или массивы массивов ведут себя совершенно одинаково. Можно сказать, что понятие «массив» абстрагировано от конкретных типов данных, которые он может в себе хранить.

Таким образом приходим к *абстрактному типу данных* (АТД). Определение из [11] гласит:

> Абстрактный тип данных — это тип данных (набор значений и совокупность операций для этих значений), доступ к которому осуществляется только через *интерфейс*.

При этом внутренняя структура АТД и его реализация «спрятаны» от программиста. Например, чтобы работать с массивом, совсем не обязательно знать формулу (3.27), она спрятана внутри компилятора, в качестве интерфейса же выступает обращение к элементам с помощью целочисленных индексов.

Понятие АТД является весьма мощным инструментом, позволяющим достичь модульности программы.

Одним из возможных АТД является *стек*, который может хранить произвольное количество элементов с методом доступа к ним «последним пришел, первым вышел», или LIFO[5]. Часто стек сравнивают со стопкой тарелок, у которой свободно помещать и снимать тарелки—«элементы» можно только сверху.

Со стеком возможны следующие операции:

1) Поместить элемент на вершину стека.

2) Проверить, есть ли в стеке хотя бы один элемент.

3) Получить элемент, находящийся на вершине.

4) Удалить элемент с вершины.

[5]Last In First Out (англ.).

© www.phys.nsu.ru

Это и есть *интерфейс* стека. Он абстрагирован как от реализации (не указано, как именно будут храниться, добавляться и удаляться элементы), так и от типа данных (не указано, какого типа элементы будут использоваться).

Принцип стека является полезной и удобной абстракцией, часто возникая в задачах с явной или неявной рекурсией.

На стек похожа *очередь*, также хранящая элементы и предоставляющая к ним доступ по принципу «первым пришел, первым вышел», или FIFO[6]. АТД «Очередь» обладает следующим интерфейсом:

1) Поместить элемент в конец очереди.

2) Проверить, есть ли в очереди хотя бы один элемент.

3) Получить элемент, находящийся в начале очереди.

4) Удалить элемент, находящийся в начале очереди.

Конкретная реализация АТД «Стек» и «Очередь» может быть разной. Если максимальное количество элементов известно и невелико, можно обойтись статическим массивом. Если это условие не удовлетворено, следует использовать динамический массив (см. разд. 4.2) или одну из разновидностей связного списка (гл. 5).

[6]First In First Out (англ.).

© www.phys.nsu.ru

Глава 5

Списки

5.1. Связные списки

Одним из фундаментальных АТД является *связный список*. Список — это упорядоченная структура данных, состоящая из произвольного количества элементов, где каждый элемент связан со следующим, кроме последнего *(хвоста списка)*, при этом циклы не допускаются. Первый элемент называется *головой списка* и полностью определяет собой список, поскольку остальные элементы последовательно доступны через связи.

Структура связного списка изображена на рис. 2, а интерфейс АТД «Список» таков:

1) Вставка элемента в голову списка.

2) Вставка элемента *после заданного*.

3) Удаление элемента *после заданного*.

4) Обход списка (применение заданной операции ко всем элементам).

5) Поиск элемента (поиск первого элемента, начиная с головы, для которого определенное условие будет истинным).

В сравнении с динамическим массивом список позволяет

© www.phys.nsu.ru

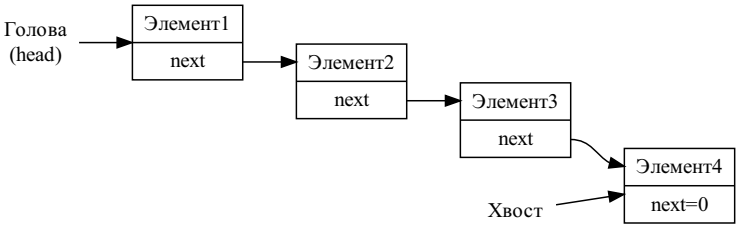


Рис. 2. Структура связного списка

очень быстро (за константное время) выполнять вставку и удаление элементов из середины[1], но поиск элемента по номеру n происходит медленнее (понадобится n раз пройти по связям).

Связные списки были изобретены в 1955–56 гг. для решения задач, связанных с искусственным интеллектом. Позднее они стали одним из базовых типов в языке Лисп.

Обычно для реализации списка используется динамическая память. В C элемент списка объявляется с помощью ключевого слова **struct**; структура содержит в себе указатель на следующий элемент и данные, соответствующие самому элементу. Например, список строк может быть реализован на базе следующей структуры:

```
struct StringList {
    char *string;
    struct StringList *next;
};
```

Для реализации списков из произвольных типов данных в C можно либо отдельно выделять память, храня бестиповый указатель:

[1] В массиве потребуется копировать элементы на одну позицию, начиная с места вставки/удаления до конца массива.

© www.phys.nsu.ru

```
struct List {
    void *data;
    struct List *next;
}; // ,
```

либо выделять дополнительную память в рамках **struct** List:

```
struct List {
    struct List *next;
    char data[1];
};

struct List *list_create_element(void *data, int size)
{
    struct List *el = (struct List *)
        malloc(sizeof(struct List)-1+size);

    if (el) {
        el->next = 0;
        memcpy(el->data, data, size);
    }

    return el;
}
```

В рамках этого раздела будет развиваться последний подход.

На листинге 5.1 приведен заголовочный файл с интерфейсом для списка.

Реализация функций list_insert_head и list_insert_after является очевидной (см. листинг 5.2). Первая из них возвращает новую голову списка, которой становится элемент new_el. Вторая вставляет new_el так, что он оказывается между элементами after и after->next.

Функция удаления, которой передается *предыдущий* элемент перед удаляемым, исправляет указатель after->next, исключая тем самым удаляемый элемент из цепи. Он также становится возвращаемым значением, что позволяет освобождать память таким образом:

```
free(list_delete_after(el));
```

© www.phys.nsu.ru

Листинг 5.1. Интерфейс списка (`list.h`)

```
typedef struct List {
    struct List *next;
    char data[1];
} List;

List *list_create_element(void *data, int size);
List *list_insert_head(List *head, List *new_el);
void list_insert_after(List *after, List *new_el);
List *list_delete_after(List *after);
void list_traverse(List *head,
                   void (*)(List *, void *), void *);
List *list_find(List *head,
                int (*)(List *, void *), void *);
```

Листинг 5.2. Функции вставки и удаления из связного списка

```
List *list_insert_head(List *head, List *new_el) {
    new_el->next = head;
    return new_el;
}

void list_insert_after(List *after, List *new_el) {
    new_el->next = after->next;
    after->next = new_el;
}

List *list_delete_after(List *after) {
    List *del = after->next;
    if (del) {
        after->next = del->next;
        del->next = 0;
    }
    return del;
}
```

© www.phys.nsu.ru

Листинг 5.3. Функции обхода и поиска в списке

```
void list_traverse(List *head,
    void (*func)(List *e, void *arg), void *arg) {

    while (head) {
        List *next = head->next;
        func(head, arg);
        head = next;
    }
}

List *list_find(List *head,
    int (*predicate)(List *e, void *arg), void *arg) {

    for (; head; head = head->next)
        if (predicate(head, arg))
            return head;

    return 0;
}
```

Если в функцию удаления будет передан последний элемент списка, то переменная del будет равна 0 и функция вернет нулевой указатель.

На листинге 5.3 приведены функции обхода list_traverse и поиска элементов list_find. Сами по себе они не делают никаких операций над элементами, делегируя это функциям-обработчикам, которые вызываются по указателям.

Рассмотрим, как их можно использовать. С помощью функции list_traverse, например, можно подсчитать количество элементов в списке — см. функцию list_size на листинге 5.4. Третий аргумент функции обхода передается в качестве аргумента пользовательской функции, вызываемой по указателю (здесь это функция _list_counter), что позволяет накапливать результат в локальной переменной size.

© www.phys.nsu.ru

Листинг 5.4. Функции подсчета количества элементов в списке и уничтожения списка

```
static void _list_counter(List *el, void *arg) {
    ++*(int *)arg;
}

int list_size(List *head) {
    int size = 0;

    list_traverse(head, _list_counter, &size);

    return size;
}

static void _list_free_el(List *el, void *arg) {
    free(el);
}

void list_destroy(List *head) {
    list_traverse(head, _list_free_el, 0);
}
```

© www.phys.nsu.ru

Листинг 5.5. Поиск строки в списке

```
static void _string_cmp(List *el, void *arg) {
    return !strcmp(el->data, (char *)arg);
}

int find_secret(List *head) {
    return !!list_find(head, _string_cmp, "secret");
}
```

«Осторожная» реализация внутреннего цикла в list_traverse (указатель на следующий элемент сохраняется перед вызовом пользовательской функции) позволяет на базе обхода реализовать и уничтожение списка (см. функцию list_destroy на листинге 5.4).

Функция list_find получает в качестве аргумента *предикат* — функцию, которая должна вернуть ненулевое значение, если переданный ей элемент списка соответствует критерию поиска. Тогда list_find вернет указатель на этот элемент. Если же на все элементы функция-предикат вернула 0, list_find также вернет 0, что будет означать «элемент не найден».

На листинге 5.5 приведен пример поиска строки в списке строк. Функция find_secret вернет 1 или 0, в зависимости от того, существует в списке строка **"secret"** или нет.

Интересный пример использования связных списков демонстрирует файловая система (ФС) FAT фирмы Microsoft. Название ФС дала структура данных, лежащая в ее основе — таблица размещения файлов (File Allocation Table, или FAT). В одной из первых версий данной ФС, FAT16, все доступное место на диске делится на кластеры (один кластер может иметь размер от 512 байт до 64 Кбайт) так, что общее их количество не превышает $65\,517 = 2^{16} - 19$. Сама таблица FAT состоит из 2^{16} 16-битных записей, соответствующих кластерам. Если кластер свободен, соответствующая позиция в таблице содержит 0. Если кластер при-

© www.phys.nsu.ru

надлежит какому-то файлу, то позиция в FAT содержит номер *следующего* кластера, принадлежащего тому же самому файлу, или специальное значение, если кластер последний. В каталоге, где хранятся имена файлов, хранится также первый кластер, принадлежащий файлу.

На рис. 3 приведен пример таблицы размещения файлов. На «диске» всего 7 кластеров. Каталог содержит три файла: `a.txt` (цепочка кластеров $3 \to 6 \to 4$), `b.txt` (цепочка $2 \to 5$), `c.txt` (кластер № 7). Кластер № 1 свободен.

Таким образом, таблица FAT — это целое множество связных списков, по списку на каждый файл.

5.2. Циклические и двусвязные списки

Связные списки обладают одним недостатком: они позволяют двигаться по связям только в одну сторону. Простейшим способом исправить положение является *циклический список*, у которого хвост ссылается на голову (см. рис. 4). Это дает возможность, начиная с любого элемента, получить доступ ко списку целиком. Также повышается степень симметрии структуры списка, и при работе со списком можно не заботиться о том, где находится последний или первый узел. Кроме того, некоторые задачи сами по себе обладают циклической природой: например, список может использоваться для хранения вершин многоугольника, где каждая вершина связана со следующей, но среди них нельзя явно выделить ни первую, ни последнюю.

Поддержка циклических списков требует внесения изменений в реализацию. Так, в функциях обхода и поиска условие выхода по достижению последнего элемента должно быть заменено на условие «совершения полного круга», когда текущим элементом становится тот, который служил отправной точкой.

Если задавать циклический список не через первый, а через *последний* элемент, как это предложено Кнутом [5, разд. 2.2.4], то появляется возможность простой вставки и в хвост, и в голову списка, что позволит легко реализовать на базе такого списка

© www.phys.nsu.ru

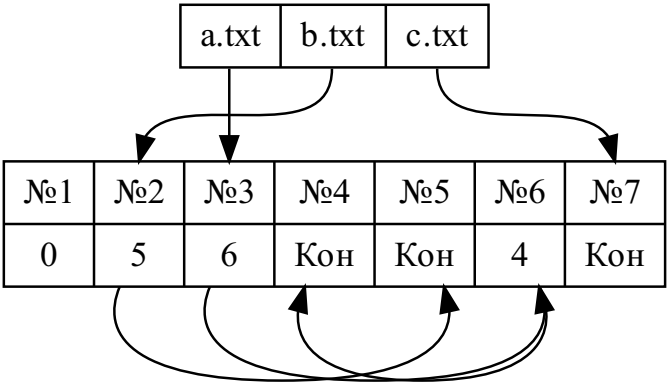


Рис. 3. Пример таблицы размещения файлов

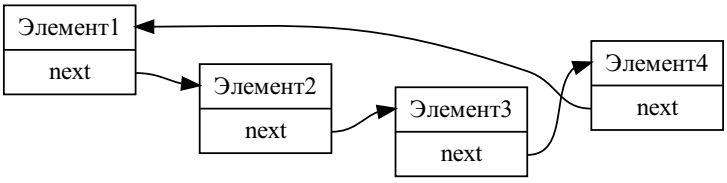


Рис. 4. Циклический список

© www.phys.nsu.ru

АТД «Стек» и «Очередь».

Другое решение проблемы «односторонности» связного списка состоит в том, чтобы сделать его *двусвязным*, добавив связи в обратную сторону (см. рис. 5). Двусвязный список занимает больше памяти, но предоставляет больше удобства в работе. Например, чтобы удалить элемент X из такого списка, достаточно иметь ссылку на X, а не на предыдущий элемент. Кроме того, двусвязный список позволяет вставлять элемент не только справа от заданного, но и слева, а также совершать обход списка в обратном порядке (от хвоста к голове).

И, наконец, последней возможностью является *циклический двусвязный список* (см. рис. 6), являющийся самым сложным среди прочих видов списков, но и предоставляющий максимум возможностей.

Более подробную информацию о списках можно найти в [2, разд. 4.3], [5, разд. 2.2], [8, разд. 10.2] и [11, разд. 3.3].

© www.phys.nsu.ru

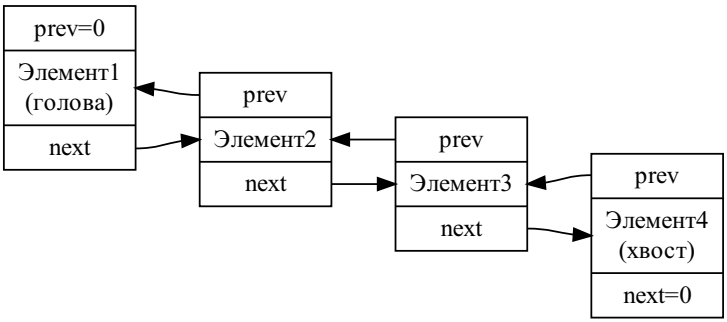


Рис. 5. Двусвязный список

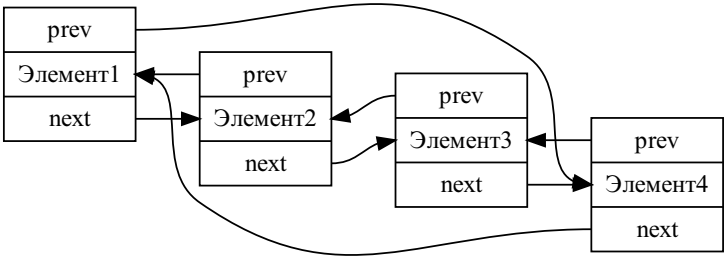


Рис. 6. Циклический двусвязный список

© www.phys.nsu.ru

Глава 6

Деревья

6.1. Определения

Наиболее важной нелинейной структурой, которая встречается при работе с компьютерными алгоритмами, является *дерево*. Древовидная структура задает для узлов отношение «ветвления», которое во многом напоминает строение обычного дерева.

Существенное свойство дерева состоит в том, что оно является рекурсивной структурой данных, что фактически следует из его определения [5]:

Дерево — конечное множество T одного или более узлов со следующими свойствами:

1) существует один выделенный узел, а именно — *корень (root)* данного дерева T;

2) остальные узлы (за исключением корня) распределены среди $m \geq 0$ непересекающихся множеств $T_1, \ldots, T_m$, и каждое из этих множеств, в свою очередь, является деревом; деревья $T_1, \ldots, T_m$ называются *поддеревьями* данного корня.

Узлы, у которых нет поддеревьев, называются *листьями*.[1] Прочие узлы называются *внутренними*, или *узлами ветвления*.

Для обозначения отношений между узлами в дереве обычно используется терминология, заимствованная из генеалогических

[1] Это вполне соответствует деревьям в природе.

© www.phys.nsu.ru

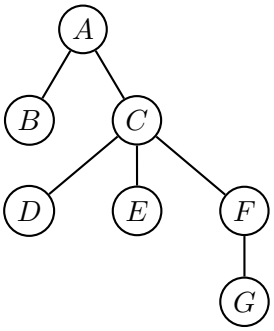


Рис. 7. Пример дерева

деревьев (родовых схем). Каждый узел называется *родителем (parent)* корней его поддеревьев, а сами корни называются *братьями—сестрами (siblings)*, а также *детьми (children)*, или *дочерними узлами* своего родителя. Из определений следует, что у узлов—листьев нет детей.

Вводится также понятие *уровня узла*. Если какой-то узел имеет уровень l, то его дочерние узлы имеют уровень $l + 1$; уровень корня дерева принимается равным 0. *Высотой* дерева называется количество различных уровней узлов, присутствующих в нем.

Проиллюстрируем введенные понятия с помощью рис. 7. Узел A является корнем[2] дерева, изображенного на рисунке, с двумя поддеревьями: $\{B\}$ и $\{C, D, E, F, G\}$. Корень дерева $\{C, D, E, F, G\}$ — узел C, он имеет три поддерева: $\{D\}$, $\{E\}$ и $\{F, G\}$. Узлы B, D, E и G — листья в большом дереве; A, C и F — внутренние узлы. Родителем узла F является узел C, братьями-сестрами — узлы D и E, единственным ребенком — узел G и т. д. Распишем также узлы по уровням:

0: A.

1: B, C.

[2]Несмотря на то, что деревья в природе растут снизу вверх, в информатике деревья обычно отображаются в перевернутом виде, когда корень находится сверху.

© www.phys.nsu.ru

2: D, E, F.
3: G.

Дерево называется *упорядоченным*, если имеет относительный порядок поддеревьев $T_1, \ldots, T_m$ (п. 2 приведенного выше определения).

Кроме графического изображения (как на рис. 7) для наглядного представления упорядоченных деревьев используется также Лисп-нотация:

$$(\text{Корень Поддерево}_1 \ \ldots \text{Поддерево}_n),$$

где Поддерево_k — это k-е поддерево в такой же нотации. Если поддерево состоит из одного лишь корня, скобки можно опустить. Дерево с рис. 7 в Лисп-нотации будет выглядеть так:

$$(A\ B\ (C\ D\ E\ (F\ G))),$$

что иллюстрирует интересную связь между деревьями и списками: дерево может быть представлено как список, состоящий из корня и поддеревьев. Лисп-нотация, как можно догадаться, восходит к языку Лисп, в котором она положена за основу синтаксиса (см. листинг 2.3 на стр. 25).

Реализация деревьев, как и в случае со списками, обычно опирается на динамическую память. Поскольку каждый узел может содержать в себе произвольное количество связей, приходится либо хранить их в динамическом массиве, либо использовать представление *с левым дочерним и правым сестринским узлами* [8, разд. 10.4], проиллюстрированное рис. 8. В этом представлении каждый узел обладает двумя связями: левой и правой. Левая связь указывает на самого первого ребенка данного узла, правая — на следующий по счету сестринский узел. Таким образом, дочерние узлы образуют связный список. В языке C данное представление может быть реализовано следующим образом:

© www.phys.nsu.ru

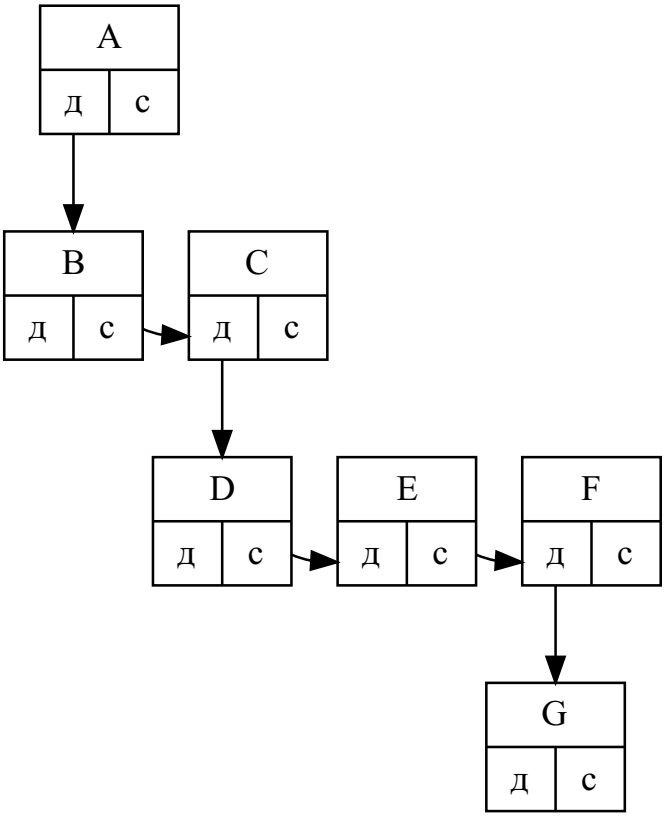


Рис. 8. Дерево с рис. 7 в представлении с левым дочерним и правым сестринским узлами

© www.phys.nsu.ru

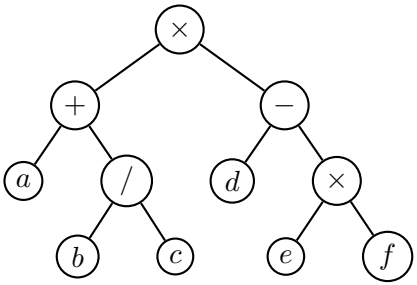


Рис. 9. Двоичное дерево, соответствующее выражению $(a+b/c)\cdot(d-ef)$

```
struct TreeNode {
    struct TreeNode *parent,   /* родитель */
                    *child,    /* ребенок */
                    *sibling; /* сестринский */
    /* ... */
};
```

6.2. Двоичные деревья

Важным типом деревьев являются *двоичные (бинарные) деревья*. Каждый узел двоичного дерева имеет не более двух поддеревьев, причем в случае только одного поддерева следует различать левое и правое. Строго говоря, двоичное дерево — это конечное множество узлов, которое является пустым или состоит из корня и двух непересекающихся двоичных деревьев, которые называются левым и правым поддеревьями данного корня [5, разд. 2.3].

Двоичные деревья естественным образом возникают в некоторых задачах. Например, двоичным деревом является схема теннисного турнира, где каждая игра — это узел, обозначенный ее победителем, а дочерние узлы — две предыдущие игры соперников. Арифметическое выражение с бинарными операциями также может быть представлено в виде двоичного дерева: знаки операций представляют из себя внутренние узлы с операндами-поддеревьями (см. рис. 9).

© www.phys.nsu.ru

Реализация двоичного дерева проще, чем произвольного. Обычно в C для хранения узлов используется динамическая память, а в структуре узла присутствует два указателя на левый и правый дочерние узлы:

```
struct Node {
    struct Node *left, *right;
    char data[1];
};
```

Данные узла здесь хранятся тем же способом, что и в реализации связного списка (см. стр. 69).

6.3. Обходы двоичных деревьев

Многие алгоритмы, работающие с древовидными структурами, опираются на понятие *обхода (traversing)* дерева. В процессе обхода каждый узел посещается ровно один раз, то есть обход задает линейное упорядочение узлов.

Для двоичного дерева существует три базовых способа обхода[3], которые определяются рекурсивно:

1) Сверху вниз, или *в прямом порядке* (preorder). Если дерево пусто, то для обхода делать ничего не надо, в противном случае обход выполняется в три этапа:

   а) попасть в корень;

   б) пройти левое поддерево;

   в) пройти правое поддерево.

2) Слева направо, или *в центрированном порядке* (inorder).

   Аналогично п. 1, со следующим порядком этапов:

   а) пройти левое поддерево;

[3] Существует еще два способа: *в ширину (по уровням)* и в *глубину*, — которые не будут рассматриваться в данной книге. См. [11, разд. 5.6].

© www.phys.nsu.ru

б) попасть в корень;

в) пройти правое поддерево.

3) Снизу вверх, или *в обратном порядке* (postorder).

Аналогично п. 1, со следующим порядком этапов:

а) пройти левое поддерево;

б) пройти правое поддерево;

в) попасть в корень.

Применим три варианта обхода к дереву арифметического выражения на рис. 9:

Сверху вниз	$\times + a/bc - d \times ef.$
Слева направо	$a + b/c \times d - e \times f.$
Снизу вверх	$abc/ + def \times - \times.$

В этих последовательностях можно узнать три способа записи арифметических выражений. Обход сверху вниз дает *префиксную запись* (она также называется польской записью в честь своего изобретателя Яна Лукасевича), снизу вверх — *постфиксную запись* (обратную польскую), а слева направо приводит к привычной *инфиксной записи*, хотя и без скобок, уточняющих порядок выполнения операций.

Рекурсивная реализация трех видов обходов на C получается очень простой и приведена на листинге 6.1. Каждая из функций в качестве аргумента получает корень дерева root, указатель f на функцию—«обходчик», которая будет вызвана для каждого узла, а также дополнительный аргумент arg для последней.[4]

Обходы двоичного дерева могут быть реализованы и без рекурсии, с использованием стека.

Процедуру уничтожения всего дерева (листинг 6.2) можно выразить через обход *снизу вверх*.

[4]По аналогии с функцией обхода списка list_traverse (листинг 5.2, стр. 70).

© www.phys.nsu.ru

Листинг 6.1. Реализация обходов деревьев

```
void tree_preorder(struct Node *root,
        void (*f)(struct Node *n, void *a), void *a) {
    if (root) {
        f(root, a);

        if (root->left)
            tree_preorder(root->left, f, a);
        if (root->right)
            tree_preorder(root->right, f, a);
    }
}

void tree_inorder(struct Node *root,
        void (*f)(struct Node *n, void *a), void *a) {
    if (root) {
        if (root->left)
            tree_inorder(root->left, f, a);

        f(root, a);

        if (root->right)
            tree_inorder(root->right, f, a);
    }
}

void tree_postorder(struct Node *root,
        void (*f)(struct Node *n, void *a), void *a) {
    if (root) {
        if (root->left)
            tree_postorder(root->left, f, a);
        if (root->right)
            tree_postorder(root->right, f, a);

        f(root, a);
    }
}
```

© www.phys.nsu.ru

Листинг 6.2. Уничтожение дерева

```
void _node_free(struct Node *n, void *a) {
    free(n);
}

void tree_destroy(struct Node *root) {
    tree_postorder(root, _node_free, 0);
}
```

6.4. Деревья поиска

Одно из частых применений двоичных деревьев — это *деревья поиска*. Предположим, что каждому узлу дерева сопоставлен некоторый уникальный ключ. Исходное дерево будет являться деревом поиска, если для каждого узла n будет выполнено следующее условие:

> Ключи всех узлов из левого поддерева *меньше*, чем ключ n, а ключи из правого поддерева, наоборот, *больше*.

Ключи, как правило, являются либо числами, либо строками, в последнем случае для их сравнения используется *лексикографический алгоритм сравнения строк*. Суть этого алгоритма состоит в посимвольном сравнении строк с начала; если какая-то из них кончится раньше другой, то она будет *предшествующей*, если символы в паре не равны друг другу, то *предшествующей* будет та строка, чей символ в алфавитном порядке стоит раньше. Если символы равны, происходит переход к следующей паре символов и сравнение продолжается. Таким образом, пустая строка будет предшествовать любой другой строке, $a \prec aa$, $aa \prec ab$ и т. д. (запись $x \prec y$ означает, что x предшествует y). В языке C лексикографическое сравнение выполняет функция стандартной библиотеки strcmp.

© www.phys.nsu.ru

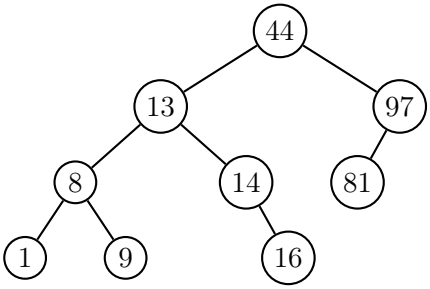


Рис. 10. Пример дерева поиска

На рис. 10 приведен пример дерева поиска. Рассмотрим, как происходит поиск. Допустим, требуется определить, присутствует ли ключ 9 в дереве. Поиск начинается с корня; $9 < 44$, следовательно, если ключ 9 где-то в дереве и присутствует, то в левом поддереве корня. Идя налево, сравниваем 9 с 13: $9 < 13$, следовательно, опять нужно идти налево. Но в следующем узле $9 > 8$; идя направо, обнаруживаем искомый узел. Таким образом, алгоритм поиска или находит узел, или завершается, не найдя его. (Если бы мы искали ключ 10, то дошли бы тем же самым путем до узла с ключом 9 и далее нужно было бы пойти направо, но правое поддерево пусто, следовательно, ключ 10 в дереве отсутствует.) Итак, для поиска потребовалось выполнить всего 4 операции сравнения при общем количестве узлов, равном 9.

Описанный алгоритм, являющийся основополагающим в АТД «Дерево поиска», обладает сравнительно простой реализацией на C, приведенной на листинге 6.3. Как и в предыдущих примерах реализации АТД, tree_search представляет собой абстрактный алгоритм поиска с «конкретикой» (сравнением узлов по ключам), вынесенной в функцию, вызываемую по указателю. Ниже приведена функция, которую можно использовать, если ключами в дереве являются обычные C-строки:

```
int _string_compare(void *k1, void *k2) {
    return strcmp((char *)k1, (char *)k2);
}
```

© www.phys.nsu.ru

Листинг 6.3. Поиск в дереве

```
struct Node *tree_search(struct Node *root, void *key,
        int (*compare)(void *k1, void *k2)) {

    for (;;) {
        int c;

        if (!root)
            return 0;

        c = compare(key, root->data);

        if (!c)
            return root;
        else
            root = (c < 0) ? root->left : root->right;
    }
}
```

Существенным свойством дерева поиска является то, что обход *слева направо* приводит к упорядочению элементов по возрастанию ключей. Обходя дерево на рис. 10, получим последовательность

1 8 9 13 14 16 44 81 97.

Обратимся теперь к задаче вставки узла в дерево поиска. Алгоритм вставки очень похож на алгоритм поиска, но имеет большее количество частных случаев:

- *Вставка в пустое дерево.* При этом изменяется значение корня (*root) — именно поэтому аргументом является двойной указатель.

- *Вставка узла с ключом, которого не было.* Обычная ситуация: функция находит место для узла, вставляет его в дерево и возвращает указатель на узел.

© www.phys.nsu.ru

- *Вставка узла с ключом, который уже присутствует в дереве.* В этом случае функция возвращает указатель на уже существующий узел, не изменяя дерево. Сравнивая переданный и возвращенный указатели, можно определить, был ли вставлен узел или нет.

Реализация алгоритма приведена на листинге 6.4.

Операция удаления ключа из дерева является более сложной, чем поиск и вставка. Если узел является листом или содержит только один дочерний узел, удалить легко — достаточно лишь изменить ссылку в родительском узле. Но что делать, если оба поддерева непусты? Тогда нужно заменить узел самым правым узлом из левого поддерева или самым левым из правого. Реализацию этого алгоритма можно найти в [2, разд. 4.4.4] и в [8, разд. 12.3].

Собрав воедино рассмотренные операции, получаем интерфейс АТД «Дерево поиска»:

1) Вставка элемента.

2) Поиск элемента по ключу.

3) Удаление элемента по ключу.

4) Обход дерева *слева направо.*[5]

5) Уничтожение дерева.

Этот АТД часто используется при работе с текстами, когда требуется, имея ключ (слово), быстро находить соответствующие данные.

6.5. Сбалансированные деревья

Необходимо отметить, что для заданного набора ключей существует более одного дерева поиска. На рис. 11 и 12 приведены

[5] Другие варианты обходов дерева поиска не имеют особого смысла.

© www.phys.nsu.ru

Листинг 6.4. Вставка в дерево поиска

```
struct Node *tree_insert(struct Node **root,
        struct Node *n,
        int (*compare)(void *k1, void *k2)) {
    Node *r = *root;

    n->left = n->right = 0;

    if (!r)
        return (*root = n);

    for (;;) {
        int c = compare(n->data, r->data);

        if (!c)
            return r;

        if (c < 0) {
            if (r->left) {
                r = r->left;
                continue;
            }

            return (r->left = n);
        } else {
            if (r->right) {
                r = r->right;
                continue;
            }

            return (r->right = n);
        }
    }
}
```

© www.phys.nsu.ru

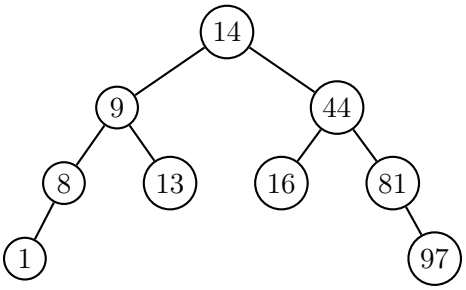


Рис. 11. Дерево поиска, эквивалентное дереву на рис. 10

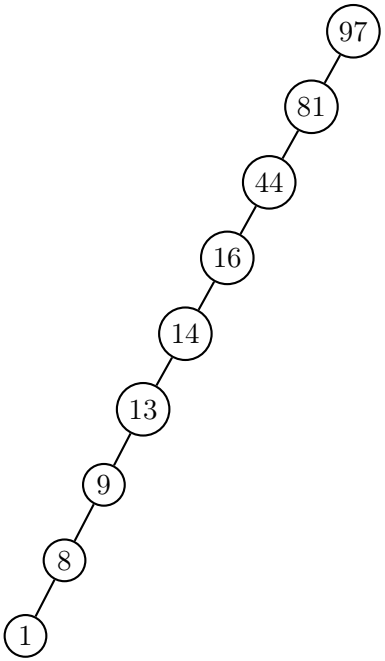


Рис. 12. Еще одно дерево поиска, эквивалентное дереву на рис. 10

© www.phys.nsu.ru

деревья, эквивалентные дереву на рис. 10. Какое из трех более желательно?

Ответ очевидным образом вытекает из предназначения дерева поиска — *быстрого поиска*. Количество операций, необходимых для нахождения узла, совпадает с количеством различных уровней в дереве (высотой). Следовательно, чем меньше уровней, тем быстрее. Легко показать, что полностью заполненное двоичное дерево с n уровнями содержит $2^n - 1$ узлов. Отсюда следует, что из m узлов можно построить только такое дерево, в котором будет не менее $\log_2(m+1)$ уровней.

В рассматриваемом нами дереве поиска $m = 9$, минимальное количество уровней равно $\lceil \log_2 10 \rceil = 4$. Следовательно, деревья на рис. 10 и 11 являются оптимальными, а дерево на рис. 12 — нет.

Как же добиваться оптимальности? Описанный алгоритм вставки не учитывает ее и может легко породить дерево, подобное тому, что на рис. 12 (для этого нужно вставлять элементы от большего ключа к меньшему).

Эту проблему решили советские математики Г. М. Адельсон-Вельский и Е. М. Ландис. Они показали, что если слегка ослабить требования к оптимальности дерева поиска, заменив оптимальность *сбалансированностью* (дерево является сбалансированным тогда, когда высоты двух поддеревьев любого узла отличаются не более чем на единицу), то операции *поиска*, *вставки* и *удаления* в худшем случае все равно будут выполняться за $O(\log m)$ операций — путь поиска по сбалансированному дереву не будет превышать путь по оптимальному дереву более чем на 45%. При этом после вставки и удаления свойство сбалансированности может быть восстановлено с помощью перебалансировки дерева по предложенному авторами алгоритму, также требующему максимум $O(\log m)$ операций.

Подробнее о сбалансированных деревьях[6] и методах балансировки см. [5, разд. 6.2.3] и [2, разд. 4.5]. Существуют и дру-

[6] Их также называют АВЛ- или AVL-деревьями, по инициалам создателей.

© www.phys.nsu.ru

гие способы поддержания оптимального состояния дерева поиска: *красно-черные деревъя* [8, гл. 13], *2-3-4-деревъя* [11, разд. 13.3] и другие.

© www.phys.nsu.ru

Глава 7

Хеш-таблицы

Быстрый поиск данных по ключу, который обеспечивается двоичным деревом поиска (см. разд. 6.4), может быть достигнут с помощью принципиально иного подхода.

Предположим сначала, что данные находятся в связном списке — тогда время поиска является линейным, $O(N)$. Но что если списков не один, а много ($N = 10, 1000, \dots$), и мы можем каким-то образом, основываясь на значении ключа, отнести каждый элемент к одному из N списков? Тогда для поиска по ключу потребуется сначала определить номер списка, а потом искать в нем. Если элементы распределились по спискам равномерно, время поиска будет существенно сокращено по сравнению со случаем линейного поиска. Если количество списков превышает количество ключей, поиск вообще может выполняться за одну операцию.

Эти рассуждения приводят нас к новому АТД. Функция, преобразующая ключ в номер, называется *хеш-функцией*, значение, соответствующее ключу — его *хеш-кодом*, а вся структура данных — *хеш-таблицей*. Обычно таблица имеет фиксированный размер, N ячеек, и данные с ключом k относятся к ячейке $h_N(k)$.

Интерфейс АТД «Хеш-таблица» практически совпадает с интерфейсом АТД «Дерево поиска»:

1) Вставка элемента.

2) Поиск элемента по ключу.

© www.phys.nsu.ru

3) Удаление элемента по ключу.

4) Обход таблицы.

5) Уничтожение таблицы.

Отличие от дерева состоит в том, что при обходе хеш-таблицы порядок данных не определен.

7.1. Хеш-функции

Из сказанного выше можно сделать некоторые выводы относительно желательных свойств хеш-функции.

Во-первых, она должна вычисляться быстро, ибо это вычисление будет происходить каждый раз при любой операции с таблицей.

Во-вторых, хеш-функция должна по возможности обеспечить равномерное распределение ключей по таблице, в противном случае поиск будет медленным. Случай, когда разным ключам соответствует одно значение хеш-функции, называется *коллизией*. Очевидно, что если ключей больше, чем ячеек в хеш-таблице, то без коллизий не обойтись, но плохая хеш-функция и при достаточном количестве места в таблице может приводить к коллизиям! Наихудший пример: $h(k) \equiv 0$ для любого k.

Рассмотрим варианты реализации хеш-функции для различных типов данных. Если ключ является целым числом, то можно воспользоваться методом деления:

$$h_N(k) = k \bmod N \tag{7.1}$$

или умножения:

$$h_N(k) = \lceil N(kA \bmod 1) \rceil, \tag{7.2}$$

где A — произвольная константа ($0 < A < 1$), а выражение $kA \bmod 1$ означает взятие дробной части произведения. В [7]

© www.phys.nsu.ru

предлагается использовать в качестве A константу золотого сечения:

$$A \approx (\sqrt{5} - 1)/2 \approx 0{,}6180339887\ldots$$

Если ключ — действительное число в диапазоне $[0, 1)$, то

$$h_N(x) = \lfloor Nx \rfloor. \tag{7.3}$$

Произвольное число с плавающей точкой можно хешировать по значению мантиссы, приведенному к диапазону $[0, 1)$. В языке C мантиссу числа можно получить с помощью функции стандартной библиотеки frexp из `math.h`.

Остается наиболее общий случай, когда ключ — цепочка байтов известной длины l (чаще всего это строка). Тривиальной хеш-функцией в данном случае будет сумма значений байт:

$$h_N(\boldsymbol{b}) = \sum_{k=0}^{l-1} b_k. \tag{7.4}$$

Недостатком хеш-функции (7.4) является ее неустойчивость к перестановкам байт: так, строкам «abc», «bac» и даже «aad» будет соответствовать одинаковое значение хеш-функции. Решением этой проблемы будет придание символу различного веса в зависимости от его положения. Кроме того, в хеш-функциях часто используется битовая операция «исключающее ИЛИ».

На листингах 7.1 и 7.2 приведены хеш-функции Дженкинса и Фаулера-Нолла-Во, применяемые в настоящее время во многих реализациях хеш-таблиц. Обе функции возвращают в качестве результата 32-битное беззнаковое число, которое с помощью формул (7.1) и (7.2) можно преобразовать к номеру ячейки в хеш-таблице.

7.2. Реализация хеш-таблицы на основе списков

В этом подходе хеш-таблица является динамическим массивом, который хранит в себе головы списков. Каждый список со-

© www.phys.nsu.ru

Листинг 7.1. Хеш-функция Боба Дженкинса

```
uint32_t jenkins_one_at_a_time_hash
        (unsigned char *key, size_t key_len) {
    uint32_t hash = 0;
    size_t i;

    for (i = 0; i < key_len; i++) {
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }
    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);
    return hash;
}
```

Листинг 7.2. Хеш-функция Фаулера-Нолла-Во

```
uint32_t fnv1_hash_32(void *buf, size_t len) {
{
    unsigned char *bp = (unsigned char *)buf;
    unsigned char *be = bp + len;
    uint32_t hval = 0x811c9dc5;

    while (bp < be) {
        hval += (hval<<1) + (hval<<4) + (hval<<7)
                          + (hval<<8) + (hval<<24);

        hval ^= (uint32_t)*bp++;
    }

    return hval;
}
```

© www.phys.nsu.ru

Листинг 7.3. Интерфейс хеш-таблицы на основе списков

```
#include "list.h"

typedef unsigned (*HashFunction)(char *str);
typedef struct HashTable {
    int size;
    struct List **table;
    HashFunction hashfunc;
} HashTable;

typedef struct ListData {
    char *key;
    void *data;
} ListData;

void ht_init(HashTable *ht, int size,
        HashFunction hf);
void ht_destroy(HashTable *ht, void (*d)(void *data));

void *ht_set(HashTable *ht, char *key, void *data);
void *ht_get(HashTable *ht, char *key);
void *ht_delete(HashTable *ht, char *key);
void ht_traverse(HashTable *ht,
        void (*f)(char *key, void *data));
```

держит данные с одинаковым хеш-кодом, соответствующим номеру списка в таблице.

Рассмотрим реализацию хеш-таблицы, сопоставляющей C-строкам ненулевые бестиповые указатели (**void** *). Интерфейс модуля приведен на листинге 7.3, реализация функций — на листингах 7.4—7.7.

Кратко рассмотрим реализацию каждой из функций.

ht_init. Заполняет структуру ht, выделяя память под головы списков. Функция calloc после выделения заполняет память ну-

© www.phys.nsu.ru

Листинг 7.4. Функция инициализации таблицы ht_init

```
void ht_init(HashTable *ht, int size,
        HashFunction hf) {
    ht->size = size;
    ht->hashfunc = hf;
    ht->table = (List **)
        calloc(size, sizeof (List *));
}
```

лями, что весьма удобно в нашем случае — получаем size пустых списков.

Переменную типа HashTable можно объявлять как локальную или глобальную — она занимает крайне мало памяти (несколько указателей и целое число).

ht_set. Здесь в первую очередь вычисляется хеш-код с использованием метода деления (7.1). Затем происходит попытка найти ключ в списке: вспомогательная функция-предикат _ht_keycomp сравнивает ключ, хранящийся в элементе списка (в структуре ListData), с заданным.

Если ключ найден, функция просто замещает значение, *возвращая старое* (это может быть нужно, например, для последующего вызова free, если значения являются указателями на динамическую память).

Если ключ не найден, функция создает новый элемент списка и вставляет его в голову. При этом ключ *копируется* с помощью функции strdup (переданный ключ мог находиться во временном массиве и мог бы быть «потерян», будь сохранен один лишь указатель на него). В этом случае функция возвращает 0.

ht_get. Вычисляет хеш-код и осуществляет поиск в соответствующем списке, возвращая найденный результат или 0, если ничего не найдено.

© www.phys.nsu.ru

Листинг 7.5. Функции установки и получения значения по ключу (ht_set и ht_get)

```
static int _ht_keycomp(List *el, void *key) {
    return !strcmp(((ListData *)el->data)->key,
            (char *)key);
}

void *ht_set(HashTable *ht,
        char *key, void *data) {

    unsigned h = ht->hashfunc(key) % ht->size;
    List *el = list_find(ht->table[h],
            _ht_keycomp, key);
    ListData ld;

    if (el) {
        void *old = ((ListData *)el->data)->data;
        ((ListData *)el->data)->data = data;
        return old;
    }

    ld.key = strdup(key);
    ld.data = data;
    ht->table[h] = list_insert_head(ht->table[h],
        list_create_element(&ld, sizeof (ld)));
    return 0;
}

void *ht_get(HashTable *ht, char *key) {
    unsigned h = ht->hashfunc(key) % ht->size;
    List *el = list_find(ht->table[h],
            _ht_keycomp, key);
    if (el)
        return ((ListData *)el->data)->data;
    else
        return 0;
}
```

© www.phys.nsu.ru

Листинг 7.6. Функция обхода хеш-таблицы

```
void _ht_traverse_1(List *el, void *arg) {
    void (*t)(char *, void *) =
        (void (*)(char *, void *))arg;
    ListData *pld = (ListData *)el->data;

    if (t)
        t(pld->key, pld->data);
}

void ht_traverse(HashTable *ht,
        void (*f)(char *key, void *data)) {
    int i;
    for (i = 0; i < ht->size; ++i)
        list_traverse(ht->table[i],
                _ht_traverse_1, f);
}
```

ht_delete. Вычислив хеш-код, определяет, лежит ли элемент в голове списка. Если это так, то голова сдвигается к следующему элементу. В противном случае происходит поиск предыдущего элемента в списке и удаление из списка (аналогично list_delete_after).

Если ключ был найден, то освобождается строка-копия ключа, затем сам элемент списка, а функция возвращает данные, соответствовавшие удаленному ключу. Если ключ не был найден, возвращается 0.

ht_destroy. Второй аргумент функции — пользовательская функция удаления данных. Если указатель на нее ненулевой, то она будет применена к каждому значению из таблицы. Это позволяет при уничтожении таблицы корректно освободить всю память.

После поэлементного удаления всех списков освобождает-

© www.phys.nsu.ru

Листинг 7.7. Функции удаления по ключу и уничтожения таблицы

```
void *ht_delete(HashTable *ht, char *key) {
    unsigned h = ht->hashfunc(key) % ht->size;
    List *el = 0, *p = ht->table[h];
    void *data;
    if (p && _ht_keycomp(p, key)) {
        el = p;
        ht->table[h] = p->next;
    } else for (; p && p->next; p = p->next)
            if (_ht_keycomp(p->next, key)) {
                el = p->next;
                p->next = el->next;
            }
    if (!el)
        return 0;
    data = el->data;
    free(((ListData *)el->data)->key);
    free(el);
    return data;
}

void _ht_destroy_1(List *el, void *arg) {
    void (*destroyer)(void *) = (void (*)(void *))arg;
    ListData *pld = (ListData *)el->data;
    if (destroyer)
        destroyer(pld->data);
    free(pld->key);
    free(el);
}

void ht_destroy(HashTable *ht, void (*d)(void *)) {
    int i;
    for (i = 0; i < ht->size; ++i)
        list_traverse(ht->table[i], _ht_destroy_1, d);
    free(ht->table);
    ht->table = ht->size = 0;
    ht->hashfunc = 0;
}
```

© www.phys.nsu.ru

ся таблица голов списков, зануляются указатели и размер хеш-таблицы (на случай, если она будет ошибочно использована после уничтожения).

7.3. Иные реализации хеш-таблиц

Кроме способа, рассмотренного в предыдущем разделе, существуют и другие, которые будут рассмотрены в этом разделе.

Открытая адресация. В этом подходе никаких списков нет, все данные хранятся в самой таблице. Рассмотрим, как происходит вставка в такую таблицу.

Сначала вычисляется хеш-код. Если соответствующая ячейка свободна, она заполняется, и вставка завершена. В противном случае происходит систематический перебор других ячеек, пока не будет найдена свободная. Другими словами, вместо одного номера $h_N(k)$ имеем целую последовательность:

$$\langle h_N(k,0), h_N(k,1), \ldots, h_N(k,N-1)\rangle,$$

которая перебирается до нахождения первой свободной ячейки. В методе открытой адресации требуется, чтобы эта последовательность являлась перестановкой множества $\langle 0,1,\ldots,N-1\rangle$, чтобы в конечном счете могли быть просмотрены все ячейки хеш-таблицы. Для построения последовательности $h_N(k,l)$ используются следующие приемы:

1) *Линейное исследование.*

$$h_N(k,l) = (h_N(k) + l) \bmod N, \tag{7.5}$$

то есть ячейки перебираются последовательно, начиная со стартового хеш-кода $h_N(k)$, с переходом к нулю через N.

Линейное исследование легко реализуется, но с ним связана проблема *первичной кластеризации*, когда возникают длинные последовательности занятых ячеек, увеличивающие среднее время поиска.

© www.phys.nsu.ru

2) *Квадратичное исследование.*

$$h_N(k,l) = (h_N(k) + c_1 l + c_2 l^2) \bmod N, \tag{7.6}$$

где c_1 и c_2 — вспомогательные константы. Этот подход работает лучше, чем линейное исследование, но требует специального выбора N, c_1 и c_2 — см. [8, стр. 314]. Кроме того, ключи с одинаковым значением стартового хеш-кода обладают одной и той же последовательностью номеров, что приводит к более мягкой *вторичной кластеризации.*

3) *Двойное хеширование.*

$$h_N(k,l) = (h_N(k) + l \cdot h'_N(k)) \bmod N, \tag{7.7}$$

где $h'_N(k)$ — вторичная хеш-функция. Для того чтобы последовательность исследования могла охватить всю таблицу, значение $h'_N(k)$ должно быть взаимно простым с размером хеш-таблицы N. Это можно сделать, выбрав $N = 2^j$ и построив $h'_N(k)$ так, чтобы она возвращала только нечетные значения. Другой способ состоит в выборе для N простого числа и в задании диапазона значений для $h'_N(k)$ в $[1, M-1]$. Тогда можно принять

$$h_N(k) = h(k) \bmod N; \tag{7.8}$$

$$h'_N(k) = 1 + (h(k) \bmod N'), \tag{7.9}$$

где N — простое число, а N' немного меньше N (например, $N-1$).

Двойное хеширование представляет собой наилучший способ использования открытой адресации, поскольку получаемые при этом перестановки обладают многими характеристиками случайно выбираемых перестановок; его производительность близка к «идеальной» схеме равномерного хеширования.

© www.phys.nsu.ru

Идеальное хеширование. В случае, если множество ключей является *статическим*, т. е. никогда не меняется, возможно построение такой системы хеширования, когда поиск любого ключа будет происходить за $O(1)$ операций (константное время). Это достигается с помощью вторичных хеш-таблиц, ассоциированных с каждой ячейкой основной хеш-таблицы, причем их хеш-функции подбираются специальным образом так, чтобы не было коллизий [8, разд. 11.5].

«Кукушиное» хеширование (cuckoo hashing). Это сравнительно новый подход, предложенный в 2001 г. [18].

Подход предполагает наличие двух хеш-таблиц T_1 и T_2 одинакового размера (N) и двух различающихся хеш-функций: $h_N^{(1)}$ и $h_N^{(2)}$. Общий принцип состоит в том, что данные, соответствующие ключу k, находятся либо в ячейке $h_N^{(1)}$ таблицы T_1, либо в ячейке $h_N^{(2)}$ таблицы T_2, но никогда в обоих местах одновременно. Такое условие делает возможным производить поиск и удаление за константное время: достаточно проверить всего две ячейки!

Название подходу дал алгоритм вставки в таблицу. Вставляя ключ k, мы вычисляем $h_N^{(1)}(k)$ и в любом случае помещаем значение в эту ячейку таблицы T_1, «выбрасывая из гнезда»[1] запись, которая там была — при ее наличии, разумеется. Если ячейка была пуста, то вставка завершается, в противном случае «выброшенная» запись перемещается на свое альтернативное место в таблицу T_2 с «выбрасыванием» той запись, которая там была, и т. д. Процесс продолжается до тех пор, пока альтернативное место одной из «выброшенных» записей не окажется свободным, позволяя успешно завершить вставку, или процесс не зациклится. В последнем случае выбираются новые хеш-функции и происходит полное перехеширование обеих таблиц.

[1] Словно кукушка, подкладывающая в гнездо свои яйца и выбрасывающая чужие!

© www.phys.nsu.ru

Глава 8

Алгоритмы сортировки

Сортировка — одна из часто встречающихся задач при обработке данных. Еще с детства людей приучают к таким вещам, как «тройка лучших учеников в классе», «горячая десятка хитов», затем уже идут рейтинг Форбс, «400 крупнейших компаний России» и т. д. Короче говоря, люди приучены воспринимать структурированную информацию. Нет ничего хуже для человека, чем хаотический набор данных, но если данные можно упорядочить по каким-либо показателям, это совершенно другое дело!

Приходим к задаче сортировки, или упорядочения, которую в общем виде можно сформулировать следующим образом [7]. Даны записи

$$R_1, R_2, \ldots, R_N, \tag{8.1}$$

которым соответствуют ключи $K_1, K_2, \ldots, K_N$. Требуется переупорядочить записи в порядке неубывания их ключей, или, другими словами, найти перестановку $p(1)p(2)\ldots p(N)$, такую, что

$$K_{p(1)} \le K_{p(2)} \le \cdots \le K_{p(N)}. \tag{8.2}$$

Сортировка подразделяется на *внутреннюю* (когда число записей N достаточно мало, чтобы весь процесс можно было провести в оперативной памяти компьютера) и *внешнюю*. Материал этой главы в основном посвящен внутренним сортировкам, за исключением раздела 8.5.

Стандартная библиотека C содержит в себе функцию qsort, построенную на основе быстрой сортировки (разд. 8.2) и годную

© www.phys.nsu.ru

в большинстве случаев. Она имеет следующий интерфейс:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Это означает «сортировать nmemb объектов, каждый размером size, которые находятся в массиве, на начало которого указывает base». Сравнение элементов производится с помощью пользовательской функции compar, которой передаются два указателя на сравниваемых объекта, а она должна вернуть -1, 0 или 1, если первый объект меньше, равен или больше второго объекта, соответственно. Такой интерфейс функции qsort позволяет абстрагироваться от сортируемых объектов.

В данном главе мы будем изучать устройство различных алгоритмов сортировки, поэтому сама по себе готовая функция qsort нас не будет интересовать. Но ее интерфейс мы позаимствуем.

8.1. Элементарные алгоритмы

Сортировка выбором (selection sort). Один из самых простых алгоритмов сортировки устроен следующим образом:

1) Найти наименьший ключ из $K_1, \ldots, K_N$ (обозначим его K_m) и поменять местами элементы R_1 и R_m.

2) Найти второй наименьший ключ ($K_{m'}$) из $K_2, \ldots, K_N$ и поменять местами элементы R_2 и $R_{m'}$.

3) ...

4) Продолжать до тех пор, пока весь массив не будет отсортирован.

Название метода происходит от того, что на каждом шаге мы *выбираем* очередной наименьший элемент и ставим его на соответствующее по счету место. Нетрудно подсчитать количество операций, требуемых для этого. На первом шаге потребуется $N - 1$ операций сравнения и 1 операция перестановки. На

© www.phys.nsu.ru

втором — $N-2$ сравнения и 1 перестановка, и т. д. Всего шагов $N-1$, то есть общее количество операций равно

$$(N-1)+(N-2)+\cdots+1 \text{ (сравн.)}+N-1 \text{ (перест.)} = O(N^2). \quad (8.3)$$

Реализация сортировки выбором представлена на листинге 8.1. Чтобы менять местами объекты, требуется дополнительная память, которая выделяется с помощью malloc в случае большого размера элемента (в противном случае используется небольшой массив buf на стеке), а сам обмен происходит в функции _exchange.

Пузырьковая сортировка (bubble sort). Этот метод сортировки не сложнее предыдущего:

1) Проходим массив от начала к концу, сравнивая на каждом шаге пару ключей K_i и K_{i+1} ($1 \le i < N$). Если $K_i > K_{i+1}$, то записи R_i и R_{i+1} меняются местами.

2) Если на предыдущем шаге хотя бы в одной паре была перестановка, проход по массиву повторяется. Если перестановок не было, то массив отсортирован — выход.

Алгоритм отчасти похож на сортировку выбором. На первом шаге свое место займет запись с наибольшим ключом (она как бы «всплывает», что и дало название алгоритму). На втором шаге то же самое произойдет с записью со вторым наибольшим ключом, и так далее. Из этого следует, что шаг №1 алгоритма можно слегка усовершенствовать, изменив условие цикла:

$$1 \le i \le N - n \quad (8.4)$$

на n-м проходе ($n \ge 1$).

На листинге 8.2 представлена реализация алгоритма.

Существует также модификация пузырьковой сортировки, называемая «коктейльной», или шейкер-сортировкой (cocktail sort, shaker sort). Здесь чередуются проходы от начала к концу

© www.phys.nsu.ru

Листинг 8.1. Сортировка выбором

```
static void _exchange(void *a, void *b,
                      void *tmp, size_t size) {
    memcpy(tmp, a, size);
    memcpy(a, b, size);
    memcpy(b, tmp, size);
}

void selection_sort(void *base, size_t count,
        size_t size,
        int (*compar)(const void *, const void *)) {

    int i;
    char buf[1024], *a = (char *)base;
    void *tmp;

    if (count < 2)
        return;

    tmp = (size > sizeof (buf)) ? malloc(size) : buf;

    for (i=0; i < count-1; i++) {
        int j, min=i;

        for (j=i+1; j < count; j++)
            if (compar(a+j*size, a+min*size) < 0)
                min = j;

        _exchange(a+i*size, a+min*size, tmp, size);
    }

    if (tmp != buf)
        free(tmp);
}
```

© www.phys.nsu.ru

Листинг 8.2. Пузырьковая сортировка

```
void bubble_sort(void *base, size_t count,
        size_t size,
        int (*compar)(const void *, const void *)) {

    int i;
    char buf[1024], *a = (char *)base;
    void *tmp;

    tmp = (size > sizeof (buf)) ? malloc(size) : buf;

    for (i = 0; i < count; ++i) {
        int j, swaps = 0;

        for (j = 0; j < count-1-i; ++j)
            if (compar(a+j*size, a+(j+1)*size) > 0) {
                _exchange(a+j*size, a+(j+1)*size,
                          tmp, size);
                swaps = 1;
            }

        if (swaps == 0)
            break;
    }

    if (tmp != buf)
        free(tmp);
}
```

© www.phys.nsu.ru

и от конца к началу, что уменьшает общее количество операций. Если в обычной пузырьковой сортировке запись с наименьшим ключом, стоящая на j-м месте, займет положенное ей 1-е место за $j-1$ проходов (сдвигаясь каждый раз на одну позицию ближе к началу массива), то «коктейльный» вариант переместит ее туда после первого же прохода от конца к началу. Иными словами, «большие» пузырьки быстро всплывают, а «маленькие» быстро тонут.

Сортировка вставками (insertion sort). Предположим, что $1 < j \le N$ и записи $R_1, \ldots, R_{j-1}$ уже отсортированы, то есть

$$K_1 \le K_2 \le \cdots \le K_{j-1}.$$

Будем по очереди сравнивать K_j с $K_{j-1}, K_{j-2}, \ldots$ до тех пор, пока не обнаружим, что R_j следует *вставить* между R_i и R_{i+1}; тогда сдвинем записи $R_{i+1}, \ldots, R_{j-1}$ на одну позицию вправо и поместим R_j в позицию $i+1$.

Можно показать, что общее количество операций и в этом случае равно $O(N^2)$. Следует, однако, заметить, что при сортировке двусвязного списка (см. разд. 5.2) вставка элемента в найденную позицию будет выполняться за константное время, что уменьшит общее время исполнения по сравнению с массивом (статическим или динамическим).

Реализацию сортировки вставками см. на листинге 8.3.

Работа алгоритма может быть отчасти улучшена с помощью *бинарных вставок*, когда вместо линейного поиска места вставки осуществляется дихотомический поиск за логарифмическое время [7, разд. 5.2.1].

Метод Шелла (shellsort). Если алгоритм сортировки каждый раз перемещает запись только на одну позицию, то среднее время его выполнения будет в лучшем случае пропорционально $O(N^2)$ [7, разд. 5.2.1]. Метод Шелла родился как простое расширение метода вставок, с повышением быстродействия за счет

© www.phys.nsu.ru

Листинг 8.3. Сортировка вставками

```
void insertion_sort(void *base, size_t count,
        size_t size,
        int (*compar)(const void *, const void *)) {

    int i;
    char buf[1024], *a = (char *)base;
    void *tmp = (size > sizeof (buf))
                ? malloc(size) : buf;

    for (i = 1; i < count; ++i) {
        int j;
        memcpy(tmp, a+i*size, size);

        for (j = i-1;
             j >= 0 && compar(a+j*size, tmp) > 0;
             --j)
            memcpy(a+(j+1)*size, a+j*size, size);

        memcpy(a+(j+1)*size, tmp, size);
    }

    if (tmp != buf)
        free(tmp);
}
```

© www.phys.nsu.ru

того, что появляется возможность обмена местами записей, находящихся далеко друг от друга.

Идея состоит в том, чтобы разделять весь массив записей на подмассивы, выбирая каждый h-й элемент:

$$[R_1, R_{h+1}, R_{2h+1}, \dots],$$
$$[R_2, R_{h+2}, R_{2h+2}, \dots],$$
$$\dots$$

и сортировать каждый подмассив по отдельности с помощью метода вставок. Результирующий массив будем называть h-сортированным. Выбрав сначала большое значение h (например, $\lfloor N/2 \rfloor$) и затем постепенно уменьшая его до 1 (при $h = 1$ алгоритм Шелла эквивалентен алгоритму вставок), получаем в итоге сортированный массив. В процессе h-сортировки записи перемещаются сразу на h позиций, что повышает быстродействие алгоритма Шелла по сравнению с обычным алгоритмом вставок.

Выбор последовательности значений h является неоднозначным, ясно лишь, что она должна быть убывающей, и последнее значение должно равняться 1. В [7] рассматриваются различные варианты, один из простейших — последовательность $h_{t-1}, h_{t-2}, \dots, h_0$ при

$$h_0 = 1; \qquad h_{s+1} = 3h_s + 1; \qquad 3h_t \geq N, \tag{8.5}$$

то есть из ряда чисел $1, 4, 13, 40, 121, 364, 1093, 3280, 9841, \dots$ выбирается наибольшее число, которое будет меньше N, а затем последовательность «раскручивается» в обратном порядке, до 1. Реализация метода Шелла с такой последовательностью приведена на листинге 8.4.

В [11, разд. 6.6] показано, что для эвристики (8.5) количество операций сравнения в худшем случае равно $O(N^{3/2})$.

8.2. Быстрая сортировка

Алгоритм быстрой сортировки (quicksort) был придуман Ч. Хоаром в 1962 г. В его основе лежит принцип «разделяй и власт-

© www.phys.nsu.ru

Листинг 8.4. Сортировка Шелла

```
void shell_sort(void *base, size_t count, size_t size,
        int (*compar)(const void *, const void *)) {

    int i, h;
    char buf[1024], *a = (char *)base;
    void *tmp = (size > sizeof (buf))
                ? malloc(size) : buf;

    /* find maximum h (h < count) */
    for (h = 1; h <= (count-1)/9; h = 3*h+1)
        ;

    for (; h > 0; h /= 3)
        /* h-insertion sort */
        for (i = h; i < count; ++i) {
            int j = i;
            memcpy(tmp, a+i*size, size);

            while (j >= h &&
                    compar(tmp, a+(j-h)*size) < 0) {
                memcpy(a+j*size, a+(j-h)*size, size);
                j -= h;
            }

            memcpy(a+j*size, tmp, size);
        }

    if (tmp != buf)
        free(tmp);
}
```

© www.phys.nsu.ru

вуй» [8, разд. 2.3.1], который предполагает *разделение* задачи на несколько подзадач меньшего размера, *покорение* — рекурсивное решение подзадач (подзадачи особо малого размера решаются непосредственно), а затем *объединение* результатов решения позадач в единое целое.

Рассмотрим с этих позиций метод быстрой сортировки. Предположим, что необходимо отсортировать записи $R_l, \ldots, R_r$ по их ключам $K_l, \ldots, K_r$.

1) *Разделение.*

   Массив записей $[R_l, \ldots, R_r]$ разбивается (путем переупорядочения его элементов) на два (возможно, пустых) подмассива $[R_l, \ldots, R_{q-1}]$ и $[R_{q+1}, \ldots, R_r]$. Каждый из ключей $K_l, \ldots, K_{q-1}$ не превышает K_q, а каждый ключ $K_{q+1}, \ldots, K_r$ не меньше K_q. Индекс q вычисляется в ходе процедуры разбиения.

2) *Покорение.*

   Подмассивы $[R_l, \ldots, R_{q-1}]$ и $[R_{q+1}, \ldots, R_r]$ сортируются путем рекурсивного вызова процедуры быстрой сортировки.

3) *Объединение.*

   Поскольку подмассивы сортируются на месте, для их объединения не нужны никакие действия: все записи $R_l, \ldots, R_r$ оказываются отсортированными.

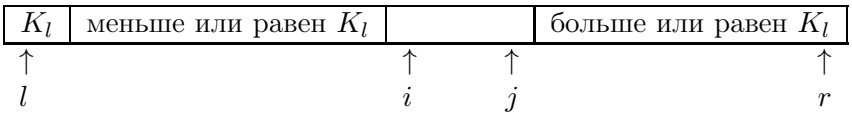
Принцип деления на две части приводит к догадке относительно быстродействия алгоритма: количество операций равно $O(N \log N)$.

Покорение и объединение тривиальны, весь вопрос состоит в том, как осуществить разбиение. Лучшая из известных процедур на данный момент состоит в следующем.

Прежде всего выбирается *разделяющий элемент*, который в процессе разбиения займет свою окончательную позицию. В его качестве можно произвольно выбрать ключ K_l. Также имеются

© www.phys.nsu.ru

два указателя i и j, причем вначале $i = l+1$ и $j = r$. Далее начинается просмотр ключей с левого конца массива с увеличением i, пока не будет найден такой ключ, что $K_i \geq K_l$ (соответствующая запись принадлежит правому подмассиву). Затем j аналогичным образом уменьшается на 1 до тех пор, пока не найдется запись R_j, которая принадлежит левому подмассиву ($K_j \leq K_l$). Если $i < j$, R_i и R_j меняются местами, после чего продолжается поиск с левого конца (с позиции $i+1$), затем снова с правого, и так далее, пока не станет $i \geq j$. Завершается процесс разделения массива обменом записей R_j с R_l. Следующая схема иллюстрирует то, что происходит в массиве во время разбиения:



Разделяющий элемент находится слева, в позиции l. Все, что слева от i, принадлежит левому подмассиву, ибо соответствующие ключи меньше или равны K_l. Все, что справа от j, принадлежит правому подмассиву. Пустая область, куда указывают i и j, еще не обработана.

Реализация алгоритма приведена на листингах 8.5 и 8.6.

Несмотря на внешнюю простоту, алгоритм быстрой сортировки обладает многими нюансами, которые необходимо учитывать при его реализации.

1) Если среди ключей встречаются одинаковые по значению, могут возникнуть трудности с фиксацией момента пересечения i и j.

   Здесь лучше дать алгоритму обменивать местами записи с одинаковыми ключами, а не пропускать их при перемещении указателей. Хотя это приводит к ненужному перемещению записей, в случае массива с полностью совпадающими ключами алгоритм сработает за логарифмическое время, а не за квадратичное.

   При большом количестве совпадающих ключей лучше ис-

© www.phys.nsu.ru

Листинг 8.5. Быстрая сортировка (функция-обертка для вызова)

```
typedef struct QuickSortData {
    char *a;
    void *tmp;
    size_t size;
    int (*compar)(const void *, const void *);
} QSD;


void quick_sort(void *base, size_t count, size_t size,
        int (*compar)(const void *, const void *)) {

    int i, h;
    char buf[1024], *a = (char *)base;
    void *tmp = (size > sizeof (buf))
              ? malloc(size) : buf;
    QSD qsd = { a, tmp, size, compar };

    if (count >= 2)
        _partial_quick_sort(&qsd, 0, count-1);

    if (tmp != buf)
        free(tmp);
}
```

© www.phys.nsu.ru

Листинг 8.6. Быстрая сортировка (основная рекурсивная реализация)

```
void _partial_quick_sort(QSD *qsd, int l, int r) {
    int i = l, j = r+1;
    if (r <= l)
        return;

#define QS_COMP(p,q) \
    (qsd->compar(qsd->a+(p)*qsd->size, \
                 qsd->a+(q)*qsd->size))
#define QS_EXCH(p,q) \
    _exchange(qsd->a+(p)*qsd->size, \
              qsd->a+(q)*qsd->size, \
              qsd->tmp, qsd->size)

    for (;;) {
        do
            ++i;
        while (i <= r && QS_COMP(i,l) < 0);

        do
            --j;
        while (QS_COMP(j,l) > 0);

        if (j < i)
            break;

        QS_EXCH(i,j);
    }

    QS_EXCH(l,j);

    _partial_quick_sort(qsd, l, j-1);
    _partial_quick_sort(qsd, j+1, r);

#undef QS_COMP
#undef QS_EXCH
}
```

© www.phys.nsu.ru

пользовать доработанный алгоритм, делящий массив на *три* части [11, разд. 7.6].

2) Быстрая сортировка является *неустойчивой* (устойчивым называется такой алгоритм сортировки, который не изменяет относительное положение записей с одинаковыми ключами).

3) Худшее время работы алгоритма быстрой сортировки равно $O(N^2)$. Это происходит, когда массив уже отсортирован — тогда на каждом шаге происходит разделение N-элементного массива на подмассивы длиной 1 и $N-1$, последнего — на 1 и $N-2$, что в итоге дает квадратичное время (!).

4) Быстродействие алгоритма в каждом конкретном случае во многом зависит от удачного выбора разделяющего элемента. Идеальный случай происходит тогда, когда выбранный элемент оказывается медианой.[1]

   Чтобы приблизиться к идеалу, вместо выбора первого элемента в качестве разделяющего, например, можно выбирать медиану из первого, последнего и серединного.

   Другое решение — *рандомизированная версия* быстрой сортировки, когда разделяющий элемент выбирается случайным образом [8, разд. 7.3].

В [15] можно найти подробное описание тонкостей и способов оптимизации алгоритма быстрой сортировки для различных случаев. Также информацию можно найти в [11, гл. 7], [1, гл. 11], [8, гл. 7], [7, разд. 5.2.2].

[1] Точное определение медианы здесь бессмысленно, ибо само по себе требует большого количества операций.

© www.phys.nsu.ru

8.3. Пирамидальная сортировка

Существует еще один довольно необычный способ *обменной сортировки*[2], основанный на структуре данных под названием *пирамида*.

Определения. Пирамида — это массив, который можно рассматривать как почти полное двоичное дерево. Каждый узел этого дерева соответствует определенному элементу массива. На всех уровнях, кроме, возможно, последнего, дерево полностью заполнено, а последний уровень заполняется слева направо до тех пор, пока в массиве не кончатся элементы.

Рассмотрим узел дерева, соответствующий элементу массива с номером n ($1 \le n \le N$) с ключом K_n. Его левый дочерний узел будет соответствовать элементу $2n$, а правый — $2n+1$. Первый узел соответствует вершине пирамиды.

Пирамиды также делятся на два вида: неубывающие и невозрастающие. У *невозрастающей* пирамиды для любого узла n выполняется условие

$$K_n \ge K_{2n} \qquad (\text{если } 2n \le N); \tag{8.6}$$

$$K_n \ge K_{2n+1} \qquad (\text{если } 2n+1 \le N). \tag{8.7}$$

Для неубывающих пирамид:

$$K_n \le K_{2n} \qquad (\text{если } 2n \le N); \tag{8.8}$$

$$K_n \le K_{2n+1} \qquad (\text{если } 2n+1 \le N). \tag{8.9}$$

(не путать пирамиды с деревьями поиска!)

На рис. 13 показана невозрастающая пирамида из массива ключей

$$16\ 14\ 10\ 8\ 7\ 9\ 3\ 2\ 4\ 1. \tag{8.10}$$

Как видно из рисунка, последовательность чисел получается, если идти по уровням сверху вниз и слева направо.

[2]Обменные сортировки построены на обмене определенных записей местами, к ним относятся все рассмотренные нами выше алгоритмы.

© www.phys.nsu.ru

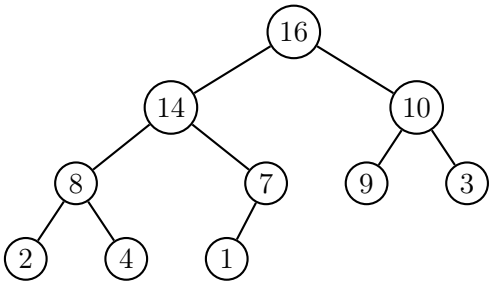


Рис. 13. Неубывающая пирамида для массива чисел (8.10)

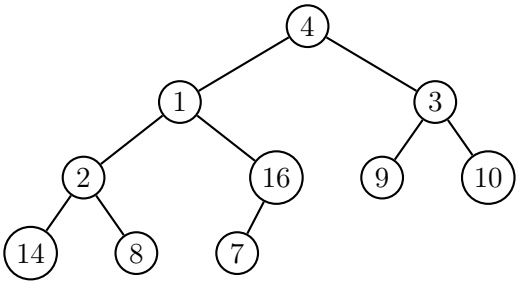


Рис. 14. Двоичное дерево из массива чисел (8.11)

Построение пирамиды. Займемся задачей построения пирамиды из массива записей $[R_1, R_2, \ldots, R_N]$. Построение будет иллюстрироваться на массиве с ключами

$$4\ 1\ 3\ 2\ 16\ 9\ 10\ 14\ 8\ 7, \tag{8.11}$$

двоичное дерево для которого представлено на рис. 14 (сам массив является перестановкой массива (8.10)).

Из рисунка видно, что узлы с ключами 9, 10, 14, 8, 7 уже удовлетворяют условию пирамиды, так как у них нет потомков (таких узлов будет всегда $\lceil N/2 \rceil$). Теперь, начиная с узла №5 (с ключом 16), будем приращивать пирамиду слева, обеспечивая выполнение условия неубывания. Узел №5 удовлетворяет условию, ибо $16 \geq 7$, поэтому двинемся дальше. Узел №4 с ключом

© www.phys.nsu.ru

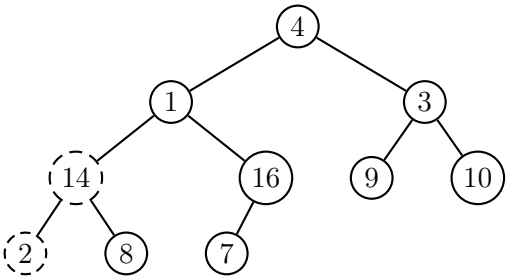


Рис. 15. Построение неубывающей пирамиды

2 не удовлетворяет условию пирамиды! Здесь включается операция *просеивания*. Из двух потомков данного узла мы выбираем тот, чей ключ является наибольшим (14), и меняем его местами с самим узлом. На рис. 15 узлы, поменявшиеся местами, помечены прерывистой линией.

Процесс приращивания пирамиды с левой стороны продолжается, будут поменяны местами ключи $10 \Leftrightarrow 3$; $1 \Leftrightarrow 16 \Leftrightarrow 7$ и $4 \Leftrightarrow 16 \Leftrightarrow 14 \Leftrightarrow 8$ (в последнем случае ключ 4 просеивается три раза подряд, в итоге находя себе место на самом нижнем уровне). Результирующая пирамида получается такой же, как на рис. 13.

Алгоритм сортировки. Итак, мы получили пирамиду, но как теперь отсортировать массив?

Оказывается, это просто. По условию неубывания элемент с самым большим ключом находится на вершине. В результате же сортировки он должен оказаться в самом конце массива. Поместим его туда, обменяв с последним элементом массива, и уменьшим размер пирамиды до $N - 1$. Элемент, который появился на вершине, возможно, нарушил условие пирамиды, но мы снова запустим процедуру просеивания, восстанавливая условие неубывания и получая в результате пирамиду размером $N - 1$ со вторым наибольшим ключом в вершине. Элемент, находящийся теперь на вершине, меняется местами с элементом под номером $N - 2$ (с одновременным уменьшением размера пирамиды до $N - 2$),

© www.phys.nsu.ru

и весь процесс повторяется до тех пор, пока пирамида не будет состоять из одного элемента ($N-1$ элементов будут уже стоять на своих местах).

В пирамиде на рис. 13 мы осуществляем обмен $16 \Leftrightarrow 1$, помещая 1 на вершину. Просеивание выводит на вершину ключ 14, который меняется с опустившимся вниз ключом 1, и т. д. В итоге получается отсортированный массив.

Время работы алгоритма равно $O(N \log N)$. Одно просеивание занимает $O(\log N)$ операций; на первом шаге, построении пирамиды, таких просеиваний будет $N/2$. На втором шаге будет N просеиваний.

Алгоритм пирамидальной сортировки является *устойчивым* и более предсказуемым, чем алгоритм быстрой сортировки (время его работы всегда равно $O(N \log N)$, вне зависимости от входных данных), однако в среднем работает немного медленнее, чем быстрая сортировка.

Реализация пирамидальной сортировки на C приведена на листингах 8.7 и 8.8.

8.4. Сортировка за линейное время

В некоторых случаях удается проводить сортировку за $O(N)$ операций, то есть за линейное время (вспомним, что лучшие в своем классе алгоритмы быстрой и пирамидальной сортировки работают за $O(N \log N)$ операций). Однако для этого данные должны обладать дополнительными свойствами, кроме того, требуется дополнительная память.

Сортировка подсчетом (counting sort). Если сортируемые записи являются целыми числами, принадлежащими интервалу $[0, M-1]$, где M — некоторая целая константа, можно завести дополнительный массив размером M и за N операций подсчитать, сколько раз каждое число встретилось в сортируемом массиве. Потом потребуется лишь (снова за N операций) заполнить мас-

© www.phys.nsu.ru

Листинг 8.7. Пирамидальная сортировка (процедура просеивания)

```
void _sift(char *a, size_t size,
           int (*compar)(const void *, const void *),
           void *tmp, int l, int r) {
    memcpy(tmp, a+l*size, size);

    for (;;) {
        int ch = 2*l+1;

        if (ch > r)
            break;

        if (ch < r &&
                compar(a+ch*size, a+(ch+1)*size)<0)
            ++ch;

        if (compar(tmp, a+ch*size) >= 0)
            break;

        memcpy(a+l*size, a+ch*size, size);
        l = ch;
    }

    memcpy(a+l*size, tmp, size);
}
```

© www.phys.nsu.ru

Листинг 8.8. Пирамидальная сортировка (основная процедура)

```
void heap_sort(void *base, size_t count, size_t size,
        int (*compar)(const void *, const void *)) {

    int i;
    char buf[1024], *a = (char *)base;
    void *tmp = (size > sizeof (buf))
                ? malloc(size) : buf;

    if (count < 2)
        return;

    for (i = count/2-1; i >= 0; --i)
        _sift(a, size, compar, tmp, i, count-1);

    for (i = count-1; i > 0; --i) {
        _exchange(a, a+i*size, tmp, size);

        _sift(a, size, compar, tmp, 0, i-1);
    }

    if (tmp != buf)
        free(tmp);
}
```

© www.phys.nsu.ru

сив числами согласно их частоте появления. Для примера возьмем массив ($M = 10$):

$$3\ 5\ 3\ 3\ 3\ 7\ 5$$

и подсчитаем, что число 3 встретилось четыре раза, число 5 — два раза, а 7 — один раз. Заполняем массив:

$$\underbrace{3333}_{\text{4 раза}}\quad\underbrace{55}_{\text{2 раза}}\quad 7.$$

Поразрядная сортировка (radix sort). Данный алгоритм использовался в машинах для сортировки перфокарт.

Сначала все элементы упорядочиваются по *младшему* разряду. Для этого нужно использовать любую *устойчивую* сортировку (можно «раскладывать» элементы по «стопкам», используя массивы или списки). Затем упорядочивание происходит по следующей цифре и т. д. Когда упорядочивание пройдет по всем цифрам, массив будет полностью сортирован.

Этот алгоритм часто используется для сортировки по сложным, или *составным* ключам. Например, деканат может захотеть отсортировать список студентов по среднему баллу, но чтобы фамилии студентов с одинаковым средним баллом шли по алфавиту. Получается составной ключ из двух позиций ⟨Средний балл, Фамилия⟩, который может быть использован для работы алгоритма поразрядной сортировки.

Карманная сортировка (bucket sort). Если числовые ключи равномерно распределены в каком-то интервале, можно поделить этот интервал на подынтервалы («карманы»), отсортировать элементы, попавшие в каждый карман, а затем последовательно перечислить элементы каждого кармана.

Например, массив строк можно распределить по карманам, соответствующим первой букве каждой строки, отсортировать каждый карман по отдельности (используя один из общих алгоритмов сортировки), а затем объединить карманы от А до Я.

© www.phys.nsu.ru

Действительные числа в интервале $[0, 1)$ можно распределить по 10 карманам: $[0, 0{,}1), [0{,}1, 0{,}2), \ldots, [0{,}9, 1)$.

Подробнее о сортировках за линейное время $O(N)$ см. [8, гл. 8].

8.5. Внешняя сортировка

Как уже говорилось выше, внешняя сортировка возникает тогда, когда сортируемые данные целиком не помещаются в память. В этом случае не подходит ни один из рассмотренных нами до этого алгоритмов.

Сортировка больших объемов данных возможна на основе следующей идеи. Разобьем все данные на части, которые все же помещаются в память, и отсортируем каждую часть отдельно. Затем *сольем* все части воедино.

Оказывается, что для слияния не требуется много памяти. В самом деле, возьмем два сортированных массива:

$$A = [1, 4, 8, 11, 13, 17] \tag{8.12}$$

$$B = [0, 7, 14, 16, 19, 41] \tag{8.13}$$

Чтобы слить их воедино, заведем для каждого массива по указателю (i и j), изначально указывающему на начало ($i \to 1$, $j \to 0$). На каждом шаге будем сравнивать числа, на которые указывают i и j, выбирать из них меньшее, записывая его в строку результата и увеличивая указатель соответствующего массива.

В примере первым в результат попадет 0, затем 1, 4, 7, 8 и т. д. Когда все указатели дойдут до конца своих массивов, результат будет готов.

Описанный алгоритм, сливающий два массива (двухпутевое слияние) может быть обобщен до N-путевого слияния [7, разд. 5.4.1].

Рассмотрим теперь рекурсивный алгоритм *сортировки слиянием*. Имея массив записей $A = [R_1, \ldots, R_N]$, поделим его попо-

© www.phys.nsu.ru

лам:

$$A_l = [R_1, \ldots, R_{\lfloor N/2 \rfloor}]; \tag{8.14}$$

$$A_r = [R_{1+\lfloor N/2 \rfloor}, \ldots, R_N]. \tag{8.15}$$

Теперь для A_l и A_r рекурсивно вызовем тот же самый алгоритм сортировки слиянием, а результаты сольем по описанной выше схеме.

Этот алгоритм отчасти похож на быструю сортировку, ибо опирается на тот же самый принцип «разделяй и властвуй». Кроме того, он всегда работает за время $O(N \log N)$, вне зависимости от того, какие входные данные.

Алгоритм может использоваться и для задач внутренней сортировки, когда данные умещаются в памяти, однако для работы ему требуется дополнительная память (всего можно обойтись двумя массивами: исходным и еще одним такого же размера).

8.6. Забавные алгоритмы

Кроме забавных языков программирования (см. Brainfuck, Befunge), существуют и забавные алгоритмы сортировки. В этом разделе мы рассмотрим несколько из них.

Блуждающая сортировка (stooge sort). Этот алгоритм построен на принципе «разделяй и властвуй», как быстрая сортировка и сортировка слиянием — предположим, что в данный момент сортируется часть массива $R_l, \ldots, R_r$.

1) Если $K_r < K_l$, поменять местами записи R_l и R_r.

2) Если $r - l \geq 2$, то:

   а) рекурсивно сортировать первые $2/3$ массива:

$$R_l, \ldots, R_{l+\lfloor \frac{2}{3}(r-l+1) \rfloor};$$

© www.phys.nsu.ru

б) рекурсивно сортировать последние $2/3$ массива:

$$R_{l+\lfloor \frac{1}{3}(r-l+1)\rfloor}, \ldots, R_r;$$

в) снова рекурсивно сортировать первые $2/3$ массива.

Данный алгоритм имеет сложность $O(n^{2{,}71\ldots})$, то есть работает *очень медленно!*

Идиотская сортировка (bogosort). Алгоритм состоит из двух шагов:

1) Проверить, является массив уже отсортированным. Если да, то алгоритм завершается.

2) Случайным образом перемешать записи в массиве и перейти к шагу 1.

Тупая сортировка (bozosort). Алгоритм является дальнейшим развитием идей идиотской сортировки (bogosort):

1) Проверить, является массив уже отсортированным. Если да, то алгоритм завершается.

2) Случайным образом поменять местами две записи в массиве и перейти к шагу 1.

© www.phys.nsu.ru

Список литературы

1. *Бентли Джон.* Жемчужины программирования, 2-е изд. — СПб.: Питер, 2002. — 272 с.

2. *Вирт Никлаус.* Алгоритмы и структуры данных. СПб: Невский Диалект, 2008. — 352 с.

3. *Керниган Брайан, Пайк Роб* Практика программирования: Пер. с англ. — СПб.: Невский Диалект, 2001. — 381 с.

4. *Керниган Брайан, Ритчи Деннис* Язык программирования С: Пер. с англ., 3-е изд., испр. — СПб.: Невский Диалект, 2001. — 352 с.

5. *Кнут Дональд.* Искусство программирования, том 1. Основные алгоритмы, 3-е изд.: Пер. с англ. — М.: Вильямс, 2000. — 720 с.

6. *Кнут Дональд.* Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд.: Пер. с англ. — М.: Вильямс, 2000. — 832 с.

7. *Кнут Дональд.* Искусство программирования, том 3. Сортировка и поиск, 2-е изд.: Пер. с англ. — М.: Вильямс, 2000. — 832 с.

8. *Кормен Томас, Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: построение и анализ, 2-е изд.: Пер. с англ. — М.: Вильямс, 2005. — 1296 с.

© www.phys.nsu.ru

9. *Макконнелл Стив.* Совершенный код. Мастер-класс: Пер. с англ. — М.: Издательство «Русская Редакция»; СПб.: Питер, 2007. — 896 с.

10. *Реймонд Эрик.* Искусство программирования для Unix. М.: Вильямс, 2005. — 544 с.[3]

11. *Седжвик Роберт.* Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ. — СПб.: ООО «ДиаСофтЮП», 2002. — 688 с.

12. *Таненбаум Энди.* Архитектура компьютера. 5-е изд. СПб: Питер, 2007. — 848 с.

13. *Уоррен Генри.* Алгоритмические трюки для программистов: Пер. с англ. — М.: Вильямс, 2004. — 288 с.

14. *Хант Эндрю, Томас Дэвид.* Программист-прагматик. Путь от подмастерья к мастеру: Пер. с англ. — СПб.: Лори, 2007. — 270 с.

15. *Bentley John, McIlroy Douglas.* Engineering a Sort Function // Software — Practice and Experience, Vol. 23 (11), pp. 1249—1265 (November 1993). `http://u.nu/758y3`

16. *Boehm Hans-J et al.* Ropes: an Alternative to Strings // Software — Practice & Experience (New York, NY, USA: John Wiley & Sons, Inc.) 25 (12): pp. 1315–1330 (December 1995). `http://u.nu/4ixx3`

17. *Goldberd David.* What Every Computer Scientist Should Know About Floating-Point Arithmetic // ACM Computing Surveys 23 (1): 5–48 (March 1991). `http://u.nu/7eux3`.

18. *Pagh Rasmus et al.* Cuckoo Hashing (2001). `http://u.nu/2eux3`.

[3]Оригинал книги на английском языке (The Art of Unix Programming by Eric S. Raymond) находится в открытом доступе: `http://u.nu/4eux3`.