

В.А.АНТОНЮК, А.П.ИВАНОВ

# Программирование и информатика

*Краткий конспект лекций*

Москва  
Физический факультет МГУ им.М.В.Ломоносова  
2015

**Антонюк Валерий Алексеевич, Иванов Алексей Петрович**

**Программирование и информатика. Краткий конспект лекций.** – М.:  
Физический факультет МГУ им. М.В. Ломоносова, 2015. – 64 с.

ISBN 978-5-8279-0126-6

Издание представляет собой краткий конспект лекций по информатике, прочитанных на первом курсе физического факультета МГУ в 2012-2015 годах авторами данного пособия. Обсуждаются как базовые численные, так и нечисленные алгоритмы: решения уравнений (методы дихотомии, хорд, касательных, итераций), вычисления определённых интегралов (формулы левых/правых/центральных прямоугольников, формула Симпсона), поиска (алгоритм Бойера–Мура), сортировки (пузырьковая, выбором, quicksort), методы Монте-Карло и способы получения случайных чисел, методы численного решения дифференциальных уравнений (Эйлера, предиктор/корректор, Рунге–Кутты, алгоритм Верле), интерполяции (многочлены Лежандра), решения систем линейных уравнений (метод Гаусса), способы организации динамических данных (вектор, стек, дека, очередь, список, двоичные деревья поиска, ассоциативные контейнеры, В-деревья, хэш-таблицы).

Рассчитано на студентов младших курсов физико-математических специальностей.

Авторы – сотрудники кафедры математического моделирования и информатики физического факультета МГУ.

Рецензенты: доцент С.А.Шлёнов, директор ЦДО Д.Н.Янышев.

Подписано в печать 28.12.2015. Объем 4 п.л. Тираж 50 экз. Заказ № .  
Физический факультет им. М.В.Ломоносова,  
119991 Москва, ГСП-1, Ленинские горы, д.1, стр. 2.  
Отпечатано в отделе оперативной печати физического факультета МГУ.

ISBN 978-5-8279-0126-6

© Физический факультет МГУ  
им. М.В. Ломоносова, 2015  
© В.А.Антонюк, А.П.Иванов, 2015

# Содержание

1.	Представление чисел в компьютерах . . . . .	5
1.1.	Представление положительных целых чисел . . . . .	5
1.2.	Представление отрицательных целых чисел: числа со знаком . . . . .	7
1.3.	Представление вещественных чисел . . . . .	8
2.	Численное решение уравнений . . . . .	10
2.1.	Метод деления отрезка пополам (метод дихотомии) . . . . .	10
2.2.	Метод хорд . . . . .	11
2.3.	Метод касательных . . . . .	12
2.4.	Метод итераций . . . . .	12
3.	Вычисление определённых интегралов . . . . .	14
3.1.	Квадратурные формулы левых и правых прямоугольников . . . . .	14
3.2.	Квадратурная формула центральных прямоугольников . . . . .	15
3.3.	Квадратурная формула трапеций . . . . .	15
3.4.	Квадратурная формула Симпсона (формула парабол) . . . . .	16
4.	Алгоритмы поиска . . . . .	17
4.1.	Поиск элемента в массиве . . . . .	17
4.2.	Алгоритм Бойера-Мура . . . . .	18
5.	Алгоритмы сортировки . . . . .	20
5.1.	Пузырьковая сортировка . . . . .	20
5.2.	Сортировка выбором . . . . .	21
5.3.	Quicksort . . . . .	22
6.	Метод Монте-Карло . . . . .	24
6.1.	Определение площадей (объёмов) фигур . . . . .	25
6.2.	Вычисление числа $\pi$ . . . . .	26
6.3.	Численное интегрирование методом Монте-Карло . . . . .	26
6.4.	Получение случайных чисел . . . . .	26
6.5.	Линейный конгруэнтный метод . . . . .	27
6.6.	Метод регистра сдвига (с линейной обратной связью) . . . . .	28
6.7.	Метод Фибоначчи с запаздываниями . . . . .	28
6.8.	«Вихрь Мерсенна» (Mersenne Twister) . . . . .	28
7.	Решение дифференциальных уравнений . . . . .	30
7.1.	Обыкновенные дифференциальные уравнения (ОДУ) . . . . .	30
7.2.	Задача Коши (задача с начальными условиями) . . . . .	31
7.3.	Ошибки приближённых методов . . . . .	31
7.4.	Устойчивость приближённого решения . . . . .	31
7.5.	Метод Эйлера (метод ломаных) . . . . .	32
7.6.	Метод средней точки (модифицированный метод Эйлера) . . . . .	32
7.7.	Метод «предиктор-корректор» (метод Эйлера с пересчётом) . . . . .	33
7.8.	Классический метод Рунге-Кутты . . . . .	33
7.9.	Алгоритм Верле . . . . .	34
8.	Интерполяция . . . . .	35
8.1.	Общая постановка задачи интерполяции . . . . .	35
8.2.	Интерполяционные многочлены Лежандра . . . . .	35
9.	Системы линейных уравнений . . . . .	38
9.1.	Метод Гаусса . . . . .	38

10.	Динамические данные: способы организации . . . . .	41
10.1.	Общие сведения . . . . .	41
10.2.	Динамический массив (вектор) . . . . .	42
10.3.	Стек, дека, очередь . . . . .	43
10.4.	Список . . . . .	45
10.5.	Двоичное (бинарное) дерево поиска . . . . .	48
10.6.	Ассоциативные контейнеры, В-деревья, хэш-таблицы . . . . .	52

# 1. Представление чисел в компьютерах

## 1.1. Представление положительных целых чисел

С раннего возраста мы привыкаем к записи чисел (количеств) в так называемой десятичной системе счисления. Каждому количеству сопоставлена некоторая последовательность символов (сами эти символы, кстати, служат для обозначения малых количеств – от нуля до основания системы счисления). Т.е., в нашем (привычном для всех) случае символы, обозначающие малые количества, – это цифры 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Они меньше основания системы счисления (т.е., числа 10) и представляют собой набор всех возможных остатков от деления на значение основания.

Поскольку остаток от деления любого числа (количества) на значение основания неизменен, мы можем построить (см. далее) для каждого количества такую последовательность остатков, которая будет уникально характеризовать это количество. А так как для остатков имеются специальные отображающие их символы, мы будем иметь для каждого количества соответствующую ему уникальную последовательность символов.

Возвращаясь к привычной десятичной записи, мы можем сказать, что количество лет, прошедших от р.Х., записывается нами как **2014**, что в реальности означает: это количество есть  $2 \cdot 10^3 + 0 \cdot 10^2 + 1 \cdot 10^1 + 4 \cdot 10^0$ .

Видно, что, хотя в применяемой нами записи и используется один и тот же набор цифр, значимость каждой цифры в записи различна. По этой причине ещё говорят, что используемая нами система записи чисел является *позиционной*: смысл каждой цифры зависит от её позиции.

Какова может быть процедура получения записи числа (количества)  $M$  в позиционной системе счисления по основанию  $N$ ? (Далее предполагаем, что  $M \geq 0$ , а  $N > 0$ ). Например, она может быть вот такой. Найдём остаток  $r_0$  от деления числа  $M$  на  $N$  (а заодно – и частное). После этих действий мы будем знать представление числа  $M$  в виде

$$M = M_0 \cdot N + r_0,$$

где  $M_0$  обозначает частное, а  $r_0$  – остаток от деления  $M$  на  $N$ . Заметим, что  $r_0 < N$ , а потому в случае, когда  $N = 10$ , мы можем вместо  $r_0$  использовать одну из наших десяти цифр. Если  $M_0 > 0$ , то сделаем то же с числом  $M_0$ , т.е., найдём остаток  $r_1$  от деления  $M_0$  на  $N$  и частное  $M_1$ :

$$M_0 = M_1 \cdot N + r_1,$$

С полученным числом  $M_1 > 0$  сделаем аналогичную процедуру и получим  $M_2$  и остаток  $r_2$ :

$$M_1 = M_2 \cdot N + r_2,$$

Понятно, что последовательность значений  $M_0, M_1, M_2, \dots$  – строго убывающая, а снизу она ограничена значением 0. Таким образом, через некоторое (конечное) количество шагов мы придём к значению  $M_n = 0$ , так что последующие значения ( $M_{n+1}$  и далее), равно как и остатки  $r_{n+1}$  и далее, будут равны нулю. Возьмём теперь последовательность остатков  $r_n, \dots, r_1, r_0$  – она и будет уникально характеризовать наше число  $M$ . В десятичной системе каждый остаток имеет собственное обозначение – цифру, значит, мы получим при  $N = 10$  уникальную запись числа  $M$  в десятичной позиционной системе счисления.

Поскольку при обсуждении процедуры мы не делали никаких предварительных предположений о величине числа  $N$ , всё это будет справедливо и для систем с другими основаниями.

Далее нас особенно будут интересовать основания **2** (так называемая двоичная система, она применяется в современных компьютерах), **8** (восьмеричная система), **16** (шестнадцатиричная система). Символы, используемые для обозначения цифр в этих системах (а также в десятичной системе), приведены в таблице.

Основание системы счисления	Цифры для изображения чисел
<b>2</b>	0,1
<b>8</b>	0,1,2,3,4,5,6,7
<b>10</b>	0,1,2,3,4,5,6,7,8,9
<b>16</b>	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Таким образом, способом получения записи числа в любой позиционной системе счисления может быть последовательное получение остатков от деления на основание системы счисления сначала самого числа, а потом – частных от предыдущего деления, – вплоть до получения нулевого результата деления. Причём неважно, в какой системе записано исходное число и основание, важно, чтобы все вычисления производились всё время в одной и той же системе счисления. Полученные остатки записываются «цифрами» системы счисления в обратном порядке.

Для получения записи числа 2014 в двоичной системе последовательно осуществлялось деление на 2 (основание системы), а сами вычисления производились в привычном десятичном виде. Таким образом, десятичное число 2014 в двоичной системе счисления будет иметь вид:

$$2014_{10} = 11111011110_2$$

Здесь – для того, чтобы указать, в какой системе счисления записано число, – использован нижний индекс основания системы счисления.

**Задание.** Найдите в качестве упражнения запись этого же числа в восьмеричной и в шестнадцатиричной системах счисления.

Как осуществляются простейшие арифметические операции в какой-либо позиционной системе счисления? Рассмотрим, например, сложение и умножение в двоичной системе (для других систем действия будут аналогичными). Прежде всего надо составить таблицы сложения и умножения для цифр рассматриваемой системы.

$$\begin{array}{r|ll}
 + & 0 & 1 \\
 \hline
 0 & \mathbf{0} & \mathbf{1} \\
 1 & \mathbf{1} & \mathbf{10}
 \end{array}
 \qquad
 \begin{array}{r|ll}
 \times & 0 & 1 \\
 \hline
 0 & \mathbf{0} & \mathbf{0} \\
 1 & \mathbf{0} & \mathbf{1}
 \end{array}$$

Таблицы сложения и умножения для двоичной системы счисления будут весьма простыми; таблицы для восьмеричной, десятичной и шестнадцатиричной систем – существенно больше. Вот, например, таблица сложения в шестнадцатиричной системе:

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Видно, что результат операции над двумя цифрами может выходить за пределы одной цифры, тогда появляется цифра переноса в следующий разряд (переносы показаны красным цветом).

Затем разряд за разрядом, начиная с младших, выполняется необходимая операция над парами цифр, а возникающие переносы корректируют цифры более старших разрядов – так же, как это делается в десятичной системе.

## 1.2. Представление отрицательных целых чисел: числа со знаком

В обычной записи чисел для указания их знака (как правило, отрицательного) используется специальный символ впереди – минус. Но как быть в случае с представлением чисел в компьютерах, где для представления чисел используются только два символа: 0 и 1?

Для указания знака используется один из разрядов числа (старший), причём единица в нём соответствует отрицательному знаку числа (в таблице ниже знаковый разряд выделен цветом).

Число	Двоичный дополнительный код
+127	0111 1111
⋮	⋮
+1	0000 0001
0	0000 0000
-1	1111 1111
⋮	⋮
-127	1000 0001
-128	1000 0000

Для изменения знака числа в двоичном дополнительном коде нужно выполнить следующие действия: инвертировать все биты числа (включая и знаковый), а затем прибавить к результату единицу (правда, для числа  $-128$  эта процедура не работает...).

### 1.3. Представление вещественных чисел

Представление вещественных чисел в компьютерах в своей основе опирается на экспоненциальную форму числа, в которой хранятся мантисса  $m$  и показатель степени  $e$  (экспонента) вместе со знаком числа, так что образуемое ими число есть

$$\pm m \cdot 2^e$$

Поскольку в подобном представлении всегда имеется неоднозначность (что мешает нам, например, удвоить, учетверить и т.д. значение  $m$ , уменьшив при этом значение экспоненты  $e$  на единицу, двойку и т.п.), используется так называемая нормализованная форма такого представления, когда  $1 \leq m < 2$ , т.е., считается, что первая цифра мантиссы – всегда единица, а потому её даже не имеет смысла хранить; в представлении числа  $m$  хранится лишь его дробная часть, а единица целой части «подразумевается».

Преимущество экспоненциальной формы представления чисел – существенно больший диапазон значений при неизменной относительной точности представления. Используемые сейчас варианты зафиксированы в стандарте IEEE 754. Для хранения вещественной величины одинарной точности (в языке C/C++ соответствующий тип данных называется **float**) отводится 32 бита: 1 – для знака числа  $s$ , 8 – для экспоненты  $e$ , 23 – для мантиссы  $m$ .

Вот как, например, будет представлено в этой форме число 1.0:

0 0 1 1 1 1 1 1 1 0

По заполнению указанных здесь полей можно узнать само число, оно будет равно:

$$(-1)^s \cdot (1 + m) \cdot 2^{e-127} = (-1)^0 \cdot (1 + 0.0) \cdot 2^{127-127} = 1 \cdot 1.0 \cdot 2^0 = 1.0$$

Обратите внимание, что поле экспоненты хранится со смещением (увеличенное на 127), т.е., знак экспоненты присутствует неявно; значения 0 и 255 в поле экспоненты используются специальным образом. Они помогают хранить нулевые значения (которые в экспоненциальной форме с ненулевой мантиссой представить невозможно) и некоторые другие полезные значения. Среди этих специальных значений имеются: два нулевых значения:  $+0$ ,  $-0$ , две бесконечности  $+\infty$  и  $-\infty$ , а также *NaNs* – не-числа (сокращение – от английского выражения *Not-a-Number*). Целые числа, большие 16 777 216 ( $2^{24}$ ) уже не представимы точно и округляются. Предусмотрено 4 режима округления (to Nearest, toward 0, toward  $+\infty$ , toward  $-\infty$ ).

Вещественные величины двойной точности (в языке C/C++ соответствующий тип данных называется **double**) хранятся аналогичным образом. Для их хранения отводится 64 бита: 1 – для знака числа, 11 – для экспоненты, 52 – для мантиссы.

Из-за фиксированности места для хранения чисел в программах мы сталкиваемся с тем, что:

- точность представления чисел – ограничена, только некоторые числа представлены точно (все непредставимые заменяются ближайшим представимым);
- диапазон представления чисел ограничен.

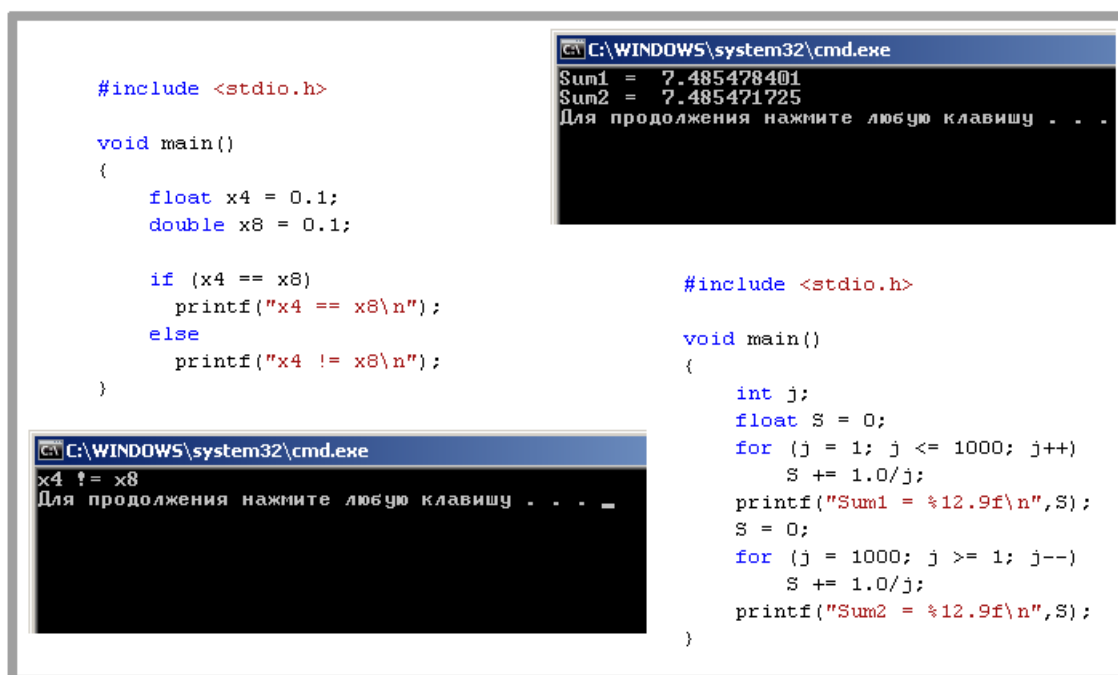


Для вещественных величин типа **float** наибольшее представимое в таком виде число – порядка  $10^{38}$ , ближайшее к нулю положительное число – порядка  $10^{-38}$ . Для вещественных величин типа **double** аналогичные величины – порядка  $10^{308}$  и  $10^{-308}$  соответственно. Таким образом, очень близкие к нулю положительные величины (то же самое верно и для ближайших к нулю отрицательных величин) становятся неотличимыми в программах от нуля, а потому возникает понятие «машинного нуля» – совокупности чисел, эквивалентных в программах нулю, но не равных ему.

По этим причинам некоторые вполне привычные операции становятся невозможными, а получаемые результаты – на первый взгляд совершенно удивительными. В одном из приведённых ниже примеров два вещественных числа разного типа, проинициализированные одной и той же константой, оказываются не равными друг другу!

Причина в том, что число 0.1 – из-за применения двоичной системы – ни в одном из способов не представляется точно, а, значит, используются некоторые приближения к нему, и эти приближения – *разные*.

Во втором примере вычисляется сумма одних и тех же величин, но в разном порядке. Видно, что две суммы заметно различаются в шестом знаке после запятой, хотя точность величин типа **float** – 7-8 десятичных цифр.



```
#include <stdio.h>

void main()
{
    float x4 = 0.1;
    double x8 = 0.1;

    if (x4 == x8)
        printf("x4 == x8\n");
    else
        printf("x4 != x8\n");
}

#include <stdio.h>

void main()
{
    int j;
    float S = 0;
    for (j = 1; j <= 1000; j++)
        S += 1.0/j;
    printf("Sum1 = %12.9f\n",S);
    S = 0;
    for (j = 1000; j >= 1; j--)
        S += 1.0/j;
    printf("Sum2 = %12.9f\n",S);
}
```

```
C:\WINDOWS\system32\cmd.exe
Sum1 = 7.485478401
Sum2 = 7.485471725
Для продолжения нажмите любую клавишу . . .
```

```
C:\WINDOWS\system32\cmd.exe
x4 != x8
Для продолжения нажмите любую клавишу . . .
```

**Выводы:** Вещественные числа в программах не имеет смысла сравнивать на совпадение друг с другом, поскольку такое совпадение практически невозможно. Погрешность при обработке накапливается, а результат вычислений зависит от порядка их исполнения.

## 2. Численное решение уравнений

После небольшого экскурса в существующие способы представления численной информации в компьютерах посмотрим, как осуществляется численное решение уравнений, причём с учётом тех особенностей, которые порождаются используемыми способами представления чисел.

Напомним, какие особенности имеются в виду:

- ограниченность диапазона представимых величин (целочисленные, вещественные)
- неточность представления большинства величин (вещественные)
- неотличимость некоторых величин от нулевого значения (вещественные)

### 2.1. Метод деления отрезка пополам (метод дихотомии)

Пусть  $f(x)$  – непрерывная функция и на концах отрезка  $[a, b]$  имеет значения разных знаков. Тогда (по теореме Вейерштрасса) она имеет на этом отрезке хотя бы один корень. Найдём его с заданной точностью  $\varepsilon$ . Будем предполагать, что на отрезке он – единственный.

Заметим, что непрерывность функции – важна, поскольку иначе корня на отрезке может и не быть. Кроме того, для последующей процедуры важна единственность корня на этом отрезке. Если их будет несколько, то заранее неизвестно, будет ли найден один из корней, и, если найден, какой именно, – всё зависит от точной реализации алгоритма и конкретных значений  $a, b$ .

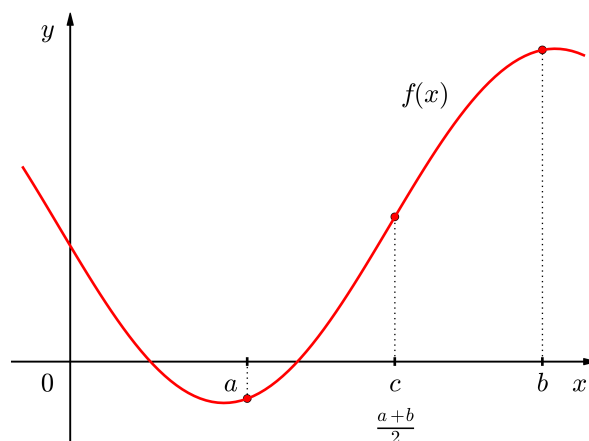


Рис. 1: Метод дихотомии

Возьмём центральную точку отрезка  $c$  и сравним знаки функции в точках  $a$  и  $c$ . Если знаки одинаковы, перейдём к отрезку  $[c, b]$ , если знаки различны – перейдём к отрезку  $[a, c]$ . Новый отрезок вдвое меньше и на нём точно присутствует корень уравнения, поэтому мы можем повторить процедуру для этой половины отрезка. В результате будет получен ещё меньший отрезок, также содержащий корень. Продолжая аналогичным образом, мы будем получать всё меньший и меньший отрезок, содержащий неизвестный корень. Но как только размеры отрезка станут меньше требуемой величины погрешности определения корня, можно остановиться и взять в качестве корня любую точку отрезка, скажем, его середину.

Существуют разные алгоритмические реализации такой процедуры; в некоторых из них можно встретить вместо сравнения знаков функции на концах отрезка проверку знака произведения  $f(a) \cdot f(b)$ . Однако это хуже сравнения знаков у двух значений функции, т.к. при вычислении произведения значений функции вполне может получиться ноль (если оба значения функции будут малы, см. замечания выше), а тогда правильность поведения алгоритма – под большим вопросом, всё будет зависеть от того, какая конкретно операция сравнения использована...

Некоторая неприятность возможна при организации цикла. Проверка малости текущего отрезка  $|b - a| < \varepsilon$  работает хорошо только если сам отрезок находится не слишком «далеко» от нуля. Поскольку вещественные значения, точно представимые в программе, по мере удаления значений от нуля располагаются всё реже и реже, при «удалённом» от нуля отрезке может случиться так, что такая проверка никогда не даст возможности выйти из цикла, так как все ненулевые длины отрезков в области этих значений превышают заданную точность...

Напоследок – ещё одна «неожиданность», вполне вероятная при очень малых отрезках (т.е., тогда, когда мы стремимся найти корень всё точнее и точнее). Формула вычисления середины отрезка при размерах отрезка, сравнимых с точностью представления чисел, может дать нам результат, лежащий на границе отрезка либо даже выходящий за пределы отрезка!

## 2.2. Метод хорд

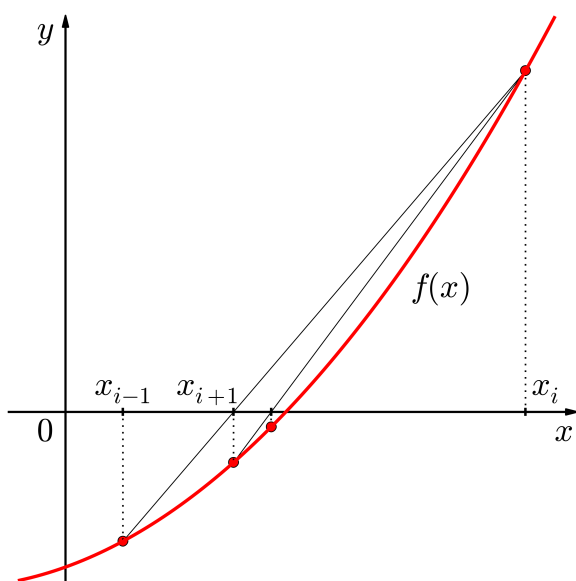


Рис. 2: Метод хорд

В этом методе на интервале  $[x_{i-1}, x_i]$ , содержащем корень, проводится прямая через концы интервала («хорда»). Точку  $x_{i+1}$ , где она пересекает ось абсцисс, мы принимаем за очередное приближение к корню. Соотношение между координатами точки  $x, y$  на прямой, проходящей через две точки  $x_{i-1}, y_{i-1} = f(x_{i-1})$  и  $x_i, y_i = f(x_i)$ , очевидно из подобия, таково:

$$\frac{x_i - x}{x_i - x_{i-1}} = \frac{f(x_i) - y}{f(x_i) - f(x_{i-1})}.$$

Пусть эта прямая пересекает ось абсцисс в точке  $x = x_{i+1}$ , тогда  $y = 0$  и, соответственно,

$$\frac{x_i - x_{i+1}}{x_i - x_{i-1}} = \frac{f(x_i)}{f(x_i) - f(x_{i-1})},$$

откуда

$$x_{i+1} = x_i - f(x_i) \cdot \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}.$$

Далее можно, как и в методе деления отрезка пополам, выбрать из двух полученных интервалов тот, на котором находится корень, а с ним проделать только описанную процедуру построения хорды и получить новое приближение к корню. При этом все хорды будут иметь общую точку в одном из концов первоначального отрезка.

Существует и другой вариант, когда мы просто после получения  $x_{i+1}$  из значений  $x_i, x_{i-1}$  по имеющейся формуле вычисляем  $x_{i+2}$  по значениям  $x_{i+1}, x_i$  и принимаем его в качестве следующего приближения. При этом проверка наличия корня на новом интервале не производится, он может лежать и за пределами интервала, а сам вариант метода скорее можно назвать не методом хорд, а методом секущих.

В любом случае повторение процедуры прекращается, если два последовательных приближения к корню уже достаточно близки друг к другу.

Известно, что если  $f(x)$  – дважды непрерывно дифференцируемая функция и знак  $f''(x)$  сохраняется на рассматриваемом отрезке, то получаемые приближения в методе хорд будут сходиться к корню монотонно.

### 2.3. Метод касательных

Предполагается, что  $f(x)$  дифференцируема на отрезке, где происходит поиск корня. Задаётся начальное приближение вблизи предполагаемого корня (или используется приближение предыдущего шага), строится касательная к функции  $f(x)$  в этой точке, тогда точка пересечения касательной с осью  $x$  даёт следующее приближение.

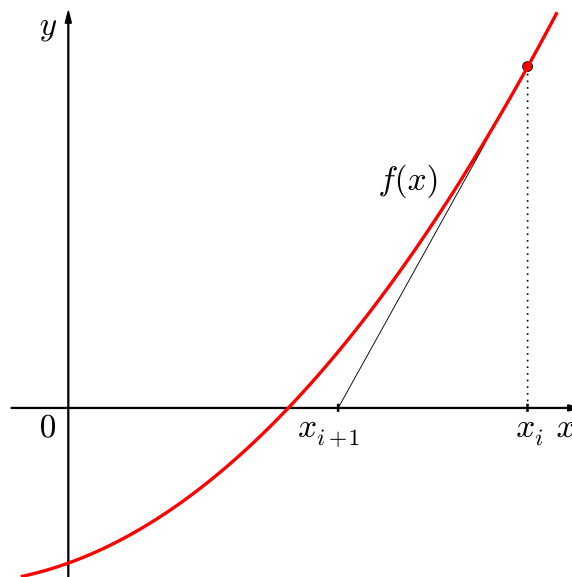


Рис. 3: Метод касательных

Значения  $x_i, x_{i+1}$  связаны уравнением для тангенса угла наклона этой касательной:

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}},$$

откуда получаем явное выражение для  $x_{i+1}$  через  $x_i$ :

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

В этом методе мы тоже останавливаемся тогда, когда два последовательных приближения достаточно близки друг к другу.

Если первоначальное приближение выбрано достаточно близко к корню, последовательность  $x_i$  сходится, причём вблизи корня – монотонно, и с той стороны, с которой  $f(x) \cdot f''(x) \geq 0$ .

Возможные неприятности в этом методе (примеры из книги Н.Н.Калиткина):

- плохо, если корень находится далеко от начального приближения; для уравнения с  $f(x) = x^3 - 2x + 2$  начальное приближение 0 – весьма неудачно, следующее значение приближения в этом методе будет равно 1, затем снова 0, и т.д., следовательно, корень никогда не будет найден;
- плохо, если производная в точке корня равна нулю; самый тривиальный пример –  $f(x) = x^2$ .

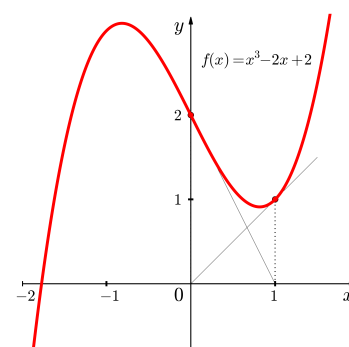
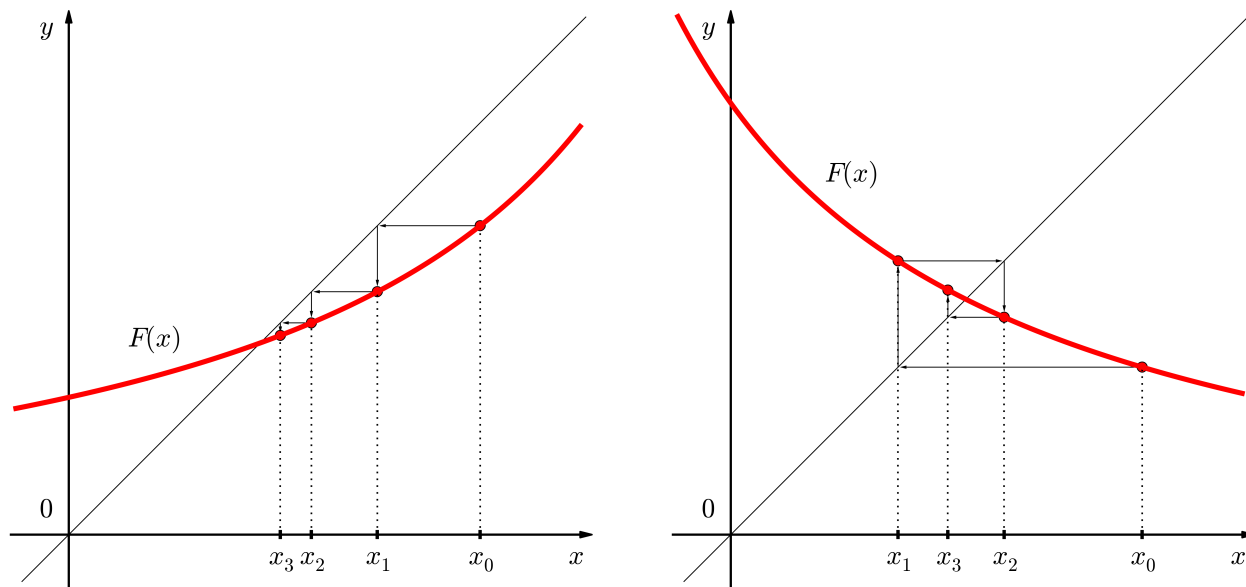


Рис. 4: «Плохой» выбор начального приближения

### 2.4. Метод итераций

Уравнение  $f(x) = 0$  заменяется равносильным, но вида  $x = F(x)$ , и строится последовательность значений  $x$ , такая, что  $x_{i+1} = F(x_i)$ ,  $i = 0, 1, 2, \dots$ , т.е., последующее значение получается из предыдущего с помощью функции  $F$ . Если  $F(x)$  определена и дифференцируема на некотором интервале, причём  $|F'(x)| < 1$ , то тогда последовательность  $x_i$  сходится

к корню уравнения на этом интервале. Иллюстрирующие этот процесс сходимости картинки приведены ниже.



(a) Возрастающая функция  $F(x)$

(b) Убывающая функция  $F(x)$

Рис. 5: Метод итераций: разные типы сходимости к корню уравнения

Обратите внимание на то, что характер сходимости определяется знаком производной  $F'(x)$ .

Таким образом, во всех разобранных методах удаётся построить последовательность приближений к неизвестному корню (в методе дихотомии эта последовательность – неявная, но в качестве неё можно взять, например, последовательность середин получаемых на каждом шаге отрезков), которая к корню сходится.

### 3. Вычисление определённых интегралов

Рассмотрим задачу о приближённом нахождении значения интеграла

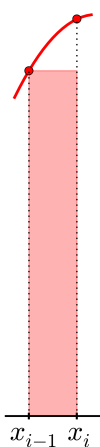
$$I = \int_a^b f(x) dx$$

Относительно  $f(x)$  будем предполагать, что она непрерывна на  $[a, b]$ , а также – когда понадобится, – что она имеет на этом отрезке производные до некоторого порядка. Вычислять значение  $I$  будем по значениям функции  $f(x)$  в некоторых точках  $x_i$ . Значения эти будем считать известными.

Как известно, геометрическая интерпретация смысла определённого интеграла – площадь криволинейной трапеции под графиком функции  $y = f(x)$  над отрезком  $[a, b]$  (в случае отрицательных значений функции часть площади учитывается со знаком минус). Этот отрезок разбивается на части точками деления  $x_1, \dots, x_{n-1}$ ;  $x_0 = a$ ,  $x_n = b$ . Вместо площади под графиком будем приближённо находить суммарную площадь узких полосок над/под отрезками разбиения  $[x_{i-1}, x_i]$ ,  $i = 1, \dots, n$ .

Формулы, по которым приближённо вычисляются значения определённых интегралов, называются **квадратурными формулами**.

#### 3.1. Квадратурные формулы левых и правых прямоугольников



Основанием каждого прямоугольника  $S_i$  служит отрезок  $[x_{i-1}, x_i]$ , высотой в первом случае (формула левых прямоугольников) является значение функции в точке  $x_{i-1}$ , во втором (формула правых прямоугольников) – значение в точке  $x_i$ . В первом случае площадь прямоугольника  $S_i$  будет равна

$$S_i = f(x_{i-1})(x_i - x_{i-1}),$$

во втором

$$S_i = f(x_i)(x_i - x_{i-1}).$$

Суммируя по всем отрезкам разбиения (т.е., по  $i$  от 1 до  $n$ ), получаем **квадратурную формулу левых прямоугольников**:

$$I \approx I_l = \sum_{i=1}^n f(x_{i-1})(x_i - x_{i-1})$$

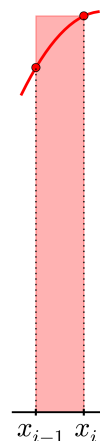
и **квадратурную формулу правых прямоугольников**:

$$I \approx I_r = \sum_{i=1}^n f(x_i)(x_i - x_{i-1}).$$

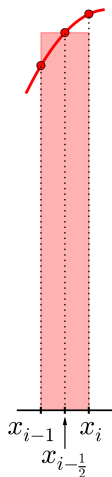
Ошибки имеют в этих формулах разные знаки, в чём легко убедиться, рассматривая для простоты сначала случай монотонной функции: видно, что прямоугольники в одном методе имеют площади немного большие, чем нужно, а в другом – немного меньшие. Можно попытаться взаимно скомпенсировать эти ошибки, взяв полусумму  $I_l$  и  $I_r$  в качестве приближённого значения интеграла

$$I \approx \frac{1}{2}(I_l + I_r) = \frac{1}{2} \sum_{i=1}^n (f(x_{i-1}) + f(x_i))(x_i - x_{i-1}).$$

То, что получилось – **формула трапеций** (см. далее).



### 3.2. Квадратурная формула центральных прямоугольников



В качестве высот рассматриваемых прямоугольников выберем высоты из середин отрезков, т.е. из точек

$$x_{i-\frac{1}{2}} = \frac{1}{2}(x_{i-1} + x_i).$$

Площадь прямоугольника  $S_i$  будет равна

$$S_i = f(x_{i-\frac{1}{2}})(x_i - x_{i-1}),$$

откуда

$$I \approx I_R = \sum_{i=1}^n f(x_{i-\frac{1}{2}})(x_i - x_{i-1}).$$

Когда отрезки разбиения одинаковы (длина одного –  $h = (b - a)/n$ ), тогда

$$I \approx h \sum_{i=1}^n f(x_{i-\frac{1}{2}}).$$

Возникающая ошибка подсчёта интеграла на отрезке зависит от знака второй производной в середине отрезка (прямоугольник равновелик трапеции, верхней стороной которой служит касательная к графику функции в срединной точке):

$$f''(x_{i-\frac{1}{2}}) < 0 \rightarrow \varepsilon_R < 0, \quad f''(x_{i-\frac{1}{2}}) > 0 \rightarrow \varepsilon_R > 0.$$

### 3.3. Квадратурная формула трапеций

Площадь под графиком функции на отрезке заменяется на площадь трапеции, тогда

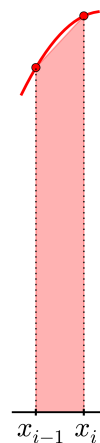
$$I \approx I_T = \frac{1}{2} \sum_{i=1}^n (f(x_{i-1}) + f(x_i))(x_i - x_{i-1}).$$

Если все отрезки одинаковы, то

$$I \approx I_T = \frac{h}{2} \sum_{i=1}^n (f(x_{i-1}) + f(x_i)).$$

В этой сумме все значения функции  $f(x_i)$  – кроме  $f(x_0) = a$  и  $f(x_n) = b$  – встречаются два раза, поэтому можно написать и так:

$$I \approx I_T = \frac{h}{2} (f(a) + f(b) + 2 \sum_{i=1}^{n-1} f(x_i)).$$



Говорят, что формула трапеций имеет второй порядок точности, т.к. при удвоении числа отрезков оценка ошибки уменьшается вчетверо, т.е., как вторая степень длины отрезка  $h$ . Легко заметить, что знаки ошибок  $\varepsilon_R$  и  $\varepsilon_T$  – противоположны, поэтому можно скомбинировать формулы трапеций и центральных прямоугольников. Поскольку, как известно, оценки этих ошибок различаются в два раза ( $\varepsilon_T \approx 2\varepsilon_R$ ), ошибки  $\varepsilon_T/2$  и  $\varepsilon_R$  будут примерно компенсировать друг друга.

$$I \approx I_{RT} = \frac{1}{3}(2I_R + I_T).$$

На отрезке  $[x_{i-1}, x_i]$ :

$$\frac{1}{3}(2f(x_{i-\frac{1}{2}}) + \frac{1}{2}(f(x_{i-1}) + f(x_i))) = \frac{1}{6}(f(x_{i-1}) + 4f(x_{i-\frac{1}{2}}) + f(x_i)),$$

поэтому (для случая одинаковых отрезков разбиения)

$$I \approx I_{RT} = \frac{h}{6} \sum_{i=1}^n (f(x_{i-1}) + 4f(x_{i-\frac{1}{2}}) + f(x_i)).$$

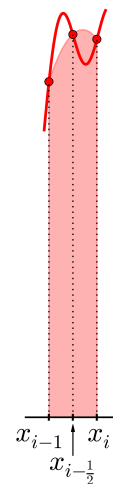
Оказывается, получилась **формула Симпсона** – четвёртого порядка точности.

### 3.4. Квадратурная формула Симпсона (формула парабол)

Эту формулу можно получить и «честно», приближая интегрируемую функцию на каждом отрезке  $[x_{i-1}, x_i]$  фрагментом параболы (квадратным трёхчленом) так, чтобы парабола имела в точках  $x_{i-1}$ ,  $x_i$  и середине отрезка  $x_{i-\frac{1}{2}}$  те же значения, что и функция, а в качестве приближенного значения интеграла от функции на отрезке используя значение интеграла для этого фрагмента параболы.

Поскольку три точки  $(x_{i-1}, f(x_{i-1}))$ ,  $(x_{i-\frac{1}{2}}, f(x_{i-\frac{1}{2}}))$ ,  $(x_i, f(x_i))$  однозначно задают некоторую параболу с коэффициентами  $a_i, b_i, c_i$ , для определения этих коэффициентов можно записать такую систему:

$$\begin{cases} a_i x_{i-1}^2 + b_i x_{i-1} + c_i = f(x_{i-1}) \\ a_i x_{i-\frac{1}{2}}^2 + b_i x_{i-\frac{1}{2}} + c_i = f(x_{i-\frac{1}{2}}) \\ a_i x_i^2 + b_i x_i + c_i = f(x_i) \end{cases}$$



Значение неизвестного интеграла  $S_i$  от функции  $f(x)$  на отрезке  $[x_{i-1}, x_i]$  затем аппроксимируется интегралом от функции  $a_i x^2 + b_i x + c_i$  на этом отрезке:

$$S_i \approx \int_{x_{i-1}}^{x_i} (a_i x^2 + b_i x + c_i) dx = a_i \int_{x_{i-1}}^{x_i} x^2 dx + b_i \int_{x_{i-1}}^{x_i} x dx + c_i \int_{x_{i-1}}^{x_i} dx = a_i \frac{x^3}{3} \Big|_{x_{i-1}}^{x_i} + b_i \frac{x^2}{2} \Big|_{x_{i-1}}^{x_i} + c_i x \Big|_{x_{i-1}}^{x_i}$$

Далее можно, конечно, решать систему и находить неизвестные  $a_i, b_i, c_i$  для этого отрезка, выражая с их помощью нужный интеграл, но мы поступим немного по-другому: мы перегруппируем слагаемые полученной приближённой формулы и увидим, что этот результат может быть выражен только через значения  $f(x_{i-1})$ ,  $f(x_{i-\frac{1}{2}})$ ,  $f(x_i)$  и  $x_i - x_{i-1}$ .

$$S_i \approx a_i \frac{x^3}{3} \Big|_{x_{i-1}}^{x_i} + b_i \frac{x^2}{2} \Big|_{x_{i-1}}^{x_i} + c_i x \Big|_{x_{i-1}}^{x_i} = a_i \frac{x_i^3 - x_{i-1}^3}{3} + b_i \frac{x_i^2 - x_{i-1}^2}{2} + c_i (x_i - x_{i-1})$$

Вынося из последнего полученного выражения общий множитель  $\frac{1}{6}(x_i - x_{i-1})$ , преобразуем оставшийся сомножитель  $2a_i(x_{i-1}^2 + x_{i-1}x_i + x_i^2) + 3b_i(x_{i-1} + x_i) + 6c_i$  далее:

$$\begin{aligned} & 2a_i x_{i-1}^2 + 2a_i x_{i-1} x_i + 2a_i x_i^2 + b_i x_{i-1} + b_i x_i + 2b_i x_{i-1} + 2b_i x_i + c_i + c_i + 4c_i = \\ & = \underbrace{a_i x_{i-1}^2 + b_i x_{i-1} + c_i}_{f(x_{i-1})} + \underbrace{a_i x_i^2 + b_i x_i + c_i}_{f(x_i)} + a_i x_{i-1}^2 + 2a_i x_{i-1} x_i + a_i x_i^2 + 2b_i x_{i-1} + 2b_i x_i + 4c_i = \\ & = f(x_{i-1}) + f(x_i) + 4 \underbrace{\left( \frac{x_{i-1} + x_i}{2} \right)^2 + 4b_i \frac{x_{i-1} + x_i}{2} + 4c_i}_{4f(x_{i-\frac{1}{2}})} = f(x_{i-1}) + 4f(x_{i-\frac{1}{2}}) + f(x_i) \end{aligned}$$

Таким образом, для случая одинаковых отрезков разбиения ( $x_i - x_{i-1} = h$ ,  $i = 1, \dots, n$ ) оценка интеграла  $I$  по всему отрезку  $[a, b]$  будет равна:

$$I \approx \sum_{i=1}^n S_i = \frac{h}{6} \sum_{i=1}^n (f(x_{i-1}) + 4f(x_{i-\frac{1}{2}}) + f(x_i)).$$



## 4. Алгоритмы поиска

### 4.1. Поиск элемента в массиве

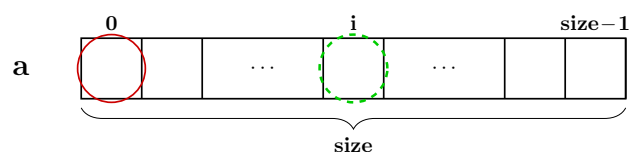
Разберём две простейших задачи: 1) поиска минимального элемента в последовательности (массиве) и 2) поиска номера минимального элемента в массиве.

Мы не знаем, какой элемент в массиве является минимальным, поэтому нам придётся просмотреть их все, сравнивая с каким-то «предполагаемым минимальным». Пусть, например, это будет первый элемент массива.

```
Min = a[0];
for (i = 1; i < size; i++)
    if (a[i] < Min)
        Min = a[i];
```

Для поиска номера – всё аналогично:

```
MinI = 0;
for (i = 1; i < size; i++)
    if (a[i] < a[MinI])
        MinI = i;
```



По завершении просмотра массива мы будем иметь в соответствующих переменных значения искомого элемента или индекса соответственно.

В приведённой реализации мы начинали просмотр от начала массива, но с таким же успехом можно было начинать и от конца. Вообще, если неизвестен порядок следования элементов, то нет никакого предпочтительного порядка просмотра элементов при поиске элемента, минимального/максимального по значению. А вот если бы элементы массива располагались в порядке возрастания, то ни минимальный, ни максимальный элемент искать было бы не нужно, поскольку они располагались бы в начале и конце массива соответственно.

Если бы в массиве присутствовали повторяющиеся значения, то – в зависимости от порядка просмотра – были бы найдены правые или левые минимальные/максимальные значения или их позиции.

Что изменилось бы в наших примерах, если бы мы искали конкретное значение или его положение в массиве? Цикл просмотра всё равно остался бы (только теперь был бы полным, т.е., начинался бы с нулевого индекса), а вот смысл проверки должен был быть другим, поскольку нужна проверка на равенство. Заметим, что, поскольку поиск конкретного значения может быть и безрезультатным, в качестве первоначального значения индекса уже нельзя брать 0 – необходимо другое, «невозможное» значение индекса, с помощью которого можно понять, что после просмотра ничего не найдено. В качестве такого значения можно использовать  $-1$ .

В случае повторяющихся значений в массиве поиск конкретного значения может завершиться «множественными» совпадениями, поэтому для такой ситуации нужно знать, что делать с получаемыми результатами (сохранять или использовать по мере обнаружения совпадения).

Если бы значения в массиве были отсортированы, то искать конкретное значение в нём можно было бы гораздо эффективнее, чем простым перебором всех значений.

Поскольку в программах текстовая информация представлена также в виде числовых значений отдельных символов, то разбираемые примеры подходят для реализации поиска отдельных символов в последовательностях символов (текстах). Можно таким же образом пытаться решить и более сложную задачу: поиска нескольких символов подряд, т.е., после обнаружения первого символа проверять на совпадение следующего со вторым и так далее, а в случае неудачи – переходить к проверке следующей позиции. Однако, такой «наивный» алгоритм весьма неэффективен: существуют более интересные альтернативы.

## 4.2. Алгоритм Бойера-Мура

Этот алгоритм предложен в 1977 году (Boyer, Moore) и считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки (иногда именуемой шаблоном) в строке. Основа его преимуществ в том, что ценой некоторых предварительных вычислений над шаблоном (подстрокой) сравнение шаблона со строкой (текстом) производится не во всех позициях – часть проверок пропускается из-за явной невозможности совпадения.

Просмотр строки производится слева направо, а сравнение с подстрокой – справа налево. Сначала совмещается начало текста (строки) и шаблона (подстроки), а проверка начинается с *последнего* символа шаблона. Если символы совпадают, производится сравнение предпоследнего символа шаблона с символом текста и так далее. Если все символы совпали – подстрока найдена. Если же какой-то символ шаблона не совпадает с соответствующим символом строки, шаблон сдвигается на несколько символов вправо и проверка опять начинается с последнего символа. На сколько производится сдвиг – зависит от символа строки в позиции несовпадения и от того, встречается ли совпавшая часть шаблона в самом шаблоне повторно. Для иллюстрации эффективности этой процедуры рассмотрим «хороший» случай: символ в строке не совпал при сравнении, но в шаблоне (искомой подстроке) его нет. Тогда сдвиг вправо можно делать сразу на длину шаблона! В приводимом примере – это переход от шага 3 к шагу 4: сразу на три позиции.

Текст:		<b>А</b>	<b>Б</b>	<b>Р</b>	<b>А</b>	<b>К</b>	<b>А</b>	<b>Д</b>	<b>А</b>	<b>Б</b>	<b>Р</b>	<b>А</b>	
Шаблон:	шаг 1	<b>Б</b>	<b>Р</b>	<b>А</b>									<i>Примечание 1</i>
	шаг 2		<b>Б</b>	<b>Р</b>	<b>А</b>								<i>Примечание 2</i>
	шаг 3					<b>Б</b>	<b>Р</b>	<b>А</b>					<i>Примечание 3</i>
	шаг 4								<b>Б</b>	<b>Р</b>	<b>А</b>		<i>Примечание 4</i>
	шаг 5									<b>Б</b>	<b>Р</b>	<b>А</b>	<i>Примечание 5</i>

1. Совпадения последнего символа нет, но символ текста на этом месте – часть шаблона. Сдвиг – на одну позицию (длина шаблона минус позиция символа в шаблоне).
2. Полное совпадение: шаблон в тексте найден.
3. Сдвиг на длину шаблона; символа последней позиции нет в шаблоне вообще – сдвиг возможен на всю длину шаблона.
4. Совпадения последнего символа снова нет, но сам символ есть в шаблоне, сдвиг на одну позицию (аналогично шагу 1).
5. Полное совпадение – шаблон найден.

В этом довольно тривиальном примере шаблон весьма прост и никакие его части в нём не повторяются. Если символ первого несовпадения в шаблоне отсутствует, сдвиг вправо будет максимален – на длину шаблона. Если же это не так, сдвиг должен быть меньше, а насколько – зависит от того, как часто в строке встречается совпавшая часть. Возможно, что мы натолкнулись на неё, нужно быть «осторожнее», и сдвиг будет меньше.

Алгоритм Бойера-Мура предполагает использование двух вспомогательных таблиц (они

строятся по содержимому шаблона заранее, до начала поиска), первой из них мы фактически воспользовались в приведённом выше простом примере поиска. Это так называемая **таблица стоп-символов**, содержащая *последнюю позицию* каждого символа в шаблоне (кроме последнего), либо 0.

Символы:	<b>Б</b>	<b>Р</b>	остальные
Позиция:	1	2	0

Сдвиг шаблона после сравнения последнего символа будет меньше длины шаблона на величину из этой таблицы для несовпавшего символа текста. Т.е., при таком шаблоне после несовпадения сдвиг будет почти всегда на три позиции (длина шаблона – три символа), за исключением букв Б и Р, когда сдвиг должен быть на две или одну позицию соответственно, чтобы буква совпала.

Из-за крайней простоты шаблона поиска (БРА) таблица стоп-символов оказалась весьма несложной; вот как она будет выглядеть для шаблона АБРАКАДАБРА (т.е., если бы мы хотели искать в тексте это слово):

Символы:	<b>А</b>	<b>Б</b>	<b>Р</b>	<b>К</b>	<b>Д</b>	остальные
Позиция:	8	9	10	5	7	0

Видно, что в случае несовпадения последнего символа сдвиги будут не такими впечатляющими для букв из шаблона (несмотря на его внушительную длину – 11), только для К и Д сдвиг будет примерно на половину длины шаблона; буквы же Б, Р и особенно А слишком часто встречаются в нём.

Простота предыдущего примера (поиск подстроки БРА в строке АБРАКАДАБРА) проявляется ещё и в том, что шаблон ни в одной из позиций не совпал частично, а потому вторая таблица нам не понадобилась. Эта вторая таблица называется **таблицей суффиксов** (завершающих символов шаблона) и содержит *наименьшую величину, на которую надо сдвинуть шаблон, чтобы он снова совпал с этим суффиксом*. Для поиска шаблона АБРАКАДАБРА в каком-либо тексте это существенно, т.к. он немножко «самоповторяется» и по этой причине сдвиги не должны быть большими, чтобы не пропустить совпадение.

Суффикс:	–	...А	...РА	...БРА	...АБРА	...ДАБРА (и более)
Сдвиг:	1	3	7	7	7	11

Если произошло частичное совпадение суффикса шаблона с текстом, то необходимый сдвиг шаблона надо узнавать на основе такой таблицы, показывающей повторяемость суффикса в теле шаблона. Если такая повторяемость есть, сдвиг будет меньше максимального и его величина уже содержится в таблице суффиксов!

## 5. Алгоритмы сортировки

Под сортировкой числовых таблиц (массивов) понимается перестановка их элементов в определённом порядке, например, по убыванию или возрастанию. После такой перестановки многие операции поиска осуществляются проще и эффективнее. Важна сортировка и для обработки символьной информации, поскольку в этом случае достигается расположение элементов в алфавитном порядке (из-за «правильно» присвоенных символам числовых кодов).

В принципе, алгоритмов сортировки существует довольно много, а различаются они своими характеристиками, такими, как сложность выполнения, эффективность использования памяти и другими. Далее мы рассмотрим реализации нескольких часто используемых алгоритмов: пузырьковой сортировки, сортировки выбором, быстрой сортировки (Quicksort). Общим для всех этих алгоритмов является то, что они не используют дополнительную память: сортировка происходит в рамках сортируемого массива, про который предполагается, что он полностью помещается в оперативную память. Элементы массива для простоты будем считать целыми числами, а сортировать его будем по возрастанию.

Для записи алгоритмов на языке C будем использовать два макроопределения, которые позволят нам сделать всё проще и нагляднее. Вот эти макроопределения:

```
#define Less(x,y) x < y
#define Swap(x,y) {int t = x; x = y; y = t;}
```

В первом результат сравнения является истинным, если первый элемент строго меньше второго. Во втором осуществляется перестановка двух указанных элементов, причём для этого используется временная (локальная) переменная `t`.

Сами алгоритмы будем записывать в виде функции, которой передаются: сортируемый массив и необходимые размеры, указывающие область, где должна происходить сортировка.

Для пузырьковой сортировки и сортировки выбором достаточно одного параметра размера: размера самого массива, поскольку «область воздействия» – весь массив, возможно ограничиваемый с одной из сторон. Для быстрой сортировки (Quicksort) нужны два параметра размера, поскольку в процессе сортировки всего массива таким же способом рекурсивно сортируются его подмножества.

Для создания тестовой программы каждого из алгоритмов сортировки ещё понадобятся: главная функция, в которой будет вызываться необходимая функция сортировки, а также вспомогательная функция отображения массива. На них – в силу их простоты – мы останавливаться не будем.

### 5.1. Пузырьковая сортировка

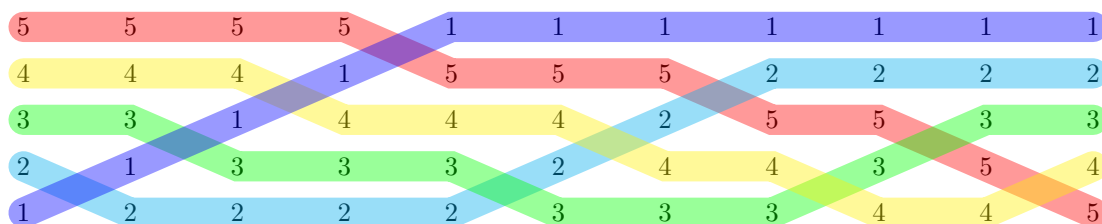
Просматриваем пары соседних элементов (например, справа) и элементы пары переставляем, если порядок «неестественный». Делаем несколько проходов, завершая перестановки на каждом последующем проходе всё раньше и раньше (поскольку минимальные элементы уже попали на своё место).

Пусть  $i$  обозначает позицию, до которой в принципе может дойти после перестановок (оставшийся) минимальный элемент,  $j$  – индекс (правого) элемента из пары соседних элементов; пара не может затронуть элементы левее позиции  $i$ . Тогда функцию для реализации алгоритма можно записать так:

```
void BubbleSort(int a[], int size)
{
    for (int i = 0; i < size-1; i++)
        for (int j = size-1; j > i; j--)
            if (Less(a[j],a[j-1]))
                Swap(a[j],a[j-1]);
}
```

Запись алгоритма включает цикл по всем конечным положениям первого (левого) элемента пары и вложенный в него цикл осуществления перестановок. Этот вложенный цикл не затрагивает элементы с индексами менее  $i$ , поскольку ранее на эти места уже были помещены элементы с минимальными значениями.

Вот как, например, в соответствии с приведённым кодом, протекает процесс сортировки небольшого массива из пяти чисел, расположенных изначально не по возрастанию (как хотелось бы), а по убыванию:



Наблюдение за процессом сортировки массива, изначально расположенного «наоборот», даёт возможность увидеть, как «всплывают» меньшие значения, перемещаясь по массиву, и как на последнем месте оказываются по очереди *все* элементы этого массива, что, конечно, затратно для этого метода сортировки, поскольку там должен стоять максимальный элемент. По этой причине пузырьковая сортировка используется только тогда, когда скорость работы не является критичной.

## 5.2. Сортировка выбором

Перебираем все позиции в массиве (кроме, быть может, последней): это места, где может стоять очередной минимальный элемент (из всех оставшихся). Для каждой позиции его возможного нахождения нужно также обнаружить этот минимальный элемент, для чего перебираются все оставшиеся элементы на предмет обнаружения минимального. Найдя его, мы переместим его на место – путём перестановки – и продолжим перебор позиций далее. К моменту достижения последней позиции окажется, что ничего перемещать будет уже не нужно, поскольку все минимальные элементы уже стоят на своих местах.

Введём обозначения:  $i$  – позиция, где будет стоять очередной минимальный (среди оставшихся) элемент,  $k$  – это позиция, где он будет найден; пока он не найден (а он может быть и не найден), полагаем, что  $k=i$ , т.е., что это – минимальный;  $j$  – текущий элемент для анализа; если выполняется условие  $a[j] < a[k]$ , значит, найден ещё меньший, чем тот, что стоит в позиции минимального.

```

void SelectionSort(int a[], int size)
{
    for (int i = 0; i < size-1; i++)
    {
        int k=i;
        for (int j = i+1; j < size; j++)
            if (Less(a[j],a[k]))
                k = j;
        if (i != k)
            Swap(a[i],a[k]);
    }
}

```

Внешний цикл – просмотр по очереди всех позиций (кроме последней) и перемещение в эту позицию минимального из оставшихся элементов (последняя позиция не нужна, поскольку последний оставшийся элемент в ней и будет находиться), внутренний – поиск минимального элемента из остающихся; если таковой нашёлся, производится обмен: найденный минимальный элемент попадает на своё место, а на его бывшее место попадает элемент из текущей позиции.

### 5.3. Quicksort

Идея алгоритма: выбираем некоторый опорный элемент, по величине которого разделяем массив на два подмассива, в одном – все элементы, не превышающие опорный, в другом – все, что не меньше опорного. Затем для каждого из подмассивов (если в них более двух элементов) рекурсивно запускаем ту же процедуру. В конце получаем отсортированный первоначальный массив.

Реализация разделения может быть такой. Вводим два индекса-«указателя» позиции в массиве:  $i$  и  $j$ . В начале алгоритма они соответствуют левому и правому крайним элементам массива. Будем двигать  $i$  слева вправо, если порядок следования текущего слева ( $a[i]$ ) и опорного ( $p$ ) – «правильный» (т.е.,  $a[i] < p$ ). Будем двигать  $j$  справа влево, если порядок следования опорного ( $p$ ) и текущего справа ( $a[j]$ ) – «правильный» (т.е.,  $p < a[j]$ ). Остановка может произойти либо на опорном элементе (всё, что находится с этой стороны стоит «правильно» по отношению к опорному), либо на «неправильно» стоящем элементе. Если позиции индексов различаются – меняем величины местами и продолжаем движение – вплоть до «смыкания» индексов. Кстати, индексы тогда будут равны как раз (новой) позиции опорного элемента...

После всех перестановок при «сближении» индексов массив разделён на два подмассива, в одном сосредоточены элементы, не превышающие опорный, во втором – не меньшие, чем он. В хорошем случае эти два подмассива не вырожденные, в «плохом» – один из них пуст или содержит «мало» элементов.

К этим подмассивам применяем такую же процедуру разделения, выбирая для каждого подмассива свой опорный элемент. В результате элементы этих подмассивов будут перегруппированы относительно своих опорных элементов и почти (в «хорошем» случае) вдвое меньше по размеру.

Идеально было бы иметь в качестве опорного элемента в каждом случае некое среднее значение по этому массиву, но сделать это невозможно без некоторой сортировки, а потому опорным часто выбирается просто средний элемент.

```

void QuickSort(int a[], int l, int r)
{
    int i = l, j = r, p = a[(l + r)/2];

    while (i <= j) {
        while (Less(a[i],p))
            i++;
        while (Less(p,a[j]))
            j--;
        if (i <= j) {
            Swap(a[i],a[j]);
            i++; j--;
        }
    }
    if (l < j)
        QuickSort(a,l,j);
    if (i < r)
        QuickSort(a,i,r);
}

```

Встречные индексы  $i$  и  $j$  «пропускают» правильно расположенные элементы и «останавливаются» на расположенных неправильно либо на опорных значениях. После «остановки» обоих индексов соответствующие им значения обмениваются местами, а каждый из индексов делает следующий шаг. Так продолжается до тех пор, пока индексы не «поменяются местами» (бывший больший станет меньше бывшего меньшего). После чего эта процедура повторяется для двух фрагментов изначального массива, причём только если размер фрагментов – более одного элемента.

## 6. Метод Монте-Карло

Это – общее название группы методов, основанных на получении большого числа реализаций случайного процесса, который подобран так, чтобы его вероятностные характеристики совпадали с искомыми величинами решаемой задачи. При этом вычисление точного значения какой-то величины заменяется *оценкой* соответствующих вероятностных характеристик. Для пояснения этих слов (возможно, не совсем понятных сейчас) рассмотрим уже знакомый нам пример с вычислением определённого интеграла – но немного с другой стороны.

Поскольку большая часть сведений, излагаемых здесь, только станет известна при будущем изучении теории вероятностей и математической статистики, мы будем вынуждены ограничиваться фактами и утверждениями, которые либо кажутся интуитивно понятными, либо в каком-то виде уже известны. Поэтому многие термины, определяемые только в рамках этих будущих дисциплин, используются здесь без обоснования и часто даже без определения.

Вычисление определённого интеграла функции  $f(x)$  на отрезке  $[a, b]$ , проделанное ранее, можно попытаться реализовать и по-другому. Ведь площадь под графиком функции на отрезке является равновеликой площади прямоугольника, одна сторона которого – сам отрезок, а вторая – некоторое «среднее» значение функции  $f(x)$  на этом интервале. Мы можем определить это среднее значение, зная величину интеграла, но если бы мы его узнали каким-то другим образом, то величину интеграла можно было бы определить по этому среднему, просто умножая его на длину отрезка!

Как же найти это «среднее» значение? Предположим, что мы случайным образом получаем некое количество  $N$  точек этого отрезка  $u_i$ ,  $i = 1, \dots, N$ . Эти точки соответствуют значениям некоторой случайной величины  $\mathbf{u}$  (здесь мы делаем различие между обозначением случайной величины ( $\mathbf{u}$ ) и её отдельными значениями ( $u_i$ )). Если случайные точки «разбросаны» по отрезку более или менее равномерно (о подобных случайных величинах принято говорить, что они имеют *равномерное распределение* на этом отрезке), то среди случайных значений  $f(u_i)$  функции  $f(x)$  все её возможные значения будут представлены достаточно равноправно. Новая случайная величина  $f(\mathbf{u})$  (как, впрочем, и любые другие случайные величины) имеет такую характеристику, как *математическое ожидание*,  $\mathbf{M}f(\mathbf{u})$ , причём известно, что оно (мат. ожидание) может быть определено на основе *плотности распределения*  $\rho(x)$  случайной величины:

$$\mathbf{M}f(\mathbf{u}) = \int_a^b f(x)\rho(x) dx$$

Как известно, для равномерно распределённой на отрезке  $[a, b]$  случайной величины плотность её распределения равна:

$$\rho(x) = \begin{cases} \frac{1}{b-a}, & x \in [a, b] \\ 0, & x \notin [a, b] \end{cases}$$

поэтому

$$\mathbf{M}f(\mathbf{u}) = \frac{1}{b-a} \int_a^b f(x) dx.$$

Видно, что математическое ожидание случайной величины  $f(\mathbf{u})$ , где  $\mathbf{u}$  – равномерно распределённая случайная величина, и есть то самое неизвестное нам «среднее» значение функции  $f(x)$  на отрезке  $[a, b]$ . Но для случайных величин существует возможность эмпирического определения (оценки) их неизвестного мат. ожидания – с помощью значения «выборочного среднего»:

$$\mathbf{M}f(\mathbf{u}) \approx \frac{1}{N} \sum_{i=1}^N f(u_i),$$



откуда

$$\int_a^b f(x) dx \approx \frac{b-a}{N} \sum_{i=1}^N f(u_i).$$

Возвращаясь к пояснению слов, сказанных ранее по поводу сути метода Монте-Карло, можно отметить, что в приведённом примере мы воспользовались взаимосвязью значения нужного нам определённого интеграла с математическим ожиданием некоторой случайной величины, а само мат. ожидание попытались оценить по полученной *выборке* её значений.

С точки зрения формулы мы не получили вроде бы ничего нового: формула неотличима от формулы центральных прямоугольников, но на самом деле точки, в которых вычисляются значения функции, расположены не в равномерной сетке, а случайным образом, так что значения их координат имеют *равномерное распределение* на отрезке  $[a, b]$ .

Разумеется, возникновение подобных численных методов решения математических задач при помощи моделирования случайных величин обусловлено развитием вычислительной техники.

Ошибка оценки пропорциональна обратной величине корня квадратного из числа испытаний, т.е., для того чтобы уменьшить ошибку в 10 раз, надо увеличить число испытаний (т.е., объём работы) в 100 раз. Понятно, что таким способом добиться высокой точности невозможно, поэтому метод Монте-Карло особенно эффективен при решении задач, где результат нужен с небольшой точностью ( $\sim 5 - 10\%$ ).

Правда теперь, поскольку мы должны при вычислениях использовать случайную величину с равномерным распределением, встает проблема получения значений такой величины.

## 6.1. Определение площадей (объёмов) фигур

Предположим, что надо определить площадь некоторой ограниченной фигуры. Если случайным образом «набросать» точек в квадрат, содержащий фигуру, то количество точек, попавших в неё, будет примерно пропорционально её площади. Т.е., отыскание отношения площадей (фигуры и квадрата) сводится к вычислению отношения двух чисел: количества точек, попавших в фигуру, и общего количества точек. Точно так же можно попытаться определить значение числа  $\pi$  (см. далее).

Аналогично можно оценивать объёмы трёхмерных фигур и фигур большей размерности. В этом и большинстве последующих примеров характеристикой, связанной с искомой величиной, является вероятность попадания случайной точки в фигуру. Эта вероятность оценивается отношением числа благоприятных исходов бросания к их общему числу. Лишь в примере с иглой Бюффона (см. далее) оценивается вероятность немного другого события.

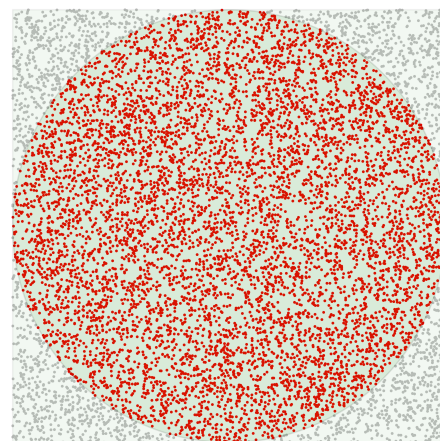
Достаточно очевидно, что такие «вычислительные» методы будут работать тогда, когда точки будут «равномерно разбросанными» по прямоугольнику (по параллелепипеду), т.е., случайная величина, которой мы будем пользоваться, будет равномерно распределённой в двух, трёх или более измерениях. Однако, не все генераторы псевдослучайных чисел могут «подходить» для этой цели.

## 6.2. Вычисление числа $\pi$

Попробуем применить изложенные идеи для вычисления числа  $\pi$ , поскольку оно входит в значение площади известной фигуры: круга.

Будем бросать случайные точки на квадрат, в который вписан круг. Площадь квадрата – это диаметр (два радиуса) круга в квадрате ( $4R^2$ ). Площадь круга – произведение  $\pi$  на квадрат радиуса круга ( $\pi R^2$ ). Отношение числа точек, попавших в круг, к общему числу точек в квадрате, оценивает величину  $\frac{\pi}{4}$ .

Существует ещё один известный способ вычисления числа  $\pi$  – так называемый **метод иглы Бюффона**. Суть его заключается в следующем. На разную равноудалёнными параллельными прямыми плоскость произвольно бросается игла, длина которой равна расстоянию между соседними прямыми, так что при каждом бросании игла либо не пересекает прямые, либо пересекает ровно одну. Можно доказать, что отношение числа пересечений иглы с какой-нибудь линией к общему числу бросков стремится к  $\frac{2}{\pi}$  при увеличении числа бросков до бесконечности.



$$\pi \approx 4 \cdot \frac{7823}{7823 + 2177} = \frac{31292}{10000} = 3.1292$$

## 6.3. Численное интегрирование методом Монте-Карло

Ещё один вариант численного интегрирования почти не отличается от способа определения площадей фигур на плоскости и объёмов в пространстве (см. изложенное выше), поэтому также применим для вычисления многократных интегралов. Выбирается область, содержащая полностью значения интегрируемой функции, причём для простоты её лучше выбрать параллелепипедом нужной размерности. В этот параллелепипед случайным образом бросается достаточное количество точек, а затем оценка вычисляемого интеграла получается как произведение объёма параллелепипеда на отношение числа точек, попавших в область интегрирования, к общему числу точек. Однократные интегралы на практике методом Монте-Карло не вычисляют, для этого есть более точные методы. А вот при переходе к многократным интегралам ситуация меняется: аналоги формул однократного интегрирования становятся значительно более сложными, а метод Монте-Карло – что весьма существенно – остаётся практически без изменений.

## 6.4. Получение случайных чисел

Способов получения случайных величин (или подобных им), в сущности, не так много. Можно попытаться использовать какие-то физические эффекты (скажем, шумы в полупроводниках) или предметы (монета, игральная кость, вращающийся барабан с цифрами и неподвижной стрелкой), где получаемое значение зависит от такого большого числа факторов, что предвидеть его просто невозможно. Однако такие способы, во-первых, бывает сложно сопрягать с устройством, которое может применяться для вычислений, а, во-вторых, не всегда они позволяют получить необходимую скорость формирования случайных значений.

Можно построить специальные таблицы случайных величин или использовать уже гото-

вые. Но тогда для работы вычислительного алгоритма понадобится значительная память, где будет храниться такая таблица, причём и всей таблицы может оказаться недостаточно, если случайных величин для решения задачи нужно очень много.

Можно также попробовать смоделировать случайные величины с помощью каких-то алгоритмов. Правда, тогда это будут уже не настоящие случайные числа, а так называемые псевдослучайные, поскольку в подобных алгоритмах последующее число генерируется на основе предыдущего (или предыдущих). Наиболее распространенными в настоящее время являются **линейный конгруэнтный метод** и **метод регистра сдвига** с линейной обратной связью. В последнее время появились и часто используются **метод Фибоначчи** с запаздываниями и так называемый **«вихрь Мерсенна»** (Mersenne Twister).

## 6.5. Линейный конгруэнтный метод

Один из простых способов получения псевдослучайных значений, в нём используются только целочисленные операции. Необходимо выбрать три натуральных числа-константы:  $M$  (модуль),  $a < M$  (множитель),  $c < M$  (приращение) и первоначальное значение псевдослучайного числа  $x_0$ , тогда последующие значения псевдослучайных чисел получаются с помощью рекуррентной формулы:

$$x_{i+1} = (ax_i + c) \bmod M$$

Эта последовательность, конечно, периодична, поскольку может содержать не более чем  $M$  различных значений, поэтому выбираются большие значения  $M$ . Но если множитель и модуль подобраны «хорошо», то результирующая последовательность чисел будет статистически очень похожа на случайную последовательность элементов множества  $\{0, 1, 2, \dots, M - 1\}$ .

«Правильный» выбор констант важен и для периода последовательности, поскольку даже взаимная простота значений не всегда может приводить к максимально возможному периоду. Тривиальные примеры:  $a = 5, c = 3, M = 11$ ;  $a = 2, c = 4, M = 7$  (проверьте самостоятельно!):

$a$	$x_i$	$c$	$M$	$x_{i+1}$
<b>5</b>		<b>3</b>	<b>11</b>	
5	· <b>1</b>	+ 3	= 8	→ 8
5	· 8	+ 3	= 43	→ 10
5	· 10	+ 3	= 53	→ 9
5	· 9	+ 3	= 48	→ 4
5	· 4	+ 3	= 23	→ <b>1</b>

Видно, что при «плохом» выборе констант период последовательности может быть заметно короче максимального. Кроме того, такой метод малоприменим для получения случайных векторов, разве что совсем малой размерности (не выше пяти).

В 1960-х годах был распространён линейный конгруэнтный генератор псевдослучайных чисел под названием RANDU. Он имел параметры  $a = 65\,539$  ( $65\,539 = 2^{16} + 3$ ),  $c = 0$  и  $M = 2^{31}$ ; значение  $x_0$  выбиралось нечётным (скажем, равным 1). Основания для выбора таких параметров были: умножение – много проще произвольного, остаток – тоже получается тривиально. Однако, оказалось, что использование его для получения случайных векторов размерности более 2 проблематично: в трёхмерном пространстве все его значения располагаются всего лишь в 15 параллельных плоскостях!

## 6.6. Метод регистра сдвига (с линейной обратной связью)

Основа алгоритма – логическая операция «Исключающее ИЛИ», производимая над некоторыми разрядами двоичного числа, представляющего текущее псевдослучайное значение. Результат её становится частью нового значения, а старое сдвигается на один разряд. Можно использовать для реализации метода программу или логическую схему, называемую регистром сдвига, дополненную схемой-вычислителем значения указанной логической операции над разрядами. Максимальным значением периода при регистре из  $N$  двоичных разрядов будет  $2^N - 1$ . Это обусловлено тем, что среди всех формируемых значений отсутствует значение, представленное нулями во всех разрядах. Однако для получения максимально возможного периода надо знать, какие именно разряды регистра и в каком количестве необходимо выбирать для формирования обратной связи. Два простых примера с трёхразрядным регистром:



## 6.7. Метод Фибоначчи с запаздываниями

Если скорость выполнения арифметических операций с вещественными числами сравнима со скоростью целочисленных операций, то для получения псевдослучайных вещественных значений можно воспользоваться так называемым методом Фибоначчи с запаздываниями. Пусть  $x_i$  далее обозначает вещественное число из диапазона  $[0, 1)$ , генерируемое на  $i$ -ом шаге, а  $a$  и  $b$  – целые положительные числа (параметры метода). Одна из возможных реализаций предписывает такую процедуру получения очередного значения по значениям, сгенерированным ранее:

$$x_i = \begin{cases} x_{i-a} - x_{i-b}, & x_{i-a} \geq x_{i-b} \\ x_{i-a} - x_{i-b} + 1, & x_{i-a} < x_{i-b} \end{cases}$$

Предполагается, что предыдущие  $\max\{a, b\}$  значений сохраняются, т.е., для работы необходим некоторый объём памяти, определяемый параметрами метода. Первоначальные значения для работы можно получить с помощью, например, линейного конгруэнтного генератора. Существуют некоторые (хорошо зарекомендовавшие себя) значения параметров  $a$  и  $b$ , которые можно использовать:  $(a, b) = (17, 5)$ ,  $(55, 24)$ ,  $(97, 33)$ . Чем больше значение константы  $a$ , тем выше размерность пространства, в котором сохраняется равномерность «расположения» случайных векторов, получаемых из генерируемых чисел.

## 6.8. «Вихрь Мерсенна» (Mersenne Twister)

Из современных генераторов псевдослучайных чисел широкое распространение также получил так называемый «вихрь Мерсенна» (Mersenne Twister, сокращённо – *MT*), предложенный в 1997 году Мацумото и Нисимурой (Matsumoto, Nishimura). Его достоинствами являются колоссальный период ( $2^{19937} - 1$ ), равномерное распределение в 623 измерениях (у линейного конгруэнтного метода – максимум в 5 измерениях), быстрая генерация



## 7. Решение дифференциальных уравнений

Поскольку многие процессы в физике описываются дифференциальными уравнениями, необходимы численные методы решения таких уравнений. Мы рассматриваем здесь методы решения так называемой задачи Коши для обыкновенного дифференциального уравнения первого порядка (дифференциальное уравнение + начальное условие).

### 7.1. Обыкновенные дифференциальные уравнения (ОДУ)

*Обыкновенное дифференциальное уравнение* (ОДУ)  $n$ -го порядка для функции  $y(x)$  аргумента  $x$  – это соотношение вида

$$F(x, y, y', y'', \dots, y^{(n)}) = 0,$$

где  $F$  – заданная функция своих аргументов. Слово «дифференциальное» подчёркивает тот факт, что в соотношение входят и производные функции  $y$ , а не только сами  $x$  и  $y$ , слово «обыкновенное» говорит о том, что искомая функция  $y$  зависит только от одного аргумента.

Решить ОДУ – значит найти *все* функции  $y(x)$ , превращающие уравнение в тождество. Семейство таких функций называется *общим решением* ОДУ, оно образуется с помощью некоторого числа произвольных констант (их число совпадает с порядком уравнения). Иногда его называют также *общим интегралом* дифференциального уравнения.

Конкретная прикладная задача может приводить к дифференциальному уравнению любого порядка. Но обыкновенное дифференциальное уравнение, скажем,  $p$ -го порядка (здесь и далее предполагается, что оно разрешимо относительно старшей производной, и такое разрешение произведено):

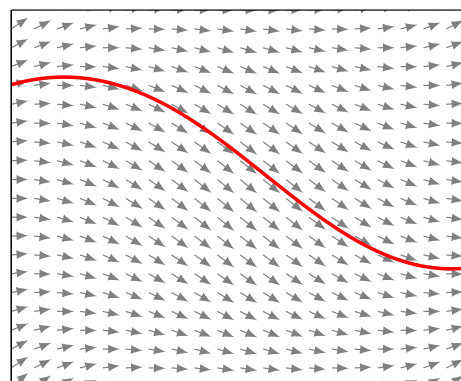
$$u^{(p)}(x) = f(x, u, u', u'', \dots, u^{(p-1)})$$

можно свести к эквивалентной системе  $p$  уравнений первого порядка с помощью замен  $u^{(k)}(x) \equiv u_k(x)$ . Аналогично, произвольную систему дифференциальных уравнений любого порядка можно заменить эквивалентной системой уравнений первого порядка, но с *большим* количеством уравнений.

Различают три основных типа задач для обыкновенных дифференциальных уравнений: *задачи Коши, краевые задачи, задачи на собственные значения*. Далее мы рассматриваем только задачу Коши, т.е., задачу с начальными условиями (в случае уравнения первого порядка – с начальным условием).

Общее ОДУ первого порядка имеет вид  $F(x, y, y') = 0$ ; если его удастся разрешить относительно производной, то оно может быть записано так:  $y' = f(x, y)$ .

Геометрически общее решение ОДУ первого порядка представляет собой семейство кривых, не имеющих общих точек и отличающихся друг от друга некоторой константой  $C$ . Эти кривые называются *интегральными кривыми* для данного уравнения. Их очевидное свойство: в каждой точке тангенс угла наклона касательной к кривой в этой точке равен значению правой части уравнения в этой точке (т.е., решаемое уравнение задаётся в плоскости  $XU$  полем



направлений касательных к интегральным кривым). Одна интегральная кривая определяет частное решение.

## 7.2. Задача Коши (задача с начальными условиями)

Для ОДУ первого порядка формулируется так:

$$\begin{cases} y' = f(x, y), & x \in [x_0, b] \\ y(x_0) = y_0 \end{cases}$$

Известно, что если  $f(x, y)$  непрерывна вместе со своей частной производной по  $y$  в некоторой области, то задача Коши имеет в этой области единственное решение.

С вычислительной точки зрения можно сказать, что имеется неизвестная функция  $y = y(x)$ , про которую известно, чему равна её первая производная  $y' = f(x, y)$  (вообще говоря, она зависит как от самой функции, так и от её аргумента), и в начальной точке  $x_0$  значение функции тоже известно:  $y(x_0) = y_0$ . Требуется найти (приблизённо), чему равна функция в других точках  $x_i$ ,  $i = 1, \dots, n$  заданного отрезка (т.е., на заданной сетке):  $x_0 < x_1 < \dots < x_n \leq b$ . Довольно часто сетка выбирается равномерной с шагом  $h$ , т.е.  $x_{i+1} - x_i = h$ ,  $i = 0, \dots, n - 1$ . Далее рассматриваются некоторые методы приближённого решения (иногда говорят – интегрирования) ОДУ первого порядка: **метод Эйлера** (метод ломаных), **метод средней точки** (модифицированный метод Эйлера), **метод «предиктор-корректор»** (метод Эйлера с пересчётом), **классический метод Рунге-Кутты**.

## 7.3. Ошибки приближённых методов

*Локальной ошибкой* называется ошибка на одном шаге метода. *Глобальная ошибка* – суммарная ошибка за все шаги. Если глобальная ошибка при уменьшении шага уменьшается примерно пропорционально уменьшению шага, то говорят, что это – метод первого порядка. Методы второго порядка имеют глобальную ошибку, уменьшающуюся пропорционально второй степени уменьшения шага и т.д.

Про рассматриваемые методы приближённого нахождения решений дифференциальных уравнений можно сказать следующее: метод Эйлера – первого порядка, метод средней точки и метод «предиктор-корректор» – методы второго порядка, классический метод Рунге-Кутты – метод четвёртого порядка.

## 7.4. Устойчивость приближённого решения

Численный метод интегрирования ОДУ называется *устойчивым* (при заданном шаге  $h$  и для определённого  $\lambda$ ), если результат численного интегрирования уравнения  $y'(t) = \lambda y(t)$  остаётся ограниченным при  $t \rightarrow \infty$ .

Устойчивость зависит от шага  $h$  и значения  $\lambda$ . На практике устойчивость зависит от произведения  $h\lambda$ . Например, для метода Эйлера (как выяснится далее)  $y_{n+1} = (1 + h\lambda)y_n$ , а потому, соответственно,  $y_{n+1} = (1 + h\lambda)^n y_0$ , значит, метод будет устойчив при  $|1 + h\lambda| \leq 1$ , т.е., когда  $-2 \leq h\lambda \leq 0$ .

## 7.5. Метод Эйлера (метод ломаных)

Подстановка начального условия в исходное ОДУ даёт значение производной функции  $y = y(x)$  в начальной точке

$$y'_0 = f(x_0, y_0)$$

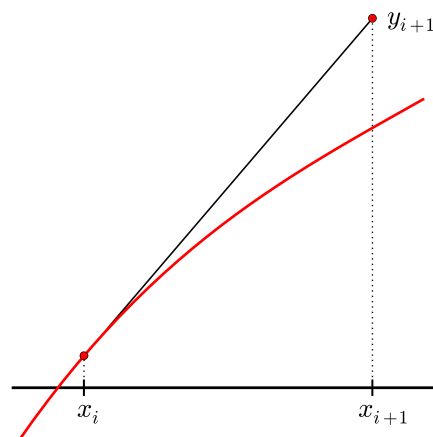
Значение в следующей точке сетки можно оценить по значению этой производной и шагу сетки

$$y_1 \simeq y_0 + (x_1 - x_0)f(x_0, y_0)$$

Поэтому общая формула метода Эйлера (на  $i$ -ом шаге) для равномерной сетки, очевидно, будет такова:

$$y_{i+1} \simeq y_i + hf(x_i, y_i), \quad i = 0, \dots, n-1.$$

Метод Эйлера является одношаговым, поскольку используется значение функции  $f$  только из предыдущего шага. Локальная ошибка метода  $\sim h^2$ , глобальная (суммарная) погрешность  $\sim h$ , поэтому говорят, что этот метод имеет первый порядок точности.



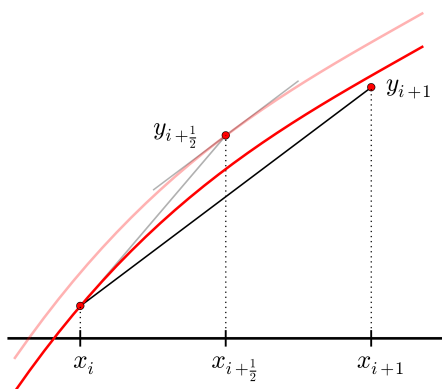
Если бы правая часть исходного ОДУ не зависела от  $y$ , то точное значение искомой функции в точке  $x_{i+1} = x_i + h$  определялось бы интегралом

$$y_{i+1} = y_i + \int_{x_i}^{x_i+h} f(x) dx$$

а это значит, что общая формула метода Эйлера – это аналог формулы численного интегрирования методом левых прямоугольников.

Этот метод из-за быстрого накопления глобальной ошибки на практике не используется, но удобен для теоретических оценок.

## 7.6. Метод средней точки (модифицированный метод Эйлера)



Оценивается значение в «половинной» точке  $x_{i+\frac{1}{2}}$  по методу Эйлера:

$$y_{i+\frac{1}{2}} \simeq y_i + \frac{h}{2}f(x_i, y_i)$$

Затем в этой половинной точке вычисляется угловой коэффициент касательной, примерно равный  $f(x_{i+\frac{1}{2}}, y_{i+\frac{1}{2}})$ , и в этом направлении совершается переход из точки  $x_i$  в точку  $x_{i+1}$ , чтобы получить новое приближённое значение искомой функции:

$$y_{i+1} \simeq y_i + hf(x_{i+\frac{1}{2}}, y_{i+\frac{1}{2}})$$

Этот метод является методом второго порядка точности.



## 7.7. Метод «предиктор-корректор» (метод Эйлера с пересчётом)

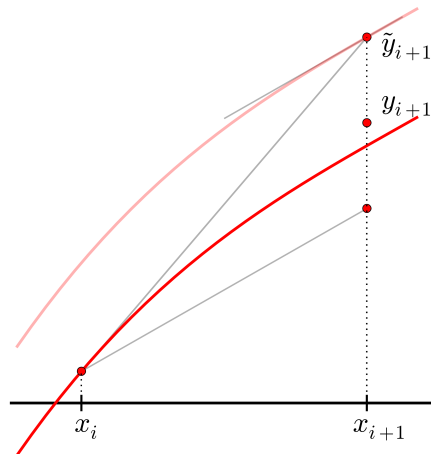
На первом этапе (прогноз) предсказывается  $\tilde{y}_{i+1}$  по методу Эйлера:

$$\tilde{y}_{i+1} \simeq y_i + hf(x_i, y_i)$$

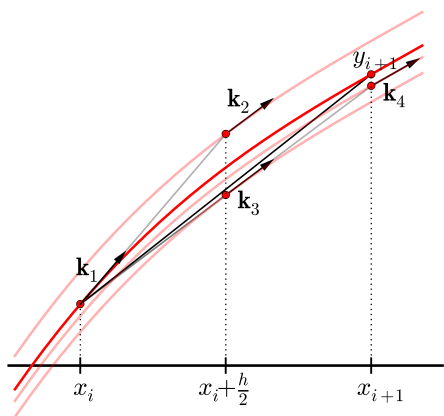
На втором этапе (коррекция) значение корректируется путём усреднения угловых коэффициентов в точках  $(x_i, y_i)$  и  $(x_{i+1}, \tilde{y}_{i+1})$ :

$$y_{i+1} \simeq y_i + h \frac{f(x_i, y_i) + f(x_{i+1}, \tilde{y}_{i+1})}{2}$$

Т.е., сначала мы «грубо» вычисляем значение функции  $\tilde{y}_{i+1}$  и наклон интегральной кривой  $f(x_{i+1}, \tilde{y}_{i+1})$  в новой точке. Затем находим средний наклон на этом шаге и по нему корректируем значение  $y_{i+1}$  в новой точке. За счёт коррекции точность метода повышается по сравнению с методом Эйлера, так что этот метод тоже является методом второго порядка точности.



## 7.8. Классический метод Рунге-Кутты



Описывается системой соотношений для  $i = 0, \dots, n-1$ :

$$\begin{aligned} y_{i+1} &= y_i + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4), \\ \mathbf{k}_1 &= f(x_i, y_i), \\ \mathbf{k}_2 &= f(x_i + \frac{h}{2}, y_i + \frac{h}{2}\mathbf{k}_1), \\ \mathbf{k}_3 &= f(x_i + \frac{h}{2}, y_i + \frac{h}{2}\mathbf{k}_2), \\ \mathbf{k}_4 &= f(x_i + h, y_i + h\mathbf{k}_3). \end{aligned}$$

Последовательно вычисляются приближающие угловые коэффициенты:  $\mathbf{k}_1$  – в исходной точке,  $\mathbf{k}_2$  – на половинном шаге (как в методе средней точки),  $\mathbf{k}_3$  – тоже на половинном шаге, но по уточнённому значению углового коэффициента  $\mathbf{k}_2$  вместо  $\mathbf{k}_1$ ,  $\mathbf{k}_4$  – на целом шаге по предыдущему значению  $\mathbf{k}_3$ . Стоит отметить, что получаемые здесь на каждом этапе  $\mathbf{k}_1, \mathbf{k}_2, \mathbf{k}_3, \mathbf{k}_4$  – угловые коэффициенты четырёх *разных* интегральных кривых в трёх точках:  $x_i, x_i + \frac{h}{2}, x_{i+1} = x_i + h$ . Для получения нового значения искомой функции на полном шаге используется взвешенное среднее этих угловых коэффициентов.

Классический метод Рунге-Кутты имеет четвёртый порядок точности. Он является явным и допускает расчёт с переменным шагом.

В случае, когда правая часть исходного ОДУ не зависит от  $y$ , легко заметить, что точное значение искомой функции в точке  $x_{i+1} = x_i + h$ , определяемое интегралом

$$y_{i+1} = y_i + \int_{x_i}^{x_i+h} f(x) dx$$

аппроксимируется в соответствии с формулой Симпсона, а это значит, что формула классического метода Рунге-Кутты – это аналог формулы численного интегрирования Симпсона.

На случай систем дифференциальных уравнений различные вариации метода Рунге-Кутты переносятся при помощи формальной замены скалярных величин на нужное число векторных компонент.

## 7.9. Алгоритм Верле

Алгоритм численного интегрирования Верле (переоткрыт L.Verlet, 1967) часто используется в молекулярной динамике для вычисления траекторий частиц. Он более устойчив, чем простой метод Эйлера. В *основной форме алгоритма Верле* вычисляется следующее местоположение частицы по текущему и прошлому положениям, причём без использования скорости. Для получения формулы запишем разложение в ряд Тейлора вектора местоположения  $\vec{r}(t)$  частицы в моменты времени  $t + \Delta t$  и  $t - \Delta t$ .

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \vec{a}(t)(\Delta t)^2/2 + \dots$$

$$\vec{r}(t - \Delta t) = \vec{r}(t) - \vec{v}(t)\Delta t + \vec{a}(t)(\Delta t)^2/2 - \dots$$

Здесь  $\vec{r}$  – положение частицы,  $\vec{v}$  – её скорость,  $\vec{a}$  – ускорение. Сложив эти два равенства и выразив  $\vec{r}(t + \Delta t)$ , получим:

$$\vec{r}(t + \Delta t) \approx 2\vec{r}(t) - \vec{r}(t - \Delta t) + \vec{a}(t)(\Delta t)^2$$

Ускорение частицы известно, поскольку пропорционально силе, действующей на частицу, и обратно пропорционально её массе. Таким образом, действительно, положение вычисляется без знания скорости. Однако, *алгоритм не является*, как говорят, *самостартующим*: с его помощью нельзя в самом начале найти текущее положение, зная только прошлое, здесь нужно воспользоваться другим способом.

Если скорость всё-таки нужна (например, для определения кинетической энергии частицы), можно воспользоваться *скоростной формой алгоритма Верле*:

1. Вычисляется скорость на «половинном» шаге:  $\vec{v}(t + \frac{1}{2}\Delta t) = \vec{v}(t) + \frac{1}{2}\vec{a}(t)\Delta t$
2. Вычисляется положение на следующем шаге:  $\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t + \frac{1}{2}\Delta t)\Delta t$
3. Вычисляется  $\vec{a}(t + \Delta t)$  в положении  $\vec{r}(t + \Delta t)$  – по действующей силе
4. Вычисляется скорость на следующем шаге:  $\vec{v}(t + \Delta t) = \vec{v}(t) + \frac{1}{2}(\vec{a}(t) + \vec{a}(t + \Delta t))\Delta t$

Или, приводя всё к двум формулам:

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t + \frac{1}{2}\vec{a}(t)(\Delta t)^2$$

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \frac{1}{2}(\vec{a}(t) + \vec{a}(t + \Delta t))\Delta t$$

## 8. Интерполяция

Интерполяцией называют такую разновидность аппроксимации, при которой кривая построенной функции проходит точно через имеющиеся точки данных. Часто нас интересует аппроксимация сложной функции другой, более простой в вычислительном смысле. Например, полиномы характерны тем, что легко вычисляются и дифференцируются.

Рассмотрим в качестве примера важный частный случай – **линейную интерполяцию**: необходимо для функции  $f(x)$ , заданной в точках  $x_1$  и  $x_2$ , построить приближение с помощью линейной функции  $Ax + B$ .

Для любой точки  $(x, y)$  прямой, проходящей через точки  $(x_1, f(x_1))$  и  $(x_2, f(x_2))$  справедливо соотношение:

$$\frac{y - f(x_1)}{x - x_1} = \frac{f(x_2) - f(x_1)}{x_2 - x_1},$$

откуда можно выразить зависимость  $y$  от  $x$ :

$$\begin{aligned} y &= f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1} \cdot (x - x_1) \\ &= f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1} \cdot x - (f(x_2) - f(x_1)) \cdot \frac{x_1}{x_2 - x_1} \\ &= \frac{f(x_2) - f(x_1)}{x_2 - x_1} \cdot x + \frac{x_2 f(x_1) - x_1 f(x_2)}{x_2 - x_1}. \end{aligned}$$

Таким образом,

$$f(x) \approx Ax + B,$$

где  $A = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$  и  $B = \frac{x_2 f(x_1) - x_1 f(x_2)}{x_2 - x_1}$ .

Можно также записать эту формулу и по-другому, перегруппировав слагаемые так, чтобы образовалась «взвешенная» сумма значений функции в обеих точках с соответствующими множителями:

$$\begin{aligned} f(x) &\approx f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1} \cdot (x - x_1) \\ &= f(x_1) - f(x_1) \cdot \frac{x - x_1}{x_2 - x_1} + f(x_2) \cdot \frac{x - x_1}{x_2 - x_1} \\ &= f(x_1) \cdot \left[ 1 - \frac{x - x_1}{x_2 - x_1} \right] + f(x_2) \cdot \frac{x - x_1}{x_2 - x_1} \\ &= f(x_1) \cdot \frac{x_2 - x}{x_2 - x_1} + f(x_2) \cdot \frac{x - x_1}{x_2 - x_1}. \end{aligned}$$

Обратите внимание на поведение множителей при значениях функции  $f(x_1), f(x_2)$ : они равны единице для той точки, в которой вычисляется функция, и нулю – для другой точки.

### 8.1. Общая постановка задачи интерполяции

Пусть даны точки  $x_i$ ,  $i = 0, 1, \dots, n$  (их называют узлами интерполяции), в которых известны значения функции  $f(x)$ :  $y_i = f(x_i)$ . Надо найти *интерполирующую функцию*  $F(x)$ , такую, что  $F(x_i) = y_i$  для  $i = 0, 1, \dots, n$ , а сама  $F$  – из заданного класса функций (чаще всего применяется интерполяция полиномами).

### 8.2. Интерполяционные многочлены Лежандра

Многочлены минимальной степени, приближающие данные значения на заданном наборе точек.

Пусть даны пары чисел  $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ , где все  $x_i$  – различны. Тогда существует единственный многочлен  $L(x)$  степени не более  $n$ , для которого  $L(x_i) = y_i, i = 0, 1, \dots, n$ .

Попытаемся построить базисные функции  $l_0(x), l_1(x), \dots, l_n(x)$ , такие, что

$$l_i(x_j) = \begin{cases} 1, & j = i \\ 0, & j \neq i \end{cases}$$

Т.е.,  $i$ -ая базисная функция равна нулю во всех узлах интерполяции, кроме  $i$ -ого, где она равна единице.

Имея такие базисные функции, мы можем сконструировать интерполирующую функцию так:

$$L(x) = \sum_{i=0}^n y_i l_i(x)$$

Рассмотрим для иллюстрации простой пример (см. рис.7): известны значения функции в точках 1, 2, 4; они равны 1,  $1/3$  и  $1/5$  соответственно. Требуется найти многочлен, проходящий через эти точки.

Таким образом, известна такая таблица значений функции:

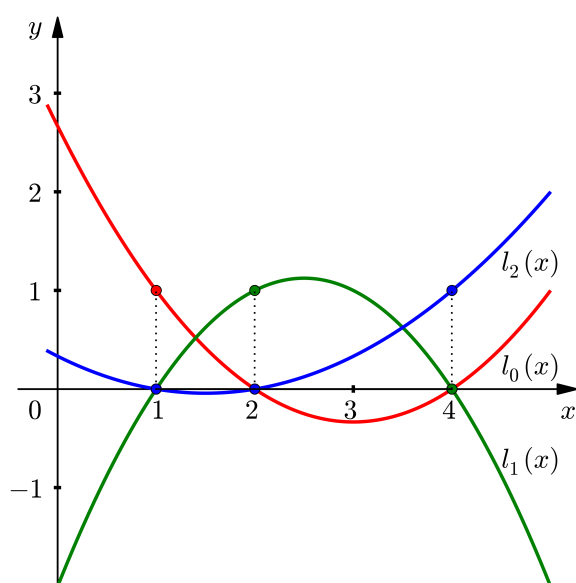
$x_j$	1	2	4
$y_j$	1	$\frac{1}{3}$	$\frac{1}{5}$

Из общих соображений понятно, что будет достаточно многочлена второго порядка. Попытаемся сконструировать его на основе трёх вспомогательных функций  $l_0(x), l_1(x), l_2(x)$ , удовлетворяющих условиям:  $l_0(x)$  равна нулю в точках  $x_1 = 2, x_2 = 4$  и единице – в точке  $x_0$ ,  $l_1(x)$  равна нулю в точках  $x_0 = 1, x_2 = 4$  и единице – в точке  $x_1$ ,  $l_2(x)$  равна нулю в точках  $x_0 = 1, x_1 = 2$  и единице – в точке  $x_2$ .

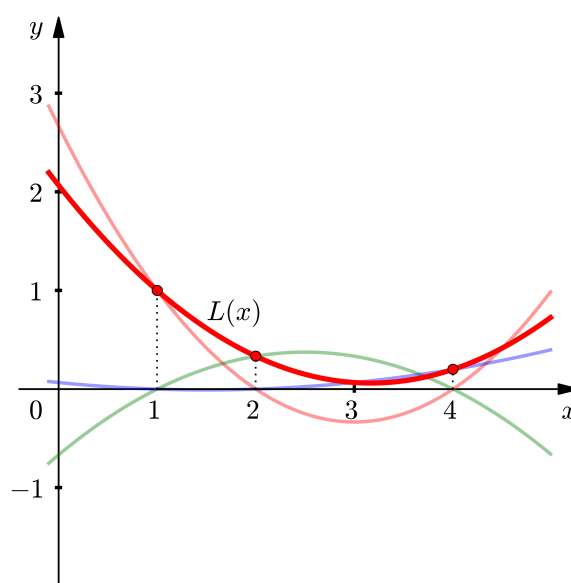
Тогда искомым многочлен можно выразить как сумму

$$L(x) = l_0(x)y_0 + l_1(x)y_1 + l_2(x)y_2$$

и он будет, очевидно, принимать в заданных точках нужные значения.



(a) Базисные функции  $l_0(x), l_1(x), l_2(x)$  для заданных точек 1, 2, 4



(b) Интерполирующий многочлен  $L(x)$  и его составляющие

Рис. 7: Пример интерполяции многочленом, проходящим через три заданные точки

Теперь уже становится понятно, что в общем случае интерполирующая функция будет иметь такой вид:

$$L(x) = \sum_{i=0}^n y_i l_i(x) = \sum_{i=0}^n y_i \overbrace{\prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}}^{l_i(x)}$$

Она называется интерполяционным многочленом Лежандра

$$L(x) = \sum_{i=0}^n y_i \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - x_j}{x_i - x_j}$$

Таким образом, получив выше уравнение прямой, проходящей через два заданных значения в данных точках, мы фактически нашли интерполяционный многочлен Лежандра первого порядка.

## 9. Системы линейных уравнений

Система линейных алгебраических уравнений (СЛАУ) – это система вида

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2n}x_n = b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \dots + a_{3n}x_n = b_3 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \dots + a_{nn}x_n = b_n \end{cases}$$

Здесь количество неизвестных величин совпадает с количеством уравнений и это самый интересный вариант, потому что в этом случае система может иметь единственное решение, тогда как в других ситуациях решений может либо не быть (количество неизвестных меньше количества уравнений), либо может быть бесконечно много (количество неизвестных больше количества уравнений). Классическим методом решения системы таких уравнений является метод Гаусса.

### 9.1. Метод Гаусса

В этом методе для приведения системы к треугольному или ступенчатому виду (когда в каждом последующем уравнении исключается одна или несколько первых неизвестных переменных) используются элементарные преобразования системы, т.е., такие, при которых новая система остаётся равносильной первоначальной. Для этого достаточно, например, таких элементарных преобразований:

- перестановка любых двух строк;
- умножение любой строки на ненулевую константу;
- прибавление к любой строке другой строки.

Когда система будет приведена к треугольному виду (если это удастся сделать), можно будет последовательно (начиная с последних переменных) найти все остальные переменные (так называемый *обратный ход* алгоритма). Если в системе в процессе преобразования получилось невозможное уравнение, то система несовместна, у неё нет ни одного решения. Если одно или несколько уравнений обратились в тождества, это значит, что они являются излишними и их можно убрать из системы.

Все эти действия можно проделывать не над всей системой, а только над её коэффициентами (матрица коэффициентов называется **матрицей системы**), включая также и правые части уравнений, т.е., над так называемой **расширенной матрицей системы**, столбец правых частей в такой матрице далее отчёркнут вертикальной чертой для наглядности. Если какие-то переменные отсутствуют в некоторых уравнениях системы, в матрице системы на соответствующем месте будут находиться нули.

**Пример 1.** Рассмотрим систему:

$$\begin{cases} x_1 + x_2 + x_3 = 4 \\ 2x_1 + 3x_2 + 4x_3 = 13 \\ 3x_1 + 4x_2 + 5x_3 = 17 \end{cases}$$

Исключим неизвестную переменную  $x_1$  из всех уравнений системы, начиная со второго. Для этого из второго уравнения вычтем первое, умноженное на 2, а из третьего – первое, умноженное на 3.

$$\left( \begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 3 & 4 & 13 \\ 3 & 4 & 5 & 17 \end{array} \right) \sim \left( \begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 5 \\ 0 & 1 & 2 & 5 \end{array} \right) \sim \left( \begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 5 \\ 0 & 0 & 0 & 0 \end{array} \right)$$

Попытка далее исключить из третьего уравнения неизвестную переменную  $x_2$  приводит к тому, что в третьей строке расширенной матрицы системы остаются одни нули: третье уравнение превратилось в тождество. Это говорит о том, что это уравнение является излишним; его можно убрать из системы (или вычеркнуть строку из расширенной матрицы). Теперь уравнений стало меньше (два для трёх переменных), для третьей переменной  $x_3$  не получится проделать обратный ход метода Гаусса, поэтому во всех уравнениях её проще перенести (с изменением знака) в правую часть и обозначить какой-нибудь другой буквой (далее используется  $\gamma$ ); она может принимать произвольное значение. Для оставшихся переменных  $x_1$  и  $x_2$  проделываем обратный ход метода, выражая через  $\gamma$  сначала  $x_2$  ( $5 - 2\gamma$ ), а через её значение –  $x_1$  ( $4 - \gamma - (5 - 2\gamma) = -1 + \gamma$ ). Окончательный результат – система совместна и имеет бесконечно много решений:  $x_1 = -1 + \gamma$ ,  $x_2 = 5 - 2\gamma$ ,  $x_3 = \gamma$  ( $\gamma$  – любое).

**Пример 2.** Рассмотрим систему:

$$\begin{cases} x_1 + x_2 + x_3 = 4 \\ 2x_1 + 3x_2 + 4x_3 = 13 \\ 3x_1 + 4x_2 + 5x_3 = 18 \end{cases}$$

При попытке исключения из третьего уравнения второй неизвестной переменной  $x_2$  теперь получается невозможное равенство, что свидетельствует о противоречивости системы.

$$\left( \begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 3 & 4 & 13 \\ 3 & 4 & 5 & 18 \end{array} \right) \sim \left( \begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 5 \\ 0 & 1 & 2 & 6 \end{array} \right) \sim \left( \begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 5 \\ 0 & 0 & 0 & 1 \end{array} \right)$$

Система несовместна, решений нет.

**Пример 3.** Рассмотрим систему:

$$\begin{cases} x_1 + x_2 + x_3 = 4 \\ 2x_1 + 3x_2 + 4x_3 = 13 \\ 3x_1 + 4x_2 + 6x_3 = 18 \end{cases}$$

Прямой ход метода Гаусса успешно завершается: получен треугольный вид системы, а излишних уравнений и противоречивых равенств не возникало. В последнем уравнении присутствует только одна переменная  $x_3$ :

$$\left( \begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 2 & 3 & 4 & 13 \\ 3 & 4 & 6 & 18 \end{array} \right) \sim \left( \begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 5 \\ 0 & 1 & 3 & 6 \end{array} \right) \sim \left( \begin{array}{ccc|c} 1 & 1 & 1 & 4 \\ 0 & 1 & 2 & 5 \\ 0 & 0 & 1 & 1 \end{array} \right)$$

Продельваем обратный ход метода:  $x_3 = 1$ , далее  $x_2 + 2x_3 = 5$ , откуда  $x_2 = 5 - 2x_3 = 3$ , и, наконец,  $x_1 = 4 - x_2 - x_3 = 0$ . Таким образом, система совместна и имеет единственное решение:  $x_1 = 0$ ,  $x_2 = 3$ ,  $x_3 = 1$ .

Если на каком-то этапе прямого хода метода Гаусса нам надо было исключать очередную переменную, а она уже оказалась исключена ранее (как в примере 1), мы переходим к исключению следующей переменной, ещё оставшейся не исключённой, либо завершаем прямой ход, если таковых уже не осталось. Если мы не получили «невозможных» равенств (что свидетельствовало бы о несовместности системы), то система совместна и имеет бесконечно много решений; неизвестные переменные, стоящие на первом месте во всех уравнениях, оставляем в левой части (их называют базисными переменными), остальные (их называют свободными переменными) — переносим в правую часть с противоположным знаком. Все они могут принимать произвольные значения.

Рассмотренные выше примеры достаточно искусственны, в реальности коэффициенты системы могут весьма сильно различаться, а потому важно не допускать чрезмерного влияния ошибок округления в промежуточных вычислениях. Для этого используют выбор

опорного элемента (*pivoting*): в качестве опорного элемента берут наибольший по модулю элемент, причём поиск его и обмен с ним надо производить всегда.

Иногда используется также метод полного исключения неизвестных (метод Гаусса-Йордана, неправильно называемый методом Гаусса-Жордана, — поскольку в немецком языке нет буквы Ж)... Если преобразуется расширенная матрица, то самый правый столбец в конце будет содержать решение.



## 10. Динамические данные: способы организации

### 10.1. Общие сведения

В программировании много технических задач обработки данных решаются выбором правильной структуры данных, которая разработана специально для эффективного решения именно этой задачи. Многообразие задач, которые приходится решать в современном программировании обусловило многообразие инструментов их решения.

Мы здесь рассмотрим устройство и применение некоторых популярных **контейнеров** – структур данных, которые предназначены для сохранения наборов данных (иногда – весьма больших) каждый из которых будет оптимальным для решения одного класса задач, но нужно помнить, что для других классов задач будет эффективен совсем другой контейнер.

Базовый синтаксис языка Си предусматривает только один *статический* контейнер – массив, размерность которого известна на этапе компиляции. Очевидно, что если мы работаем с ограниченным набором однотипных данных, причем ограничение на количество этих данных известно на момент компиляции – то мы можем выделить массив такой размерности, что его заведомо хватит для всего набора. При этом можно завести переменную, которая будет хранить количество использованных элементов в этом массиве и таким образом наша задача может быть в определенных пределах динамической – количество сохраненных в массив данных может «плавать» в пределах от нуля до его размерности, указанной при его определении. Очевидно также, что это – паллиативное решение.

В общем случае мы не знаем, сколько элементов нам потребуется сохранить (например, потому, что они поступают из внешнего мира – как измерения прибора, ввод от пользователя и т.п.). Конечно, какие-то ограничения на количество сохраняемых данных при этом все равно существуют в силу ограниченности оперативной памяти компьютера или его жесткого диска, однако, для очень широкого класса задач ресурсов современного компьютера будет хватать с избытком.

Для построения контейнеров, которые могут при необходимости увеличивать свой размер, нужны *динамическая память* и *указатели*. Динамическая память – это память, которую программа может «попросить» у операционной системы при помощи вызова специальной библиотечной функции (`malloc()` в языке Си) или специального оператора (оператор `new` в языке Си++). Если операционная система располагает необходимым количеством памяти, то она вернет нашей программе указатель на выделенный блок памяти необходимого размера:

```
#include <stdlib.h>
. . .
unsigned size = 100;
double *A = (double*)malloc(sizeof(double)*size);
```

Указатель нужно понимать как целое число, задающий смещение нужного блока памяти от «начала памяти», то есть – *адрес* этого блока. В данном примере мы выделяем блок памяти под столько вещественных чисел, сколько указано в переменной `size`. Подчеркнём – в отличие от определения статического массива, где мы обязаны задавать константный размер, здесь мы используем переменную, значение которой может быть на этапе компиляции не определено (например, введено пользователем).

Если же у операционной системы нет необходимого количества памяти, то она вернёт нам

специальный нулевой указатель (адрес памяти «нулевого байта»), который по соглашению не может использоваться ни для чего, кроме как для подобных случаев:

```
if (A == NULL) { /* сообщить об ошибке, прервать программу */ }
/* можно работать с массивом A */
```

Вся выделенная программе память будет автоматически возвращена операционной системе при завершении программы, однако для эффективного использования ресурсов компьютера выделенную память следует вернуть операционной системе как только она перестанет быть нужна нашей программе:

```
free(A);
```

В библиотеках языка Си есть также специальная функция, которая меняет ранее выделенный размер блока памяти (увеличивает или уменьшает) – с сохранением тех значений из предыдущего блока памяти, которые поместились в новый размер блока памяти:

```
size *= 2;
A = (double*)realloc(A, sizeof(double)*size);
```

## 10.2. Динамический массив (вектор)

Мы видим, что даже для простейшей динамической структуры данных, которой является динамический массив, нам необходимо поддерживать как минимум две связанные переменные: указатель начала массива (**A**) и размер нашего массива (**size**).

Чтобы хранить, а главное, передавать в функции такие связанные переменные вместе, удобно использовать структуры языка Си:

```
struct array {
    double* A;
    unsigned size;
};
```

В языке Си++ удобнее использовать классы, которые определяют как единое целое не только данные, которые надо сохранить, но и функции для работы с ними.

Итак, динамический массив – это совокупность однотипных элементов, занимающих смежные области памяти. Это самая экономная динамическая структура данных – этот контейнер не тратит никакой дополнительной памяти сверх необходимой для хранения одного элемента.

Давайте посмотрим на типичные операции с данными, которые должен поддерживать любой контейнер: вставка элемента, удаление элемента, поиск элемента по его значению. Прокомментируем производительность, с которой выполняются эти операции.

Самая элементарная операция с элементом массива – это прочитать или записать его по заданному индексу:

$A[i] = 9.8;$

Такая операция не зависит от количества элементов в массиве, её выполнение занимает константное (и при этом очень малое) время.

Вставка элемента в произвольное место массива – это операция, линейно зависящая от размера массива. Мы должны выделить память нового размера (на один элемент больше), скопировать начало массива до точки вставки, скопировать новый элемент, скопировать остаток массива после точки вставки. Потом следует освободить память массива старого размера. Очевидно, что при такой вставке будут скопированы все элементы массива, и чем массив больше, тем больше элементов придётся скопировать.

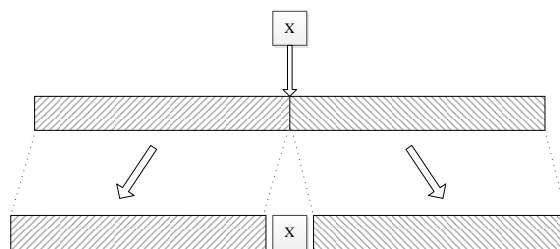


Рис. 8: Вставка элемента

Аналогично и для удаления одного элемента из произвольного места массива – даже, если мы не будем уменьшать выделенную память, нам в среднем придётся скопировать примерно половину элементов, а «половина элементов» – это линейная зависимость от размера массива.

Теперь посмотрим, как быстро можно найти элемент в массиве с заданным значением. Если не предпринимать специальных мер, то мы вынуждены будем просматривать элементы массива один за другим. В среднем мы найдём наш элемент «где-то в середине», то есть, опять же – на половине длины массива. Поэтому и поиск элемента в массиве – линейная по его размеру операция.

Ранее указывалось, что если поддерживать массив в отсортированном по возрастанию значений порядке, то можно будет найти элемент с нужным значением методом деления его длины пополам (дихотомия), что даст логарифмическую, а не линейную зависимость времени выполнения этой операции от размера массива. На больших массивах это даёт колоссальный выигрыш в производительности.

### 10.3. Стек, дека, очередь

Если сузить класс решаемых задач, то с помощью совсем небольшого усложнения структуры данных можно добиться более эффективного выполнения операций вставки и удаления элемента в конце массива.

А именно, давайте – кроме размера памяти, выделенной под хранение элементов массива, – заведём в структуре данных ещё одно поле, в котором будем хранить количество элементов в нашем массиве (буфере):

```
struct array {
    double* A;
    unsigned total; /* всего выделено память под столько элементов */
    unsigned size; /* столько из них используется реально */
};
```

Теперь мы можем в пределах размера `total` добавлять и удалять элементы в хвосте массива за *константное время*.

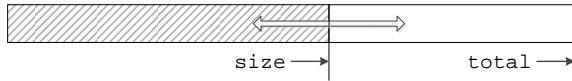


Рис. 9: Вставка-удаление в конце

То есть, такая вставка и удаление не зависят от количества элементов, ранее сохраненных в массив, фактически, нам надо скопировать (один) новый элемент и увеличить одну целочисленную переменную (**size**) с проверкой, что она не превысит полного размера массива **total**.

Если же нам размера **total** не хватит для очередной операции добавления элемента, то никуда не денешься, нужно будет выделять увеличенный размер памяти и копировать все элементы из старого буфера в новый и это будет линейная по количеству элементов операция. Однако и здесь можно применить некоторую оптимизацию, чтобы такая проблема возникала как можно реже. Обычно при нехватке буфера просто удваивают его размер, в результате размер буфера растет экспоненциально и он за несколько таких операций быстро дорастает до такого размера, когда его хватит для решения всей задачи.

Тактика удаления элементов ещё проще: при уменьшении количества элементов буфер обычно не уменьшают, при этом, конечно, возникает перерасход памяти, но зато операций, линейных по количеству элементов массива нет вовсе – любая операция удаления любого количества элементов в конце массива становится константной, не зависящей от количества элементов.

Отметим, что операция взятия элемента по индексу никуда не делась – она по-прежнему занимает константное время. А вот поиск элемента по значению возможен только линейный – мы ведь добавляем элементы только в конец, поэтому упорядочить их по значению не можем.

Задачи, связанные с добавлением-удалением элементов в конец массива возникают достаточно часто, поэтому в программировании выделяют специальный вид контейнера, называемый **стек** (stack, стопка). Такой контейнер чаще всего реализуется в виде описанного здесь динамического массива, но поддерживает он всего две операции: «добавить элемент» – **push()** и «извлечь элемент» – **pop()**. При этом элементы извлекаются в порядке, обратном поступлению – «последний пришедший уходит первым» (LIFO: LastIn, FirstOut).

Наш контейнер можно ещё усложнить для того, чтобы организовать двустороннюю вставку и удаление элементов за константное время. Для этого, кроме параметров буфера, хранящего элементы, нам нужно будет запомнить положение начала и конца области буфера, занятой под хранение элементов. Такой контейнер называется **декой** (deque, double-ended queue – двусторонняя очередь):

```
struct deque {
    double* A;
    unsigned total; /* всего выделено память под столько элементов */
    unsigned first; /* реальный индекс первого элемента в буфере */
    unsigned end; /* индекс элемента, следующего за последним в буфере */
};
```

Если **first** равен **end** – то дека пуста, если нет, то она содержит **end - first** элементов. При этом в самом начале **first** и **end** устанавливаются на середину буфера – чтобы максимизировать свободное пространство, доступное для вставки и удаления элементов в начало или конец деки.

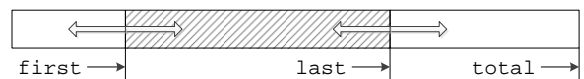


Рис. 10: Хранение элементов в deque

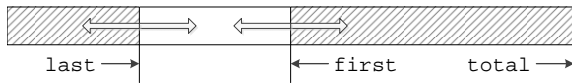


Рис. 11: Хранение элементов в деке

и отложить, если «обернуть» достигший края буфера конец нашей деки через противоположный край буфера.

При этом усложнится алгоритм определения  $i$ -того элемента, однако, эта операция все равно останется константной по отношению к количеству элементов в деке (т.е. не будет зависеть от этого количества).

Для определения момента, когда уже абсолютно необходимо увеличить размер буфера, применяют следующее правило: если `first` достигает `last` из положения «`first` меньше `last`», то это означает, что наша дека опустела, если же наоборот, `first` достигает `last` из положения «`first` больше `last`», то это означает, что свободное место в буфере закончилось и его размер нужно увеличить.

Увеличение размера буфера можно производить, пользуясь той же тактикой удвоения его размера, что и для динамического массива, только при этом следует аккуратно скопировать элементы из старого буфера в новый так, чтобы они заняли середину нового массива (см. схему 3.1).

Итак, дека поддерживает операции: `push_front()`, `pop_front()`, `push_tail()`, `pop_tail()` и выполняет их за константное время. Также за константное время можно получить адрес  $i$ -того по порядку элемента деки (операция взятия элемента по индексу), как в массиве, хотя эта операция и будет производиться дольше, чем для массива, — из-за необходимости *вычислять* реальное положение нужного элемента, — но она всё равно не будет зависеть от количества хранимых элементов.

Отметим, что реализация деки в стандартных библиотеках устроена сложнее, чем это было рассказано выше: для того, чтобы чуть быстрее вычислять индекс и более предсказуемо выделять новую память при ее нехватке, — но в целом принцип работы деки остаётся похожим на то, что было здесь рассказано. Описанная здесь реализация минимизирует накладные расходы памяти.

На практике довольно часто встречаются более жестко определенные *направленные очереди*: когда элемент вставляется в очередь с одного конца, а извлекается всегда с другого конца очереди — так же, как это происходит в реальных очередях, например, в столовой. Поэтому часто определяют для таких очередей самостоятельный контейнер, который поддерживает две операции: вставки в хвост очереди и удаление из начала очереди. Так же как для стека, их можно назвать `push()` и `pop()`, но следует помнить, что порядок извлечения элементов тут будет другой: LIFO (LastIn, FirstOut).

## 10.4. Список

В том случае, когда нужно за константное время вставить новый элемент в произвольное место контейнера, имеет смысл применить другую структуру данных, которая называется односвязным **списком**. В списке каждый элемент содержит кроме тех данных, которые надо сохранить, ещё одно служебное поле — указатель на следующий элемент списка:

```

struct ListElem {
    double val;
    struct ListElem *next;
};

```

Головной элемент списка обычно задаётся отдельной структурой, в которой хранится указатель первого элемента списка и количество хранимых элементов:

```

struct ListHead {
    struct ListElem *start;
    unsigned size;
};

```

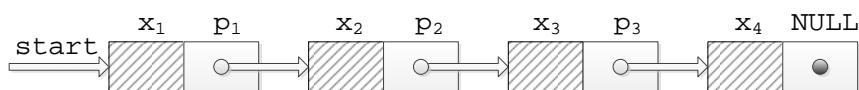


Рис. 12: Хранение элементов в списке

Последний элемент списка маркируется нулевым (NULL) указателем в поле `next`.

Рассмотрим процедуру вставки элемента после указанного:



Рис. 13: Вставка в список

То есть, мы присвоили двум указателям новые значения – и добились вставки в нужную позицию. Очевидно, что в отличие от динамического массива эта операция будет константной, не зависящей от количества элементов. Именно ради этого мы пошли на существенные накладные расходы – мы на каждый элемент храним дополнительный указатель, который занимает 4 или 8 байт. Даже если мы храним вещественные числа – это означает практически удвоение размера занимаемой контейнером памяти, что уж говорить про случай, когда хранить надо более мелкие данные (байты).

Кроме того, нам пришлось отказаться от такой полезной операции, как нахождение элемента по его индексу – в списках, чтобы добраться до  $i$ -того элемента нужно отсчитать  $i$  элементов от начала списка. То есть, операция нахождения элемента по его индексу будет такой же, как операция нахождения элемента по его значению – линейно зависящей от количества сохранённых в список элементов.

На практике чаще бывает нужно вставить новый элемент не *за* указанным элементом списка, а *перед* ним.

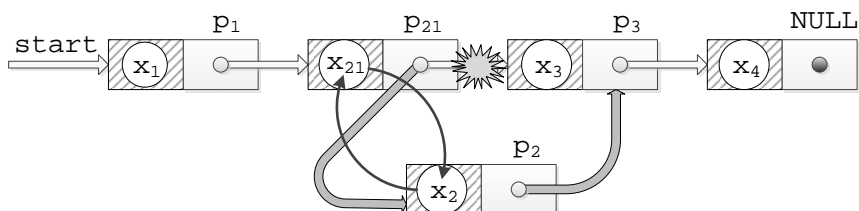


Рис. 14: Вставка в список перед нужным элементом

Для того, чтобы получить нужный результат, обычно вставляют элемент за указанным элементом, как в предыдущем случае, но после этого обмениваются значения, хранимые в двух соседних элементах.

Удаление элемента, следующего за указанным тоже константная операция:

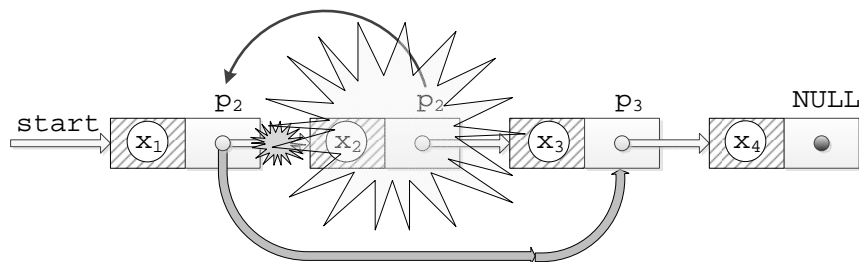


Рис. 15: Удаление из списка элемента, следующего за указанным

Аналогично тому, как это делалось при вставке, можно реализовать удаление указанного элемента: удаляется элемент, следующий за указанным, но предварительно обмениваются значения двух элементов.

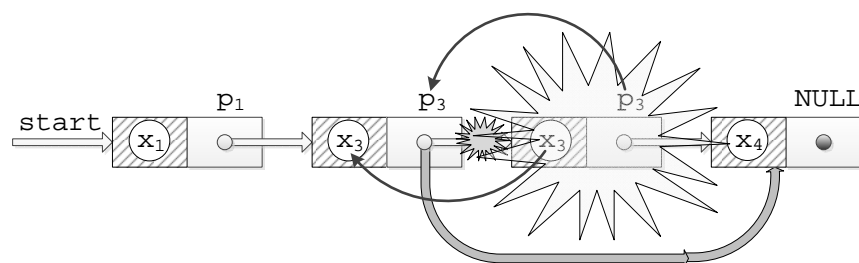


Рис. 16: Удаление из списка указанного элемента

Отметим, что константной является операция добавления в список любого количества элементов, то есть, вставка одного списка в другой выполняется практически с той же скоростью, что и вставка одного элемента (сравнить с массивом).

Довольно сильно влияет на производительность работы со списками то, в каком порядке мы храним элементы списка, в какую позицию мы их вставляем новые элементы. Типичная операция – это «поиск со вставкой», для неё логичнее всего вставлять новый элемент в хвост списка: ведь при поиске элемента мы просмотрели список от начала до конца, т.е. мы уже находимся на последнем элементе, и если нужно добавить новый элемент, то мы его добавим прямо здесь. Ничуть не сложнее вставка нового элемента в голову списка. Однако в обоих случаях, для того, чтобы убедиться, что нужного элемента ещё нет в списке, мы должны просмотреть этот список от начала до конца, перебрать все его элементы.

Можно сэкономить вдвое при поиске, если поддерживать наш список отсортированным по значениям элементов – в этом случае нам в среднем придётся просмотреть не весь список, а его половину, прежде, чем мы удостоверимся, что нужного элемента в списке нет. Вставка элемента в этом случае производится ровно в том месте, где завершился наш поиск.

На практике часто бывает нужно перейти от текущего элемента не только к последующему за ним элементу, но и к предыдущему. Если для решения этой задачи не жалко выделить еще по одному указателю на каждый элемент списка, то можно организовать двусвязный список:

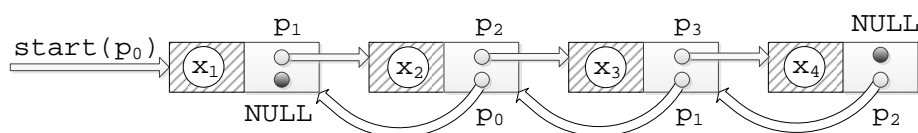


Рис. 17: Двусвязный список

## 10.5. Двоичное (бинарное) дерево поиска

### Простое дерево поиска

В тех случаях, когда основной операцией является «поиск со вставкой» элемента, реализация контейнера в виде отсортированного массива не будет эффективной по времени вставки и удаления. Желательно было бы построить контейнер, который будет обеспечивать в среднем логарифмическое время всех трёх операций: поиска элемента по значению, вставки нового элемента, удаления указанного элемента.

Списки – даже отсортированные по значению – не обеспечат логарифмического времени поиска. Для решения этой задачи используется контейнер, представляющий из себя двоичное дерево (каждый его элемент содержит два указателя – на левое поддерево и на правое поддерево):

```
struct TreeNode {
    double val;
    struct TreeNode *left;
    struct TreeNode *right;
};
```

Как и в случае списков, корень дерева определяется отдельной структурой данных, хранящей указатель на корневой элемент и общее количество элементов в дереве. Для того, чтобы можно было производить поиск элемента по значению, узлы дерева поддерживают в определенном порядке, а именно, значения (их еще называют ключами поиска) всех узлов левого поддерева данного узла должны быть меньше, а значениях всех узлов правого поддерева – больше значения, хранимого в данном узле.

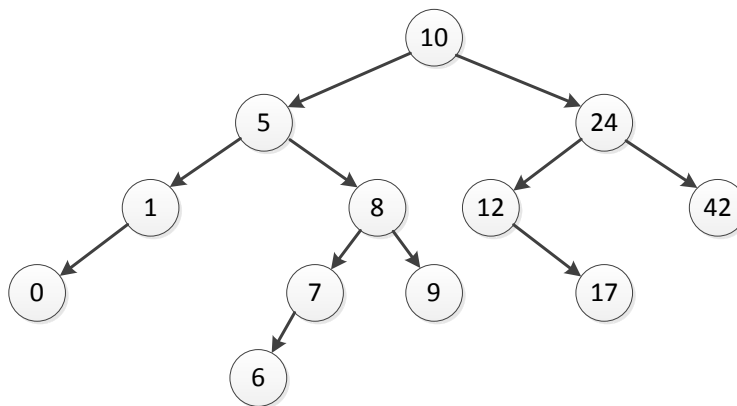


Рис. 18: Двоичное дерево поиска

Узлы, не содержащие потомков, называют *листьями* дерева. Отсутствующий потомок маркируется нулевым (NULL) указателем в родительском узле. Здесь приводится двоичное дерево, которое хранит уникальные, не повторяющиеся ключи, однако, его несложно обобщить и на случай повторяющихся ключей, например, можно считать, что в левом поддерева хранятся ключи строго меньше текущего, а в правом – большие, либо равные ему.

Поиск нужного значения в таком дереве будет несложным:

```
while (t) {
    if (t->key == x) { break; /* Найдено! */ }
    else if (x < t->key) { t = t->left; }
    else { t = t->right; }
}
```



Мы спускаемся от корня к листьям, в каждом узле выбираем направление движения либо в левое, либо в правое поддерево. Если искомое значение в дереве есть – мы его найдём, если же оно отсутствует – мы доберёмся до листа, причём, если нам надо будет вставить искомое значение – то мы вставим его прямо в этот лист. Сама операция вставки в этом случае не будет зависеть от количества узлов в нашем дереве, но она обязательно должна предваряться процедурой поиска значения в дереве, а эта операция будет пропорциональна высоте дерева.

Более-менее очевидно, что если нам ключи для поиска и вставки поступают в случайном порядке, то наше дерево будет достаточно плотно заполнено, и, следовательно, его высота не будет существенно отличаться от высоты *идеально сбалансированного дерева*. Более того, это утверждение было строго доказано: высота спонтанно заполняемого дерева поиска отличается от высоты идеально сбалансированного дерева только множителем. А высота идеально сбалансированного дерева равна, очевидно, двоичному логарифму от количества его узлов.

То есть, для двоичного дерева и поиск, и добавление нового элемента будут логарифмически зависеть от количества сохранённых данных, это очень хороший результат: например, если у нас сохранено 65 535 элементов, то высота такого дерева будет равна всего 16.

Осталось выяснить, как будут удаляться элементы? Очевидно, чтобы удалить нужное значение, его необходимо сначала найти в нашем дереве. Мы помним, что поиск занимает логарифмическое время.

Далее, если мы попали в лист – то его удалить легко: нужно просто обнулить один указатель в родительском узле. Если нам нужно удалить узел с одним потомком – мы с этим тоже легко справимся, в этом случае надо заменить в родительском узле указатель, который указывает на удаляемый узел на указатель единственного потомка удаляемого узла. Понятно, что обе эти операции будут константными по времени.

А что делать, если нужно удалить внутренний узел, у которого два потомка? Ведь у родителя удаляемого узла можно заменить только один указатель, а тут надо сохранить два указателя. В этом случае задачу удаления внутреннего узла сводят к задаче удаления листа: давайте в правом поддереве удаляемого узла найдём самый левый лист и удалим его, но перед удалением – обменяем ключи найденного листа и удаляемого узла:

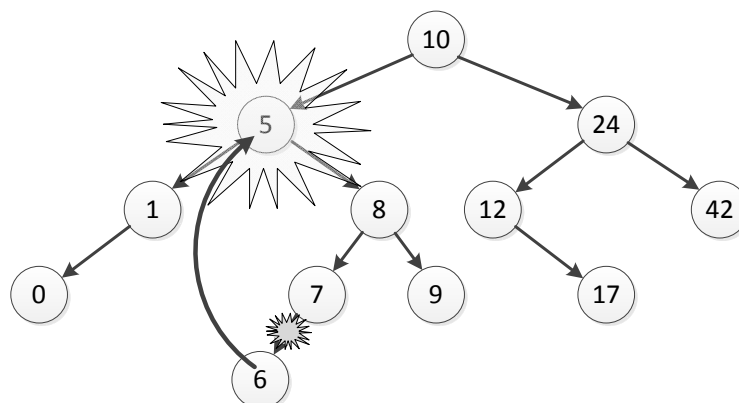


Рис. 19: Удаление внутреннего узла

Легко заметить, что такая операция сохранит главное свойство нашего дерева поиска: отношения порядка между левыми и правыми поддеревьями для любого узла, так как самый *левый* узел в *правом поддереве* – меньше всех остальных ключей этого поддерева, но одновременно он больше любого ключа *левого поддерева*.

С точки зрения производительности единственное, что нам придётся сделать – это дойти до листа, то есть пройти всю высоту дерева. Удаление же листа – константная операция. Таким образом, мы приходим к выводу, что и время на удаление любого ключа из двоичного дерева будет пропорционально логарифму от общего количества хранимых элементов.

## Балансировка двоичного дерева

То, что тут было рассказано о двоичных деревьях, показывает их эффективность для случайно заполненного дерева. Однако, если ключи поступают для заполнения не случайно (а, например, в порядке возрастания их значения), – наше дерево выродится в одну длинную линейную ветвь и его высота будет пропорциональна общему количеству узлов, поэтому ни о какой логарифмической эффективности основных операций в таком случае речь идти не может.

Для того, чтобы застраховаться от таких «неудачных» случаев, нужно при каждой операции вставки или удаления ключа предусмотреть специальную процедуру перестроения дерева для того, чтобы длины его ветвей были как-то сбалансированы.

Отметим, что понятие идеально сбалансированного дерева – неэффективно с вычислительной точки зрения: в идеально сбалансированном дереве количество узлов в левом и правом поддеревьях любого узла различается не более, чем на единицу. То есть, даже для того, чтобы проверить в каждом узле идеальную сбалансированность, нам надо пересчитать все дочерние узлы. В среднем мы будем находиться где-то в середине дерева, при этом количество дочерних узлов будет пропорционально общему количеству узлов дерева, что и показывает неэффективность этого подхода.

Но если наложить ограничение не на количество узлов в поддеревьях узла, а лишь на разность высот левого и правого поддерева (а ведь именно высота дерева влияет на время поиска элемента!), то алгоритм можно получить гораздо более эффективный.

Назовём дерево *сбалансированным* (пусть и неидеально) *АВЛ-деревом* (по первым буквам фамилий: Адельсон-Вельский и Ландис), если для любого его узла выполняется правило: высоты его левого и правого поддеревьев могут различаться не более, чем на единицу.

Отечественные математики Адельсон-Вельский и Ландис строго доказали, что принципиально различных случаев разбалансированности двоичного дерева может быть всего два, остальные получаются из них отражением вокруг вертикальной оси.

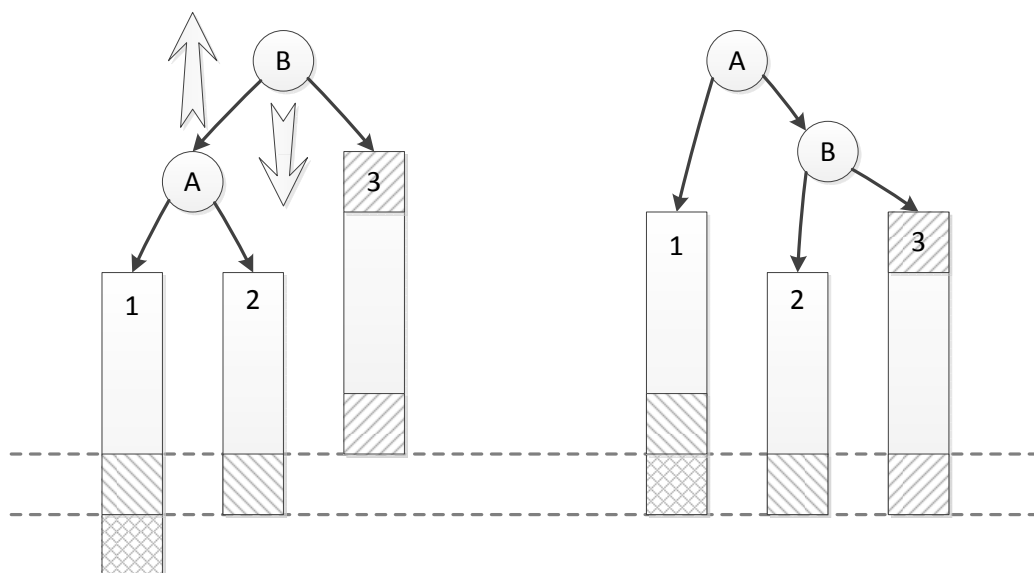


Рис. 20: АВЛ-балансировка первого типа

Рассмотрим первый вариант разбалансированного дерева, который может возникнуть, например, при вставке нового узла в поддерево 1 (см. схему 5.3). Заметим следующее соотношение между ключами узлов и деревьев в этом случае: ключи дерева 1 меньше ключа узла А, ключ узла А меньше ключей дерева 2, ключи дерева 2 меньше ключа узла В, ключ узла В меньше ключей дерева 3.

В исходном состоянии дерево 2 является поддеревом узла А, но в силу приведённого соотношения дерево 2 имеет право быть и поддеревом узла В. Поэтому поменяем в узле В левую связь на указатель поддерева 2, а правую связь узла А – направим на узел В. Если после этого подтянуть узел А на одну позицию вверх, а узел В – опустить на одну позицию вниз, то мы получим сбалансированное дерево, показанное на схеме 5.3 справа. То есть, мы устранили разбалансировку этого типа.

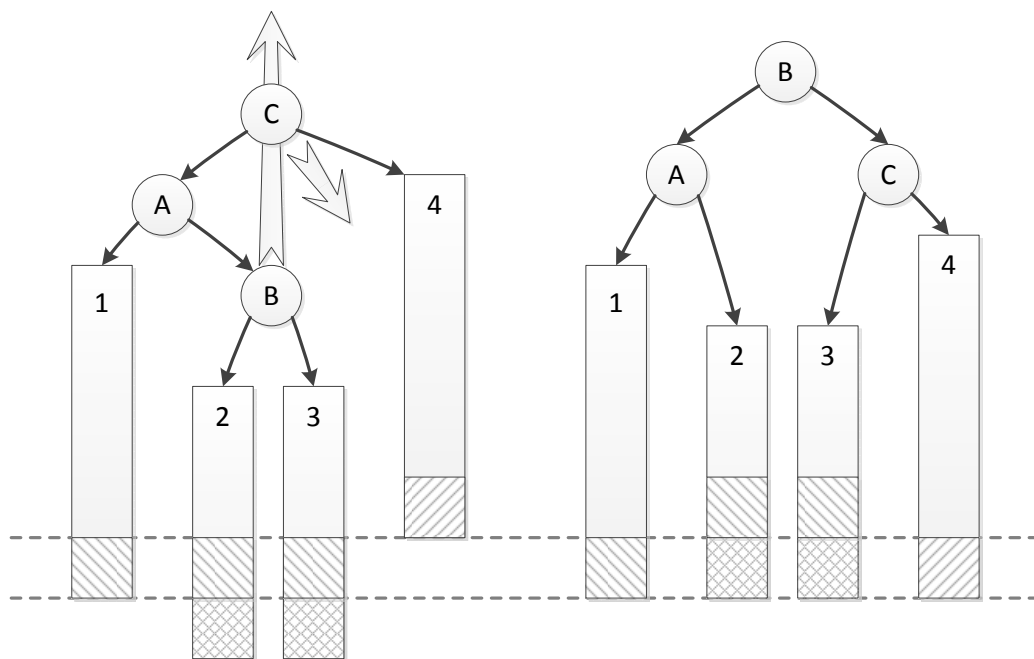


Рис. 21: AVL-балансировка второго типа

Так как вставка в двоичное дерево осуществляется в листья, то разбалансировка второго типа не может возникнуть в результате обычной вставки — так как сверх допустимого опустились бы вниз сразу два поддерева, а уже единственная вставка привела бы к разбалансировке, которую мы устранили бы описанным выше способом. Но такое разбалансированное дерево может возникнуть как результат удаления узла, например, в поддереве 4. И эту разбалансированность тоже предстоит устранить, пользуясь тем же свойством упорядоченности поддеревьев: ключи дерева 1 меньше ключа узла А, ключ узла А меньше ключей дерева 2, ключи дерева 2 меньше ключа узла В, ключ узла В меньше ключей дерева 3, ключи дерева 3 меньше ключа узла С, ключ узла С меньше ключей дерева 4. Поэтому дерево 2 имеет право оказаться правым поддеревом узла В, а дерево 3 имеет право оказаться левым поддеревом узла С. В свою очередь, узел А может находиться на левой связи узла В, а узел С имеет право находиться на правой связи узла В. Если после такого обмена связями подравнять получившуюся высоту дерева, подтянув узел В на две позиции вверх, то мы опять придём к полностью сбалансированному дереву, показанному на схеме 5.4 справа – так как деревья 2 и 3 поднимутся на одну позицию, а дерево 4 – на одну позицию опустится.

Алгоритмов балансировки двоичных деревьев разработано много, однако, почти все они тем или иным способом ограничивают высоты левого и правого поддеревьев любого узла. Так, алгоритм «красно-чёрных деревьев», лежащий в основе некоторых контейнеров стандартных библиотек, допускает, что высоты левого и правого поддеревьев могут отличаться не более, чем в два раза. При этом деревья получаются более высокими, чем в описанном здесь случае, зато затраты времени на балансировку будут гораздо меньше, так как балансировку можно производить реже и сама она менее накладна. Говоря проще, красно-чёрные деревья проигрывают AVL-деревьям по времени поиска, но выигрывают

– по времени вставки и удаления элементов, оставаясь, тем не менее, логарифмически эффективными по всем трём операциям.

## 10.6. Ассоциативные контейнеры, В-деревья, хэш-таблицы

### Ассоциативные контейнеры

Описанная здесь структура данных эффективна для решения задачи поиска значения (ключа) в данном контейнере и при этом поддерживаются столь же эффективные операции добавления и удаления элементов. В стандартных библиотеках такой вид контейнера называется множеством (**set**), так как для множеств типичными операциями как раз и будут: добавление элемента в множество, удаление элемента из него и определение – входит ли элемент в данное множество?

Другой класс задач возникает, когда по указанному ключу надо быстро найти связанное с ним значение, при этом ключ и значение могут быть произвольного типа, например, ключ – строка, а значение число или наоборот.

По своему устройству такой контейнер мало чем отличается от описанного здесь двоичного дерева, просто вместо одного ключа в каждом узле хранится пара «ключ-значение». Но так как здесь решается задача в немного другой постановке, то программный интерфейс (набор функций) удобно сделать отличающимся от контейнера **set**.

Контейнеры, которые эффективно решают задачу поиска значения, связанного с указанным ключом, называются *ассоциативными контейнерами* (словарями), так как с каждым ключом ассоциируется некоторое значение. В стандартных библиотеках такие ассоциативные контейнеры представлены контейнерами, называемыми **map** (от слова «отображение»).

Ассоциативный контейнер можно построить на любой из структур данных, описанных выше: и на упорядоченном массиве, и на двоичном дереве, и на хэш-таблице, – не следует только забывать о принципиально разной эффективности каждой из структур данных при решении различных классов задач.

### В-деревья

Всё, что обсуждалось до сих пор, прекрасно решает различные задачи в программировании, когда весь набор данных *помещается в оперативную память компьютера*. Если же данные не могут быть размещены в памяти, возникает необходимость в других способах их организации.

Большие файлы с данными (базы данных) могут храниться на жёстком диске компьютеров, которые уже на протяжении многих лет имеют на порядок большую ёмкость по сравнению с оперативной памятью. К сожалению, диски к тому же имеют ещё и на несколько порядков меньшее время доступа к данным по сравнению с оперативной памятью – в силу необходимости механического перемещения читающей магнитной головки к нужному участку диска.

А в таких больших массивах данных тоже бывает нужно добавлять новые данные, удалять избранные данные либо что-то находить — и всё это надо делать с максимальной эффективностью.

Сейчас появились и быстро дешевет твердотельные электронные диски, в которых отсутствуют механические части, однако их ёмкость всё ещё существенно уступает ёмкости обычных механических жёстких дисков (а время доступа всё равно гораздо больше времени доступа к данным в оперативной памяти), поэтому алгоритмы, разработанные для решения задачи поиска во «внешней памяти», ещё значительное время будут сохранять актуальность.

Давайте попробуем решать те же задачи (поиск, вставка и удаление ключей), которые мы решали с помощью двоичного дерева, для больших данных, хранящихся на стандартном жёстком диске компьютера.

Как известно, операционная система предоставляет специальные системные вызовы для позиционирования следующей операции чтения на любое нужное смещение от начала файла. Однако такое позиционирование по меркам времени центрального процессора будет очень долгим, поэтому пытаться применить алгоритм двоичного дерева поиска довольно бессмысленно — он будет чрезвычайно медленным. Стоит также вспомнить и о том, что жёсткий диск — устройство блочного чтения, позиционирование — операция долгая, но после начала чтения быстро считывается и запоминается во временном буфере достаточно большая непрерывная область данных. Происходит так потому, что при чтении сама читающая головка неподвижна, а под ней со скоростью 5-10 тысяч оборотов в секунду вращается жёсткий диск.

Размер одномоментно считываемой области зависит от настроек файловой системы (которые задаются при форматировании диска средствами операционной системы). Называется такая область кластером или страницей жёсткого диска. Для больших жёстких дисков размер страницы достигает 64 килобайт и более — представьте себе, сколько ключей, содержащих целые числа, может поместиться на такой странице! Десятки тысяч.

Так как в двоичном дереве поиска основное количество операций производится со смежными узлами, будем такие смежные узлы держать на одной странице:

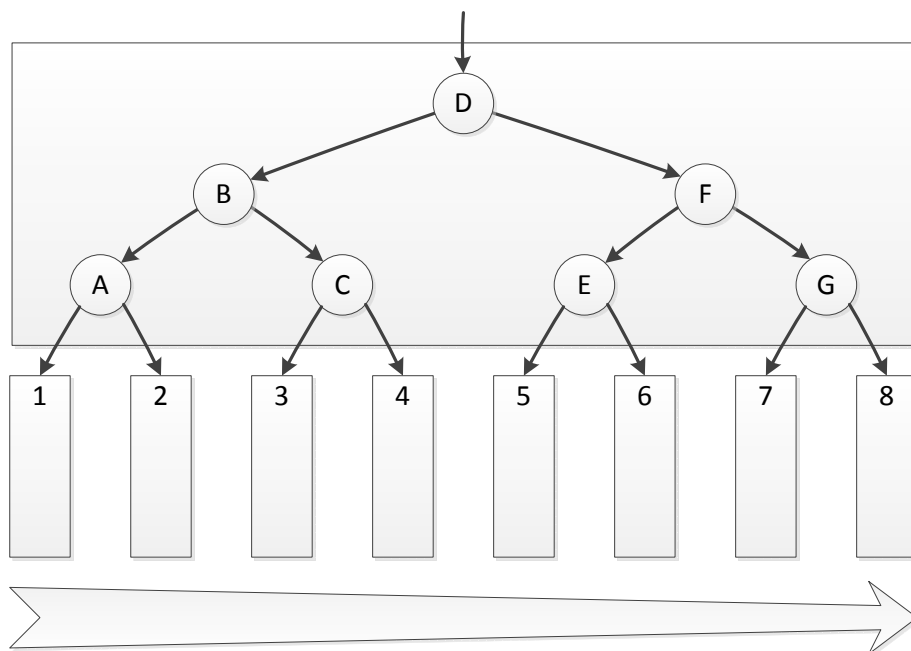


Рис. 22: Страничная организация двоичного дерева

Обратите внимание, относительно таких страниц мы всё равно видим дерево, но не двоичное, а сильно ветвящееся — на полностью заполненной странице количество дочерних страниц будет равно количеству узлов на странице плюс одна страница. Ещё одно важное свойство таких дочерних страниц будет заключаться в том, что ключи в них образуют

непересекающиеся упорядоченные слева направо множества. Очевидно, что для достижения максимальной эффективности все страницы должны быть одинакового размера, соответствующего размеру кластера файловой системы. Поэтому если какая-то страница заполнена не полностью, то она всё равно имеет тот же размер, а места отсутствующих ключей на такой странице будут вакантными.

Теперь вспомним, что операции с данными в уже считанной странице на много порядков быстрее чтения этой страницы в память. Поэтому держать на ней сбалансированное дерево ключей или что-то подобное – неэффективно, лучше весь доступный размер страницы отвести на хранение полезных данных. То есть, можно «сплющить» двоичное дерево в отсортированный массив ключей (самая компактная структура данных) и параллельный ему массив ссылок на дочерние страницы (размерности на единицу больше, чем размерность массива ключей).

Заметим, что при таком «сплющивании» каждый ключ данной страницы займёт позицию между двумя ссылками на дочерние страницы, левая дочерняя страница будет содержать ключи, меньшие данного ключа, а правая – большие (по-прежнему считаем все ключи уникальными, хотя задача достаточно тривиально обобщается на случай, когда есть не уникальные ключи):

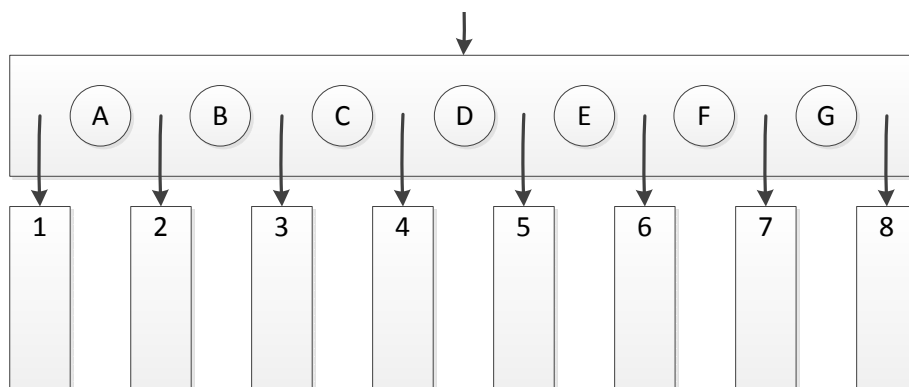


Рис. 23: Страничная организация двоичного дерева

Здесь вместо ссылок будут фигурировать смещения начала каждой дочерней страницы от начала файла – это смещение надо указать системному вызову операционной системы для перемещения позиции следующего чтения к началу нужной страницы. Помним также о том, что в силу того, что массив ключей страницы отсортирован, в нём можно быстро искать дихотомией (делением массива пополам).

Так как получаемые страницы достаточно велики (дерево страниц сильно ветвится), то мы сможем буквально за пару-тройку перемещений позиции чтения находить данные в базе данных, хранящей многие миллиарды ключей.

Разберёмся, как такое дерево страниц построить – как в него добавлять ключи? Вставка новых страниц «как в двоичном дереве» тут невозможна – будет очень неэффективно «раздвигать» файл на жёстком диске, как мы это делали с массивом в оперативной памяти. Вместо этого все новые страницы будут добавляться просто в конец файла, с указанием правильного смещения к ней в нужном месте родительской страницы. Но в этих операциях добавления новой страницы тоже можно сильно сэкономить, если на *каждой* странице предусмотреть достаточное количество *вакантных мест* под ключи. Ну, и сами страницы должны образовывать каким-то образом сбалансированное сильно ветвящееся дерево – чтобы избежать длинных цепочек страниц в каком-нибудь неудачном случае.

Именно такой алгоритм и разработали Бэйер и Маккрейт (Bayer, McCreight) в 1970 го-

ду, в честь одного из них структура данных получила название *B-дерево* (*B-дерево* – не бинарное!).

**Определение.** *B-деревом* называется сильно ветвящееся дерево страниц, в котором каждая страница, кроме, быть может, корневой, содержит не менее  $K$  и не более  $2K$  ключей. Число  $K$  называется порядком *B-дерева*.

Довольно очевиден алгоритм поиска в таком дереве: начинаем с корневой страницы (которую для простоты можно просто всю целиком держать в памяти), находим между какой парой ключей расположено значение искомого ключа и проходим к дочерней странице по связи, лежащей между этими двумя ключами. Если искомое значение меньше минимального ключа страницы или больше максимального, то проходим по самой левой или самой правой связи соответственно. В случае сбалансированного дерева очевидно, что такой поиск займет  $O(\log_K N)$  операций.

Рассмотрим алгоритм вставки нового ключа с одновременной балансировкой дерева страниц:

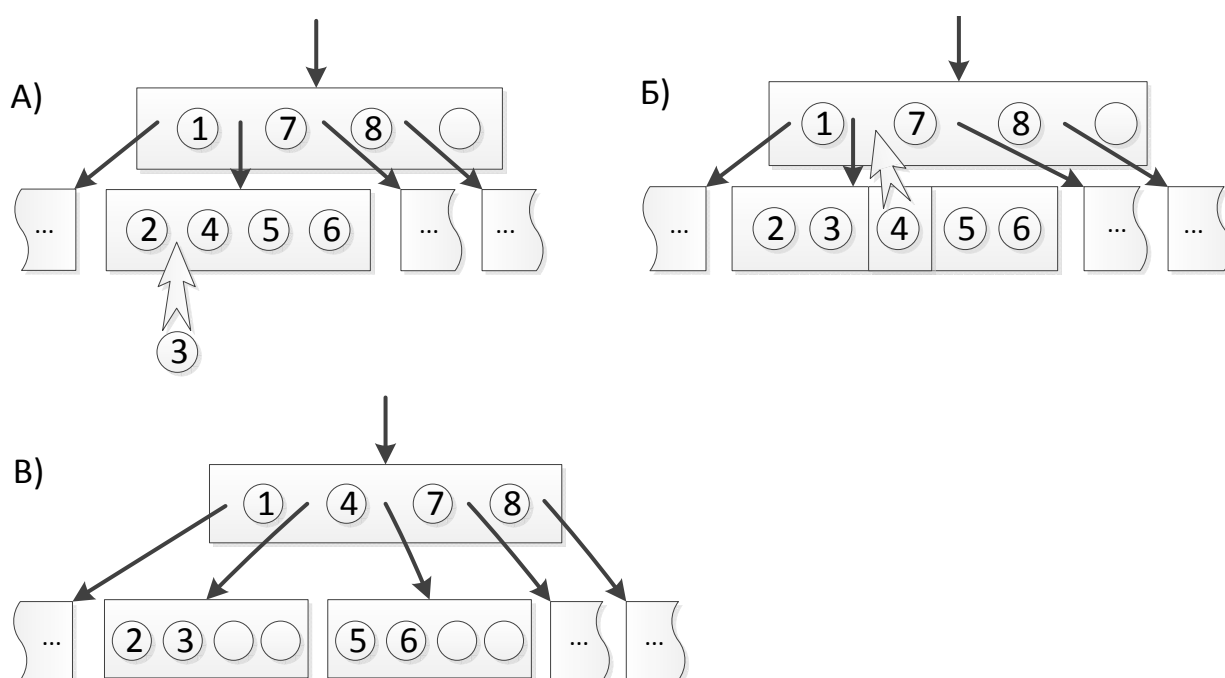


Рис. 24: Вставка в *B-дерево*

1. Мы дошли до листовой страницы и не нашли искомого ключа.
2. Если на странице меньше  $2K$  ключей (она не заполнена) – то просто вставляем ключ в в нужное место отсортированного массива ключей этой страницы. Все дочерние связи у листовой страницы – нулевые.
3. Если вставлять некуда (рис.24 А: страница заполнена, на ней уже есть  $2K$  ключей плюс один ключ, который мы хотим вставить, всего –  $2K + 1$  ключей), то всё равно вставляем этот ключ в отсортированный массив (рис.24 Б: массив уже в памяти!), но сразу после этого – делим его пополам:  $K$  первых по порядку ключей оставляем на текущей странице, для  $K$  последних по порядку ключей создаем новую листовую страницу (и записываем её в конец файла), а один средний ключ – отправляем в родительскую страницу.
4. Если на родительской странице есть вакансия – вставляем «лишний» ключ в нужное место и образуем справа от него новую связь на новую листовую страницу (рис.24 В).

5. Если вакансии на родительской странице нет – повторяем для неё п.3.
6. Если дошли до корня, в котором нет вакансии – по алгоритму п.3 образуем новый корень ровно с одним ключом в нём (определение  $B$ -дерева делает специальную оговорку для корня).

Получается, что  $B$ -дерево растёт от листьев к корню, – в отличие от бинарного дерева, где рост происходит в листьях. В самом худшем случае мы поднимемся до корня, то есть пройдем всю высоту дерева, поэтому оценка производительности вставки элемента будет такой же, как и для времени поиска:  $O(\log_K N)$ .

Алгоритм удаления ключа – просто обратный к алгоритму вставки: вместо разбиения страниц, при необходимости, происходит их слияние с заимствованием одного элемента из родительской страницы и с уничтожением ставшей ненужной одной связи в родительской странице. Естественно, оценка производительности удаления ключа останется логарифмической:  $O(\log_K N)$ .

Однако, что происходит с местом на диске при удалении страницы? Ведь удаляем мы её из *середины* файла! Здесь нас выручит то, что все страницы одного размера – мы будем просто в отдельном контейнере в оперативной памяти поддерживать список таких опустевших внутренних страниц – и, если возникнет нужда через какое-то время создать новую страницу – сначала проверим, есть ли незанятые страницы в этом списке? Если есть – займём такую ранее выделенную страницу под новые данные, если нет – создадим новую страницу в конце файла, как раньше. То есть, при удалении страницы никакие смещения, записанные в рабочих страницах – не меняются и с этими страницами возможна нормальная работа.

$B$ -дерево порядка 1 (его ещё называют 2–3-деревом, т.к. внутренние узлы содержат либо две, либо три дочерние связи) довольно успешно используется для балансировки дерева поиска в оперативной памяти компьютера.

Ещё один популярный вариант этой структуры данных называется  $B^+$ -деревом. Он экономит используемую оперативную память в том случае, когда хранятся пары «ключ-значение, связанное с этим ключом», причём размер данных «значения» достаточно велик. От  $B$ -дерева она отличается тем, что абсолютно все ключи (или пары ключ-значение) представлены в листьях, которые, к тому же, ещё провязаны в линейный одно- или даже двухсвязный список, образуя упорядоченное по возрастанию ключей множество пар ключ-значение – для удобства их упорядоченного перебора. Внутренние, не-листовые страницы используются только для поиска и в них хранятся только ключи, некоторые из которых дублируются на каждом из вышележащих уровней:

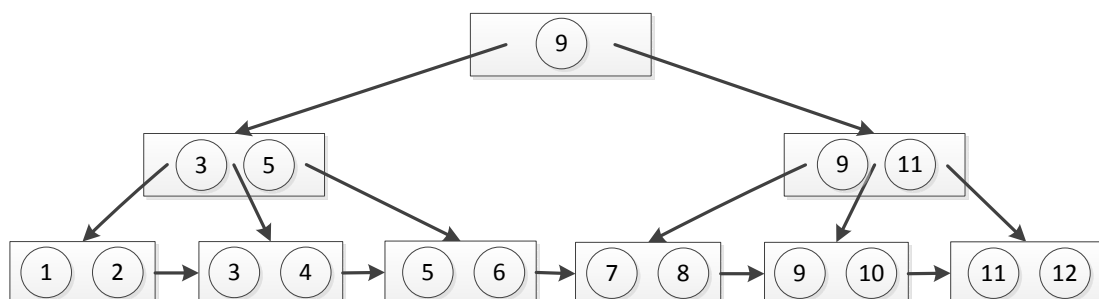


Рис. 25:  $B^+$ -дерево



## Битовые вектора, хэш-таблицы

Рассмотрим такой интересный вопрос: а можно ли построить ассоциативный контейнер, в котором можно будет искать значения за константное, а не за логарифмическое время? Т.е., можно ли сделать так, чтобы время поиска вообще не зависело от того, сколько в контейнере хранится элементов?

Если бы нам надо было бы найти по целочисленному ключу некоторое связанное с ним значение (контейнер типа *map*), то можно было бы просто использовать массив, длина которого равна максимально возможному ключу плюс один и сохранять значения просто в элементах этого массива. То есть, роль ключа будет играть просто индекс в этом массиве. Очевидно, что операция взятия элемента по его индексу не зависит от размера массива, нам ведь достаточно просто прибавить смещение, равное индексу массива к указателю на начало массива – это ровно одна операция!

Если нам нужно решать задачи для ассоциативного контейнера типа *set* (множество), то есть просто быстро давать ответ на вопрос «есть такое-то целое число во множестве или нет?», то можно существенно сэкономить занимаемую память, если просто хранить битовый вектор. Если  $i$ -тый бит вектора установлен в единицу – то мы считаем, что число  $i$  входит в множество, если в ноль – то число отсутствует в нём. Такой вектор будет компактнее в восемь раз по сравнению с вектором значений байтов «истина» и «ложь» и в 32 раза компактнее массива целых чисел.

Однако даже с целыми числами возникает вопрос: если мы априори не знаем, какое может быть максимальное значение ключа во множестве, или это максимальное значение слишком велико (миллиард) – то вектора понадобятся слишком длинные, что потребует многих гигабайт памяти. Что уж говорить о ключах произвольной природы типа вещественных чисел, строк или даже произвольных структур данных! А ведь понятно, что с данными такой природы тоже могут возникнуть задачи поиска, мы же их успешно решали с помощью деревьев поиска.

Решением будет вычисление специально подобранной функции с ключом в качестве аргумента. Пусть пока для простоты ключи у нас будут целочисленными, но произвольными, любой величины. Мы бы хотели подобрать такую функцию, которая бы отобразила любой ключ на целое число, которое можно было бы использовать в качестве индекса в массиве некоторого заранее зафиксированного размера, превышающего общее количество ключей, которые мы бы хотели сохранить в контейнере.

Неплохим вариантом такой функции, которая отображает произвольное целое число в диапазон от нуля до некоторого наперёд заданного  $N$  будет операция нахождения остатка от деления ключа на число  $N$ :

$$H(k) = k \% N.$$

Действительно, ведь все возможные значения остатка как раз и будут занимать диапазон от нуля до  $N - 1$ , то есть пробегать все возможные значения индекса массива размерности  $N$ . Такую функцию называют хэш-функцией (от английского слова *hash* – мелко рубленное, перемешанное блюдо, литературный перевод на русский язык – винегрет), а массив, куда будут сохраняться значения – хэш-таблицей.

Теперь возникает следующий вопрос: если у нас значения ключей не ограничены, а значения хэш-функции – ограничены, то, весьма вероятно, что функция не будет взаимно-однозначной, то есть разным ключам с некоторой ненулевой вероятностью может сопоставиться один и тот же индекс хэш-таблицы. В таком случае, говорят, что в хэш-таблице возникает коллизия:

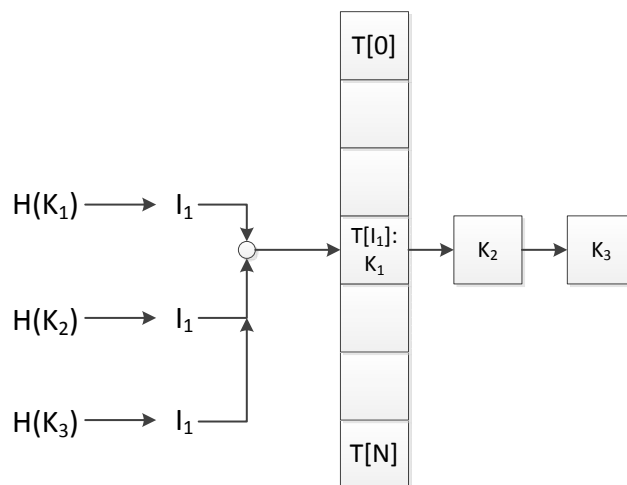


Рис. 26: Коллизии в хэш-таблице и их разрешение методом цепочек

Простейший метод их разрешения – просто стартовать в каждой ячейке хэш-таблицы односторонний список, в котором надо будет искать значения исходного ключа простым перебором. Такой метод разрешения называется *методом цепочек*. Если мы не найдём ключ в этой «цепочке», то можно быть уверенным, что ключ отсутствует в контейнере. Если мы его захотим добавить – достаточно будет его вписать в конец цепочки.

Очевидно, что хэш-функция тем лучше, чем более короткие цепочки она порождает для данного размера хэш-таблицы. В идеале она должна была бы просто равномерно «размазать» все возможные значения ключей по всей таблице, то есть, оптимальной с точки зрения минимизации числа коллизий будет просто генератор псевдослучайных чисел с равномерной плотностью распределения вероятностей. Псевдо – потому, что такой генератор должен давать однозначные результаты, то есть, независимо от порядка, в котором подаются ключи на хэширование – он каждому ключу должен сопоставить одну и ту же ячейку хэш-таблицы.

Известно, что наиболее часто используемый генератор псевдослучайных чисел как раз и основан на функции взятия остатка от деления – и именно поэтому функция взятия остатка показывает неплохие результаты в качестве функции хэширования.

С другой стороны, в отличие от деревьев поиска хэш-таблицы никак не упорядочивают сохраняемые в них ключи, этот порядок вообще-то будет зависеть от порядка, в которых мы подаем ключи на хэширование, то есть, может быть разным для одного и того же набора ключей.

Очевидное соображение: если мы ожидаем, что цепочки будут длинными, то в качестве «хранилища коллизий» лучше использовать не связный список, а дерево поиска или даже хэш-таблицу второго уровня.

Еще одно очевидное соображение: чем меньше заполнена таблица – тем меньше вероятность коллизии, то есть, количество коллизий будет зависеть от соотношения планируемого количества хэшируемых ключей и размера хэш-таблицы. Это говорит в пользу того, что размер хэш-таблицы, – который выбирается заранее, – следует выбирать как можно больше.

Но вот нетривиальный факт: в типовых случаях среднее количество коллизий оказывается небольшим даже при 75% заполнении хэш-таблицы. В теории вероятностей это можно доказать строго, мы же здесь ограничимся только результатами этого доказательства (см. табл.1).

Дробные числа тут можно понимать в смысле: «105 коллизий на 100 операций вставки в среднем при коэффициенте заполнения таблицы 10%». Мы видим, что среднее число коллизий растет не очень быстро, поэтому рекомендация по выбору размера хэш-таблицы отсюда следует довольно простая: если вы планируете добавить в таблицу известное количество ключей – увеличьте её размер на четверть относительно этого количества и при этом среднее количество коллизий на одну операцию будет всё ещё меньше двух, то есть, в среднем операция вставки будет занимать константное время.

коэффициент заполнения таблицы	среднее число коллизий на одну операцию
10%	1.05
25%	1.15
50%	1.39
75%	1.85
90%	2.56
95%	3.15
99%	4.66

Таблица 1: Количество коллизий при разной заполненности хэш-таблицы

Увы, эти соображения не страхуют от возможных неприятностей – случайно можно получить такое соотношение размера таблицы, хэш-функции и ключей, что какая-то одна цепочка окажется очень длинной. Но можно надеяться, что это будет весьма редкой ситуацией. Для того, чтобы немного застраховаться от подобных неприятностей рекомендуют размер таблицы (и модуль в хэш-функции) выбирать простым числом. То есть, полная рекомендация по выбору размера хэш-таблицы звучит так: увеличьте размер на 25% относительно планируемого количества ключей, после чего найдите ближайшее сверху от этого числа *простое число* – и используйте его в качестве модуля хэш-функции и размера хэш-таблицы.

Другая прагматическая рекомендация – в ходе добавления ключей можно каждый раз подсчитывать максимальное или среднее количество коллизий и сохранять его где-то рядом с хэш-таблицей. Если это число покажется вам чрезмерным, можно устроить *рехэширование*: увеличить размер таблицы в два раза (опять найти ближайшее сверху простое число) и переместить все ключи из старой таблицы в новую, вычисляя для каждого ключа хэш-функцию с новым модулем. Понятно, что такая операция будет линейной относительно размера хэш-таблицы, но зато есть надежда, что при тактике удвоения размера применять рехэширование нужно будет достаточно редко.

Если задача заключается в том, чтобы заранее скомпилировать хэш-таблицу из некоторого множества известных ключей, а потом только пользоваться ею для поиска ключей (отвечать на вопрос – «есть ключ в контейнере или нет?», либо искать связанное с ключом значение в ассоциативном контейнере), то можно построить более компактный по памяти алгоритм разрешения коллизий – метод открытой адресации:

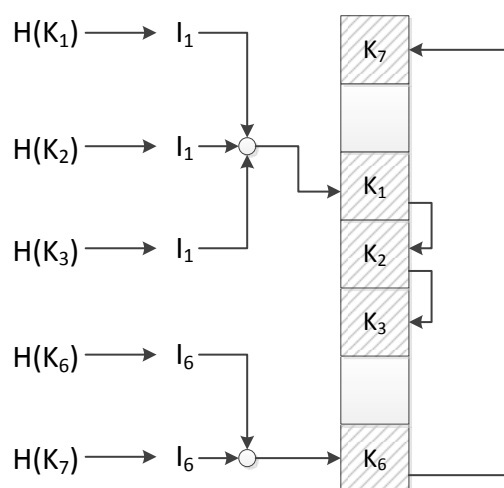


Рис. 27: Разрешение коллизий методом линейных проб при открытой адресации

Попад в какую-либо ячейку хэш-таблицы сравниваем хранящийся там ключ с искомым, и, если они не совпадают – переходим к следующей по порядку ячейке таблицы. Если она пустая – то сохраним в эту ячейку текущий ключ, если заполнена – опять сравниваем искомым ключ с ранее сохранённым. При необходимости – опять перейдем к следующей ячейке, а если мы находимся в конце массива – перепрыгнем в его начало. Рано или поздно мы найдём свободную ячейку, на которой поиск завершится, либо вернемся в самую первую проверенную ячейку, что будет означать, что наша таблица переполнилась и ничего больше в неё сохранить нельзя.

Математически этот алгоритм можно выразить как последовательный перебор разных хэш-функций, одна из которых рано или поздно приведёт нас к успеху:

$$H_0(k) = k \% N, H_1(k) = (k + 1) \% N, H_2(k) = (k + 2) \% N, \dots, H_i(k) = (k + i) \% N$$

Обратите внимание, что этот набор функций линеен по  $i$ , поэтому его и называют методом линейных проб при открытой адресации. Можно применить и более сложные хэш-функции, например, метод квадратичных проб:

$$H_i(k) = (k + i^2) \% N$$

Но обычно и метод линейных проб дает вполне приемлемые результаты.

Понятно, что в любом методе открытой адресации некоторые коллизии будут зависеть от порядка, в котором мы хэшируем ключи: при одном порядке данный ключ будет порождать коллизию, а при другом – не будет. Но среднее количество коллизий все равно будет зависеть от коэффициента заполнения таблицы образом, подобным приведённому в таблице выше, а поэтому это не будет приносить никаких неприятностей для операций поиска и вставки.

Но, в отличие от таблиц с разрешением коллизий методом цепочек, удалять ключи из такой таблицы будет нельзя: удалив какое-то значение мы можем прервать непрерывную последовательность ячеек, в которую сохраняли коллизии, поэтому последующие ключи просто не найдутся, если что-то из таблицы удалить.

В завершение вернёмся к вопросу выбора хэш-функции для нецелочисленных, и, возможно, вообще для нечисловых ключей. Рассмотрим эту задачу на примере хэширования строк, так как объекты любой природы можно свести к строкам различной длины, просто преобразовав их к шестнадцатиричному виду байт, хранимых в оперативной памяти для данного объекта, или перекодировав эти байты в строки другим, более экономным способом.

Нам нужно сделать два шага: первым действием надо преобразовать строку в целое число, а вторым шагом – отобразить его в хэш-таблицу уже известным из предыдущей части лекции образом. Первое, что приходит в голову для преобразования строки в число – это просто просуммировать все байты этой строки, ведь каждый байт – это просто число от нуля до 255. Однако, такая хэш-функция будет неоптимальной по количеству коллизий: она будет давать совершенно одинаковые значения для строк, различающихся порядком следования одних и тех же букв.

Приведем пример неплохой хэш-функции для строк, которая реализована в одной из известных библиотек языка Си++:

```
unsigned hashkey( const char* str )
{
    unsigned hash = 0;
    while (*str) hash = (hash << 5) + hash + *str++;
    return hash;
}
```

Здесь операция  $\ll 5$  означает «битовый сдвиг левого операнда влево на 5 позиций» (или же – что то же самое – «целочисленное умножение на 32»).

Приведённая хэш-функция будет зависеть не только от букв строки, но и от порядка их следования. И не забудьте потом сделать второй шаг – взять остаток от деления возвращённого этой функцией числа на размер хэш-таблицы.

Изобретение эффективных по коллизиям хэш-функций – увлекательное занятие, важно лишь вовремя остановиться: если количество коллизий для простой хэш-функции вас устраивает, то незачем искать более совершенную хэш-функцию. Сэкономленное время работы программы не будет окупать затраты времени на изобретение новой хэш-функции и её программирование.

# Литература

- [1] Н. Н. Калиткин *Численные методы* // М., Наука, 1978.
- [2] R. S. Boyer, J. S. Moore *A fast string searching algorithm* // Comm. ACM – 1977 – **20**, 762–772.
- [3] M. Matsumoto, T. Nishimura *Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator.* // ACM Trans. on Modeling and Computer Simulations – 1998 – **8** (1), 3-30.
- [4] Г. М. Адельсон-Вельский, Е. М. Ландис *Один алгоритм организации информации* // Доклады АН СССР – 1962 – Т.**146**, № 2. – С. 263–266.
- [5] R. Bayer, E. McCreight *Organization and Maintenance of Large Ordered Indexes* // Acta Informatica – 1972 – **1** (3), 173–189.

# Список иллюстраций

1	Метод дихотомии . . . . .	10
2	Метод хорд . . . . .	11
3	Метод касательных . . . . .	12
4	«Плохой» выбор начального приближения . . . . .	12
5	Метод итераций: разные типы сходимости к корню уравнения . . . . .	13
6	Mersenne Twister: изменение состояния . . . . .	29
7	Пример интерполяции многочленом, проходящим через три заданные точки . . . . .	36
8	Вставка элемента . . . . .	43
9	Вставка-удаление в конце . . . . .	44
10	Хранение элементов в деке . . . . .	44
11	Хранение элементов в деке . . . . .	45
12	Хранение элементов в списке . . . . .	46
13	Вставка в список . . . . .	46
14	Вставка в список перед нужным элементом . . . . .	46
15	Удаление из списка элемента, следующего за указанным . . . . .	47
16	Удаление из списка указанного элемента . . . . .	47
17	Двусвязный список . . . . .	47
18	Двоичное дерево поиска . . . . .	48
19	Удаление внутреннего узла . . . . .	49
20	АВЛ-балансировка первого типа . . . . .	50
21	АВЛ-балансировка второго типа . . . . .	51
22	Страничная организация двоичного дерева . . . . .	53
23	Страничная организация двоичного дерева . . . . .	54
24	Вставка в В-дерево . . . . .	55
25	$B^+$ -дерево . . . . .	56
26	Коллизии в хэш-таблице и их разрешение методом цепочек . . . . .	58
27	Разрешение коллизий методом линейных проб при открытой адресации . . . . .	59

# Предметный указатель

- алгоритм
  - Бойера-Мура, 18
  - Верле, 34
- алгоритм, 18, 34
- ассоциативные контейнеры, *см.* контейнеры
- балансировка дерева, 50
- бинарное дерево поиска**, 48
- битовый вектор**, 57
- дека**, 43
- двоичный дополнительный код**, 7
- формула
  - Симпсона, 16
  - центральных прямоугольников, 15
  - левых прямоугольников, 14
  - правых прямоугольников, 14
  - трапеций, 15
- формула, 14–16
- хэш-таблица**, 57
- коллизии**, 58
- контейнеры**, 41
- квadrатурные формулы**, 14
- машинный ноль**, 9
- метод
  - Эйлера, 32
  - Эйлера модифицированный, 32
  - Эйлера с пересчётом, 33
  - Гаусса, 38
  - Монте-Карло, 24
  - Рунге-Кутты, 33
  - дихотомии, 10
  - хорд, 11
  - итераций, 13
  - касательных, 12
  - ломаных, 32
  - предиктор-корректор, 33
  - средней точки, 32
- метод, 10–13, 24, 32, 33, 38
- многочлены Лежандра, 36
- очередь**, 45
- определённые интегралы, 14
- представление вещественных чисел, 8
- разрешение коллизий, 59
- решение
  - ОДУ, 30
  - уравнений, 10
  - решение*, 10, 30
- системы линейных уравнений, 38
- сортировка
  - массива, 20
  - пузырьковая, 20
  - выбором, 21
  - quicksort, 22
- сортировка*, 20–22
- список**, 45
- стек**, 44
- таблица
  - стоп-символов, 19
  - суффиксов, 19
- таблица*, 19
- вектор (массив)**, 42
- вещественные величины
  - двойной точности, 8
  - одинарной точности, 8
- задача интерполяции, 35
- AVL-дерево**, 50
- B-дерево**, 54