2

Алгоритмы и структуры данных

В конечном счете, только знакомство с методами и средствами, применяемыми на практике, может помочь найти правильное решение для поставленной задачи, и только наличие определенного опыта позволит неизменно достигать профессиональных результатов.

Рэймонд Филдинг. "Техника съемки спецэффектов"
(Raymond Fielding, The Technique of Special Effects Cinematography*)*

Изучение алгоритмов и структур данных составляет фундамент науки информатики и программирования; это огромная область знаний, изобилующая элегантными методами и тонкими математическими доказательствами. Теорию алгоритмов и структур данных не следует воспринимать как игрушку для ученых-теоретиков. Хороший алгоритм или структура делает возможным секундное решение задачи, на которую могли бы уйти годы работы.

В таких специализированных областях, как графика, управление базами данных, синтаксический анализ, численные методы, моделирование, способность решать задачи почти целиком зависит от знания разработанных в этой области алгоритмов и структур данных. Если вы пишете программу из области, новой для вас, то вы просто *обязаны* выяснить, что уже наработано другими людьми в этой области. Иначе можно потратить время зря, делая плохо то, что другие давно сделали хорошо.

Алгоритмы и структуры данных используются в каждой программе, но лишь в очень немногих программах приходится изобретать что-то действительно новое. Даже в таких сложных программах, как компиляторы и Web-браузеры, структуры данных представляют собой большей частью массивы, списки, деревья и хэш-таблицы. Если потребуются какие-то более сложные объекты, то почти наверняка они будут основаны на перечисленных простых конструкциях. Соответственно, для большинства программистов задача заключается в том, чтобы узнать, какие алгоритмы и структуры данных существуют для решения задач определенного типа, и правильно выбрать нужные из их числа.

Имеется всего лишь несколько фундаментальных алгоритмов, встречающихся почти во всех программах (в основном это алгоритмы поиска и сортировки), причем они нередко включены в состав библиотек. Аналогично, почти все структуры данных являются производными от нескольких базовых типов. Поэтому материал,

рассматриваемый в этой главе, знаком практически всем программистам. Мы написали рабочие версии кода для того, чтобы иметь конкретный материал для обсуждения, и читатель может использовать соответствующие фрагменты кода практически дословно. Однако желательно делать это только после тщательного изучения ассортимента средств, предлагаемого тем или иным языком и его библиотеками.

2.1. Поиск

Если говорить о хранении статических табличных данных, то ничто не может сравниться по эффективности с массивом. Инициализация на этапе компиляции позволяет конструировать массивы самыми простыми средствами и без больших затрат ресурсов. (В языке Java инициализация происходит на этапе выполнения программы, но эти подробности реализации несущественны, если массив не слишком велик.) Пусть, например, мы пишем программу, подсчитывающую в тексте "слова-паразиты". Такие слова обычно встречаются слишком часто в плохо написанном тексте.

```
char *flab[] = {
    "actually",
    "just",
    "quite",
    "really",
    NULL
};
```

Подпрограмма поиска должна знать, сколько элементов содержится в массиве. Один из способов дать ей это понять заключается в том, чтобы передать длину в качестве аргумента, а другой способ (используемый здесь) состоит в том, чтобы поместить маркер NULL в конец массива:

```
/* lookup: последовательный поиск слова в массиве */
int lookup(char *word, char *array[])
{
    int i;
    for (i = 0; array[i] != NULL; i++)
        if (strcmp(word, array[i]) == 0)
            return i;
    return -1;
}
```

В языках C и C++ параметр, представляющий собой массив строк, можно объявить как `char *array[]` или `char **array`. Хотя эти формы эквивалентны, первый способ яснее указывает на способ использования объекта данных.

Данный алгоритм поиска называется *последовательным поиском*, потому что в нем перебираются все элементы подряд, пока не будет найден нужный. Если объем данных сравнительно невелик, последовательный поиск работает достаточно быстро. Существуют стандартные библиотечные функции для выполнения последовательного поиска в совокупностях данных различных типов. Например, такие функции, как `strchr` и `strstr`, разыскивают первое вхождение соответственно символа и подстроки в строке C/C++; класс `String` языка Java содержит метод `indexOf`

для той же цели; нетипизированные алгоритмы `find` языка C++ применимы к большинству типов данных. Если для типа данных, с которым вы работаете, уже есть стандартная функция, воспользуйтесь ею.

Последовательный поиск легко реализуется, но объем выполняемых в нем операций прямо пропорционален объему имеющихся данных. Удвоение количества элементов в массиве приведет к удвоению времени на поиск элемента, особенно если его нет среди данных. Это линейная зависимость, т.е. время поиска является линейной функцией объема данных; вот почему этот метод еще называется *линейным поиском*.

Ниже приведен фрагмент массива более реалистичного размера, взятый из программы синтаксического анализа языка HTML. В этом массиве определяются символические имена для более чем ста текстовых символов:

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
};
/* Символы HTML; например, Aelig обозначает лигатуру A и E. */
/* Значения соответствуют кодировке Unicode/ISO10646. */
Nameval htmlchars[] = {
    "AElig",    0x00c6,
    "Aacute",   0x00c1,
    "Acirc",    0x00c2,
    /* ... */
    "zeta", 0x03b6,
};
```

Для таких больших массивов, как этот, удобнее выполнять *поиск делением пополам* (*дихотомией*), или *двоичный поиск*. Алгоритм двоичного поиска — это математическая формализация того способа, которым мы, люди, разыскиваем слово в словаре. Сначала берем средний элемент. Если он больше того, который мы ищем, переходим к поиску в первой половине массива; в противном случае ищем во второй половине. Повторяем процедуру до тех пор, пока не найдем нужный элемент или не удостоверимся, что его нет среди имеющихся данных.

Для выполнения двоичного поиска таблица данных должна быть отсортирована. В данном случае это так; да и вообще, данные полезно сортировать — это и признак хорошего стиля, и помощь в поиске нужной информации. Кроме того, необходимо знать длину таблицы. В этом может помочь макрос `NELEMS`, приведенный в главе 1:

```
printf("The HTML table has %d words\n", NELEMS(htmlchars));
```

Функция двоичного поиска в этой таблице может выглядеть, например, таким образом:

```
    /* lookup: двоичный поиск строки name в массиве tab; возвращает
индекс */
    int lookup(char *name, Nameval tab[], int ntab)
    {
        int low, high, mid, cmp;

        low = 0;
        high = ntab - 1;
        while (low <= high) {
```

```
            mid = (low + high) / 2;
            cmp = strcmp(name, tab[mid].name);
            if (cmp < 0)
                high = mid - 1;
            else if (cmp > 0)
                low = mid + 1;
            else    /* элемент найден */
                return mid;
        }
        return -1;  /* элемент не найден */
    }
```

Подведем итог. Чтобы найти в массиве `htmlchars` индекс символа S ("одна вторая"), необходимо записать такой оператор:

```
half = lookup("frac12", htmlchars, NELEMS(htmlchars));
```

В ходе двоичного поиска на каждом шаге отбрасывается половина оставшихся данных. Таким образом, процедура включает столько шагов, сколько раз можно поделить количество элементов на 2, пока не останется один элемент. Это равно $\log_2 n$, не считая ошибки округления. Если, например, массив содержит 1000 элементов, то линейный поиск может занять до 1000 шагов, тогда как двоичный — около 10. В массиве из миллиона элементов линейный поиск требует порядка миллиона операций сравнения, а двоичный — порядка 20. Чем больше в массиве элементов, тем большее преимущество имеет алгоритм двоичного поиска над линейным. Начиная с некоторого объема исходных данных (зависящего от конкретной реализации), двоичный поиск всегда работает быстрее, чем линейный.

2.2. Сортировка

Алгоритм двоичного поиска работает только в том случае, если элементы массива отсортированы. Если в некотором наборе данных предстоит часто выполнять поиск, то имеет смысл отсортировать его один раз, а затем использовать процедуру двоичного поиска для нахождения нужных элементов. Если набор данных известен заранее, его можно отсортировать на этапе написания программы и окончательно сформировать при компиляции. Если же это не так, то его придется сортировать на этапе выполнения программы.

Одним из лучших универсальных методов сортировки является алгоритм *быстрой сортировки* (*quicksort*), изобретенный в 1960 году Ч.Э.Р. Хоаром (C.A.R. Hoare). Этот алгоритм — превосходная демонстрация того, как можно обойтись без лишних вычислений. Для его реализации массив делится на "большие" и "малые" элементы:

выбирается один элемент в массиве (назовем его *опорным элементом* — *pivot*);

все остальные элементы делятся на две группы:

"малые", т.е. меньшие, чем опорный;

"большие", т.е. превосходящие по значению опорный или равные ему;

каждая из групп подвергается рекурсивной сортировке.

По окончании этого процесса массив оказывается упорядоченным. Этот алгоритм работает очень быстро, поскольку если известно, что какой-либо элемент меньше опорного, то уже нет нужды сравнивать его со всеми "большими" элементами. Аналогично, "большие" элементы не сравниваются с "малыми". Поэтому данный алгоритм значительно быстрее, чем более простые наподобие сортировки простыми вставками или методом пузырьков, в которых каждый элемент непосредственно сравнивается со всеми остальными.

Алгоритм быстрой сортировки весьма практичен и эффективен; проведено множество исследований его свойств и разработано огромное количество его вариантов. Версия, которую мы приводим здесь, — одна из простейших, но далеко не самых быстрых.

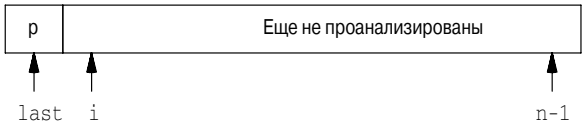
Приведенная ниже функция `quicksort` сортирует массив целых чисел.

```
/* quicksort: сортирует v[0]...v[n-1] в порядке возрастания */
void quicksort(int v[], int n)
{
    int i, last;

    if (n <= 1)  /* ничего не нужно делать */
        return;
    swap(v, 0, rand() % n);  /* переместить опору в v[0] */
    last = 0;
    for (i = 1; i < n; i++)  /* разбиение */
        if (v[i] < v[0])
            swap(v, ++last, i);
    swap(v, 0, last);        /* восстановить опору */
    quicksort(v, last);      /* рекурсивная сортировка */
     quicksort(v+last+1, n-last-1); /* каждой из частей */
}
```

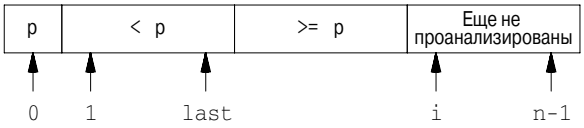
Операция `swap`, меняющая местами два элемента, фигурирует в функции `quicksort` три раза, поэтому ее лучше всего оформить в виде отдельной функции:

```
/* swap: меняет местами элементы v[i] и v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = v[i];
}
```

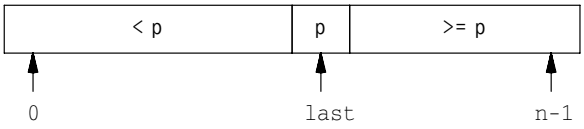
При разбиении массива в нем случайным образом выбирается опорный элемент; затем он временно перебрасывается в начало массива; выполняется перебор остальных элементов, причем "малые" (меньшие, чем опорный) перемещаются в начало (в позицию `last`), а "большие" — в конец (в позицию `i`). В начале этой процедуры, сразу после перемещения опорного элемента в начало, имеет место равенство `last = 0`, а элементы от `i = 1` до `n-1` еще не проанализированы:



Что касается перебора в цикле `for`, то элементы от `1` до `last` строго меньше, чем опорный, элементы от `last+1` до `i-1` больше или равны опорному, а элементы от `i` до `n-1` еще не проанализированы. До тех пор, пока не начнет выполняться соотношение `v[i] >= v[0]`, элемент `v[i]` может участвовать в обмене с самим собой; из-за этого тратится некоторое время, но не так уж много, чтобы об этом беспокоиться.



После разбиения всех элементов на группы элемент `0` меняется местами с элементом `last`, чтобы поместить опорный элемент в его окончательную позицию. Это позволяет сохранить правильный порядок. Теперь массив выглядит следующим образом:



Эта же процедура применяется к левому и правому подмассивам; как только она закончена, массив оказывается упорядоченным.

Насколько быстр алгоритм быстрой сортировки? В наилучшем возможном случае:

на первом этапе n элементов разбиваются на две группы примерно по $n/2$ элементов в каждой;

на втором этапе две группы по примерно $n/2$ элементов разбиваются на четыре по примерно $n/4$ элементов;

далее четыре группы разбиваются на восемь по $n/8$ элементов и т.д.

Процесс протекает примерно в $\log_2 n$ этапов, так что общий объем операций в наилучшем случае пропорционален $n + 2 \times n/2 + 4 \times n/4 + 8 \times n/8...$ ($\log_2 n$ слагаемых), что составляет $n \log_2 n$. В среднестатистическом случае эта процедура лишь немногим более трудоемка. Логарифм по основанию 2 — это стандарт, поэтому основание можно не писать и считать, что затраты времени на быструю сортировку пропорциональны $n \log n$.

Данная реализация быстрой сортировки наиболее удобна для демонстрации, но в ней есть и слабые места. Если при каждом выборе опорного элемента элементы разбиваются на две почти одинаковых по численности группы, то выполненный нами анализ правилен; однако если разбиение на группы часто оказывается слишком неравномерным, то время работы растет примерно по закону n^2. В нашей реализации опорный элемент выбирается случайно. Это снижает вероятность того, что необычность входных данных приведет к слишком большому количеству неравномерных разбиений. Но если все исходные данные одинаковы, то в нашем варианте каждый раз от группы будет отделяться всего один элемент, и время работы приобретет порядок n^2.

Эффективность работы некоторых алгоритмов сильно зависит от исходных данных. Случайно или преднамеренно искаженный характер данных может привести к тому, что вполне стабильный алгоритм вдруг станет работать очень медленно или использовать слишком много памяти. В случае быстрой сортировки наша простейшая реализация иногда работает слишком медленно, однако более совершенные варианты сводят вероятность аномального поведения алгоритма почти к нулю.

2.3. Библиотечные средства

Стандартные библиотеки C и C++ содержат функции сортировки, которые ведут себя устойчиво по отношению к любым входным данным и развивают максимально возможное быстродействие.

Библиотечные функции предназначены для сортировки данных любого типа, однако взамен пользователь должен приспособиться к их интерфейсу, который часто бывает несколько сложнее, чем рассмотренный ранее. В языке C библиотечная функция называется `qsort`, и для ее работы необходимо подготовить функцию сравнения, которая будет вызываться из `qsort` всякий раз, когда необходимо сравнить два значения. Поскольку значения могут иметь любой тип, в функцию сравнения передаются два указателя типа `void*` на сравниваемые элементы данных. Функция приводит указатели к требуемому типу, извлекает значения данных, сравнивает их и возвращает результат (отрицательный, нулевой или положительный, в зависимости от того, является ли первое значение бо́льшим, равным или меньшим, чем второе).

Ниже показан вариант, предназначенный для сортировки массива строк (это часто встречающаяся задача). Определим функцию `scmp`, которая приводит аргументы к нужному типу и вызывает функцию `strcmp` для непосредственного сравнения:

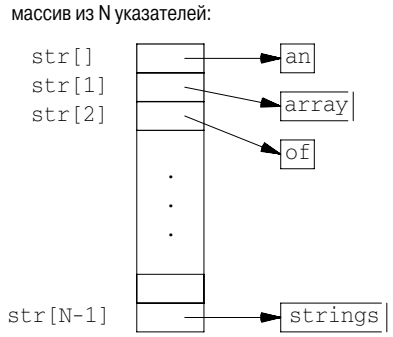
```
/* scmp: сравнение строк *p1 и *p2 */
int scmp(const void *p1, const void *p2)
{
    char *v1, *v2;

    v1 = *(char **) p1;
    v2 = *(char **) p2;
    return strcmp(v1, v2);
}
```

Все это можно было бы записать в одну строчку, но код читать удобнее с применением временных переменных.

Чтобы отсортировать элементы от `str[0]` до `str[N-1]` в массиве строк, в функцию `qsort` следует передать такие аргументы, как сам массив, его длина, размер сортируемых элементов и указатель на функцию сравнения:

```
char *str[N];
qsort(str, N, sizeof(str[0]), cmp);
```

Функцию `strcmp` нельзя использовать непосредственно как функцию сравнения, потому что `qsort` перебирает адреса всех элементов в массиве в виде `&str[i]` типа `char**`, а не `str[i]` типа `char*`, как показано на этом рисунке:



А вот аналогичная функция `icmp` для сравнения целых чисел:

```
/* icmp: сравнение целых чисел *p1 и *p2 */
int icmp(const void *p1, const void *p2)
{
    int v1, v2;

    v1 = *(int *) p1;
    v2 = *(int *) p2;
    if (v1 < v2)
        return -1;
    else if (v1 == v2)
        return 0;
    else
        return 1;
}
```

Можно было бы написать:

```
?    return v1-v2;
```

Но в том случае, если `v2` велико и положительно, а `v1` — велико и отрицательно (или наоборот), получившееся арифметическое переполнение приведет к неправильному ответу. Непосредственное сравнение длиннее, но безопаснее.

И в этом случае вызов функции `qsort` требует передачи массива, его длины, размера элемента и указателя на функцию сравнения в качестве аргументов:

```
int arr[N];

qsort(arr, N, sizeof(arr[0]), icmp);
```

В ANSI C также определена функция двоичного поиска под именем `bsearch`. Как и `qsort`, функции `bsearch` требуется указатель на функцию сравнения (часто это та же самая, что используется в `qsort`). Она возвращает указатель на найденный элемент или `NULL`, если элемент отсутствует. Далее приведена функция `lookup` для поиска символов HTML в кодовой таблице, переписанная с использованием процедуры `bsearch`.

```
/* lookup: ищет строку name в массиве tab, возвращает индекс */
int lookup(char *name, Nameval tab[], int ntab)
{
    Nameval key, *np;

    key.name = name;
    key.value = 0;  /* не используется; может быть любым */
    np = (Nameval *) bsearch(&key, tab, ntab,
                        sizeof(tab[0]), nvcmp);
    if (np == NULL)
        return -1;
    else
        return np-tab;
}
```

Как и в случае `qsort`, функция сравнения получает адреса сравниваемых элементов, поэтому ключ `key` должен иметь соответствующий тип. В данном примере приходится конструировать фиктивную запись `Nameval`, чтобы было что передать в функцию сравнения. Сама функция носит имя `nvcmp` и сравнивает два элемента `Nameval` путем вызова стандартной функции `strcmp` с ее строковыми компонентами в качестве аргументов. Числовые значения игнорируются.

```
/* nvcmp: сравнивает два имени типа Nameval */
int nvcmp(const void *va, const void *vb)
{
    const Nameval *a, *b;
    a = (Nameval *) va;
    b = (Nameval *) vb;
    return strcmp(a->name, b->name);
}
```

Эта функция аналогична `scmp`, но отличается тем, что строки представлены в виде полей структур.

Неудобство, связанное с использованием ключа, показывает, что функция `bsearch` предоставляет пользователю меньшие функциональные возможности, чем `qsort`. Хорошая подпрограмма сортировки общего назначения занимает несколько страниц кода, тогда как функция двоичного поиска обычно не длиннее, чем те несколько операторов, которые обеспечивают интерфейс к ней в программе. И тем не менее рекомендуется пользоваться `bsearch`, а не писать свою собственную функцию. Уже в течение многих лет задача двоичного поиска является на удивление трудной для большинства программистов.

Стандартная библиотека C++ содержит нетипизированный алгоритм под названием `sort`, который гарантирует количество операций порядка $O(n \log n)$. Его интерфейс проще, чем у рассмотренных функций, потому что он не требует ни приведения типов, ни знания размера элементов, ни явно заданной функции сравнения, лишь бы тип исходных данных предполагал отношение упорядоченности.

```
int arr[N];

sort(arr, arr+N);
```

В библиотеке C++ также имеются нетипизированные процедуры двоичного поиска с теми же преимуществами интерфейса.

Упражнение 2.1. Самая естественная форма алгоритма быстрой сортировки — это рекурсивная. Напишите этот же алгоритм в итерационной форме и сравните две его версии. (Ч. Хоар рассказывал, как тяжело было реализовать быструю сортировку итерационным способом и как все стало на свои места, стоило только перейти к рекурсивной форме.)

2.4. Быстрая сортировка в Java

Ситуация в языке Java несколько отличается от C. В ранних версиях Java не было стандартной функции сортировки, и приходилось писать собственные. А вот в более новых версиях функция `sort` уже есть, и она работает с классами, реализующими интерфейс `Comparable`, так что теперь можно сортировать данные библиотечными средствами. Но поскольку сама техника сортировки может пригодиться и в других приложениях, далее мы проработаем все детали реализации алгоритма быстрой сортировки на языке Java.

Нетрудно адаптировать функцию `quicksort` к данным любых типов, сортировка которых может нам понадобиться, но будет более поучительно реализовать процедуру нетипизированной сортировки, которую можно вызывать для любого объекта, т.е. в стиле интерфейса `qsort`.

Существенное отличие этого случая от C и C++ состоит в том, что в языке Java невозможно передать функцию сравнения в другую функцию: там нет указателей на функции. Вместо этого создается *интерфейс*, содержащий одну функцию, которая сравнивает два объекта типа `Object`. Затем для каждого класса сортируемых данных создается класс с методом, реализующим интерфейс для этого конкретного типа. Экземпляр этого класса передается в функцию сортировки, которая в свою очередь пользуется функцией сравнения из состава класса для того, чтобы сравнивать сортируемые элементы.

Начнем с определения интерфейса `Cmp`, в котором объявляется один член — функция сравнения `cmp`, сравнивающая между собой два объекта `Object`.

```
interface Cmp {
    int cmp(Object x, Object y);
}
```

Теперь можно написать функции сравнения для реализации этого интерфейса. Например, в следующем классе определяется функция для сравнения двух объектов типа `Integer`:

```
// Icmp: сравнение объектов Integer
class Icmp implements Cmp {
    public int cmp(Object o1, Object o2);
    {
        int i1 = ((Integer) o1).intValue();
        int i2 = ((Integer) o2).intValue();
        if (i1 < i2)
            return -1;
```

```
        else if (i1 == i2)
            return 0;
        else
            return 1;
    }
}
```

В этом классе сравнению подвергаются объекты `String`:

```
// Scmp: String comparison
class Scmp implements Cmp {
    public int cmp(Object o1, Object o2)
    {
        String s1 = (String) o1;
        String s2 = (String) o2;
        return s1.compareTo(s2);
    }
}
```

Таким способом можно сортировать только данные типов, производных от `Object`; элементы базовых типов наподобие `int` или `double` подобным образом отсортировать нельзя. Вот почему в нашем примере мы сортируем объекты класса `Integer`, а не типа `int`.

Имея эти компоненты, уже можно перевести функцию `quicksort` с языка C на язык Java и сделать так, чтобы она вызывала функцию сравнения из объекта `Cmp`, переданного в нее как аргумент. Наиболее существенное изменение состоит во введении индексов `left` и `right`, поскольку в Java нет указателей на массивы.

```
// Quicksort.sort: быстрая сортировка элементов v[left]..v[right]
static void sort(Object[] v, int left, int right, Cmp cmp)
{
    int i, last;

    if (left >= right ) // ничего не нужно делать
        return;
    swap(v, left, rand(left, right)));  // переместить опорный элемент
    last = left;                        // в v[left]
    for (i = left+1; i <= right; i++)   // разбиение диапазона
        if (cmp.cmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);                // восстановить опорный элемент
    sort(v, left, last-1, cmp);         // рекурсивная сортировка
    sort(v, last+1, right, cmp);        // обеих частей диапазона
}
```

Функция `Quicksort.sort` использует функцию `cmp` для сравнения пары объектов между собой, а затем вызывает `swap`, как и раньше, чтобы поменять их местами.

```
// Quicksort.swap: поменять местами v[i] и v[j]
static void swap(Object[] v, int i, int j)
{
    Object temp;

    temp = v[i];
    v[i] = v[j];
```

```
        v[j] = temp;
    }
```

Генерирование случайных чисел осуществляется отдельной функцией в диапазоне от `left` до `right` включительно:

```
static Random rgen = new Random();

// Quicksort.rand: возвращает случайное целое число
//                 в диапазоне [left,right]
static int rand(int left, int right)
{
    return left + Math.abs(rgen.nextInt())%(right-left+1);
}
```

Здесь с помощью функции `Math.abs` вычисляется абсолютное значение, потому что генератор случайных чисел Java выдает и положительные, и отрицательные числа.

Функции `sort`, `swap` и `rand`, а также объект-генератор `rgen` являются членами класса `Quicksort`.

Наконец, чтобы воспользоваться функцией `Quicksort.sort` для сортировки массива данных типа `String`, запишем следующее:

```
String[] sarr = new String[n];

// Здесь заполняется n элементов массива sarr...

Quicksort.sort(sarr, 0, sarr.length-1, new Scmp());
```

В результате этого функция `sort` вызывается с аргументом в виде объекта для сравнения строк, созданного специально для данного случая.

Упражнение 2.2. Наша реализация быстрой сортировки на Java изобилует операциями приведения типов, поскольку элементы постоянно преобразовываются из их исходного типа (например, `Integer`) в тип `Object` и наоборот. Напишите версию `Quicksort.sort` для сортировки данных вполне определенного типа и поэкспериментируйте с обоими вариантами функции, чтобы выяснить, какое влияние на быстродействие оказывают преобразования типов.

2.5. *O*-оценка

Ранее объем операций, выполняемый тем или иным алгоритмом, уже характеризовался через n — количество элементов в наборе исходных данных. Поиск в несортированных данных требует времени, пропорционального n; если воспользоваться алгоритмом двоичного поиска в отсортированных данных, то затрачиваемое время окажется пропорциональным $\log n$. Время сортировки данных может измеряться порядком n^2 или $n \log n$.

Нам необходим способ формулировать подобные зависимости более четко и конкретно, в то же время абстрагируясь от таких деталей, как быстродействие процессора или качество компилятора (а также квалификация программиста). Желательно было бы сравнивать время выполнения и требуемые объемы памяти тех или иных алгоритмов независимо от языка программирования, компилятора, аппаратной

архитектуры, быстродействия процессора, загрузки операционной системы и других усложняющих анализ факторов.

Для этой цели имеется стандартная форма записи или метод оценки, именуемый *O*-оценкой. Основным ее параметром является n, т.е. размер исследуемой задачи, причем время выполнения алгоритма или его *сложность* представляется в виде функции n. Буква *O* представляет собой сокращение от слова *order* ("порядок"). Таким образом, выражение "двоичный поиск имеет характер $O(\log n)$" означает, что требуется порядка $\log n$ шагов для поиска в массиве из n элементов. Запись $O(f(n))$ означает, что при возрастании n время выполнения алгоритма растет пропорционально $f(n)$, например $O(n^2)$ или $O(n \log n)$. Подобные асимптотические оценки играют большую роль в теоретическом анализе, а также очень помогают при грубом сравнении алгоритмов, но на практике возникают значительные отличия в деталях. Например, алгоритм порядка $O(n^2)$ с небольшим коэффициентом пропорциональности может работать быстрее, чем алгоритм порядка $O(n \log n)$ с высоким коэффициентом для относительно малых n, но по мере роста n преимущество будет оставаться за алгоритмом с медленнее растущей функцией в *O*-оценке.

Следует также различать поведение алгоритмов в *наихудшем* и *среднестатистическом* случаях. Что такое "среднестатистический случай", определить довольно трудно, поскольку это зависит от предполагаемого характера входных данных. Наихудший случай обычно можно определить более-менее точно, хотя он легко может дезориентировать программиста. Например, для алгоритма быстрой сортировки наихудший случай соответствует количеству операций порядка $O(n^2)$, а его среднее быстродействие характеризуется оценкой $O(n \log n)$. Всякий раз тщательно выбирая опорный элемент, можно свести вероятность квадратичного, т.е. $O(n^2)$, объема операций практически к нулю. На практике хорошо написанная процедура быстрой сортировки почти всегда выполняет порядка $O(n \log n)$ операций.

Ниже приведены наиболее важные на практике порядковые оценки.

Оценка	*Характер зависимости*	*Пример операции*
$O(1)$	Постоянная	Обращение по индексу
$O(\log n)$	Логарифмическая	Двоичный поиск
$O(n)$	Линейная	Сравнение строк
$O(n \log n)$	Пропорциональная $n \log n$	Быстрая сортировка
$O(n^2)$	Квадратичная	Простая сортировка
$O(n^3)$	Кубическая	Умножение матриц
$O(2^n)$	Экспоненциальная	Разбиение на группы

Обращение к элементу в массиве имеет порядок $O(1)$, т.е. не зависит от количества элементов. Алгоритм, который на каждом шаге отбрасывает половину оставшихся данных, наподобие двоичного поиска, обычно требует $O(\log n)$ операций. Сравнение двух строк из n элементов с помощью функции `strcmp` имеет характер $O(n)$. Традиционный алгоритм умножения матриц отнимает $O(n^3)$ времени, поскольку каждый элемент на выходе образуется путем перемножения n пар исходных элементов и сложения результатов, а всего в матрице n^2 элементов.

Алгоритмы с экспоненциальным порядком количества операций часто включают в себя вычисление или перебор всех возможных сочетаний и вариантов. Так, в множестве из n элементов имеется 2^n подмножеств, поэтому если в алгоритме выполняется перебор по всем подмножествам, то он имеет характер $O(2^n)$. Экспоненциальные алгоритмы обычно обходятся слишком дорого, если n достаточно велико, потому что добавление в задачу всего одного элемента удваивает количество операций. К сожалению, существует немало задач, в том числе знаменитая "задача коммивояжера", для решения которых известны только алгоритмы экспоненциального порядка. Если необходимо решить именно такую задачу, очень часто прибегают к альтернативным методам, которые дают то или иное приближение к точному правильному ответу.

Упражнение 2.3. Выясните, какие исходные данные являются наихудшими для алгоритма быстрой сортировки. Попытайтесь найти несколько массивов данных, которые бы заставили библиотечную реализацию этого алгоритма работать как можно медленнее. Автоматизируйте процесс экспериментирования так, чтобы подбор и анализ исходных данных выполнялись сами по себе.

Упражнение 2.4. Разработайте и реализуйте алгоритм, сортирующий массив из n целых чисел как можно медленнее. Играть нужно честно: алгоритм должен постепенно сходиться к результату и успешно заканчиваться; нельзя применять такие обманные трюки, как холостые прогоны циклов специально для задержки времени. Какой порядок имеет этот алгоритм по отношению к количеству элементов n?

2.6. Расширяемые массивы

В предыдущих разделах использовались статические массивы, т.е. такие, размер и содержимое которых фиксировались еще на этапе компиляции. Если бы таблицы "слов-паразитов" или символов HTML приходилось модифицировать в процессе выполнения программы, то в качестве базовой структуры данных больше подошла бы хэш-таблица. Добавление в отсортированный массив n элементов по одному за раз является операцией порядка $O(n^2)$, и ее лучше избегать при больших n.

Тем не менее иногда бывает необходимо держать все данные непостоянного объема в одной составной переменной, причем этих данных не должно быть слишком много. Самой удачной структурой данных для подобной задачи по-прежнему остается массив. Чтобы минимизировать затраты на распределение памяти, приращение массива следует выполнять блоками, и для поддержания хорошего стиля программирования массив нужно хранить вместе со статистической информацией, необходимой для его обработки. В языках C++ и Java это делается с помощью классов из стандартных библиотек, а в C — с помощью структур.

В приведенном ниже фрагменте определяется растущий массив элементов типа `Nameval`; новые элементы добавляются в конец массива, который расширяется по мере необходимости. Имеется возможность обратиться к любому элементу массива по его индексу за фиксированное время. Получается некий аналог векторных классов из библиотек Java и C++.

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
};

struct Nvtab {
    int     nval;        /* текущее количество элементов */
    int     max;         /* количество выделенных ячеек */
    Nameval *nameval;    /* массив пар "имя-значение" */
}

enum ( NVINIT = 1, NVGROW = 2 };

/* addname: добавляет новое имя и значение в структуру nvtab */
int addname(Nameval newname)
{
    Nameval *nvp;

    if(nvtab.nameval == NULL) { /* первый раз */
       nvtab.nameval =
            (Nameval *) malloc(NVINIT * sizeof(Nameval));
       if (nvtab.nameval == NULL)
            return -1;
       nvtab.max = NVINIT;
       nvtab.nval = 0;
    } else if (nvtab.nval >= nvtab.max) { /* расширение */
       nvp = (Nameval *) realloc(nvtab.nameval,
                (NVGROW*nvtab.max) * sizeof(Nameval));
       if (nvp == NULL)
            return -1;
       nvtab.max *= NVGROW;
       nvtab.nameval = nvp;
    }
    nvtab.nameval[nvtab.nval] = newname;
    return nvtab.nval++;
}
```

Функция `addname` возвращает индекс только что добавленного элемента или `-1`, если произошла ошибка.

При вызове функции `realloc` массив расширяется до нового размера, сохраняя все уже имеющиеся в нем элементы. Возвращается указатель на новый буфер или `NULL`, если оказалось недостаточно памяти. Удвоение размера массива при каждом вызове `realloc` позволяет сохранить постоянным средний объем операций по копированию одного элемента. Если бы при каждом вызове этой функции добавлялась всего одна ячейка, время таких операций имело бы порядок $O(n^2)$. Поскольку адрес массива может изменяться при перераспределении памяти, в остальной части программы следует обращаться к элементам по индексам, а не через указатели. Обратите внимание, что в коде не употребляется такая конструкция:

```
?    nvtab.nameval = (Nameval *) realloc(nvtab.nameval,
?             (NVGROW*nvtab.max) * sizeof(Nameval));
```

В этом случае, если бы при перераспределении памяти произошла ошибка, все накопленные в исходном массиве данные были бы потеряны.

Работа начинается с массива очень малого исходного размера (`NVINIT = 1`). Это заставляет программу немедленно расширить его и таким образом гарантирует, что данный блок программы сработает. Первоначальный размер массива можно и увеличить перед тем, как выпускать программу для профессионального использования, но, вообще-то, затраты на инициализацию не так уж велики, чтобы об этом беспокоиться.

Возвращаемое из `realloc` значение не обязательно приводить к его окончательному типу, потому что в языке C это делается с указателями `*void` автоматически. А вот в C++ это не так; там явное приведение типа необходимо. Можно поспорить о том, что безопаснее: приводить типы (это более явная запись, в хорошем стиле) или не приводить (приведение типов может скрыть различные тонкие ошибки). Мы решили включить в функцию приведение типов, чтобы сделать код безошибочным с точки зрения как C, так и C++. В результате компилятор C менее тщательно проверяет ошибки, но это уравновешивается дополнительной возможностью проверки сразу двумя компиляторами.

Удаление имени из массива может оказаться довольно хитрой задачей, поскольку нужно решить, что делать с освободившейся ячейкой. Если порядок элементов не имеет значения, то легче всего перебросить последний элемент в образовавшуюся свободную позицию. Если же порядок необходимо соблюсти, то придется сдвинуть на одну ячейку назад все элементы после свободной позиции:

```
    /* delname: удаление первой найденной строки nameval из массива
nvtab */
    int delname(char *name)
    {
        int i;

        for (i = 0; i < nvtab.nval; i++)
            if (strcmp(nvtab.nameval[i].name, name) == 0) {
                memmove(nvtab.nameval+i, nvtab.nameval+i+1,
                    (nvtab.nval-(i+1)) * sizeof(Nameval));
                nvtab.nval--;
                return 1;
            }
        return 0;
    }
```

При вызове функции `memmove` массив “сдавливается”, сдвигаясь внутрь себя на одну позицию. Эта функция принадлежит к стандартной библиотеке и предназначена для копирования блоков памяти произвольной длины.

В стандарте ANSI C определены две функции: `memcpy`, которая работает быстро, но затирает память, если буфер-источник и буфер назначения перекрываются; и `memmove`, работающая медленнее, зато неизменно правильно. Бремя выбора между правильностью и быстродействием не следовало бы возлагать на программиста, так что в идеале функция должна быть только одна. Представим себе, что так оно и есть, и всегда будем пользоваться `memmove`.

Вызов функции memmove можно было бы заменить следующим циклом:

```
int j;
for (j = i; j < nvtab.nval-1; j++)
    nval.nameval[j] = nvtab.nameval[j+1];
```

Мы предпочитаем memmove, потому что ее использование позволяет избежать характерной ошибки копирования элементов в неправильном порядке. Если бы выполнялась вставка, а не удаление, элементы было бы необходимо перебирать в обратном порядке, чтобы не затереть их. Вызывая memmove, мы избавляемся от необходимости всякий раз думать об этом.

Подход, альтернативный перемещению элементов в массиве, — это пометить удаленные элементы как неиспользуемые. Затем, чтобы добавить новый элемент, вначале необходимо найти свободную ячейку, и только в том случае, если таковых нет, расширить массив до нового размера. В данном примере, чтобы пометить элемент как неиспользуемый, в его поле name можно записать значение NULL.

Массивы — это простейший способ группировки данных. Неудивительно, что во многих языках имеются эффективные и удобные средства для их хранения и индексации и что строки представляются именно массивами символов. Массивы просты в использовании, требуют $O(1)$ операций для обращения к любым элементам, отлично подходят для применения двоичного поиска и быстрой сортировки, компактно хранятся в памяти. Для целей хранения наборов данных фиксированного размера, формируемых в процессе компиляции, или относительно небольших совокупностей элементов переменной длины массивы не знают себе равных. Но работа с постоянно меняющимся набором элементов в массиве переменной длины может отнять много времени и ресурсов, поэтому если количество элементов непредсказуемо и может стать очень большим, то лучше обратиться к другим структурам данных.

Упражнение 2.5. В приведенном выше коде функция delname не вызывает функцию realloc, чтобы возвратить память, освободившуюся при удалении. Стоит ли это делать? Как бы вы определили, делать это или нет?

Упражнение 2.6. Внесите необходимые изменения в функции addname и delname так, чтобы удалять имена, помечая их как неиспользуемые. Насколько остальная часть программы изолирована от этих изменений?

2.7. Списки

Вслед за массивами самыми распространенными структурами данных в программах являются списки. Во многих языках есть встроенные списковые типы данных, а некоторые языки (такие как Lisp) вообще основаны на списках. Но в языке C приходится конструировать их самостоятельно. В C++ и Java имеются библиотечные реализации списков, но все равно нужно знать, когда и как ими пользоваться. В этом разделе рассматриваются списки в C, но эта информация может пригодиться и в более широком контексте.

Однонаправленный список — это набор элементов, каждый из которых состоит из хранимых данных и указателя на следующий элемент. Голова списка указывает на его первый элемент, а конец списка обозначается нулевым указателем. Здесь показан список из четырех элементов:

Между массивами и списками есть несколько важных различий. Во-первых, массив имеет фиксированную длину, тогда как список — в точности такую, какая нужна для хранения всех его данных, плюс некоторое дополнительное пространство для хранения указателей. Во-вторых, список можно переупорядочить, всего лишь поменяв местами несколько указателей, что гораздо удобнее, чем перемещать туда-сюда блоки данных. Наконец, при добавлении или удалении новых элементов остальные элементы никуда не перемещаются. Если, например, указатели на элементы списка хранятся в какой-то другой структуре данных, то ни один из них не потеряет свой смысл при добавлении новых элементов.

Все эти различия подсказывают, что если набор данных должен часто изменяться, в частности, если количество элементов заранее не предсказуемо, то такие данные следует организовать в виде списка. А вот массив больше подходит для сравнительно статичных данных.

Имеется всего лишь несколько базовых операций над списком: добавление нового элемента в начало или в конец списка; поиск элемента; добавление нового элемента перед определенным или после него; иногда удаление элемента. Простота устройства списков позволяет по необходимости добавлять дополнительные операции.

Вместо того чтобы определять отдельный тип `List`, в языке C обычно берут структурный тип данных для хранения элементов (например, рассмотренный выше `Nameval`) и добавляют указатель для связи со следующим элементом:

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *next;  /* следующий в списке */
};
```

Непустой список трудно инициализировать на этапе компиляции, поэтому в отличие от массивов списки чаще создаются динамически. Вначале необходимо получить возможность конструировать элементы. Самый прямой способ — это выполнить распределение памяти соответствующей функцией, которую мы назовем `newitem`:

```
/* newitem: создает новый элемент по имени и значению */
Nameval *newitem(char *name, int value)
{
    Nameval *newp;

    newp = (Nameval *) emalloc(sizeof(Nameval));
```

```
    newp->name = name;
    newp->value = value;
    newp->next = NULL;
    return newp;
}
```

Функция `emalloc` будет часто использоваться в этой книге. В ней вызывается `malloc`, а если при распределении памяти происходит ошибка, то выводится сообщение об ошибке и программа завершается. Ее код будет приведен в главе 4, а пока будем воспринимать ее как функцию распределения памяти, никогда не дающую ошибочного результата.

Самый простой и быстрый способ собрать список — это добавлять каждый новый элемент по очереди в голову списка:

```
/* addfront: добавляет newp в голову списка listp */
Nameval *addfront(Nameval *listp, Nameval *newp)
{
    newp->next = listp;
    return newp;
}
```

При модификации списка его первый элемент может измениться, как это и происходит в случае вызова функции `addfront`. Функции, модифицирующие список, должны возвращать указатель на его новый первый элемент, который хранится в переменной, обозначающей весь список в целом. Функция `addfront` и другие функции в этой группе возвращают указатель на первый элемент; типичный способ их вызова имеет вид

```
nvlist = addfront(nvlist, newitem("smiley", 0x263A));
```

Эта конструкция работает даже в том случае, если существующий список — пустой (нулевой), и позволяет легко комбинировать функции в выражениях. Она удобнее, чем альтернативный способ с передачей указателя на указатель, содержащий голову списка.

Добавление элемента в конец списка — это операция порядка $O(n)$, поскольку список необходимо сначала перебрать от начала до конца:

```
/* addend: добавляет newp в конец списка listp */
Nameval *addend(Nameval *listp, Nameval *newp)
{
    Nameval *p;

    if (listp == NULL)
        return newp;
    for (p = listp; p->next != NULL; p = p->next)
        ;
    p->next = newp;
    return listp;
}
```

Если нужно придать функции `addend` характер $O(1)$ по быстродействию, можно завести отдельный указатель на конец списка. Недостаток этого подхода, кроме необходимости хранить и отслеживать хвостовой указатель, состоит в том, что спи-

сок больше нельзя будет представить одной-единственной адресной переменной. Поэтому будем придерживаться более простого стиля.

Для поиска элемента с тем или иным именем следует пройти по цепочке указателей `next`:

```
/* lookup: последовательный поиск имени name в списке listp */
Nameval *lookup(Nameval *listp, char *name)
{
    for ( ; listp != NULL; listp = listp->next)
        if (strcmp(name, listp->name) == 0)
            return listp;
    return NULL;    /* элемент не найден */
}
```

Эта операция занимает $O(n)$ времени, и в целом улучшить ее быстродействие не представляется возможным. Даже если список отсортирован, все равно необходимо перебрать его последовательно для нахождения нужного элемента. Двоичный поиск к спискам неприменим.

Для вывода всех элементов списка можно написать функцию, которая бы перебирала его и последовательно выводила каждый элемент. Чтобы вычислить длину списка, можно написать функцию с простым перебором и инкрементированием счетчика. Есть и альтернативный подход — написать функцию `apply`, выполняющую перебор списка и вызывающую другую заданную функцию для каждого его элемента. Функцию `apply` можно сделать очень гибкой, включив в ее параметры аргумент для передачи в ту, другую функцию. Итак, `apply` будет принимать три аргумента: сам список; указатель на функцию, вызываемую для каждого элемента списка; аргумент для передачи в эту функцию.

```
/* apply: выполняет fn для каждого элемента списка listp */
void apply(Nameval *listp,
        void (*fn)(Nameval*, void*), void *arg)
{
    for ( ; listp != NULL; listp = listp->next)
        (*fn)(listp, arg);  /* вызов функции */
}
```

Второй аргумент функции `apply` — это указатель на функцию, которая принимает два аргумента и возвращает `void`. Она имеет довольно неуклюжий, хотя и стандартный синтаксис:

```
void (*fn)(Nameval*, void*)
```

Здесь `fn` объявляется как указатель на функцию, возвращающую пустое значение `void`, т.е. это переменная, содержащая адрес функции, которая ничего не возвращает. Функция принимает два аргумента. Один из них — это указатель на элемент списка `Nameval*`, а второй — нетипизированный указатель на аргумент для передачи в вызываемую функцию.

Чтобы использовать `apply`, например, для вывода элементов списка, можно написать тривиальную функцию с аргументом в виде строки формата:

```
/* printnv: вывести имя и значения по строке формата arg */
void printnv(Nameval *p, void *arg)
{
    char *fmt;

    fmt = (char *) arg;
    printf(fmt, p->name, p->value);
}
```

В данном случае функция `apply` вызывается таким образом:

```
apply(nvlist, printnv, "%s: %x\n");
```

Для подсчета элементов определяется функция с аргументом в виде указателя на целочисленный счетчик, который нужно инкрементировать:

```
/* inccounter: инкрементирует счетчик *arg */
void inccounter(Nameval *p, void *arg)
{
    int *ip;

    /* p не используется */
    ip = (int *) arg;
    (*ip)++;
}
```

Эта функция вызывается следующим образом:

```
int n;

n = 0;
apply(nvlist, inccounter, &n);
printf("%d elements in nvlist\n", n);
```

Не каждую операцию со списком можно реализовать в таком виде. Например, для уничтожения списка необходимо действовать более осторожно:

```
/* freeall: освобождение всех элементов списка listp */
void freeall(Nameval *listp)
{
    Nameval *next;

    for ( ; listp != NULL; listp = next) {
        next = listp->next;
        /* name освобождается в другом месте */
        free(listp);
    }
}
```

Содержимое памяти нельзя использовать после ее освобождения, поэтому `listp->next` необходимо сохранить в локальной переменной под именем `next` прежде, чем освобождать элемент, на который указывает переменная `listp`. Почему нельзя записать этот цикл, как и прежде, в следующем виде?

```
?    for ( ; listp != NULL; listp = next) {
?        free(listp);
```

В этом случае значение `listp->next` пропадет при вызове функции `free`, и произойдет ошибка.

Обратите внимание, что функция `freeall` не освобождает поле `listp->name`. Предполагается, что поле `name` каждого объекта `Nameval` освобождается где-то в другом месте или же что память для него вообще не отводилась. Чтобы распределение и освобождение памяти для элементов происходило по единой схеме, необходимо согласовать между собой функции `newitem` и `freeall`. Необходимо найти компромисс, гарантирующий освобождение памяти и в то же время сохранение тех элементов, которые нужно сохранить в памяти. В подобных случаях очень часто возникают ошибки. В других языках, таких как Java, эту проблему автоматически решает механизм сборки мусора. Мы еще вернемся к теме управления ресурсами в главе 4.

Удаление одного элемента из списка требует большего труда, чем добавление:

```
/* delitem: удаляет первое имя name из списка listp */
Nameval *delitem(Nameval *listp, char *name)
{
    Nameval *p, *prev;

    prev = NULL;
    for (p = listp; p != NULL; p = p->next) {
        if (strcmp(name, p->name) == 0) {
            if (prev == NULL)
                listp = p->next;
            else
                prev->next = p->next;
            free(p);
            return listp;
        }
        prev = p;
    }
    eprintf("delitem: %s not in list", name);
    return NULL;    /* сюда управление не доходит */
}
```

Как и `freeall`, функция `delitem` не освобождает поля `name`.

Функция `eprintf` выводит на экран сообщение об ошибке и завершает работу программы. Это неуклюже, если не сказать больше. Корректная обработка ошибок представляет собой довольно трудную задачу и требует более подробного обсуждения, которое мы отложим до главы 4. В главе 4 также будет продемонстрирована реализация функции `eprintf`.

Этих базовых структур и операций со списками вполне достаточно для реализации подавляющего большинства распространенных алгоритмов. Но есть и ряд альтернативных средств. Некоторые библиотеки, например стандартная библиотека шаблонов (STL) языка C++, поддерживают двунаправленные списки, в которых каждый узел содержит два указателя — на следующий и предыдущий элементы. В двунаправленных списках несколько больше дополнительные затраты памяти, но зато операции нахождения последнего элемента и удаления текущего имеют характер $O(1)$. В некоторых версиях указатели списков хранятся отдельно от данных, ко-

торые они связывают в список. Такие списки сложнее в обращении, но зато позволяют включать одни и те же элементы сразу в несколько разных списков.

Кроме того, что списки хорошо подходят для тех задач, где выполняется вставка и удаление элементов из середины структуры данных, они еще удобны для управления неупорядоченными данными варьирующегося объема, особенно если обращение к ним выполняется в стековом стиле — "последним вошел, первым вышел". Списки обеспечивают более эффективное использование памяти, чем массивы, в тех случаях, когда сразу несколько стеков должно расширяться и сокращаться независимо друг от друга. Удобно применять списки и тогда, когда данные упорядочены оригинальным способом, а их объем неизвестен априори, как это бывает в случае цепочки слов в текстовом документе. Но если приходится сочетать частое обновление данных с возможностью прямого доступа к ним, то лучше прибегнуть к структуре данных с менее ярко выраженной линейностью, например, к дереву или хэш-таблице.

Упражнение 2.7. Реализуйте некоторые из других возможных операций со списком: копирование, слияние, разбиение, вставку перед конкретным элементом или после него. Как две различные операции вставки отличаются друг от друга по сложности? В какой степени можно воспользоваться приведенными выше функциями, а что придется написать самостоятельно?

Упражнение 2.8. Напишите рекурсивную и итерационную версии функции `reverse`, которая бы изменяла порядок следования элементов в списке на противоположный. Не создавайте новых элементов списка — используйте только существующие.

Упражнение 2.9. Напишите нетипизированное определение типа `List` на языке C. Самый простой способ — это включить в состав каждого элемента списка указатель на данные типа `void*`. Проделайте то же самое на языке C++ в виде шаблона, а также на Java, определив класс для списка, содержащего элементы типа `Object`. Каковы сильные и слабые стороны разных языков в реализации этой задачи?

Упражнение 2.10. Придумайте и реализуйте набор тестов для проверки правильности написанных вами функций работы со списками. Стратегия и тактика тестирования рассматриваются в главе 6.

2.8. Деревья

Дерево — это еще одна иерархическая структура данных, содержащая набор элементов. Каждый элемент содержит значение определенного типа, может указывать на один, нуль или несколько других элементов, и при этом на него указывает только один другой элемент. Исключение составляет *корень* дерева — на этот элемент не указывает никакой другой.

Существует много типов деревьев, соответствующих сложным структурам данных из практических задач, — например, синтаксическое дерево, описывающее синтаксис предложения или оператора языка, или фамильное дерево, описывающее родственные отношения между людьми. Мы проиллюстрируем общие принципы на примере двоичных деревьев (точнее, деревьев двоичного поиска), которые в каждом узле имеют ровно две связи. Такие деревья легче всего реализовать, и при этом они

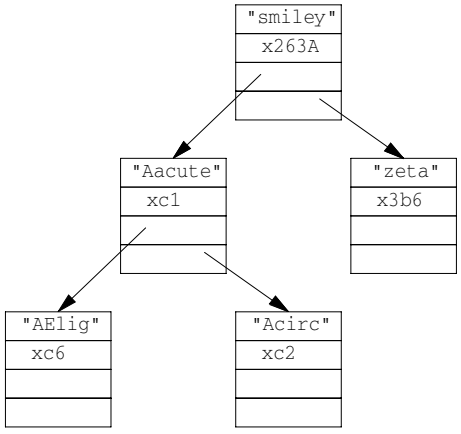
демонстрируют все существенные свойства этого класса структур данных. Каждый узел в двоичном дереве содержит значение и два указателя, `left` и `right`, указывающие соответственно на левый и правый дочерние узлы. Один или оба из этих указателей могут быть равны `NULL`, если узел имеет менее двух потомков. В дереве двоичного поиска значения в узлах полностью определяют структуру дерева: все потомки слева от какого-либо узла имеют меньшие значения, а все потомки справа от него — бо́льшие. В силу этого свойства в таком дереве можно использовать вариант алгоритма двоичного поиска для быстрого нахождения узла по значению.

Вариант структурного типа `Nameval` для использования в дереве строится очень легко:

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *left;  /* меньшее значение */
    Nameval *right; /* большее значение */
};
```

Комментарии о меньшем и большем значениях соответствуют свойствам связей в таком дереве: в левых потомках узла хранятся меньшие значения, тогда как в правых — бо́льшие, чем в самом узле.

Ниже на рисунке приведен конкретный пример дерева — подмножество таблицы символов и их имен, организованное в виде двоичного дерева узлов типа `Nameval`, отсортированных по ASCII-кодам символов.



Поскольку в каждом узле имеются два указателя на другие элементы дерева, многие операции порядка $O(n)$ в списках или массивах имеют характер $O(\log n)$ в деревьях. Наличие двух указателей существенно уменьшает объем операций по перебору дерева, поскольку сужает круг узлов, перебираемых для нахождения элемента.

Дерево двоичного поиска (далее будем называть его просто "деревом") строится таким образом: выполняется рекурсивный спуск с разветвлением влево или вправо, пока не найдется подходящее место для вставки нового узла. Узел должен представлять собой корректно инициализированный объект типа `Nameval` с именем, значением и двумя пустыми указателями. Новый узел добавляется в виде *листа* дерева, временно не имея потомков.

```
/* insert: вставляет newp в дерево treep, возвращает treep */
Nameval *insert(Nameval *treep, Nameval *newp)
{
    int cmp;

    if (treep == NULL)
        return newp;
    cmp = strcmp(newp->name, treep->name);
    if (cmp == 0)
        weprintf("insert: duplicate entry %s ignored",
            newp->name);
    else if (cmp < 0)
        treep->left = insert(treep->left, newp);
    else
        treep->right = insert(treep->right, newp);
    return treep;
}
```

Ранее мы ничего не сказали о дублирующихся записях в дереве. Данная версия функции `insert` сообщает о попытке вставить в дерево дублирующийся узел (при `cmp == 0`). Функция вставки нового элемента в список ничего подобного не делала, поскольку для этого следовало бы перебрать весь список, и операция вставки приобрела бы характер $O(n)$, а не $O(1)$. А вот в случае дерева, во-первых, такая проверка не требует дополнительных усилий, и, во-вторых, качество всей структуры данных пострадало бы от наличия дублирующихся записей. Правда, иногда бывает полезно разрешить вставку дублирующихся записей или же вообще игнорировать факт дублирования.

Функция `weprintf` представляет собой вариант `eprintf`. Она выводит сообщение об ошибке, в начале которого стоит слово `warning` ("предупреждение"), но в отличие от `eprintf` не завершает работу программы.

Дерево, в котором любой путь от корня к листу имеет примерно одинаковую длину, называется *сбалансированным*. Преимущество сбалансированного дерева заключается в том, что поиск по нему имеет характер $O(\log n)$, поскольку, как и в двоичном поиске, на каждом шаге отбрасывается половина оставшихся данных.

Если узлы добавляются в дерево по мере их поступления, дерево может оказаться несбалансированным; более того, оно может стать исключительно разбалансированным. Например, если элементы прибывают в отсортированном виде, то спуск всегда будет выполняться до самого низа одной из ветвей дерева, фактически представляя собой список по указателю `right`. Этот случай характеризуется всеми проблемами быстродействия, присущими спискам. Но если элементы поступают в случайном порядке, то подобная ситуация маловероятна и дерево будет более-менее сбалансированным.

Достаточно сложно реализовать такое дерево, которое гарантированно будет сбалансированным. Это одна из причин, по которой существует так много типов деревьев. Для целей нашего изложения мы оставим все эти вопросы в стороне и будем полагать, что поступающие данные достаточно стохастичны, чтобы поддерживать сбалансированность дерева.

Код функции поиска `lookup` похож на `insert`:

```
/* lookup: ищет имя name в дереве treep */
Nameval *lookup(Nameval *treep, char *name)
{
    int cmp;

    if (treep == NULL)
        return NULL;
    cmp = strcmp(name, treep->name);
    if (cmp == 0)
        return treep;
    else if (cmp < 0)
        return lookup(treep->left, name);
    else
        return lookup(treep->right, name);
}
```

Нужно отметить следующее по поводу функций `lookup` и `insert`. Во-первых, они очень напоминают реализации двоичного поиска, о которых упоминалось в начале этой главы. Это не случайно, поскольку в них заложена та же идея, что и в двоичном поиске ("разделяй и властвуй"), откуда происходит и быстродействие логарифмического порядка.

Во-вторых, эти функции являются рекурсивными. Если их переписать в итерационной форме, они станут еще более похожими на алгоритм двоичного поиска. Фактически итерационный вариант функции `lookup` можно получить одним изящным преобразованием из ее рекурсивной версии. Если элемент не найден, то последняя операция в `lookup` — это возвращение результата вызова себя самой, т.е. рекурсивный вызов выполняется в конце. Такой вызов легко преобразовать в итерационную форму, подставив нужные аргументы и передав управление в начало. Самый прямой способ — это воспользоваться оператором `goto`, но цикл `while` лучше с точки зрения стиля:

```
/* nrlookup: нерекурсивный поиск имени name в дереве treep */
Nameval *nrlookup(Nameval *treep, char *name)
{
    int cmp;

    while (treep != NULL) {
        cmp = strcmp(name, treep->name);
        if (cmp == 0)
            return treep;
        else if (cmp < 0)
            treep = treep->left;
        else
            treep = treep->right;
    }
```

```
        return NULL;
    }
```

Как только появляется возможность обхода дерева, дальнейшие операции получаются сами собой. Можно воспользоваться некоторыми методами работы со списками, например, написать общую функцию перебора узлов с вызовом другой заданной функции в каждом узле. Однако на этот раз встает вопрос выбора: когда выполнять операции над узлом и когда обрабатывать остальную часть дерева? Ответ зависит от характера данных, представляемых деревом. Если это упорядоченные данные наподобие дерева двоичного поиска, то левая половина всегда обходится раньше правой. Но иногда упорядоченность данных имеет нетривиальный характер, как, например, в фамильном дереве, и порядок обхода дерева зависит от представляемых взаимоотношений между узлами.

Симметричный (*in-order*) обход имеет место в том случае, когда операция выполняется после обхода левого поддерева и до обхода правого:

```
    /* applyinorder: симметричное применение функции fn к treep */
    void applyinorder(Nameval *treep,
            void (*fn)(Nameval*, void*), void *arg)
    {
        if (treep == NULL)
            return;
        applyinorder(treep->left, fn, arg);
        (*fn)(treep, arg);
        applyinorder(treep->right, fn, arg);
    }
```

Эта процедура применяется, если узлы необходимо обработать в порядке сортировки, например, для вывода их всех по порядку:

```
    applyinorder(treep, printnv "%s: %x\n");
```

Она также обеспечивает рациональный метод сортировки: вставить все элементы в дерево, разместить в памяти массив подходящей длины, а затем скопировать узлы по порядку в массив симметричным обходом дерева.

При *концевом* (*post-order*) обходе дерева операция над узлом инициируется только после обхода всех его потомков:

```
    /* applypostorder: концевой обход с вызовом fn */
    void applypostorder(Nameval *treep,
            void (*fn)(Nameval*, void*), void *arg)
    {
        if (treep == NULL)
            return;
        applypostorder(treep->left, fn, arg);
        applypostorder(treep->right, fn, arg);
        (*fn)(treep, arg);
}
```

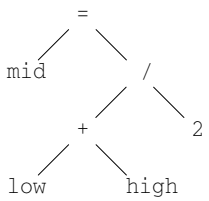
Концевой обход применяется в тех случаях, когда операция над узлом зависит от результата аналогичных операций над его потомками. Примеры таких операций — это вычисление высоты дерева (равной наибольшей из высот двух поддеревьев плюс единица), размещение графического изображения дерева на странице или экране в программе построения диаграмм (для этого нужно выделить каждому поддереву место в рабочем пространстве и суммировать их), измерение общей емкости дерева.

Третий способ обхода — *прямой* (*pre-order*) — применяется очень редко, так что не будем здесь тратить время на его обсуждение.

На практике деревья двоичного поиска применяются довольно редко, хотя B-деревья, характерные очень интенсивным ветвлением, используются для хранения информации на вспомогательных носителях. В повседневной работе программиста часто встречается такое применение деревьев, как разложение структуры оператора или выражения. Например, следующий оператор можно представить в виде *синтаксического дерева*:

```
mid = (low + high) / 2;
```

Синтаксическое дерево этого оператора показано на рисунке.



Чтобы проделать вычисления по даному дереву, следует выполнить его концевой обход, применяя соответствующие операции в каждом узле. Более подробно синтаксические деревья рассматриваются в главе 9.

Упражнение 2.11. Сравните быстродействие функций `lookup` и `nvlookup`. Какова разница между рекурсивной и итерационной формами?

Упражнение 2.12. Напишите функцию сортировки с симметричным обходом. Какой порядок по быстродействию имеет данная операция? При каких условиях она может работать плохо? Каковы ее характеристики по сравнению с алгоритмом быстрой сортировки и с библиотечными функциями?

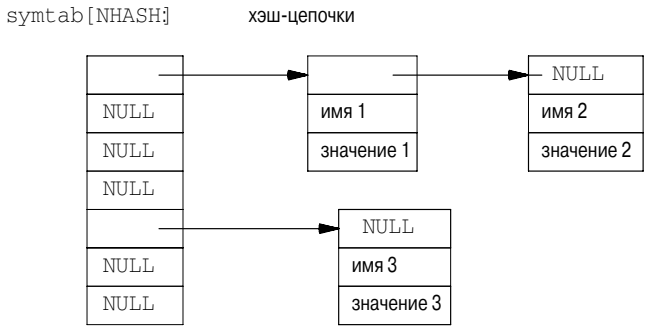
Упражнение 2.13. Придумайте и реализуйте набор тестов для проверки правильности функций работы с деревьями, рассмотренных выше.

2.9. Хэш-таблицы

Хэш-таблицы — это одно из величайших достижений в науке программирования. В них объединены черты массивов, списков и некоторых сложных математических методов. В результате получаются эффективные структуры для хранения и обработки динамических данных. Типичным применением хэш-таблиц являются *таблицы символов*, устанавливающие ассоциации между некоторыми значениями (*данными*) и элементами динамического набора строк (*ключей*). Почти в любом компиляторе

используется хэш-таблица, хранящая информацию о всех переменных компилируемой программы. Web-браузеры часто используют такие таблицы для учета просмотренных страниц, а программы связи с Internet прибегают к хэш-таблицам для кэширования доменных имен и IP-адресов.

Общая идея состоит в том, чтобы передать ключ в специальную *хэш-функцию* и получить *хэш-код*, значение которого принадлежало бы к ограниченному целочисленному диапазону, равномерно заполненному такими ключами. Затем полученный хэш-код используется для индексирования таблицы, в которой непосредственно хранится информация. В языке Java имеется стандартный интерфейс для работы с хэш-таблицами. В C и C++ имеется свой стиль их реализации. Обычно с каждым хэш-кодом, или ячейкой (*bucket*) таблицы, ассоциируется список элементов, соответствующих этому коду, как показано на рисунке.



На практике хэш-функция задается заранее, и для массива данных заблаговременно распределяется необходимый объем памяти (часто даже на этапе компиляции программы). Каждый элемент массива представляет собой список, который связывает элементы с общим хэш-кодом в цепочку. Другими словами, хэш-таблица из n элементов является массивом списков, средняя длина которых составляет $n /$ (размер массива). Извлечение элемента из таблицы есть операция порядка $O(1)$ при условии, что хэш-функция подобрана удачно и списки не слишком разрастаются в длину.

Поскольку хэш-функция представляет собой массив списков, тип ее элементов должен быть тот же, что и у списка:

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *next;      /* следующий в цепочке */
};

Nameval *symtab[NHASH]; /* таблица символов */
```

Для работы с отдельными цепочками ячеек применяются методы, рассмотренные в разделе 2.7. Если подобрана хорошая хэш-функция, то не будет никаких проблем: вычисляйте хэш-код, идите в нужную ячейку и перебирайте список, пока не найдете требуемый элемент. Ниже приведен код функции поиска и добавления элемента.

Если функция находит элемент в хэш-таблице, она возвращает указатель на него. Если элемент не найден и при этом установлен флаг создания `create`, то элемент добавляется в таблицу. Копия имени не создается; предполагается, что вызывающая функция сама позаботится об этом.

```
    /* lookup: находит имя name в таблице symtab, по необходимости
добавляя */
    Nameval* lookup(char *name, int create, int value)
    {
        int h;
        Nameval *sym;

        h = hash(name);
        for (sym = symtab[h]; sym != NULL; sym = sym->next)
            if (strcmp(name, sym->name) == 0)
                return sym;
        if (create) {
            sym = (Nameval *) emalloc(sizeof(Nameval));
            sym->name = name; /* размещается в другом месте */
            sym->value = value;
            sym->next = symtab[h];
            symtab[h] = sym;
        }
        return sym;
    }
```

Это сочетание поиска и создания нового элемента по мере необходимости широко распространено. Если не сочетать эти операции, работа будет дублироваться. Придется писать так:

```
    if (lookup("name") == NULL)
        additem(newitem("name", value));
```

В этом случае хэш-код вычисляется дважды.

Каким должен быть размер массива? Общая идея состоит в том, что массив должен быть достаточно велик, чтобы цепочка каждого хэш-кода содержала поменьше элементов, и операция поиска имела характер $O(1)$. Например, в компиляторе таблица может содержать несколько тысяч ячеек, поскольку большой файл исходного кода обычно содержит несколько тысяч строк, и ожидается, что новые идентификаторы будут встречаться с примерной частотой по одному на строку.

Теперь необходимо решить, что же будет вычислять хэш-функция `hash`. Функция должна работать по детерминистическому алгоритму, выдавать код достаточно быстро и при этом равномерно распределять данные по массиву. В одном из распространенных алгоритмов хэш-кодирования строк хэш-код строится путем прибавления каждого байта строки по очереди к уже накопленному хэш-коду, умноженному на некоторый коэффициент. Умножение позволяет распределить биты из нового байта по накопленному коду; в конце цикла кодирования получается хорошо перемешанная “каша” из байтов входных данных. Для строк в кодировке ASCII эмпирически выведенное значение коэффициента, используемого при таком хэш-кодировании, составляет от 31 до 37.

```
enum { MULTIPLIER = 31; }
/* hash: вычисляет хэш-код строки */
unsigned int hash(char *str)
{
    unsigned int h;
    unsigned char *p;

    h = 0;
    for (p = (unsigned char *) str; *p != '\0'; p++)
        h = MULTIPLIER * h + *p;
    return h % NHASH;
}
```

В этой операции явным образом используются символы типа `unsigned char` (без знака), поскольку стандарт C или C++ не гарантирует наличие или отсутствие знака у символьных переменных, а хэш-код должен быть положительным.

Хэш-функция возвращает остаток от деления накопленного результата на длину массива. Если хэш-функция распределяет ключи равномерно, точный размер массива не имеет значения. Однако стопроцентную надежность хэш-функции гарантировать трудно, и даже наилучшие функции могут столкнуться с проблемами при обработке некоторых наборов входных данных. Поэтому рекомендуется сделать размер массива равным простому числу и тем самым дополнительно подстраховаться, поскольку в этом случае размер массива, коэффициент хэш-кодирования и вероятные значения данных не будут иметь общего делителя.

Эксперименты показывают, что для очень широкого круга строк трудно построить хэш-функцию, которая бы работала заметно эффективнее, чем приведенная выше. Зато очень легко сконструировать такую, которая бы работала хуже. В ранней версии Java существовала хэш-функция для кодирования строк, которая работала тем эффективнее, чем длиннее была строка. Эта хэш-функция экономила время, принимая в расчет только 8 или 9 символов через равномерные интервалы, начиная с начала, если кодируемая строка была длиннее 16 символов. К сожалению, хоть эта функция и работала быстро, ее плохие статистические показатели сводили на нет все преимущества быстродействия. Пропуская фрагменты строки, она игнорировала наиболее значимые ее части. Например, имена файлов начинаются с длинных, практически одинаковых префиксов — имен каталогов — и отличаются только последними несколькими символами (`.java` или `.class`). URL-адреса обычно начинаются с `http://www.`, а заканчиваются `.html`, имея тенденцию отличаться только в середине. Таким образом, хэш-функция часто принимала во внимание только неизменную часть строки и в результате строила длинные списки элементов в ячейках, снижавшие скорость поиска. Проблема была решена заменой этого алгоритма на другой, аналогичный показанному выше (с коэффициентом 37), который принимал во внимание все символы строки.

Хэш-функция, хорошо работающая с данными одного вида (например, короткими именами переменных), может оказаться неудачной в работе с другими (такими как URL-адреса), поэтому хэш-функцию для своей программы следует тестировать на типичных наборах входных данных. Хорошо ли она кодирует короткие строки? А длинные? А строки одинаковой длины с небольшими отличиями?

Строки — это не единственные объекты, которые можно организовывать в хэш-таблицы (хэш-кодировать). Например, в программе физического моделирования и расчета можно хэш-кодировать трехмерные координаты частиц, тем самым организуя хранилище данных в виде линейной таблицы порядка O(*количество частиц*) вместо трехмерного массива порядка $O(размерX \times размерY \times размерZ)$.

Одно из замечательных применений хэш-кодирования можно найти в программе Джерарда Хольцмана (Gerard Holzmann) под названием Supertrace, предназначенной для анализа протоколов и систем параллельной обработки данных. Программа Supertrace получает на вход полную информацию о всех возможных состояниях анализируемой системы и хэширует ее, чтобы получить адрес единственного бита памяти. Если этот бит установлен, значит, состояние уже было зарегистрировано ранее; если сброшен, то нет. В программе Supertrace используется хэш-таблица длиной во много мегабайт, но в каждой ячейке хранится всего один бит. Цепочки (списки) данных не создаются; если вдруг два состояния *конфликтуют*, получая один и тот же хэш-код, программа просто игнорирует этот факт. Программа полагается на низкую вероятность подобного конфликта (эта вероятность не обязана быть нулевой, поскольку алгоритм работы Supertrace носит вероятностный, а не детерминистический характер). Поэтому хэш-функция в ней написана особенно тщательно, с использованием *циклического избыточного контроля* (*cyclic redundancy check*), который позволяет получить очень равномерную "смесь" исходных данных.

Хэш-таблицы прекрасно подходят для организации таблиц символов, поскольку обеспечивают доступ к любому элементу за время порядка $O(1)$. Но у них есть и ряд недостатков. Если хэш-функция написана плохо или размер таблицы недостаточен, то списки в ячейках вырастают до нерациональной длины. Поскольку эти списки не отсортированы, время обращения к данным возрастает до $O(n)$. Элементы нельзя непосредственно расположить в отсортированном порядке, но можно подсчитать их, выделить массив нужной длины, заполнить его указателями на элементы и отсортировать их. И все же при правильной организации хэш-таблиц им нет равных среди других структур данных по таким показателям, как объем операций, требуемый для поиска, вставки и удаления элементов.

Упражнение 2.14. Наша хэш-функция имеет довольно общий характер и удобна при работе со строками. Однако с некоторыми исходными данными она может справляться недостаточно эффективно. Сконструируйте набор данных, который бы заставил эту функцию работать плохо. Насколько трудно построить такой набор для различных значений `NHASH`?

Упражнение 2.15. Напишите функцию для обращения к последовательным элементам хэш-таблицы в несортированном порядке.

Упражнение 2.16. Модифицируйте функцию `lookup` так, чтобы при превышении средней длиной списка некоторого порога `x` массив расширялся бы автоматически с коэффициентом пропорциональности `y` и чтобы хэш-таблица подвергалась перестройке.

Упражнение 2.17. Разработайте хэш-функцию для хранения координат точек в двумерном пространстве. Насколько легко адаптировать вашу функцию к изменениям типа координат (например, от целочисленных к вещественным), системы координат (от декартовой к полярной) или размерности (от двух к более высокой)?

2.10. Резюме

Алгоритм для решения своей задачи следует выбирать в несколько этапов. Вначале проанализируйте потенциально пригодные алгоритмы и структуры данных. Оцените, какой объем данных придется обрабатывать программе. Если в задаче участвует сравнительно немного данных, ограничьтесь простыми методами; если объем данных может возрасти в будущем, отбросьте методы, не приспосабливающиеся к увеличению размерности задачи. Затем выберите средство языка или стандартной библиотеки для решения поставленной проблемы. Если такового не существует, постарайтесь написать (или заимствовать) короткую, простую и легко понятную программную реализацию решения. Протестируйте ее. Только если тесты покажут, что она работает слишком медленно, следует переходить к более усовершенствованным методам.

Хотя существует множество структур данных, и многие из них играют критическую роль для быстродействия программ, на практике большинство программных систем основано на использовании массивов, списков, деревьев и хэш-таблиц. Каждая из этих структур поддерживает определенный набор базовых операций, таких как создание нового элемента; поиск, добавление, удаление элемента; применение некоторых операций ко всем элементам сразу.

Каждая из операций имеет свое характерное время выполнения, которое часто определяет, насколько хорошо конкретный тип данных (или его реализация) подходит для того или иного приложения. Например, массивы предоставляют такой доступ к своим элементам, время которого не зависит от их количества, но зато неспособны достаточно гибко расти или уменьшаться в размерах. В списках удобно выполнять операции добавления и удаления, но для обращения к конкретному элементу требуется порядка $O(n)$ операций. Деревья и хэш-таблицы являются разумным компромиссом: быстрый доступ к отдельным элементам сочетается с простыми механизмами расширения, по крайней мере при соблюдении некоторого критерия сбалансированности.

Для специализированных задач часто создаются более сложные и изощренные структуры данных, но перечисленного набора вполне достаточно для программной реализации подавляющего большинства прикладных задач.

Дополнительная литература

Доступное изложение большого количества разнообразных алгоритмов можно найти в серии книг: Bob Sedgewick, *Algorithms* (Addison-Wesley). Третье издание книги *Algorithms in C++* из этой серии (1998 г.) содержит полезный материал по хэш-функциям и вопросам объема хэш-таблиц. Надежным источником строго и последовательно проанализированных алгоритмов является книга Donald Knuth, *The Art of Computer Programming* (Addison-Wesley). В 3-м томе этой книги (2-е издание, 1998 г.) как раз рассматриваются алгоритмы сортировки и поиска[1].

Программа Supertrace описана в книге Gerard Holzmann, *Design and Validation of Computer Protocols* (Prentice Hall, 1991).

Построение эффективного и устойчивого алгоритма быстрой сортировки рассматривается в статье J. Bentley, D. McIlroy, "Engineering a sort function", *Software — Practice and Experience*, 1993, **23**, 1, p. 1249–1265.

[1] Все три книги Дональда Кнута "Искусство программирования" вышли на русском языке в ИД "Вильямс" в 2000 году. — *Прим. ред.*