Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Кубанский государственный технологический университет»

В. Н. Марков

# АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Утверждено методическим советом
ФГБОУ ВО «Кубанский государственный технологический университет»
в качестве учебного пособия

Краснодар
2022

УДК 004.02
ББК 32.973
М 26

Р е ц е н з е н т ы:

**О.А. Финько** – д-р техн. наук, проф. кафедры № 22
ФГКВОУ ВО «Краснодарское высшее военное
орденов Жукова и Октябрьской Революции
Краснознамённое училище имени генерала армии С. М. Штеменко»;
**А.В. Павлова** – д-р физ.-мат. наук, зав. кафедрой
математического моделирования ФГБОУ ВО
«Кубанский государственный университет»;
**В.М. Трофимов** – д-р физ.-мат. наук, проф. кафедры
информационных систем и программирования ФГБОУ ВО
«Кубанский государственный технологический университет»

М 26 **Марков В.Н.**

Алгоритмы и структуры данных: учеб. пособие/ В.Н. Марков. – Краснодар: Изд. ФГБОУ ВО «КубГТУ», 2022. – 207 с.

ISBN 978-5-8333-1091-5

Изложены основные абстрактные структуры данных и способы их представления на языке программирования высокого уровня C#. Рассмотрены алгоритмы обработки этих структур и пояснена оценка их вычислительной сложности. Даны рекомендации по выбору структур данных и алгоритмов их обработки. Приведены задачи для самостоятельного решения и выполнения лабораторных работ.

Учебное пособие предназначено для студентов направлений подготовки 09.03.03 Прикладная информатика и 09.03.04 Программная инженерия, а также для магистрантов и аспирантов, занимающихся разработкой и исследованием алгоритмов и программ.

**УДК 004.02**
**ББК 32.973**

ISBN 978-5-8333-1091-5

**Оглавление**

## Введение

Учебное пособие подготовлено в рамках дисциплины «Алгоритмы и структуры данных». Пособие содержит учебный материал по основным разделам рабочей программы курса: алгоритмам обработки данных, абстрактным структурам данных и обработке строк. Всем приводимым алгоритмам дана оценка вычислительной сложности. Учебное пособие включает подробное описание богатой коллекции исходных текстов программ на языке программирования C#, демонстрирующих излагаемые абстрактные типы данных и алгоритмы их обработки. Все главы снабжены перечнем задач для самостоятельного решения с целью приобретения студентами умений и навыков в вопросах программной реализации алгоритмов и абстрактных структур данных на лабораторных занятиях. Каждая глава заканчивается перечнем контрольных вопросов для самопроверки.

В первую главу вынесены вопросы рекурсивного программирования и рассмотрен ряд комбинаторных задач, которые выгодно решаются с помощью рекурсии и мемоизации. Вторая и третья главы посвящены эффективной обработке массивов, оптимизации кэширования данных и алгоритмам быстрого умножения матриц. Следующие три главы раскрывают тайны сортировки коллекций на примере шести наилучших, по мнению автора, алгоритмов.

Главы с 7 по 15 объединены общим признаком – ссылочными структурами данных: списками, деревьями и графами. Это самый важный и ёмкий раздел дисциплины. Он включает создание и использование связных списков, стеков, деков, очередей, барабанов, различных видов деревьев и графов, пирамид, дерамид. Большое внимание уделено балансировке деревьев и поиску минимальных путей на абстрактных графах и графах, привязанных к координатной сетке.

Третий раздел дисциплины вмещает главы с 16 по 18. В нём освещены вопросы хеширования текстовых данных, разработки хеш-функций и построения хеш-таблиц. Отдельный упор сделан на освоение алгоритмов поиска строк в тексте с использованием кольцевых хеш-функций, префикс-функций, таблиц стоп-символов и хороших суффиксов, а также на основе бора и его модификации в орграф. Последняя глава завершается рекомендациями по выбору и использованию структур данных и алгоритмов, изученных в рамках дисциплины.

Теоретический материал пособия иллюстрирован 123 поясняющими рисунками и 9 таблицами. Изложение прикладных вопросов подкреплено 113 комментированными исходными текстами программ и примерами. Лабораторные работы и практические занятия представлены 44 задачами и 188 контрольными вопросами.

# 1 Рекурсия

## 1.1 Понятие рекурсии

*Рекурсивная функция* – определение функции посредством её повторного вызова. *Рекурсия* (рекурсивный вызов) – вызов функции самой себя с новыми значениями параметров. Новые значения параметров, как правило, имеют меньшую величину и тем самым сокращают область построения решения или пространства поиска. При выполнении условия останова вместо рекурсивного вызова с помощью оператора `return` возвращается вычисленное значение. Это место в программе называется *дном* рекурсии. Начиная со своего дна, рекурсия выполняет цепочку возвратов вплоть до точки первого вызова. Оператор `return` выполняется столько раз, сколько раз был выполнен рекурсивный вызов.

Рекурсия может быть *оптимизированной* или *неоптимизированной*. Рекурсия называется *оптимизированной* тогда, когда все вычисления выполняются до рекурсивного вызова, а сам рекурсивный вызов является в определении функции последним (хвостовым). По этому признаку оптимизированную рекурсию ещё называют *хвостовой*. Многие современные компиляторы преобразуют хвостовую рекурсию в цикл. Достоинством оптимизированной рекурсии является экономия стековой памяти и, как следствие, надёжность вычислений.

Рассмотрим простейшую оптимизированную рекурсию на примере метода поэлементного вывода массива на экран.

```
static void Main()
{
    int[] m = { 12,13,14,15 };        // исходный массив
    P(m, 0);        // вызов рекурсивного метода: 12, 13, 14, 15,
    Console.ReadLine();
}
static void P(int[] m, int i)         // рекурсивный метод
{
    if (i == m.Length) return;        // дно рекурсии
    else
    {
        Console.Write(m[i] + ", ");   // вывод элемента
        P(m, i+1);                    // рекурсивный вызов
    }
}
```

В этом примере при каждом рекурсивном вызове мы инкрементировали индекс `i` от нуля до максимального значения.

Признаком *неоптимизированной* рекурсии являются вычисления, выполняемые после рекурсивного вызова. Такое название рекурсия имеет

потому, что перед рекурсивным вызовом надо запомнить адрес возврата и все те параметры, которые потребуются для выполнения вычислений после рекурсивного вызова. Это ведёт к неоптимальному расходу стековой памяти, что является недостатком. Бóльшая выразительная способность и гибкость программирования составляют достоинства.

Обратите внимание на реверсный порядок вывода элементов массива неоптимизированной рекурсией.

```
static void Main()
{
    int[] m = { 12,13,14,15 };       // исходный массив
    P(m, 0);        // вызов рекурсивного метода: 15, 14, 13, 12,
    Console.ReadLine();
}
static void P(int[] m, int i)        // рекурсивный метод
{
    if (i == m.Length) return;       // дно рекурсии
    else
    {
        P(m, i+1);                   // рекурсивный вызов
        Console.Write(m[i] + ", ");  // вывод элемента
    }
}
```

Полезные вычисления могут выполняться до рекурсивного вызова и после него.

```
static void Main()
{
    int[] m = { 12,13,14,15 };
    P(m, 0);                  // 12, 13, 14, 15, 15, 14, 13, 12,
    Console.ReadLine();
}
static void P(int[] m, int i)        // рекурсивный метод
{
    if (i == m.Length) return;       // дно рекурсии
    else
    {
        Console.Write(m[i] + ", ");  // прямой порядок вывода
        P(m, i+1);                   // рекурсивный вызов
        Console.Write(m[i] + ", ");  // обратный порядок вывода
    }
}
```

**Накопление результата в оптимизированной рекурсии**

В случае оптимизированной рекурсии результаты вычислений на каждом шаге передаются в рекурсивный вызов в качестве параметров функции. Для такой передачи в рекурсивной функции предусматривается хотя бы один дополнительный параметр.

Рассмотрим использование дополнительного параметра `acc` на примере вычисления факториала с помощью оптимизированной рекурсии. Значения параметров `acc1` и `n1` вычисляются до рекурсивного вызова. В аккумуляторе `acc1` накапливается произведение. Параметр `n1` – это число, на которое будет умножен аккумулятор при следующем рекурсивном вызове. Признаком достижения дна рекурсии является истинность условия `n==0`, при котором значение факториала находится в аккумуляторе `acc`.

Листинг 1 – Оптимизированная рекурсия

```
static void Main()
{
    Console.WriteLine(Fact(1,3));    // вычисляем 3!
    Console.ReadLine ();
}
static ulong Fact(ulong acc, uint n)  // рекурсивная функция
{
    if (n == 0) return acc;    // на дне результат в аккумуляторе
    else
    {
        ulong acc1 = acc * n;      // накопление произведения
        uint n1 = n - 1;           // следующий множитель
        ulong f = Fact(acc1,n1);   // рекурсивный вызов
        return f;                  // возвращаем факториал
    }
}
```

Накопление результата в первом параметре происходит во время цепочки рекурсивных вызовов. По достижении дна рекурсии мы определяем значение факториала – он находится в аккумуляторе. При цепочке возвратов, выполняемых операторами `return`, накопленный результат возвращается в точку первого вызова функции

```
Fact(1,3) → Fact(1×3,2) → Fact(3×2,1) → Fact(6×1,0)
    6     ←      6      ←      6      ←     6
```

Возврат из рекурсии называется *обратным ходом*. В нашем примере на обратном ходе результат передаётся без изменений, так как в хвостовой рекурсии после оператора `return` вычисления не производятся.

Следует заметить, что в таком развёрнутом и пошаговом виде рекурсивные вычисления не описывают. Обычно используют лаконичное определение рекурсии.

```
static ulong Fact(ulong acc, uint n)    // рекурсивная функция
{
    if (n == 0) return acc;             // результат в аккумуляторе
    else return Fact(acc * n, n - 1);   // возвращаем факториал
}
```

**Накопление результата в неоптимизированной рекурсии**

Рассмотрим вычисление факториала с помощью неоптимизированной рекурсии. В этом случае дополнительный параметр не потребуется.

Листинг 2 – Неоптимизированная рекурсия

```
static void Main()
{
    Console.WriteLine(Fact(3));
    Console.ReadKey();
}
static ulong Fact(uint n)
{
    if (n == 0) return 1;     // дно рекурсии: 0!=1
    else
    {
        uint n1 = n - 1;      // следующий множитель
        ulong f = Fact(n1);   // рекурсивный вызов
        ulong r = n * f;      // накопление произведения
        return r;             // возвращаем произведение
    }
}
```

На прямом ходе рекурсии мы получаем ряд чисел от `n` до `1`, играющих роль очередных множителей. На обратном ходе посредством умножения вычисляется факториал.

```
Fact(3) → Fact(2) → Fact(1) → Fact(0)
 3×2    ←   2×1   ←   1×1   ←   1
```

Лаконичное определение неоптимизированной рекурсии вычисления факториала.

```
static ulong Fact(uint n)
{
    if (n == 0) return 1; else return n * Fact(n - 1);
}
```

Для полноты картины представим итеративный цикл вычисления факториала.

```
uint n = 20;              // вычисляем 20!
ulong f = 1;              // аккумулятор
for (uint i = n; i > 0; i--)
    f = f * i;            // накопление произведения
Console.WriteLine(f);
```

**Использование стека и кучи для организации рекурсии**

Для работы программ в памяти компьютера выделяются две области памяти: *стек* (англ. stack) и *куча* (англ. heap).

Стек состоит из последовательности стековых кадров (англ. frame). Стековый кадр содержит локальные переменные метода/функции и аргу-

менты, с которыми его вызывали, а также адрес возврата из метода (адрес оператора, следующего после вызываемого метода). Вызванный метод считывает свои аргументы из стека. Для возврата в программу из стека выталкивается адрес возврата и ему передаётся управление. При этом вершину стека занимает предыдущий/родительский стековый фрейм, содержащий локальные переменные и аргументы метода, в который вернулась прграмма. Таким образом, программа возобновляет работу с места, находящегося после вызова метода. Каждому вычислительному потоку выделяется свой стек неизменного размера, как праивло 1 Мб.

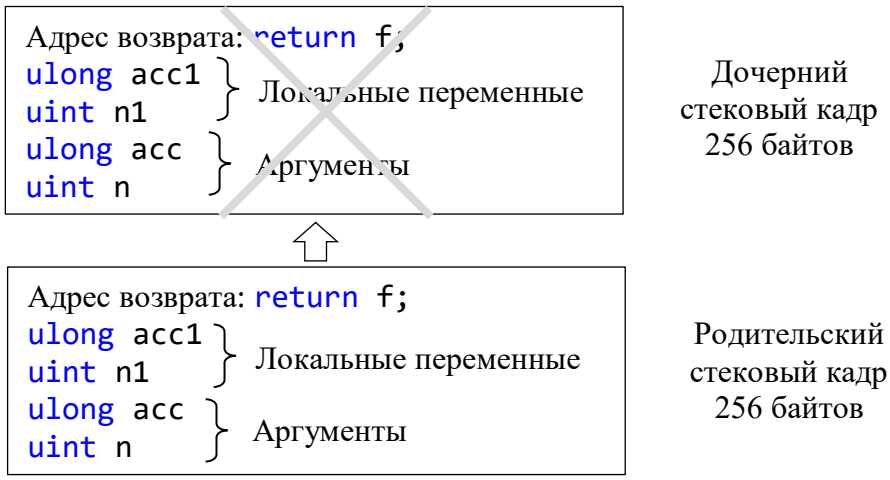



Рис. 1.1. Стек оптимизированной рекурсии (Листинг 1)

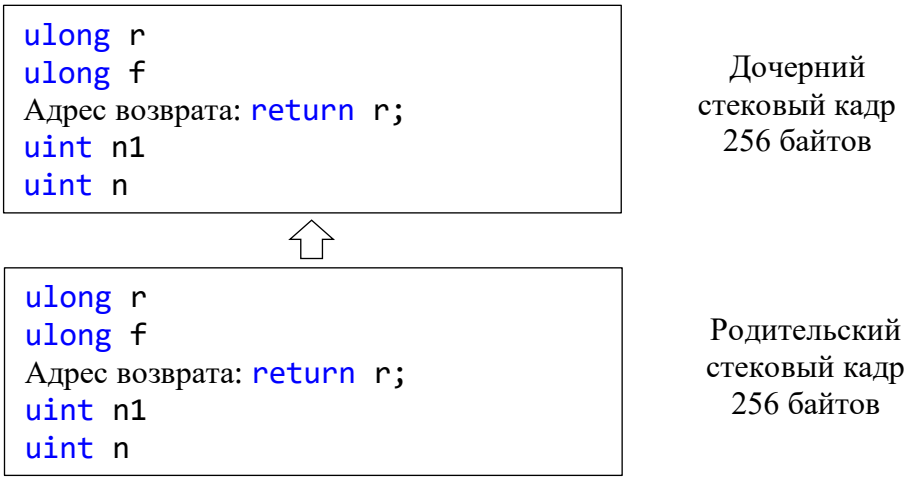



Рис. 1.2. Стек неоптимизированной рекурсии (Листинг 2)

В оптимизированной рекурсии после возврата никаких действий не происходит, следовательно, этот дочерний кадр можно вовсе не вставлять в стек (рис. 1.1), превращая рекурсию вы итеративный цикл.

В неоптимизированной рекурсии после возврата из дочернего кадра в родительский выполняются два оператора присваивания. Поэтому оба кадра в стеке должны присутствовать.

Куча общая для всех потоков приложения и используется для выделения памяти глобальным переменным и ссылочным структурам, допускающим изменение размера. Куча больше стека и может пополняться операционной системой и очищаться по запросу программы.

Как правило, неоптимизированная рекурсия требует много шагов для своего завершения. Каждый шаг – это вызов метода/функции и, как следствие, расход стековой памяти. Чем больше параметров передаётся в рекурсивном вызове, тем быстрее расходуется стек. В силу этого важно оценивать максимальную глубину используемой рекурсии, т.е. число вызовов метода/функции. В случае большой глубины может наступить истощение стека, что приведёт к аварийному завершению программы.

Заметьте, функция `Fact` возрастает настолько быстро, что значение выше 20! подсчитать в беззнаковых целых числах не получится. Число 21! окажется настолько большим, что превысит 64 бита и будет представлена только младшими 64 битами, старшие биты будут безвозвратно потеряны. Поэтому, с одной стороны, при вычислении факториала не надо заботиться о расходе стека: 20 вызовов стек не переполнят. С другой стороны, надо контролировать переполнение разрядной сетки, иначе для всех `n>20` будут получены неправильные результаты.

```
static ulong Fact(ulong n)
{
    try
    {
        if (n == 0) return 1;
        else
            return checked(n * Fact(n - 1));
    }
    catch (OverflowException)       // при переполнении
    {
        Console.WriteLine("Переполнение");
        return 0;
    }
}
```

Результат

```
20! = 2432902008176640000
Переполнение
21! = 0
```

На 128-разрядном процессоре можно вычислить максимальный факториал 34!, на 32-разрядном – всего 12!.

### 1.2 Возвратная рекурсия

*Возвратная* рекурсия использует более одного рекурсивного вызова. Это расширяет выразительные возможности рекурсии, но увеличивает время работы и размер требуемой памяти. С помощью возвратной рекурсии определяются функции, зависящие более чем от одного предыдущего значения.

В качестве примера возвратной рекурсии рассмотрим функцию вычисления членов ряда Фибоначчи `Fib(n)`. Напомним, что очередной член ряда Фибоначчи является суммой двух предыдущих

$$Fib(n) = Fib(n-1) + Fib(n-2).$$

Первые члены ряда Фибоначчи: 0, 1, 1, 2, 3, 5, 8, 13, 21. Иногда, по договорённости, ноль исключают из этого ряда. Так поступим и мы. В следующей программе индексация начинается с единицы.

```
static void Main()
{
    DateTime t1 = DateTime.Now;     // засекаем начало
    uint n = 38;                    // номер члена ряда
    ulong f = Fib(n);               // вызов рекурсивной функции
    DateTime t2 = DateTime.Now;     // засекаем конец
    Console.WriteLine($"Fib({n}) = {f}
({t2.Subtract(t1).TotalSeconds} сек.)");
    Console.ReadKey();
}
static ulong Fib(uint n)
{
    try
    {
        if (n < 3) return 1;        // первый и второй члены равны 1
        else
            return checked((ulong)(Fib(n - 1) + Fib(n - 2)));
    }
    catch (OverflowException)
    {
        Console.Write("Переполнение");
        return 0;
    }
}
```

Программа выводит на экран значение $n$-го члена ряда Фибоначчи и время его вычисления на процессоре AMD Ryzen 5 3,6 ГГц.

```
Fib(38) = 39088169 (1,1871661 сек.)
Fib(39) = 63245986 (1,8307231 сек.)
Fib(40) = 102334155 (2,9333761 сек.)
```

Такое большое время связано с многократным дублированием одних и тех же вычислений. К примеру, для получения шестого члена ряда Фибоначчи (рис. 1.3) надо вычислить величину `Fib(5)` – 1 раз, `Fib(4)` – 2 раза, `Fib(3)` – 3 раза, `Fib(2)` – 5 раз, `Fib(1)` – 3 раза.

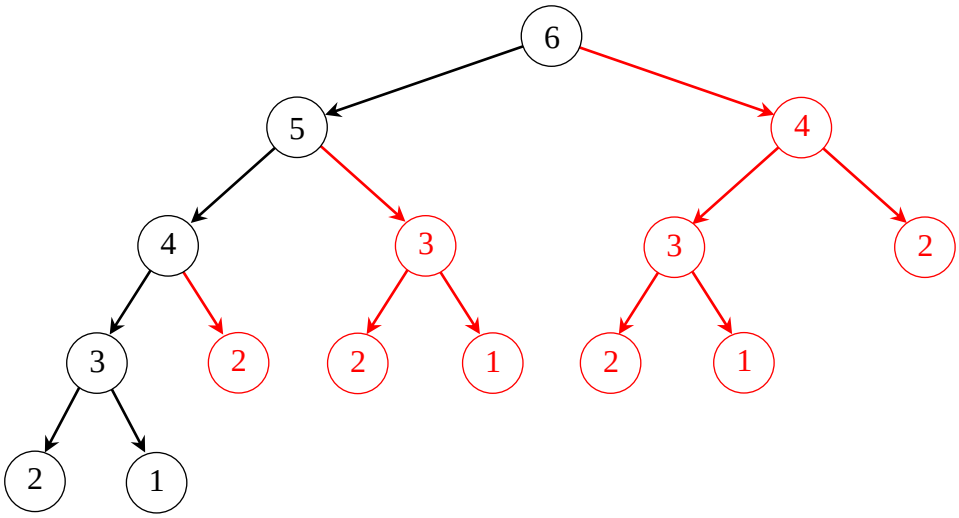



Рис. 1.3. Дерево вызовов при вычислении `Fib(6)`

В некоторых случаях возвратную рекурсию удаётся преобразовать в рекурсию с одним вызовом или даже в итеративный цикл, например, ряд Фибоначчи можно строить рекурсивным преобразованием кортежа $(fib_{i-1}, fib_i, i)$ по правилу

$$(a, b, i) \rightarrow (b, a + b, i + 1).$$

Начальный кортеж (1,1,2). Первые два параметра являются первыми двумя членами ряда Фибоначчи. Третий параметр – индекс члена ряда, занимающего место второго параметра. Последовательно преобразуя кортежи по вышеуказанному правилу, получим ряд

$$(1,1,2) \rightarrow (1,2,3) \rightarrow (2,3,4) \rightarrow (3,5,5) \rightarrow (5,8,6) \rightarrow (8,13,7) \rightarrow \ldots,$$

в котором кортеж (8,13,7) содержит седьмой член ряда, равный 13.

Когда надо найти член ряда по его индексу, например, равному семи, то третий параметр удобно задать искомым индексом и преобразовать кортежи, декрементируя индекс до значения 2. Тогда цепочка

$$(1,1,7) \rightarrow (1,2,6) \rightarrow (2,3,5) \rightarrow (3,5,4) \rightarrow (5,8,3) \rightarrow (8,13,2)$$

закончится кортежем (8,13,2), во втором параметре которого лежит искомый седьмой член ряда, равный 13.

```
static void Main()
{
```

```
    DateTime t1 = DateTime.Now;     // засекаем начало
    uint n = 40;                    // номер члена ряда
    ulong f = Fib(n);               // вызов рекурсивной функции
    DateTime t2 = DateTime.Now;     // засекаем конец
    Console.WriteLine($"Fib({n}) = {f}
({t2.Subtract(t1).TotalSeconds} сек.)");
    Console.ReadKey();
}
static ulong Fib(uint n)      // функция вычисления n-го члена ряда
{
    if (n < 3) return 1;      // первый и второй члены равны 1
    else
        return Fi(1, 1, n);  // вызов функции преобразования кортежа
}
static ulong Fi(ulong a, ulong b, uint n)
{
    try
    {
        if (n == 2) return b;       // дно рекурсии
        else return Fi(b, checked(a + b), n - 1);
    }
    catch (OverflowException)
    {
        Console.Write("Переполнение");
        return 0;
    }
}
```

Время работы

```
Fib(38) = 39088169 (0 сек.)
Fib(39) = 63245986 (0 сек.)
Fib(40) = 102334155 (0 сек.)
Fib(93) = 12200160415121876738 (0.001 сек.)
```

Обратите внимание, `Fib(93)` – максимальный член ряда, представимый в 64-разрядной сетке.

### 1.3 Мемоизация

Существует способ ускорения рекурсии. Он основан на исключении повторных вычислений посредством *мемоизации*: запоминания результатов промежуточных вычислений. Мемоизацию можно использовать для ускорения не только рекурсии, но и любых других вычислительных процессов.

Для мемоизации вычисления членов ряда Фибоначчи добавим в программу одномерный массив `f` длиной 100, инициализированный нулями. На самом деле 100 элементов слишком много, так как максимальный

член ряда, не превышающий 64 разряда, имеет индекс 93. В массиве `f` будем сохранять значения всех вычисленных членов ряда по их индексу за исключением первых двух, которые по определению равны единице. Перед вычислением *n*-го члена ряда `Fib(n)` мы будем обращаться сначала к элементу массива `f[n]` в надежде, что он там есть. Равенство `f[n]` нулю, является признаком отсутствия `Fib(n)` в массиве, что вынудит нас вычислять его в виде суммы двух предыдущих членов ряда. Если же `f[n]` больше нуля, то вычислять `Fib(n)` не надо, так как он равен значению элемента массива `f[n]`.

```
static void Main()
{
    DateTime t1 = DateTime.Now;     // засекаем начало
    uint n = 93;                    // номер члена ряда
    ulong f = Fib(n);               // вызов рекурсивной функции
    DateTime t2 = DateTime.Now;     // засекаем конец
    Console.WriteLine($"Fib({n}) = {f}
({t2.Subtract(t1).TotalSeconds} c.)");
    Console.ReadKey();
}
public static ulong[] f = new ulong[100];  // хранилище
static ulong Fib(uint n)
{
    try
    {
        if (n < 3) return 1;        // первые два члена ряда равны 1
        else
        {
            if (f[n] > 0) return f[n];  // читаем Fib(n) из массива
            else
            {
                ulong u = checked(Fib(n - 1) + Fib(n - 2));
                f[n] = u;           // запоминаем Fib(n) в массиве
                return u;
            }
        }
    }
    catch (OverflowException)
    {
        Console.Write("Переполнение");
        return 0;
    }
}
```

Время работы

```
Fib(93) = 12200160415121876738 (0 c.)
```

Обратите внимание на то, что рекурсия осталась неоптимизированной и возвратной, но использование мемоизации резко повысило скорость вычислений, практически сравняв её со скоростью выполнения итеративного цикла.

### 1.4 Комбинаторика

До сих пор мы рассматривали рекурсивные решения задач, которые легко можно решить итеративным циклом. Эти задачи были использованы в показательных целях. Однако существует ряд задач, не имеющих простого и очевидного решения с помощью итеративных циклов, но достаточно легко решаемых рекурсивно.

К таким задачам относятся комбинаторные задачи. Рассмотрим рекурсивный алгоритм получения всех перестановок элементов списка `w`, заданного перечислением элементов, например `{1,2,3}`

```
List<int> w = new List<int>() { 1, 2, 3 };
```

Требуется получить в списке `p` все его перестановки по одной: 123, 132, 213, 231, 312, 321. Число перестановок равно $n!$, $n$ – длина списка `w`.

**Рекурсивный алгоритм получения всех перестановок**

1. Передадим функции `Per` (англ. permutation – перестановка) в качестве параметров исходный список `w` и пустой список `p`, в копиях которого будем строить все варианты перестановок.

2. В цикле от 1 до `n` будем делать копии списка `w` (назовём копии именем `w1`) и вырезать из них по одному элементу на каждом витке. В результате получим вырезанный элемент и остаток `w1`, например, на первом витке мы получим число 1 и остаток `w1`={2,3}, на втором витке – число 2 и остаток `w1`={1,3}, на третьем витке – число 3 и остаток `w1`={1,2}.

3. В этом же теле цикла будем делать копии списка `p` (назовём их `p1`), в конец каждой копии будем добавлять вырезанный из списка `w` элемент. Поэтому на первом витке получим `p1`={1} и `w1`={2,3}, на втором витке `p1`={2} и `w1`={1,3}, на третьем витке `p1`={3} и `w1`={1,2}.

4. Самый главный шаг – в конце тела цикла делаем рекурсивный вызов функции `Per` с параметрами `w1` и `p1`, например, когда параметры `p`={1} и `w`={2,3}, то функция `Per` в цикле станет вырезать из списка {2,3} по одному элементу, добавляя их к списку {1}. Таким образом, на этом первом витке первого рекурсивного вызова мы получим `p1`={1,2} и `w1`={3}, на втором витке получим `p1`={1,3} и `w1`={2}.

5. Когда список `w` станет пустым – список `p` будет содержать очередной вариант перестановки.

6. Рекурсия остановится, когда во всех витках цикла индекс `i` пробежит все значения.

Дерево построения всех перестановок элементов списка (рис. 1.4) показывает перетекание элементов из списка `w` в список `p` по одному.

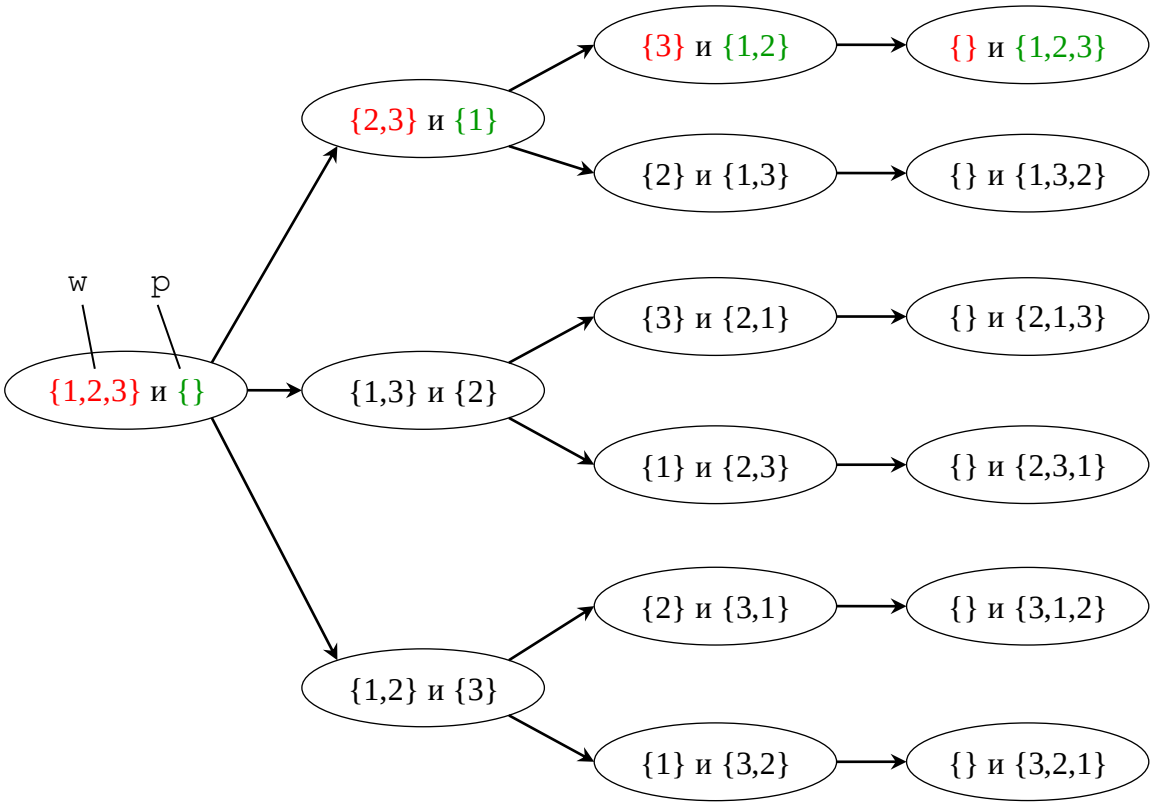



Рис. 1.4. Дерево рекурсивного построения всех перестановок элементов списка {1, 2, 3}

Получение всех перестановок оптимизированной рекурсией интенсивно расходует кучу, но экономит стек. Это оправдано, так как куча большая и всегда может быть пополнена операционной системой.

```
static void Main(string[] args)
{
    List<int> w = new List<int>() { 1, 2, 3 };  // исходный список
    List<int> p = new List<int>();              // список перестановок
    Per(w, p);                              // вызов рекурсивной функции
    Console.ReadKey();
}
static void Per(List<int> w, List<int> p)
{
    if (w.Count > 0)              // условие продолжения рекурсии
    {
```

```
        for (int i = 0; i < w.Count; i++)
        {
            List<int> w1 = new List<int>();  // новый список w1
            w1 = w.GetRange(0, w.Count);  // копируем w → w1
            int e = w1[i];                // получаем i-й элемент
            w1.RemoveAt(i);               // удаляем i-й элемент
            List<int> p1 = new List<int>();  // новый список p1
            p1 = p.GetRange(0, p.Count);  // копируем p → p1
            p1.Add(e);                    // добавляем элемент в p1
            Per(w1, p1);                  // рекурсивный вызов
        }
    }
    else                                  // дно рекурсии
    {
        Console.WriteLine(string.Join(" ",p)); // одна перестановка
    }
}
```

Результат – все перестановки

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

Неоптимизированная рекурсия экономит кучу, но интенсивно расходует стек. Это опасно, так как стек может переполниться, что вызовет аварийное завершение программы. Ещё одним отличием неоптимизированной рекурсии от оптимизированной является отказ от использования копий списков `w1` и `p1`. Перед рекурсивным вызовом списки `w` и `p` изменяются и передаются параметрами в рекурсивный вызов. После возврата из рекурсивного вызова списки `w` и `p` восстанавливаются.

```
static void Main(string[] args)
{
    List<int> w = new List<int>() { 1, 2, 3 }; // исходный список
    List<int> p = new List<int>();    // здесь один вариант решения
    Per(w, p);                        // вызов рекурсивной функции
    Console.ReadKey();
}
static void Per(List<int> w, List<int> p)
{
    if (w.Count > 0)                // условие продолжения рекурсии
    {
        for (int i = 0; i < w.Count; i++)
        {
            int e = w[i];             // получаем i-й элемент из w
```

```
            w.RemoveAt(i);              // удаляем i-й элемент из w
            p.Insert(p.Count, e);       // добавляем элемент в p
            Per(w, p);                  // рекурсивный вызов
            w.Insert(i, e);             // восстанавливаем w
            p.RemoveAt(p.Count - 1);    // восстанавливаем p
        }
    }
    else                                // дно рекурсии
    {
        Console.WriteLine(string.Join(" ",p)); // одна перестановка
    }
}
```

### 1.5 Ханойские башни

Задача о ханойских башнях эффектно демонстрирует выразительную мощь неоптимизированной рекурсии.

Даны три стержня *A*, *B*, *C* и, например, четыре диска с отверстиями. На стержень *A* насажены все четыре диска пирамидкой: сверху диск наименьшего размера, снизу – наибольшего. Требуется переместить все диски со стержня *A* на стержень *C*, используя стержень *B* как вспомогательный. Больший диск помещать сверху меньшего нельзя.

Идея решения заключается в уменьшении размерности задачи в каждом рекурсивном вызове. Для того чтобы переместить четыре диска, надо найти способ перемещения верхних трёх дисков со стержня *A* на стержень *B*, а оставшийся нижний диск переместить со стержня *A* на стержень *C* труда не составит. Эта же логика действует при перемещении трёх дисков: надо найти способ перемещения двух верхних дисков, а третий диск переместить легко. С каждым рекурсивным вызовом число перемещаемых дисков уменьшается на единицу и на дне рекурсии число дисков станет равным нулю.

Функция `H(n,"A","B","C")` описывает перемещение `n` дисков со стержня `"A"`, указанного вторым параметром, на стержень `"C"`, указанный четвёртым параметром, используя стержень `"B"`, указанный третьим параметром, в качестве вспомогательного.

```
static void Main()
{
    H(4,"A","B","C");
    Console.ReadLine();
}
static void H(int n, string x, string y, string z)
{
    if (n == 0) return;     // дно рекурсии
    H(n - 1,x,z,y); // перенос n-1 дисков с x на y, z – вспомогат.
```

```
    Console.WriteLine($"диск {n}: {x} -> {z}"); // один диск x -> z
    H(n - 1,y,x,z); // перенос n-1 дисков с y на z, x – вспомогат.
}
```

Предложенная рекурсия позволяет перемещать любое количество дисков с вычислительной сложностью $O(2^n)$.

### 1.6 Управление размером стека

Для борьбы с переполнением стека при вызове неоптимизированной рекурсии можно увеличить размер отводимого рекурсии стека. Для этого рекурсивную функцию надо вызывать в отдельном вычислительном потоке, при запуске которого следует указать необходимый для этого потока размер стека. Размер стека по умолчанию 1 Мб. Поэтому указать надо бóльшую величину.

Рассмотрим тривиальный пример неоптимизированной рекурсии `Rec(N)`, подчитывающей количество единиц в некотором числе `N` путём прибавления единицы к значению функции после каждого рекурсивного вызова, уменьшающего исходное число на 1.

```
static void Main()
{
    Console.WriteLine(Rec(10000));
    Console.ReadKey();
}
static int Rec(int n)
{
    if (n == 0) return 0;  // дно рекурсии
    return Rec(n - 1) + 1; // неоптимизированная рекурсия
}
```

При N = 10000 размера стека по умолчанию 1 Мб вполне хватает. Однако при N = 20000 происходит переполение стека. Давайте вызовем нашу функцию `Rec(20000)` в отдельном вычислительном потоке, которому отведём стек размером, например, 2 Мб.

```
static void Main()
{
    Thread T = new Thread(F,1<<21); // 2Мб
    T.Start();
    Console.ReadKey();
}
public static void F()
{
    Console.WriteLine(Rec(20000);
}
static int Rec(int n)
{
```

```
    if (n == 0) return 0; // дно рекурсии
    return Rec(n - 1) + 1; // неоптимизированная рекурсия
}
```

Результат 20000 будет выведен на экран.

Описанный способ увеличения размера стека действенен, но увлекаться им не стоит, так как он зависит от глубины рекурсии, верхнюю границу которой не для всех задач можно оценить. Поэтому надо научиться писать такие рекурсии, для которых не надо бесконечно увеличивать размер стека.

**Задача 1.** На основании рекурсии, описанной в подразделе 1.4, разработать рекурсивный алгоритм и функцию поиска всех размещений элементов списка. Например, список {1,2,3} имеет 15 вариантов размещения элементов. По одному: {1}, {2}, {3}; по двум: {1,2}, {1,3}, {2,1}, {2,3}, {3,1}, {3,2} и по трём элементам: {1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2}, {3,2,1}. Эти 15 вариантов могут быть выведены на экран в произвольном порядке. Получить экспериментальную оценку вычислительной сложности.

**Задача 2.** Получить экспериментальную оценку вычислительной сложности функции `H(int n, string x, string y, string z)`, описанной в подразделе 1.5. Сравнить с теоретической оценкой.

**Контрольные вопросы**

1. Раскройте понятие оптимизированной рекурсии.
2. Что такое хвостовая рекурсия?
3. В чём заключается отличие оптимизированной и неоптимизированной рекурсии?
4. За счёт чего происходит накопление результата в рекурсии?
5. Что такое обратный ход рекурсии?
6. Когда рекурсия достигает своего дна?
7. Раскройте понятие возвратной рекурсии.
8. Какова роль стека и кучи в организации рекурсии?
9. Для чего применяется мемоизация рекурсии?

# 2 Представление и обработка массивов

## 2.1 Размещение массивов в оперативной памяти

Массив располагается в куче (англ. heap). Куча представляет собой непрерывную область памяти, с помощью которой реализуется динамически распределяемая память приложения. Куча разделяется на занятые и свободные области различного размера. При запуске процесса операционная система выделяет ему память для размещения кучи. В дальнейшем память для кучи может быть увеличена динамически: по запросу программы.

Массивы бывают двух видов: статические и динамические. При инициализации *статического* массива в куче выделяется область памяти неизменяемого размера, после чего запоминается адрес первого элемента. При инициализации *динамического* массива в куче выделяется область памяти определённой ёмкости (англ. capacity), которая больше объявленного размера массива, после чего запоминается адрес первого элемента. Пока реальный размер (англ. size) динамического массива меньше выделенной ему ёмкости, размер массива можно быстро увеличивать, выделяя дополнительные ячейки из свободной ёмкости. Когда размер массива сравняется с ёмкостью, то для дальнейшего увеличения размера массива операционная система выполняет ряд действий:

1. В куче ищется и выделяется новая область памяти, превышающая размер массива. Если в куче нет области нужного объёма, то она выделяется в свободной оперативной памяти, расширяя тем самым кучу.
2. Содержимое массива копируется во вновь выделенную область памяти. При большом размере массива на это копирование уйдёт много времени!
3. Сохраняется новый адрес первого элемента, так как элементы массива мы скопировали в новые адреса.
4. Выполняется команда освобождения предыдущей ёмкости памяти массива. Если язык программирования поддерживает автоматическое управление памятью, то возвращение объёма памяти операционной системе выполняется сборщиком мусора автоматически.

Увеличение размера динамического массива выполняется медленно в случае необходимости увеличения его ёмкости (рис. 2.1). Поэтому ёмкость увеличивается сразу на значительную величину для того, чтобы массив копировался на новое место как можно реже. Новые элементы добавляются в конец массива. Как добавлять новые элементы в середину или начало массива будет объяснено далее.

Элементами массива могут быть либо непосредственные значения, либо ссылки. При создании массива значений его элементам присваива-

ются значения по умолчанию, например, значением по умолчанию для всех числовых типов данных является ноль, для типа `bool` – значение `false`. При создании массива ссылок его элементам присваиваются нулевые ссылки `null`.

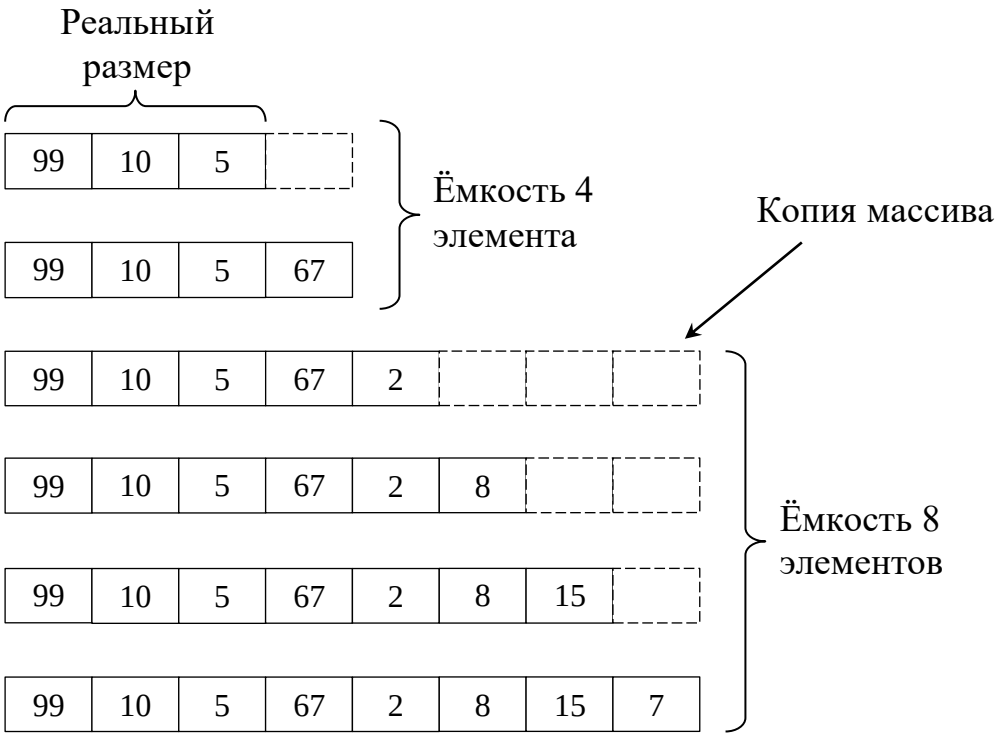



Рис. 2.1. Рост ёмкости и размера динамического массива

Размеры элементов массива значений одинаковые. Такие массивы, как правило, хранят символы, числа, булевы значения. С другой стороны, элементами массива могут быть строки разной длины или другие массивы. В этом случае такие массивы хранят ссылки на строки или массивы. Размеры ссылок одинаковые.

Платформа .NET позволяет создавать массивы различных типов и размеров, так как в основе всех типов этой платформы лежит базовый класс `System.Object`. В этом случае переменные различных типов хранятся в массиве как объекты `object`.

```
object[] m = { 'z', 123, "abc", 9.999, false };
Console.WriteLine(string.Join(" ",m));  // z 123 abc 9,999 False
```

Каждый элемент массива имеет свой уникальный индекс, по которому можно быстро вычислить адрес данного элемента в памяти. Достоинством массивов является быстрый доступ к элементам по чтению и записи. В языках программирования индексация элементов по умолчанию начинается с нуля. Язык C# позволяет создавать массивы с произвольны-

ми нижними границами индексации каждого измерения. Ниже представлено создание одномерного массива `m` длиной 4, индексация которого начинается с -6.

```
int[] len = new int[] {4};  // длина первого измерения
int[] lowerbound = new int[] {-6};  // нижняя граница индексации
Array m = Array.CreateInstance(typeof(int), len, lowerbound);
m.SetValue(99, -6);  // m[-6] = 99
Console.WriteLine(m.GetValue(-6));
```

**Размещение одномерных массивов в памяти**

Рассмотрим три одномерных массива.

```
int[] w = { 99, 10, 5, 67, 2 };     // 5 четырёхбайтовых чисел
short[] m = { 99, 10, 5, 67, 2 };   // 5 двухбайтовых чисел
string[] z = { "qwe", "r", "ty" }; // 3 строки
```

Размещение этих массивов в памяти зависит от типов элементов. Элементы одномерных массивов располагаются в смежных ячейках памяти друг за другом (рис. 2.2). Массив `w` имеет адрес 100, массив `m` – адрес 200, а массив `z` – адрес 300. При этом массивы w и m являются массивами *значений*, а массив `z` является массивом *ссылок* на строки.

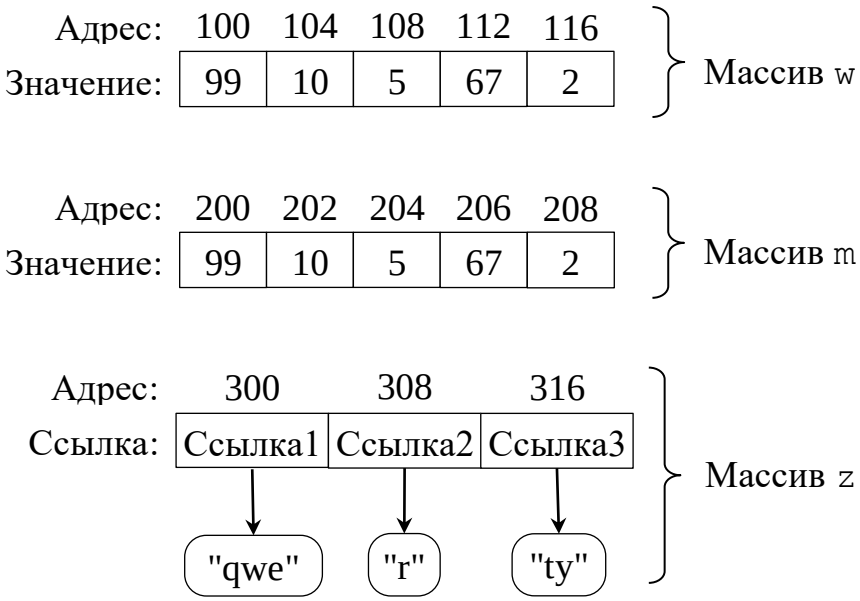



Рис. 2.2. Размещение одномерных массивов в памяти

Адрес *i*-го элемента *Addr*[*i*] массива `w` легко получить, зная адрес первого элемента *Addr*[0] и размер элемента массива *SizeElem* = 4 байта,

$$Addr[i] = Addr[0] + SizeElem \cdot i,$$

например, элемент `w[3]` = 67 расположен по адресу

$$Addr[3] = 100 + 4 \cdot 3 = 112.$$

**Размещение многомерных массивов в памяти**

В качестве примера многомерных массивов рассмотрим двумерный и трёхмерный массивы

```
int[,] a = { { 1, 2, 3 }, { 4, 5, 6 } };          // массив 2×3
int[,,] b = { { { 1, 2, 3 }, { 4, 5, 6 } },
              { { 7, 8, 9 }, { 10, 11, 12 } } };  // массив 2×2×3
```

Элементы двумерных массивов располагаются в памяти в смежных ячейках друг за другом (рис. 2.3): вначале следуют элементы первой строки, потом – второй и т.д. Доступ к элементам двумерного массива `a[i,j]` осуществляется по вычисляемому адресу

$$Addr[i,j] = Addr[0,0] + SizeElem \cdot J \cdot i + SizeElem \cdot j,$$

где *Addr*[0,0] – адрес первого элемента `a[0,0]`;

*SizeElem* – размер элемента массива, в нашем случае четыре байта;

*J* – количество элементов в строке, в нашем случае *J*=3.

На рисунках 2.3 и 2.4 показано, что массив `a` имеет адрес 100, массив `b` имеет адрес 200. На самом деле адреса массивов назначаются при загрузке программы в память, а до этого они неизвестны.

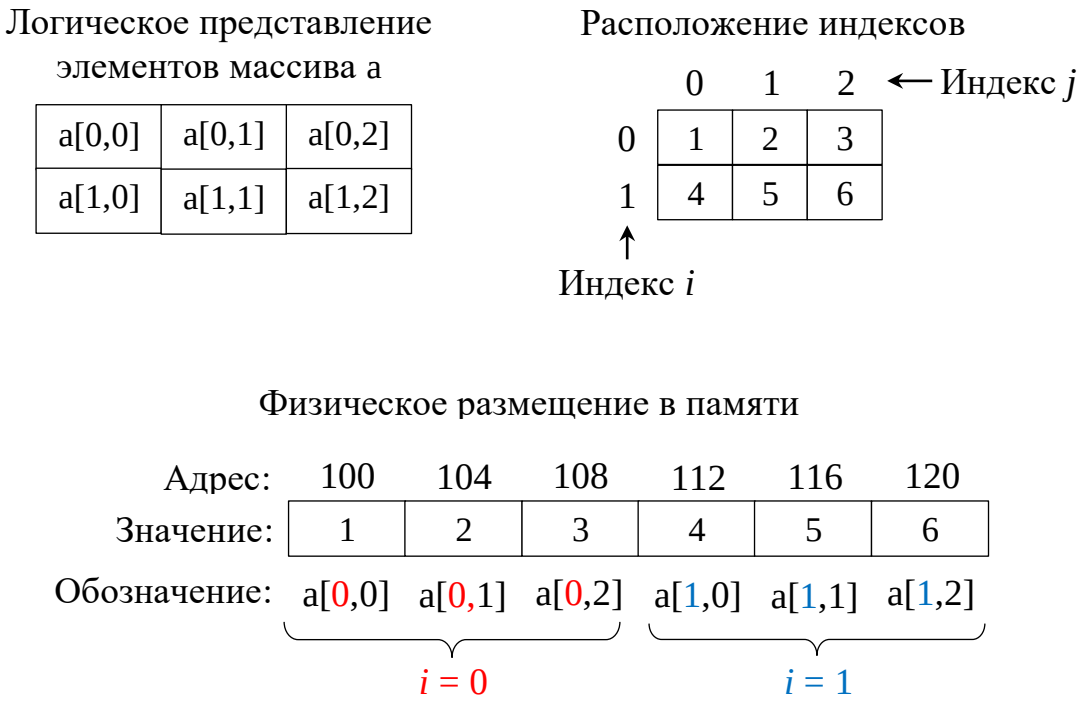
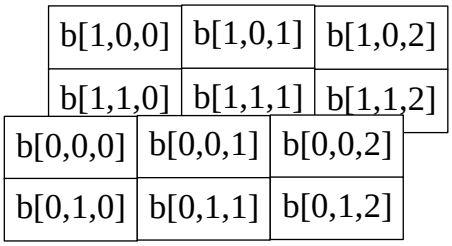
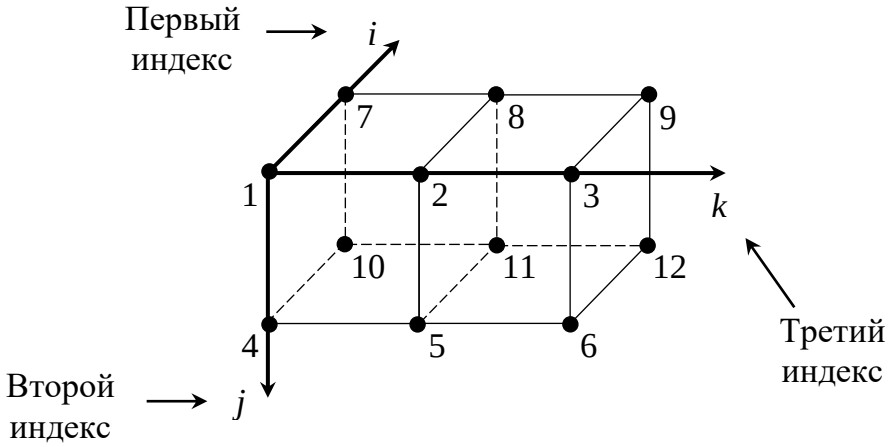



Рис. 2.3. Размещение двумерного массива a в памяти

Логическое представление элементов массива b




Расположение и направление осей




Физическое размещение в памяти

| Адрес: | 200 | 204 | 208 | 212 | 216 | 220 | 224 | 228 |
|---|---|---|---|---|---|---|---|---|
| Значение: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Обозначение: | b[0,0,0] | b[0,0,1] | b[0,0,2] | b[0,1,0] | b[0,1,1] | b[0,1,2] | b[1,0,0] | b[1,0,1] |

Рис. 2.4. Размещение трёхмерного массива в памяти

Элементы трёхмерных массивов располагаются в памяти в смежных ячейках друг за другом: вначале следуют элементы первой двумерной компоненты – передней поверхности, её элементы имеют первый индекс $i$=0, затем – второй двумерной компоненты, имеющей индекс $i$=1, и т.д. Доступ к элементам трёхмерного массива осуществляется по адресу

$$Addr[i,j,k] = Addr + SizeElem \cdot J \cdot K \cdot i + SizeElem \cdot K \cdot j + SizeElem \cdot k,$$

где *Addr* – адрес первого элемента `b[0,0,0]`;

*SizeElem* – размер элемента массива, в нашем случае четыре байта;

*J* – количество строк в передней гране, т.е. размерность *J*, в нашем случае *J*=2;

*K* – количество столбцов в передней гране, в нашем случае *K*=3.

### 2.2 Удаление элементов из динамического массива

Удаление и вставка элементов являются долгими операции и могут выполняться только над динамическими массивами. Удаление одного элемента из одномерного массива выполняется путём смыкания массива в позиции удаляемого элемента за счёт перемещения правой части массива влево на столько байтов, сколько занимает удаляемый элемент.

Например, для удаления элемента 5, находящегося во второй позиции одномерного массива `w` (рис. 2.5), надо произвести копирование всех элементов, начиная с третьего, находящихся справа от него в левые смежные ячейки. При этом размер массива уменьшается на количество байтов, занимаемое одним элементом, а размерность массива уменьшается на один элемент. В данном примере свободная ёмкость массива увеличилась на четыре байта, занимающие адреса от 116 по 119.

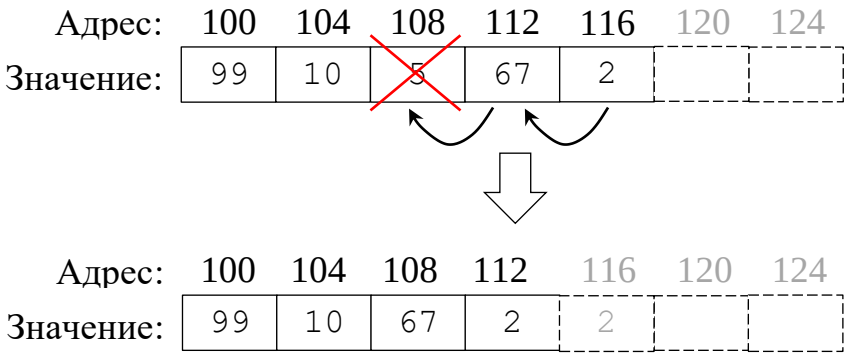



Рис. 2.5. Удаление элемента из динамического массива

Чем ближе удаляемый элемент к началу массива, тем дольше выполняется операция удаления. В самом худшем случае удаление элемента из нулевой позиции повлечёт копирование всех остальных элементов на одну позицию влево.

Удаление одного элемента из многомерного массива нарушит логическую целостность массива, так как он будет содержать строки и столбцы различной длины, что недопустимо. В двумерном массиве можно удалить целиком одномерную компоненту: строку или столбец. В трёхмерном массиве можно удалить не менее чем одну двумерную компоненту.

### 2.3 Вставка элементов в динамический массив

Вставка нового элемента в одномерный массив выполняется путём размыкания массива в позиции вставляемого элемента за счёт перемещения правой части массива вправо. Новый элемент записывается в освободившуюся позицию, например, для вставки числа 6 во вторую позицию одномерного массива `w` (рис. 2.6) надо произвести копирование всех элементов в правые ячейки, начиная с последней позиции и до второй, и не забыть записать новый элемент во вторую позицию. При этом размер массива увеличится на количество байт, занимаемое одним элементом, а размерность массива увеличивается на один элемент. В данном примере из свободной ёмкости массива программа взяла четыре байта, расположенные по адресам от 120 до 123 включительно.

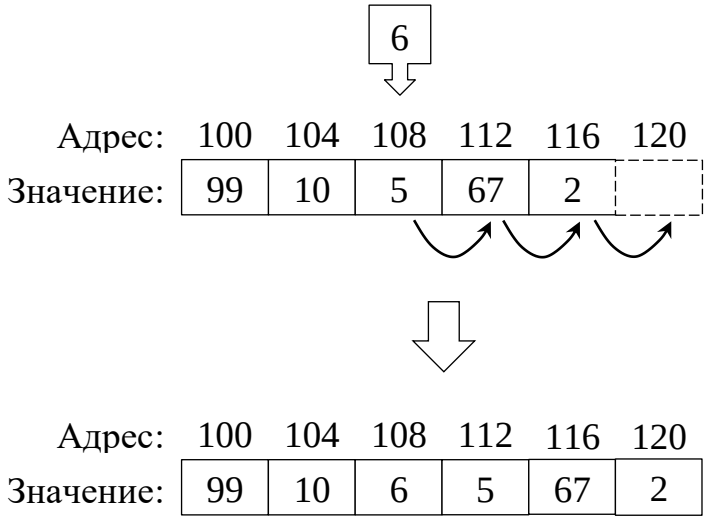



Рис. 2.6. Вставка элемента в динамический массив

Чем ближе к началу массива располагается позиция вставляемого элемента, тем дольше выполняется операция вставки. В самом худшем случае вставка элемента в нулевую позицию массива повлечёт копирование всех остальных элементов на одну позицию вправо.

Вставка элемента в многомерный массив нарушит логическую целостность массива, так как он будет содержать строки и столбцы различной длины, что недопустимо. В двумерный массив можно вставить целиком одномерную компоненту: строку или столбец. В трёхмерный массив можно вставить не менее чем одну двумерную компоненту.

## 2.4 Последовательный поиск в неупорядоченном массиве

Поиск элемента в неупорядоченном массиве выполняется последовательно от первого элемента к последнему до тех пор, пока не будет найден искомый элемент или проверен последний. В случае удачи функция поиска возвращает индекс найденного элемента, иначе возвращает значение -1.

Если массив содержит дубликаты искомого элемента, то функцию поиска можно реализовать так, чтобы она возвратила индексы всех дубликатов.

Рассмотрим вычислительную сложность последовательного поиска в неупорядоченном массиве, содержащем $N$ уникальных элементов.

**Анализ наихудшего случая**

В наихудшем (англ. worst) случае искомый элемент последний или его нет вовсе. Тогда алгоритму придётся выполнить $N$ сравнений. Следовательно, верхняя граница вычислительной сложности $W(N) = N$.

**Анализ среднего случая**

Для оценки среднего (англ. average) случая необходимо суммировать сложность нахождения искомого числа, находящегося во всех возможных позициях, и сумму разделить на величину $N$. Тем самым мы получим среднюю сложность $A(N)$. Наши рассуждения таковы. Если искомый элемент стоит на первой позиции, то он будет найден за одно сравнение. Если на второй позиции – за два сравнения и т.д. Следовательно, нам надо сложить числа от 1 до $N$ и сумму поделить на $N$

$$A(N) = \frac{1}{N}(1 + 2 + \cdots + N) = \frac{1}{N} \cdot \frac{N(1+N)}{2} = \frac{N+1}{2}.$$

Сумма арифметической прогрессии $(1 + 2 + \cdots + N)$ была заменена известным выражением $\frac{N(1+N)}{2}$.

## 2.5 Двоичный поиск в упорядоченном массиве

Если массив упорядочен, то последовательный поиск в нём будет неэффективен, так как не использует информацию о порядке следования элементов. Для поиска в таких массивах эффективен двоичный (бинарный) поиск. Он заключается в следующем. Пусть одномерный массив упорядочен по возрастанию. На первом шаге искомый элемент сравнивается с элементом, занимающим позицию в середине массива. При сравнении возможны три варианта: либо элементы равны, либо искомый элемент меньше, либо – больше. Если элементы равны, то поиск завершён. Если искомый элемент меньше, то, очевидно, его надо искать в левой половине

массива. Если больше – в правой. Заметьте, что средний элемент не входит в левую и правую половины массива. На втором шаге указанные операции повторяются для той половины массива, которая выбрана на первом шаге. Продолжая действовать таким способом, на каждом шаге мы уменьшаем область поиска в два раза.

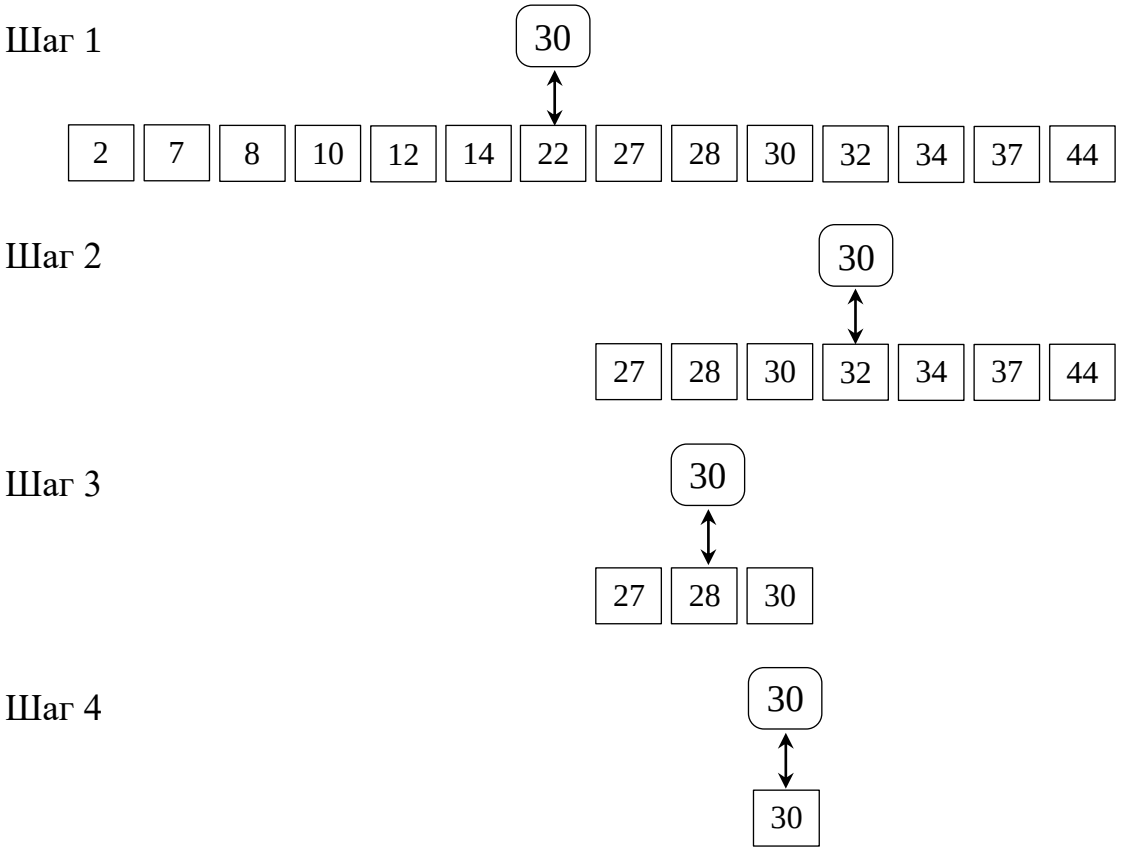



Рис. 2.7. Двоичный поиск

Поиск числа 30 в массиве, содержащем 14 элементов, выполняется в следующем порядке (рис. 2.7). На первом шаге выбирается элемент из середины массива. Здесь надо отметить, что среднюю позицию занимают два числа: 22 и 27. Однако двоичный поиск не налагает ограничений на точный выбор средней позиции в случае чётной длины массива. В данном примере выбрано число 22. Так как 30>22 (на рисунке сравнение показано знаком ↕), то поиск продолжается в правой половине массива. На втором шаге 30<32. Следовательно, поиск уходит в левую область. На третьем шаге 30>28 и мы выбираем правую часть, содержащую всего один элемент. А на четвёртом шаге наступает момент истины: либо число 30 будет найдено, либо нет. В этом примере нам повезло, число 30 мы нашли. Однако для этого мы затратили максимальное число шагов, равное четырём.

**Анализ наихудшего случая**

Оценим количество сравнений (высоты дерева) $K$ в зависимости от числа элементов массива $N$. Для этого установим соответствие между величинами $K$ и $N$ (рис. 2.8):

$$K = 1 \ – \ N = 1;$$
$$K = 2 \ – \ N = 3;$$
$$K = 3 \ – \ N = 7;$$
$$K = 4 \ – \ N = 15.$$

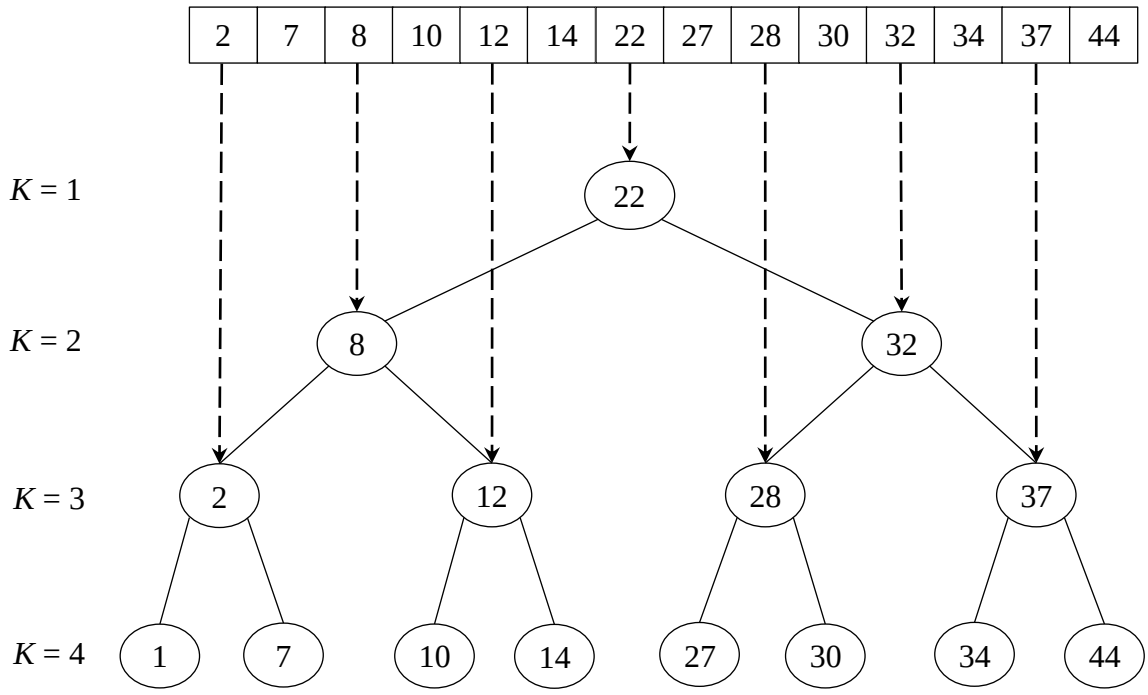



Рис. 2.8. Дерево двоичного поиска в одномерном массиве

В нашем примере (рис. 2.8) на четвёртом шаге проверены 14 элементов вместо 15, так как массив вмещает всего 14 элементов. Но мы сделаем расчёт для полного дерева, то есть для 15 элементов. Обратите внимание, что число элементов $N$ растёт как показательная функция от числа шагов $K$

$$N = 2^K - 1.$$

Следовательно, в наихудшем случае

$$W(N) = K = \lceil \log_2(N+1) \rceil.$$

Скобки $\lceil x \rceil$ означают округление $x$ в бóльшую сторону, $\lfloor x \rfloor$ – в меньшую.

**Анализ среднего случая**

Для оценки среднего случая необходимо суммировать сложность нахождения искомого числа во всех возможных позициях, и сумму разделить на количество всех позиций. Оценим количество элементов массива, достижимых при двоичном поиске (рис. 2.8):

- на первом шаге – 1 элемент ($2^0$ элементов);
- на втором шаге – 2 элемента ($2^1$ элементов);
- на третьем шаге – 4 элемента ($2^2$ элементов);
- на четвёртом шаге – 8 элементов ($2^3$ элементов).

Полное число сравнений определяется суммой произведений числа шагов и количества вершин на каждом уровне дерева. Для получения средней оценки не забываем делить на общее количество элементов

$$\frac{1}{15} \cdot (1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3) = 3{,}2(6).$$

В общем случае средняя сложность

$$A(N) = \frac{1}{N} \cdot \sum_{i=1}^{K} i \cdot 2^{i-1} = \frac{K(N+1)}{N} - 1.$$

Максимальное количество шагов определяется по вышеуказанной формуле $K = \lceil \log_2(N+1) \rceil$. При большом значении $N$ величина $\frac{N+1}{N}$ стремится к единице и

$$A(N) \approx K - 1 = \lceil \log_2(N+1) \rceil - 1.$$

## 2.6 Интерполяционный поиск в упорядоченном массиве

*Интерполяционный поиск* предсказывает позицию искомого числа по функции интерполяционной кривой. Эта функция определяется единожды для конкретного упорядоченного массива и может быть использована вплоть до его существенной модификации.

Простейший интерполяционный поиск предполагает, что значения элементов распределены в массиве равномерно или близко к этому. В этом случае можно применять линейную интерполяцию. Минимальный элемент будет иметь индекс 0, а максимальный – индекс $N$-1.

Когда распределение элементов неравномерно, то поиск производиться на основе предвычисленной функции, интерполирующей последовательность значений элементов в массиве. Преимущества интерполяционного поиска состоят в уменьшении запросов на чтение из памяти и, как следствие, уменьшение числа сравнений.

В среднем интерполяционный поиск производит $O(\log\log N)$ операций. Реальное число операций сравнения зависит от точности интерполяционной функции. В случае большого разброса значений относительно интерполирующей функции поиск может потребовать до $O(N)$ операций.

Рассмотрим интерполяционный алгоритм на примере поиска числа $A = 37$ в упорядоченном массиве с линейной интерполяцией (рис. 2.9)

```
int[] m = {20, 25, 28, 31, 32, 37, 39, 40, 52, 78, 112}; // 10 чисел
```

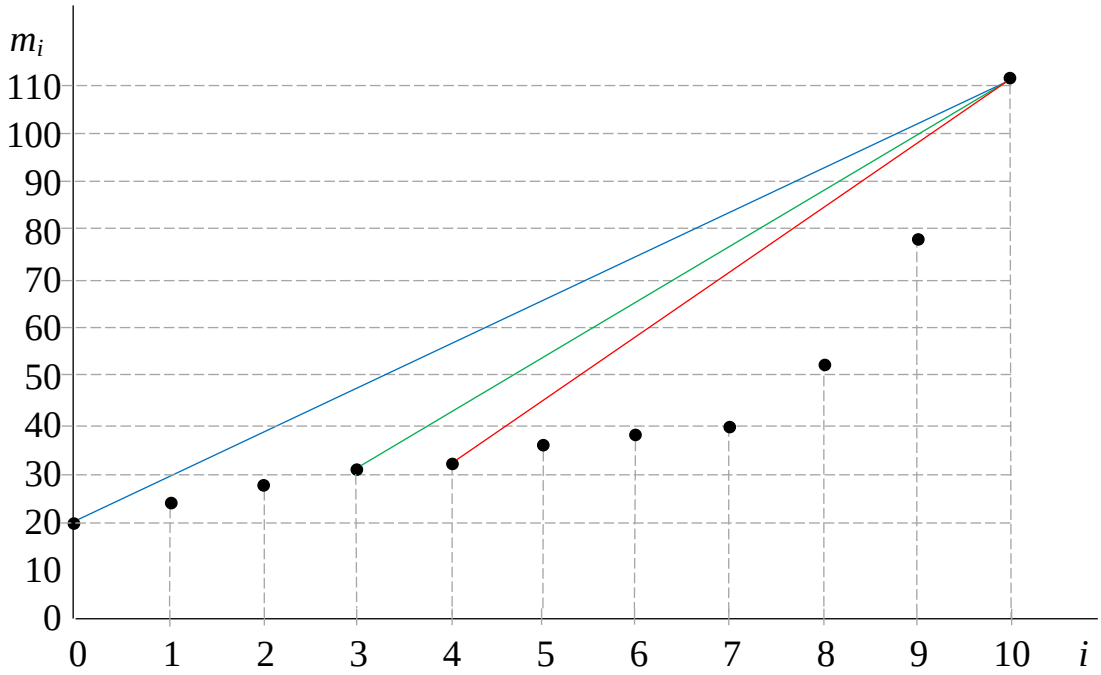



Рис. 2.9. Линейная интерполяция

Примем обозначения: $low$ – индекс левого элемента $m_{low}$; $high$ – индекс правого элемента $m_{high}$.

Шаги поиска при линейной интерполяции следующие:

1. Определяем предполагаемый индекс элемента $A$

$$ind = low + \frac{(high - low)(A - m_{low})}{m_{high} - m_{low}},$$

здесь деление – целочисленное, т.к. индексы – целые числа.

2. Если $A = m_{ind}$, то поиск завершён. Возвращаем индекс $ind$.

3. Если $A < m_{ind}$, то сужаем область поиска, сдвигая правую границу влево $high = ind - 1$. Переходим к п. 1.

4. Если $A > m_{ind}$, то сужаем область поиска, сдвигая левую границу вправо $low = ind + 1$. Переходим к п. 1.

5. Если вышли за пределы массива, то искомое число не найдено. Возвращаем -1.

Выполним данный алгоритм для числа $A = 37$ и массива (рис. 2.9).

1. Первый виток цикла описывает массив синим отрезком

$$ind = 0 + \frac{(10-0)(37-20)}{112-20} = 2,$$

$$low = ind + 1 = 3.$$

2. Второй виток цикла описывает массив зелёным отрезком

$$ind = 3 + \frac{(10-3)(37-31)}{112-31} = 4,$$

$$low = ind + 1 = 5.$$

3. Третий виток цикла описывает массив красным отрезком

$$ind = 5 + \frac{(10-5)(37-37)}{112-37} = 5,$$

$$A = m_5.$$

Большое число шагов обусловлено тем, что элементы массива плохо описываются линейной интерполяцией.

Рассмотрим эту же задачу, предварительно описав элементы массива степенной функцией (рис. 2.10). Будем искать число $A = 39$.

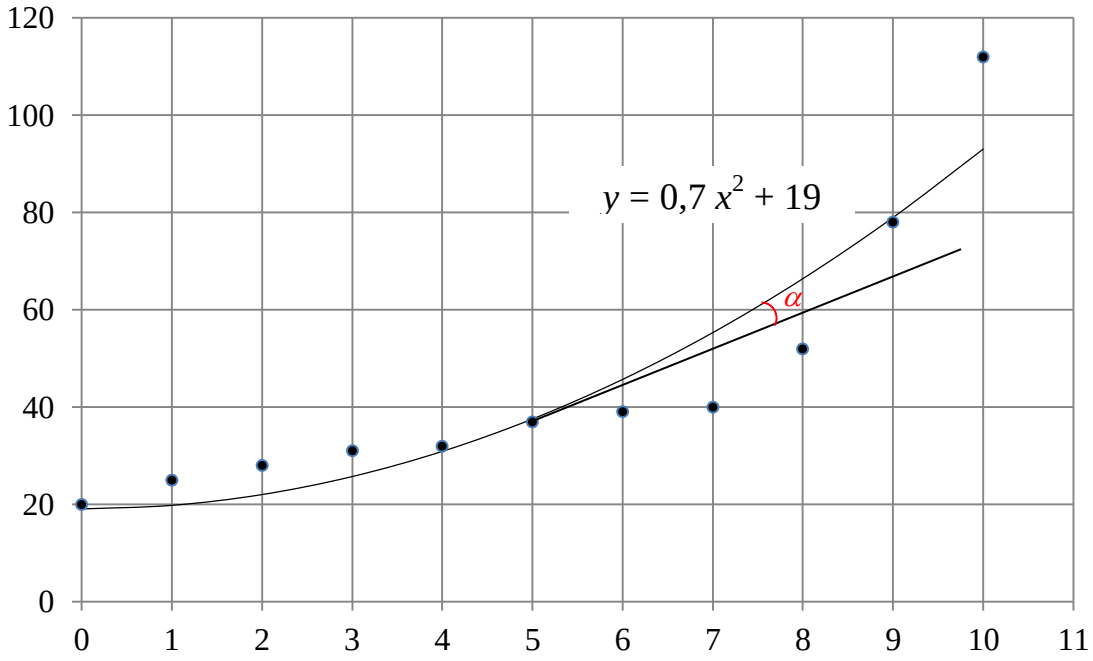



Рис. 2.10. Нелинейная интерполяция

Шаги интерполяционного поиска следующие:

1. По значению искомого числа $A$ предсказываем его индекс $i_0$, квадратные скобки означают округление

$$i_0 = \left[\sqrt{\frac{A-19}{0{,}7}}\right] = \left[\sqrt{\frac{20}{0{,}7}}\right] = 5$$

2. Заглядываем в массив по этому индексу и видим, что ошиблись

$$m[5] = 37.$$

3. Определяем величину и знак ошибки $E$

$$E = A - m[3] = 39 - 37 = +2.$$

4. Зная ошибку $E$, вычисляем приращение индекса $\Delta i$, понимая, что в точке $i_0 = 5$ тангенс угла наклона касательной равен $\tan\alpha = 1{,}4 \cdot 5 = 7$

$$\Delta i = \begin{cases} \left[\frac{E}{k}\right], \left|\frac{E}{k}\right| < 1, \\ \left[\frac{E}{k}\right], \left|\frac{E}{k}\right| \geq 1, \end{cases}$$

$$\Delta i = \left[\frac{E}{7}\right] = \left[\frac{2}{7}\right] = 1.$$

5. Вычисляем новое значение индекса

$$i_1 = i_0 + \Delta i = 5 + 1 = 6.$$

и переходим к п.2. Индекс элемента найден:

$$m[6] = 39.$$

Подобный циклический процесс будет продолжаться до успешного сравнения в п.2. Признаком отсутствия искомого числа в массиве служит смена знака ошибки $D$ или выход за пределы массива. Округление в большую по модулю сторону в п. 4 предотвращает зацикливание на одном индексе в случае малой величины ошибки, как в этом примере.

Высокая достоверность при нелинейной интерполяции уменьшает число шагов поиска. Недостатком может быть сложная формула функции интерполяции.

### 2.7 Выборка из неупорядоченного массива

Задача выборки $K$ наибольших (наименьших) элементов из неупорядоченного массива размером $N$ имеет различные решения в зависимости от соотношения величин $K$ и $N$.

*Решение 1*. Если $K \geq \log_2 N$, то эффективный алгоритм – это сортировка массива. Вычислительная сложность $O(N \cdot \log_2 N)$.

*Решение 2*. Если $K < \log_2 N$, то выборка делается за $K$ проходов.

1. На первом проходе определяем индекс *Ind* максимального элемента *Max*. Элемент *Max* сохраняем в выходном массиве размером *K*. Если максимальных элементов несколько, то сохраняем минимальный индекс.

2. На следующем проходе определяем индекс *Pos* максимального элемента $P_{max}$ такого, что $P_{max} < Max$ или $(P_{max} = Max)\ \&\ (Pos > Ind)$. Элемент $P_{max}$ сохраняем в выходном массиве. Запоминаем найденные параметры $Max = P_{max}, Ind = Pos$.

3. Пункт 2 повторяем ещё *K*-2 раза.

В худшем случае, когда массив упорядочен по возрастанию, задача решается за *K* проходов с вычислительной сложностью $W(N,K) = N \cdot K$.

Однако *наилучшим* решением является использование пирамиды для записи и хранения *K* наибольших (наименьших) элементов. Тогда задача решается за один проход по исходному массиву (см. задачу 3 главы 5).

### 2.8 Методы и свойства класса Array языка C#

Частоиспользуемые методы и свойства класса Array

```
int[] m = { -4, -3, -2, -1, 0, 1, 2, 3, 4 };
int[] m1 = new int[5];
Array.Reverse(m);             // реверс массива
Array.Resize(ref m, 8);       // уменьшение размера до 8 элементов
Array.Copy(m, 2, m1, 0, 5);   // 5 элементов со 2-й позиции в 0-ю
Array.Sort(m);                // сортировка массива
var index = Array.BinarySearch(m, -3); // бинарный поиск в массиве
bool f = Array.Exists(m, i => i == 10);  // наличие по предикату
var x = Array.Find(m, i => i > 0);       // поиск по предикату
var xx = Array.FindAll(m, i => i > 0);   // массив по предикату
x = Array.IndexOf(m, -3);                // индекс элемента
int[] n = (int[])m.Clone();    // неполная копия (только ссылки)
f = m.All(i => i > -10); // проверка всех элементов по предикату
var a = m.Average();     // среднее арифметическое
f = m.Contains(2);       // проверка наличия элемента
a = m.First();           // первый элемент
a = m.Last();            // последний элемент
int len = m.Length;      // количество элементов
a = m.Max();             // максимальный элемент
a = m.Min();             // минимальный элемент
a = m.Sum();             // сумма элементов
var b = m.Take(3);    // взять первые три элемента в коллекцию b
```

**Задача 1.** Разработать функцию двоичного поиска заданного числа в одномерном упорядоченном массиве. Функция должна возвращать индекс вхождения элемента в массив или величину -1 в противном случае.

**Задача 2.** Разработать эффективный алгоритм и программу выборки $K$ наибольших элементов из одномерного неупорядоченного массива длиной $N$ для случая $K < \log_2 N$. Массив модифицировать запрещено.

**Контрольные вопросы**

1. Раскройте понятие динамического массива.
2. В чём заключается отличие статического и динамического массивов?
3. В чём заключается отличие ёмкости и размера динамического массива?
4. Каковы достоинства и недостатки статических и динамических массивов?
5. Каков критерий выбора между статическими и динамическими массивами?
6. Каким образом можно создавать массивы из элементов различных типов на платформе .Net?
7. По какой формуле вычисляется адрес для доступа к элементам двумерного массива?
8. Какова вычислительная сложность двоичного и интерполяционного поиска?

# 3 Умножение матриц

## 3.1 Эффективное использование кэш-памяти

*Кэш-память* (англ. cache, от фр. cacher – прятать) – промежуточная память малого размера (16-32 Мб), расположенная непосредственно на процессорной плате. Она предназначена для быстрого доступа к данным, элементы которых расположены по смежным адресам, т.е. соблюдается принцип *локальности данных*. Если из оперативной памяти данные читаются за несколько десятков тактов, то чтение из кэша третьего уровня потребует менее 10 тактов за счёт его исполнения на статических элементах памяти. Кэш-память состоит из множества *кэш-строк* одинакового размера. Размер кэш-строк равен 64 байтам на процессорах x86. При чтении переменной из оперативной памяти её значение помещается в кэш-строку. Если в ней остаётся свободное место, то загружаются данные, хранимые по смежным адресам.

*Умный совет.* Для повышения скорости обработки массивов следует читать элементы, которые, вероятнее всего, уже находятся в кэше. А в кэше находятся данные, расположенные рядом с ранее прочитанными.

*Толковый совет.* Следует избегать чтения элементов массива, расположенных по произвольным адресам, так как кэш будет постоянно обновлять кэш-строки, затрачивая на это приличное время.

***Пример 1.*** Дана матрица-столбец $A$ размером $n$=6 и матрица-строка $B$ размером $m$=9

$$A = \begin{bmatrix} 2 \\ 1 \\ 2 \\ 3 \\ 4 \\ 1 \end{bmatrix} \quad \text{и} \quad B = [5 \quad 3 \quad 7 \quad 4 \quad 1 \quad 2 \quad 6 \quad 1 \quad 3].$$

Пусть для чтения их элементов используется по одной кэш-строке, вмещающей три числа, и других строк в кэш-памяти нет. Необходимо построить матрицу $M$ размером $n \times m$ путём попарного перемножения элементов матриц $A$ и $B$ по принципу «каждый с каждым». Элементы матрицы $M$ вычисляются по формуле $M_{i,j} = A_i \cdot B_j$. В конечном счёте, надо построить матрицу размером 6×9

$$M = \begin{bmatrix} 10 & 6 & 14 & 8 & 2 & 4 & 12 & 2 & 6 \\ 5 & 3 & 7 & 4 & 1 & 2 & 6 & 1 & 3 \\ 10 & 6 & 14 & 8 & 2 & 4 & 12 & 2 & 6 \\ 15 & 9 & 21 & 12 & 3 & 6 & 18 & 3 & 9 \\ 20 & 12 & 28 & 16 & 4 & 8 & 24 & 4 & 12 \\ 5 & 3 & 7 & 4 & 1 & 2 & 6 & 1 & 3 \end{bmatrix}.$$

**Кэширование данных при традиционном умножении матриц**

Индексацию элементов матриц начинаем с нуля

```
int[] a = { 2, 1, 2, 3, 4, 1 };           // матрица A
int[] b = { 5, 3, 7, 4, 1, 2, 6, 1, 3 };  // матрица B
int[,] m = new int[6, 9];                 // матрица M
for (int i = 0; i < a.Length; i++)
    for (int j = 0; j < b.Length; j++)
        m[i, j] = a[i] * b[j];
```

Как видно из кода, вначале строится первая строка массива `M` путём умножения элементов матрицы `B` на первый элемент массива `A`. При выполнении оператора

```
m[0,0] = a[0] * b[0];
```

естественно, будет промах в кэш и в кэш-строку вместе с элементом `a[0]` загружаются элементы `a[1]` и `a[2]`. В другую кэш-строку вместе с `b[0]` загружаются элементы `b[1]` и `b[2]`. Выполнение двух операторов

```
m[0,1] = a[0] * b[1];
m[0,2] = a[0] * b[2];
```

будет быстрым, так как элементы `b[1]` и `b[2]` уже находятся в кэше. Далее, при выполнении оператора

```
m[0,3] = a[0] * b[3];
```

будет промах в кэш при чтении `b[3]`. Поэтому кэш-строка для массива `B` обновится новой тройкой `b[3] b[4] b[5]`. Выполнение операторов

```
m[0,4] = a[0] * b[4];
m[0,5] = a[0] * b[5];
```

вновь будет быстрым, но затем опять последует промах и кэш-строка обновится элементами `b[6] b[7] b[8]`. Выполнение операторов

```
m[0,6] = a[0] * b[6];
m[0,7] = a[0] * b[7];
m[0,8] = a[0] * b[8];
```

завершит построение первой строки массива `M`. Рассмотренное построение строки массива четыре раза обновляло кэш-строки.

Построение второй строки массива `M` трижды обновит кэш-строку элементами массива `B`. При построении третьей строки массива `M` также трижды обновится кэш-строка для массива `B`.

Таким образом, для построения первых трёх строк массива `M` кэш-память будет обновлена 10 раз. Для построения остальных трёх строк будет единожды обновлена кэш-строка, содержащая элементы массива `A` и девять раз обновлена кэш-память при чтении элементов массива `B`. Итого, традиционное умножение 20 раз обновит кэш.

**Кэширование данных при блочном способе построения массива**

*Блочная* матрица – это матрица, разделённая на блоки. *Блок* – матрица меньшего размера. Матрица и её блоки могут быть квадратными или прямоугольными, но размер блоков обязательно одинаков.

Для уменьшения числа промахов в кэш представим результирующую матрицу $M$ шестью блоками, каждый размером 3×3,

$$M = \begin{bmatrix} 10 & 6 & 14 & 8 & 2 & 4 & 12 & 2 & 6 \\ 5 & 3 & 7 & 4 & 1 & 2 & 6 & 1 & 3 \\ 10 & 6 & 14 & 8 & 2 & 4 & 12 & 2 & 6 \\ 15 & 9 & 21 & 12 & 3 & 6 & 18 & 3 & 9 \\ 20 & 12 & 28 & 16 & 4 & 8 & 24 & 4 & 12 \\ 5 & 3 & 7 & 4 & 1 & 2 & 6 & 1 & 3 \end{bmatrix} =$$

$$= \begin{bmatrix} \begin{bmatrix} 10 & 6 & 14 \\ 5 & 3 & 7 \\ 10 & 6 & 14 \end{bmatrix} & \begin{bmatrix} 8 & 2 & 4 \\ 4 & 1 & 2 \\ 8 & 2 & 4 \end{bmatrix} & \begin{bmatrix} 12 & 2 & 6 \\ 6 & 1 & 3 \\ 12 & 2 & 6 \end{bmatrix} \\ \begin{bmatrix} 15 & 9 & 21 \\ 20 & 12 & 28 \\ 5 & 3 & 7 \end{bmatrix} & \begin{bmatrix} 12 & 3 & 6 \\ 16 & 4 & 8 \\ 4 & 1 & 2 \end{bmatrix} & \begin{bmatrix} 18 & 3 & 9 \\ 24 & 4 & 12 \\ 6 & 1 & 3 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} M_1 & M_2 & M_3 \\ M_4 & M_5 & M_6 \end{bmatrix}.$$

Будем строить эти блоки в указанном порядке от $M_1$ до $M_6$. Переменные `ii` и `jj` введены в программу для организации блочного перебора элементов массивов

```
for (ii = 0; ii < n; ii += 3)
    for (jj = 0; jj < n; jj += 3)
        for (i = ii; i < ii + 3; i++)
            for (j = jj; j < jj + 3; j++)
                m[i, j] = a[i] * b[j];
```

Согласно блочному способу загруженные кэш-строки надо использовать максимально эффективно, т.е. всё, что можно вычислить на их основе, надо вычислить, например, при загрузке кэш-строк

```
a[0] a[1] a[2];
b[0] b[1] b[2];
```

не следует останавливаться на получении элементов `m[0,0]`, `m[0,1]`, `m[0,2]`. Надо продолжать вычисление всего блока $M_1$

```
m[0,0] m[0,1] m[0,2];
m[1,0] m[1,1] m[1,2];
m[2,0] m[2,1] m[2,2];
```

И только тогда, когда на загруженных в кэш данных больше вычислить ничего нельзя, следует обновить кэш-строку следующими элементами

```
b[3] b[4] b[5];
```

и вычислить блок $M_2$

```
m[0,3] m[0,4] m[0,5];
m[1,3] m[1,4] m[1,5];
m[2,3] m[2,4] m[2,5];
```

после чего вновь обновить кэш-строку

```
b[6] b[7] b[8];
```

и вычислить правый верхний блок $M_3$

```
m[0,6] m[0,7] m[0,8];
m[1,6] m[1,7] m[1,8];
m[2,6] m[2,7] m[2,8];
```

На данный момент всего за четыре загрузки кэш-строк у нас построена верхняя половина матрицы $M$. Нижняя половина также строится за четыре загрузки кэш-строк. В итоге блочный способ потребует всего восемь обновлений кэш-строк, что на 12 меньше, чем при умножении традиционным способом.

Количество загрузок кэш-строк можно ещё уменьшить, если вычислять блоки *змейкой*

$$M = \begin{bmatrix} M_1 & M_2 & M_3 \\ M_6 & M_5 & M_4 \end{bmatrix}.$$

Тогда для вычисления блока $M_4$ следует обновить только одну кэш-строку матрицы $A$, так как в кэш-строке матрицы $B$ остались нужные нам данные после вычисления блока $M_3$. Этот способ требует модификации всего одной кэш-строки при построении каждого блока, кроме первого: для него надо загружать обе кэш-строки. В этом примере нам потребуется семь обновлений кэш-строк.

### 3.2 Традиционное умножение матриц

Умножение квадратных матриц $G$ и $H$ некоммутативно, т.е. результаты $G{\cdot}H$ и $H{\cdot}G$ могут отличаться и это надо помнить. Матрицы в программе задаются в виде массивов соответствующей размерности. Умножаемые матрицы должны иметь общую размерность. Произведением 2×3-матрицы и 3×4-матрицы является 2×4-матрица с общей размерностью равной трём

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \cdot \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \end{bmatrix} =$$

$$= \begin{bmatrix} aA + bE + cI & aB + bF + cJ & aC + bG + cK & aD + bH + cL \\ dA + eE + fI & dB + eF + fJ & dC + eG + fK & dD + eH + fL \end{bmatrix}.$$

Такое произведение требует 24 умножения и 16 сложений.

В общем случае, произведение $n \times m$-матрицы и $m \times p$-матрицы требует $n \cdot m \cdot p$ умножений и $n \cdot (m-1) \cdot p$ сложений. Произведение квадратных матриц требует $n^3$ умножений и $n^3 - n^2$ сложений.

***Пример 2.*** Традиционное умножение матрицы $G$ размером $n \times m$ на матрицу $H$ размером $m \times p$

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < p; j++)
    {
        R[i, j] = 0;
        for (int k = 0; k < m; k++)
            R[i, j] = R[i, j] + G[i, k] * H[k, j];
    }
```

Здесь `R` – матрица-произведение. Такое умножение неэффективно использует кэш-память и имеет вычислительную сложность $O(n^3)$.

### 3.3 Блочное умножение матриц

Алгоритм *блочного умножения* матриц позволяет эффективно использовать кэш-память процессора, что ведёт к уменьшению времени умножения матриц в сравнении с традиционным умножением. Однако блочное умножение в целом сохраняет высокую вычислительную сложность $O(n^3)$. Размер блока должен быть кратен размеру кэш-строки.

Рассмотрим блочное умножение двух квадратных матриц $A$ и $B$

$$C = A \cdot B = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1} & A_{n,2} & \cdots & A_{n,n} \end{bmatrix} \cdot \begin{bmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,n} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ B_{n,1} & B_{n,2} & \cdots & B_{n,n} \end{bmatrix} =$$

$$= \begin{bmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,n} \\ C_{2,1} & C_{2,2} & \cdots & C_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n,1} & C_{n,2} & \cdots & C_{n,n} \end{bmatrix}.$$

где $A_{i,j}$, $B_{i,j}$ и $C_{i,j}$ – блоки матриц.

Результирующая матрица формируется поблочно. Каждый блок $C_{i,j}$ вычисляется по формуле

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}.$$

Причём каждый блок $C_{i,j}$, $A_{i,k}$ и $B_{k,j}$ в свою очередь рекурсивно разбивается на блоки меньшего размера. Собственно, умножение элементов мат-

риц начинаются на дне рекурсии, когда размер каждого блока достаточно мал, и их произведение можно вычислить конечной формулой без циклов. Полное произведение формируется на выходе из рекурсии.

***Пример 3.*** Блочное умножение матриц

$$\begin{bmatrix}1&1&2&2\\1&1&2&2\\3&3&4&4\\3&3&4&4\end{bmatrix}\cdot\begin{bmatrix}1&2&3&4\\1&2&3&4\\4&3&2&1\\4&3&2&1\end{bmatrix}=\begin{bmatrix}\begin{bmatrix}1&1\\1&1\end{bmatrix}&\begin{bmatrix}2&2\\2&2\end{bmatrix}\\\begin{bmatrix}3&3\\3&3\end{bmatrix}&\begin{bmatrix}4&4\\4&4\end{bmatrix}\end{bmatrix}\cdot\begin{bmatrix}\begin{bmatrix}1&2\\1&2\end{bmatrix}&\begin{bmatrix}3&4\\3&4\end{bmatrix}\\\begin{bmatrix}4&3\\4&3\end{bmatrix}&\begin{bmatrix}2&1\\2&1\end{bmatrix}\end{bmatrix}=$$

$$=\begin{bmatrix}\begin{bmatrix}1&1\\1&1\end{bmatrix}\cdot\begin{bmatrix}1&2\\1&2\end{bmatrix}+\begin{bmatrix}2&2\\2&2\end{bmatrix}\cdot\begin{bmatrix}4&3\\4&3\end{bmatrix}&\begin{bmatrix}1&1\\1&1\end{bmatrix}\cdot\begin{bmatrix}3&4\\3&4\end{bmatrix}+\begin{bmatrix}2&2\\2&2\end{bmatrix}\cdot\begin{bmatrix}2&1\\2&1\end{bmatrix}\\\begin{bmatrix}3&3\\3&3\end{bmatrix}\cdot\begin{bmatrix}1&2\\1&2\end{bmatrix}+\begin{bmatrix}4&4\\4&4\end{bmatrix}\cdot\begin{bmatrix}4&3\\4&3\end{bmatrix}&\begin{bmatrix}3&3\\3&3\end{bmatrix}\cdot\begin{bmatrix}3&4\\3&4\end{bmatrix}+\begin{bmatrix}4&4\\4&4\end{bmatrix}\cdot\begin{bmatrix}2&1\\2&1\end{bmatrix}\end{bmatrix}=$$

$$=\begin{bmatrix}\begin{bmatrix}2&4\\2&4\end{bmatrix}+\begin{bmatrix}16&12\\16&12\end{bmatrix}&\begin{bmatrix}6&8\\6&8\end{bmatrix}+\begin{bmatrix}8&4\\8&4\end{bmatrix}\\\begin{bmatrix}6&12\\6&12\end{bmatrix}+\begin{bmatrix}32&24\\32&24\end{bmatrix}&\begin{bmatrix}18&24\\18&24\end{bmatrix}+\begin{bmatrix}16&8\\16&8\end{bmatrix}\end{bmatrix}=\begin{bmatrix}\begin{bmatrix}18&16\\18&16\end{bmatrix}&\begin{bmatrix}14&12\\14&12\end{bmatrix}\\\begin{bmatrix}38&36\\38&36\end{bmatrix}&\begin{bmatrix}34&32\\34&32\end{bmatrix}\end{bmatrix}=$$

$$=\begin{bmatrix}18&16&14&12\\18&16&14&12\\38&36&34&32\\38&36&34&32\end{bmatrix}.$$

### 3.4 Алгоритм Штрассена

Алгоритм Фолькера Штрассена развивает идею блочного умножения матриц и уменьшает асимптотическую оценку числа умножений. Традиционный алгоритм умножения матриц размером 2×2 выполняется за восемь умножений и четыре сложения

$$\begin{bmatrix}a&b\\c&d\end{bmatrix}\cdot\begin{bmatrix}A&B\\C&D\end{bmatrix}=\begin{bmatrix}aA+bC&aB+bD\\cA+dC&cB+dD\end{bmatrix}.$$

Штрассен в 1969 году предложил способ умножения матриц размером 2×2 за семь умножений. Способ предполагает предварительное вычисление семи параметров за семь умножений и 10 сложений

$$\begin{aligned}x_1&=(a+d)(A+D),\\x_2&=(c+d)A,\\x_3&=a(B-D),\\x_4&=d(C-A),\\x_5&=(a+b)D,\\x_6&=(c-a)(A+B),\\x_7&=(b-d)(C+D).\end{aligned}$$

После чего элементы матрицы-произведения вычисляются за восемь сложений без единого умножения

$$aA + bC = x_1 + x_4 - x_5 + x_7,$$
$$aB + bD = x_3 + x_5,$$
$$cA + dC = x_2 + x_4,$$
$$cB + dD = x_1 - x_2 + x_3 + x_6.$$

В итоге получается семь умножений и 18 сложений, т.е. вместо одного умножения выполняется 14 сложений.

В отличие от традиционного алгоритма умножения матриц размером 2×2, работающего за $N^{\log_2 8} = N^3$ умножений, алгоритм Штрассена выполняет $N^{\log_2 7} = N^{2,81}$ умножений. Алгоритм работает только с квадратными матрицами размером $2^n \times 2^n$. На первом рекурсивном вызове он разбивает матрицу на квадратные блоки размером $2^{n-1} \times 2^{n-1}$. Рекурсивный процесс разбиения продолжается до тех пор, пока на $i$-ом шаге размер блока $2^{n-i} \times 2^{n-i}$ не станет настолько малым, чтобы использовать традиционный способ умножения. Так делают потому, что алгоритм Штрассена теряет эффективность по сравнению с традиционным алгоритмом на малых размерах матриц в силу большего числа сложений. Оптимальный размер матриц для перехода к традиционному умножению зависит от характеристик процессора и на практике лежит в диапазоне от 32 до 128.

Алгоритм Штрассена – первый алгоритм, использующий менее чем $N^3$ умножений. На больших матрицах он настолько эффективен, что выгодно расширить исходные матрицы до требуемого размера $2^n \times 2^n$ путём добавления нулевых строк и столбцов.

### 3.5 Умножение матриц по Винограду

Умножение матриц по Винограду имеет теоретическую оценку времени $O(n^3)$, но различные точные оценки сложности для матриц с чётной и нечётной общей размерностью.

Рассмотрим случай нечётной общей размерности. Традиционное умножение (24 умножения и 16 сложений) двух матриц

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \cdot \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \end{bmatrix} =$$

$$= \begin{bmatrix} aA + bE + cI & aB + bF + cJ & aC + bG + cK & aD + bH + cL \\ dA + eE + fI & dB + eF + fJ & dC + eG + fK & dD + eH + fL \end{bmatrix}$$

перепишем в виде

$$aA + bE + cI = (a + E)(b + A) + cI - ab - AE,$$
$$aB + bF + cJ = (a + F)(b + B) + cJ - ab - BF,$$
$$aC + bG + cK = (a + G)(b + C) + cK - ab - CG,$$
$$aD + bH + cL = (a + H)(b + D) + cL - ab - DH,$$

$$dA + eE + fI = (d + E)(e + A) + fI - de - AE,$$
$$dB + eF + fJ = (d + F)(e + B) + fJ - de - BF,$$
$$dC + eG + fK = (d + G)(e + C) + fK - de - CG,$$
$$dD + eH + fL = (d + H)(e + D) + fL - de - DH.$$

Получилось 22 умножения и 40 сложений, так как некоторые произведения встречаются многократно: $ab$ – 4 раза, $de$ – 4 раза, $AE$ – 2 раза, $BF$ – 2 раза, $CG$ – 2 раза, $DH$ – 2 раза. Эти произведения вычисляются предварительно для каждой строки первой матрицы и каждого столбца второй матрицы, и записываются во вспомогательные матрицы. В итоге мы сэкономили два умножения, но получили 24 дополнительные операции сложения, т.е. вместо одного умножения выполняется 12 сложений.

При умножении матриц размером $a \times b$ и $b \times c$ с нечётной общей размерностью, количество операций растёт кубически $O(n^3)$ (табл. 3.1).

Т а б л и ц а 3.1

Оценка сложности умножения матриц с нечётной общей размерностью

| | Количество умножений | Количество сложений |
|---|---|---|
| Предварительная обработка $a \times b$ | $a\frac{b-1}{2}$ | $a\left(\frac{b-1}{2} - 1\right)$ |
| Предварительная обработка $b \times c$ | $c\frac{b-1}{2}$ | $c\left(\frac{b-1}{2} - 1\right)$ |
| Вычисление $a \times c$ | $ac\left(\frac{b-1}{2} + 1\right)$ | $\frac{ac}{2}(3b + 1)$ |
| Всего | $\frac{b-1}{2}(ac + a + c) + ac$ | $(a + c)\left(\frac{b-1}{2} - 1\right) + \frac{ac}{2}(3b + 1)$ |

Рассмотрим случай чётной общей размерности. Произведением 2×4-матрицы и 4×3-матрицы является 2×3-матрица. Для её получения требуется 24 умножения и 18 сложений.

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix} \cdot \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \\ J & K & L \end{bmatrix} =$$
$$\begin{bmatrix} aA + bD + cG + dJ & aB + bE + cH + dK & aC + bF + cI + dL \\ eA + fD + gG + hJ & eB + fE + gH + hK & eC + fF + gI + hL \end{bmatrix}.$$

Перепишем каждый элемент матрицы-произведения в виде

$$aA + bD + cG + dJ = (a + D)(b + A) + (c + J)(d + G) - ab - cd - AD - GJ,$$

$aB + bE + cH + dK = (a + E)(b + B) + (c + K)(d + H) - ab - cd - BE - HK,$
$aC + bF + cI + dL = (a + F)(b + C) + (c + L)(d + I) - ab - cd - CF - IL,$
$eA + fD + gG + hJ = (e + D)(f + A) + (g + J)(h + G) - ef - gh - AD - GJ,$
$eB + fE + gH + hK = (e + E)(f + B) + (g + K)(h + H) - ef - gh - BE - HK,$
$eC + fF + gI + hL = (e + F)(f + C) + (g + L)(h + I) - ef - gh - CF - IL.$

Получилось 22 умножения и 47 сложений, так как некоторые произведения встречаются многократно: *ab*+*cd* – 3 раза, *ef*+*gh* – 3 раза, *AD*+*GJ* – 2 раза, *BE*+*HK* – 2 раза, *CF*+*IL* – 2 раза. В итоге мы сэкономили два умножения, но получили 29 дополнительных сложений, т.е. вместо одного умножения выполняем 14,5 сложений.

При умножении матриц размером $a \times b$ и $b \times c$ с чётной общей размерностью, количество операций также растёт кубически $O(n^3)$ (табл. 3.2).

Таблица 3.2

Оценка сложности умножения матриц с чётной общей размерностью

| | Количество умножений | Количество сложений |
|---|---|---|
| Предварительная обработка $a \times b$ | $\frac{ab}{2}$ | $a\left(\frac{b}{2} - 1\right)$ |
| Предварительная обработка $b \times c$ | $\frac{cb}{2}$ | $c\left(\frac{b}{2} - 1\right)$ |
| Вычисление $a \times c$ | $\frac{abc}{2}$ | $ac\left(3\frac{b}{2} + 1\right)$ |
| Всего | $\frac{b}{2}(ac + a + c)$ | $(a + c)\left(\frac{b}{2} - 1\right) + \frac{ac}{2}(3b + 2)$ |

Сравнение эффективности различных алгоритмов умножения матриц показывает (табл. 3.3, рис. 3.1), что алгоритм Штрассена обладает малой вычислительной сложностью, устойчив к исходным данным и применим к матрицам любого размера. Однако при реализации этого алгоритма надо следить за размером свободного стека, так как алгоритм рекурсивный и может его переполнить на больших матрицах.

Таблица 3.3

Оценка сложности умножения матриц размером $N \times N$

| Алгоритм | Количество умножений | Количество сложений |
|---|---|---|
| Традиционный | $N^3$ | $N^3 - N^2$ |
| Винограда | $\frac{N^3 + 2N^2}{2}$ | $\frac{3N^3 + 4N^2 - 4N}{2}$ |
| Штрассена | $N^{2,81}$ | $6N^{2,81} - 6N^2$ |

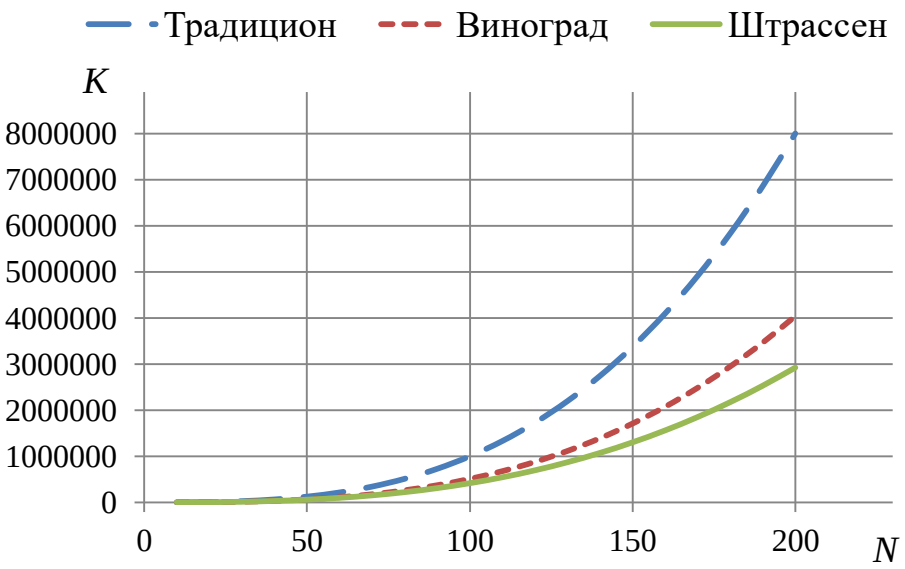



Рис.3.1. Зависимость количества умножений $K$ от размера $N$

Умножение матриц по Винограду не улучшает алгоритм Штрассена, но комбинация и усовершенствование этих алгоритмов послужили базой для алгоритма Копперсмита – Винограда с оценкой вычислительной сложности $O(n^{2,3727})$. Стоит отметить, что алгоритм Копперсмита – Винограда на практике не применяется, так как он эффективен только на матрицах астрономического размера.

**Задача 1.** Разработать функцию блочного умножения матрицы-столбца на матрицу-строку произвольных размеров.

**Задача 2.** Построить экспериментальную кривую вычислительной сложности традиционного алгоритма умножения квадратных матриц.

**Контрольные вопросы**

1. Каково назначение кэш-памяти?
2. Какая информация записывается в кэш-строку?
3. В чём заключается суть эффективного использования кэш-памяти?
4. Раскройте понятие блочной матрицы.
5. В чём заключается блочный способ построения массива?
6. Какова идея алгоритма Штрассена?
7. Какова оценка вычислительной сложности алгоритма Штрассена?
8. Каковы недостатки традиционного умножения матриц?

# 4 Быстрая сортировка

## 4.1 Введение в сортировку коллекций

Алгоритм *сортировки* – это алгоритм упорядочивания элементов в коллекции (массив, список и т.п.) по заданному признаку. В случае, когда элемент коллекции является структурой, то поле, по которому упорядочивается массив, называется *ключевым*. На практике в качестве ключа часто выступает одно поле, а остальные поля хранят какие-либо данные, никак не влияющие на сортировку. В некоторых случаях ключом служит некоторая функция над полями.

К примеру, задан массив структур `Point`, хранящих координаты точек на декартовой плоскости в полях *x* и *y*. Ключевым полем сортировки можно задать координату *x* или *y*. В качестве ключа можно выбрать сумму координат $x + y$, или расстояние точки до центра координат $\sqrt[2]{x^2 + y^2}$, или любую другую функцию над этими координатами.

**Классификация алгоритмов сортировки**

**По типу:**

а) *устойчивая* сортировка не меняет порядок сортируемых элементов, имеющих одинаковые ключи;

б) *неустойчивая* сортировка может изменить порядок сортируемых элементов, имеющих одинаковые ключи;

в) *внутренняя* сортировка обрабатывает коллекцию в оперативной памяти. Алгоритм сортировки должен хорошо сочетаться с алгоритмами кэширования и подкачки;

г) *внешняя* сортировка обрабатывает данные, расположенные на периферийных устройствах и не вмещающихся в оперативную память. Доступ к носителю осуществляется последовательно – в каждый момент времени можно прочитать или записать элемент, следующий за текущим. Наиболее часто внешняя сортировка используется в СУБД. Стоит отметить, что внутренняя сортировка значительно эффективней внешней, так как на обращение к оперативной памяти тратится меньше времени, чем к внешним носителям;

д) *естественность поведения* – повышение эффективности при обработке уже упорядоченных или частично упорядоченных коллекций;

е) потребность в *дополнительной памяти*;

ж) *знание структуры* сортируемых данных, выходящей за рамки операции сравнения.

**По способу:**

а) обменная:

– пузырьковая (англ. bubble sort) $O(N^2)$, устойчивая;

– перемешиванием (англ. cocktail sort) $O(N^2)$, устойчивая;

– гномья/«глупая» (англ. gnome sort; stupid sort) $O(N^2)$, устойчивая;

– быстрая (англ. quick sort) от $O(N \log_2 N)$ до $O(N^2)$, неустойчивая, можно сделать устойчивой;

– расчёской (англ. comb sort) $O(N^2)$, неустойчивая;

б) выбором:

– выбором (англ. selection sort) $O(N^2)$, неустойчивая;

– пирамидальная (англ. heapsort) $O(N \log_2 N)$, неустойчивая;

– плавная (англ. smoothsort) $O(N \log_2 N)$, неустойчивая;

в) вставками:

– вставками (англ. insertion sort) $O(N^2)$, устойчивая;

– Шелла (англ. Shell sort) $O(N(\log_2 N)^2)$, неустойчивая;

г) слиянием, (англ. merge sort) $O(N \log_2 N)$, устойчивая;

д) сортировка с помощью двоичного дерева (tree sort) $O(N \log_2 N)$, устойчивая;

е) без сравнений:

– подсчётом (англ. counting sort) $O(N + K)$, устойчивая;

– поразрядная (англ. radix sort) $O(N \cdot K)$, устойчивая;

– блочная (англ. bucket sort) $O(N)$, устойчивая.

### 4.2 Быстрая сортировка (сортировка Хоара)

*Быстрая сортировка* (англ. quick sort или qsort) – широко известный алгоритм сортировки, разработанный английским учёным Чарльзом Хоаром в 1960 г. в СССР, где он обучался в МГУ компьютерному переводу и занимался разработкой русско-английского разговорника. Хоар разработал этот метод применительно к машинному переводу.

**Алгоритм быстрой сортировки**

1. Выбрать элемент массива в качестве *опорного*. Идеальный выбор – медиана, но для её определения нужно время. *Медиана* разделяет массив на две равные части: одна часть содержит элементы, меньшие медианы, другая – бóльшие. В качестве опорного иногда выбирают первый, или последний, или элемент со случайным индексом. Опорный элемент может быть числом, не принадлежащим массиву.

2. Разделить массив на две части так, чтобы в левой части были элементы меньшие или равные опорному, в правой – бóльшие или равные опорному. Алгоритм разделения обычно использует встречный поиск.

2.1. Продвигаясь по массиву слева направо, ищем элемент, больший или равный опорному.

2.2. Продвигаясь по массиву справа налево, ищем элемент, меньший или равный опорному.

2.3. Если такие элементы найдены, то меняем их местами и возвращаемся к подпункту 2.1.

2.4. Если встречный поиск пересёкся, то считаем, что массив разделён на две части. Границей разделения является элемент, на котором произошла встреча. Этот элемент может остаться неупорядоченным (см. элемент 79 на рис. 4.1), поэтому он передаётся сразу в два подмассива.

3. Рекурсивно повторить пункты 1 и 2 для двух частей массива, полученных в подпункте 2.4. Условием останова рекурсии является длина массива меньшая трёх. Два элемента легко упорядочить обменом, а один упорядочивать не надо.

Ниже представлена функция быстрой сортировки с выводом промежуточных результатов на экран.

```
public static void QuickSort(int[] A, int low, int high)
{
    int i = low;   // запоминаем наименьший индекс
    int j = high;  // запоминаем наибольший индекс
    int v = A[(low + high) >> 1];  // получаем опорный элемент
    Console.Write($"\nот {low} до {high}  ");
    Console.WriteLine($"опорный: m[{(low + high) / 2}]={v}");
    do
    {
        while (A[i] < v) ++i;  // ищем элемент ≥ опорному
        while (A[j] > v) --j;  // ищем элемент ≤ опорному
        Console.WriteLine($"нашли: i={i} j={j}");
        if (i <= j)            // если индексы не пересеклись
        {
            Console.WriteLine($"меняем: {A[i]} {A[j]}");
            int temp = A[i];   // то меняем элементы местами
            A[i] = A[j];
            A[j] = temp;
            i++; j--;      // индексы сдвигаем навстречу друг другу
            Console.WriteLine(string.Join(" ", A));
        }
    } while (i < j);                  // пока индексы не встретились
    if (low < j) QuickSort(A, low, j);   // сорт. левый подмассив
    if (i < high) QuickSort(A, i, high); // сорт. правый подмассив
}
```

Проверьте функцию `QuickSort` с помощью кода

```
int[] m = { 61, 67, 75, 44, 31, 12, 94, 59, 26, 79 };
Console.WriteLine(string.Join(" ", m));
QuickSort(m, 0, m.Length - 1);
Console.WriteLine(string.Join(" ", m));
```

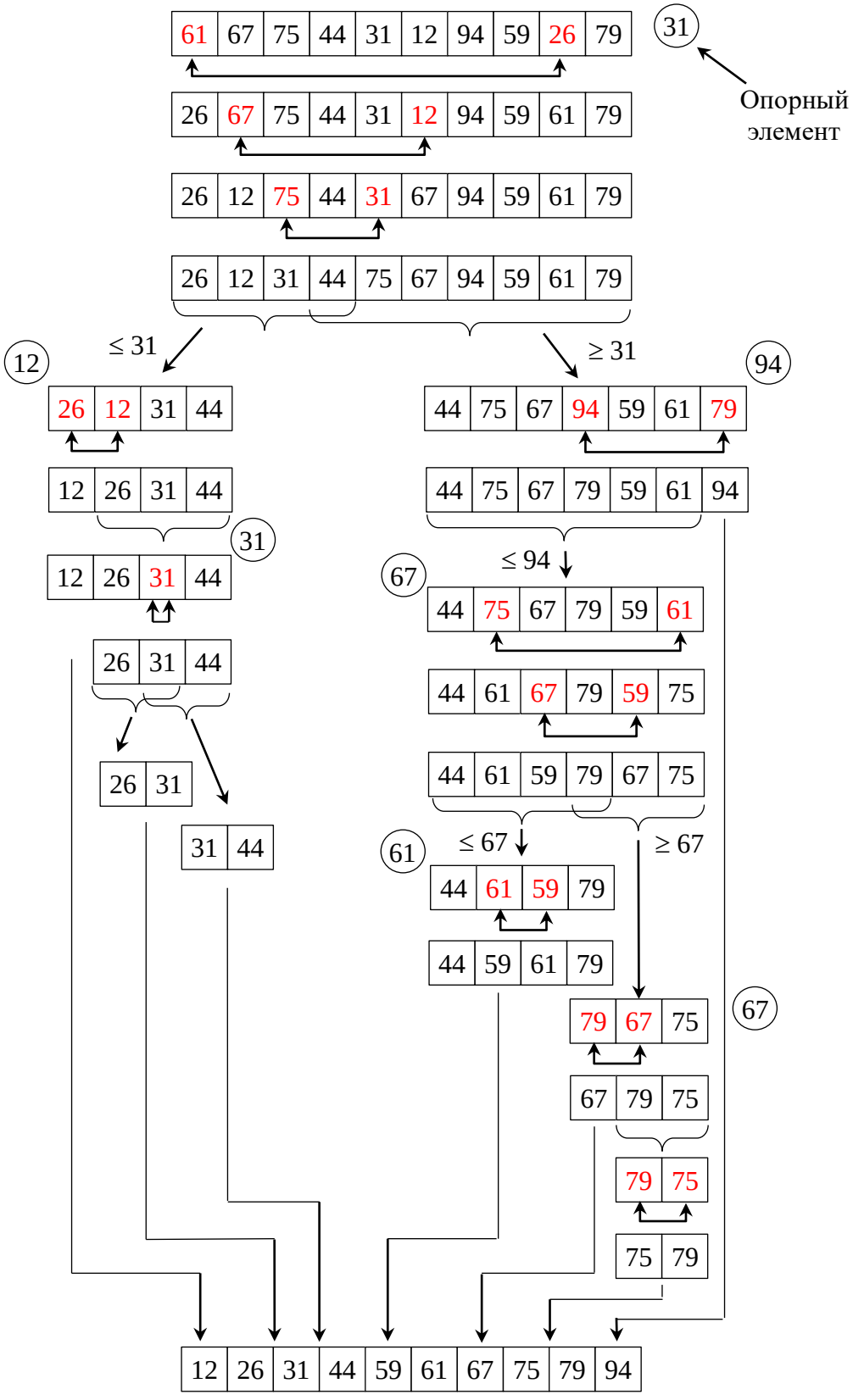



Рис. 4.1 Выполнение алгоритма быстрой сортировки

Представленная программа в качестве опорного элемента выбирает элемент, занимающий среднюю позицию в массиве. Опорные элементы заключены в кружки (рис. 4.1). Стрелками показан обмен элементов местами. Рекурсия останавливается, когда массив содержит меньше двух элементов.

**Оценка эффективности алгоритма Хоара**

*Лучший случай.* Для этого алгоритма самый лучший случай – когда опорный элемент делит массив на две равных части. Тогда количество сравнений равно $N \log_2 N$, что является наименьшим временем.

*Средний случай.* Оценка количества обменов $O(N \log_2 N)$. На практике средний случай имеет место при случайном порядке элементов и выборе опорного элемента из элементов массива.

*Худший случай.* Опорный элемент разделяет массив на себя и массив из всех остальных элементов. Такое может произойти, если в качестве опорного на каждом этапе будет выбран либо наименьший, либо наибольший элемент. Худший случай даёт $O(N^2)$ обменов.

**Варианты улучшения алгоритма:**

1. При выборе опорного элемента случайным образом худший случай становится очень маловероятным и ожидаемое время выполнения алгоритма сортировки равно $O(N \log_2 N)$.

2. Выбрать опорным элементом средний по величине среди трёх элементов: первого, последнего и находящегося в середине массива. Такой выбор также направлен против худшего случая.

3. При приближении опасной глубины рекурсии возможно устранение одной ветви рекурсии: рекурсивный вызов делать только для меньшего подмассива, а больший обрабатывать циклически в пределах этого же вызова функции `qSort`. Очевидно, что глубина рекурсии не превысит $\log_2 N$, а в худшем случае она будет не более 2, так как вся обработка пройдёт в цикле первого уровня рекурсии.

4. Разбивать массив не на две, а на три части: по признаку меньше, равно и больше опорного элемента.

**Достоинства быстрой сортировки:**

а) одна из самых быстродействующих (на практике) внутренних сортировок общего назначения;

б) проста в реализации;

в) сортирует на месте;

г) требует лишь $\log_2 N$ дополнительной памяти для стека адресов возврата, а неулучшенный алгоритм в худшем случае – $O(N)$ памяти;

д) хорошо сочетается с механизмами кэширования и виртуальной памяти;

е) существует эффективная модификация (алгоритм Седжвика) для сортировки строк: сначала сравнение с опорным элементом выполняется только по нулевому символу строки, далее применение аналогичной сортировки для «большего» и «меньшего» массивов тоже по нулевому символу, и для «равного» массива – сравнение уже по первому символу;

ж) работает на связанных списках и других структурах с последовательным доступом.

**Недостатки быстрой сортировки:**

а) при неудачном выборе опорных элементов (например, `m[low]`) сильно деградирует по скорости до $O(N^2)$, что может случиться, когда массив упорядочен или почти упорядочен. Таблеткой для лечения является модификация алгоритма, известная как `Introsort`, – при приближении к исчерпанию стека следует перейти на пирамидальную сортировку или выбирать опорный элемент случайно, а не фиксированным образом. Так работает метод `Sort` класса `Array`. В качестве опорного элемента он выбирает среднее значение между первым, средним и последним элементами массива. Этот метод переключается на пирамидальную сортировку при достижении глубины рекурсии $2\log_2 N$. Если размер массива не превышает 16 элементов, то применяется алгоритм вставками;

б) наивная реализация алгоритма может привести к ошибке переполнения стека, так как ей может потребоваться сделать $O(N)$ рекурсивных вызовов. В улучшенных реализациях, в которых рекурсивный вызов происходит только для сортировки меньшей из двух частей массива, глубина рекурсии гарантированно не превысит $O(\log_2 N)$;

в) неустойчив, если требуется устойчивость – приходится расширять ключ.

### 4.3 Примеры быстрой сортировки с различными ключами

***Пример 1.*** Рассмотрим способы сортировки методом `QuickSort` массива точек на декартовой плоскости, задаваемых координатрами `x` и `y`. Для этого определим структуру `Point`, хранящую координаты точки на плоскости, с полями `x` и `y`, а также имеющую функцию `CompareTo` сравнения двух точек. Выберем ключевым полем сортировки координату `x`. Для сортировки массива структур `Point` мы должны указать, что при сравнении двух структур надо сравнивать поля `x`. Для этого наследуем обобщённый интерфейс `IComparable<Point>`, определённый для структуры `Point`. Данный интерфейс для сравнения использует функцию `CompareTo`. Эта функция возвращает целое число, показывающее, как соотносится текущая структура `this` с некоторой другой структурой `p`. В исходном коде мы указали это как соотношение полей `this.x` и `p.x`.

Функция `CompareTo` работает по следующим правилам:

а) если `this.x < p.x`, то функция вернёт отрицательное число;

б) если `this.x = p.x`, то функция вернёт ноль;

в) если `this.x > p.x`, то функция вернёт число, большее нуля.

Структура `Point` имеет следующий вид

```
public struct Point : IComparable<Point>
{
    public int x;  // поле координаты x
    public int y;  // поле координаты y
    public int CompareTo(Point p)  // функция сравнения структур
    {
        return this.x.CompareTo(p.x);  // сравниваем по полям x
    }
}
```

Для того чтобы метод `QuickSort` являлся полиморфным и был примим к массиву любых типов данных, надо поместить его в обобщённых класс, например, `Library<T>`, который наследует обобщённый интерфейс `IComparable<T>`. В этом случае метод `QuickSort` сможет сравнивать любые типы данных с помощью функции `CompareTo`, которая видима через интерфейс `IComparable<T>`.

```
public class Library<T> where T : IComparable<T>
{
    public static void QuickSort(T[] A, int low, int high)
    {
        int i = low;  // запоминаем наименьший индекс
        int j = high; // запоминаем наибольший индекс
        T v = A[(low + high) >> 1]; // получаем опорный элемент
        do
        {
            while (A[i].CompareTo(v) < 0) ++i; // пока A[i] < v
            while (A[j].CompareTo(v) > 0) --j; // пока A[j] > v
            if (i <= j)         // если индексы не пересеклись
            {
                T temp = A[i]; // то меняем элементы местами
                A[i] = A[j];
                A[j] = temp;
                i++; j--; // индексы сдвигаем навстречу друг другу
            }
        } while (i < j); // пока индексы не встретились
        if (low < j) QuickSort(A, low, j); // сорт. левый массив
        if (i < high) QuickSort(A, i, high); // сорт. правый массив
    }
}
```

Зададим массив `m` из трёх структур `Point`, который математически можно описать в виде $\{(4,-2),(1,2),(2,-1)\}$, и выполним быструю сортировку с помощью вызова `Library<Point>.QuickSort`. Обратите внимание, что в этом вызове указан тип `Point`, с которым будет работать метод сортировки. Поэтому во время сравнения двух элементов массива он обратится к той функции `CompareTo`, которая описана именно в структуре `Point`.

```
static void Main()
{
    Point[] m = { new Point { x = 4, y = -2},     // первая точка
                  new Point { x = 1, y = 2 },     // вторая точка
                  new Point { x = 2, y = -1 } };  // третья точка
    Library<Point>.QuickSort(m, 0, m.Length - 1);
    foreach (Point e in m)
        Console.Write($"({e.x},{e.y}); ");  // вывод результата
    Console.ReadLine();
}
```

Результат сортировки

```
(1,2); (2,-1); (4,-2);
```

Обратите внимание: поле `y` не принимало участие в сортировке, и было прицеплено к полю `x` «бесплатным грузом». Для выбора поля `y` в качестве ключа, достаточно указать функцию сравнения по полям `y`

```
return this.y.CompareTo(p.y)
```

***Пример 2.*** Для сравнения точек по признаку суммы их координат `x+y` следует указать эту сумму в функции сравнения структур

```
return (this.x + this.y).CompareTo(p.x + p.y);
```

Результат сортировки

```
(2,-1); (4,-2); (1,2);
```

Заметьте, что в последнем примере ключом сортировки является функция над полями `x` и `y`.

***Пример 3.*** Сортировка по убыванию методом `Array.Sort` может быть выполнена путём модификации функции сравнения `CompareTo`. Достаточно изменить знак величины, которую она возвращает, чтобы получить нужный эффект

```
return -(this.x + this.y).CompareTo(p.x + p.y);
```

или поменять местами сами сравниваемые структуры

```
return (p.x + p.y).CompareTo(this.x + this.y);
```

Результат сортировки в обоих случаях будет одинаковым

```
(1,2); (4,-2); (2,-1);
```

Справедливости ради стоит отметить, что в документации Microsoft указан иной способ сортировки по убыванию – с помощью класса пользовательского компаратора `Compare`, который меняет порядок сортировки. Однако его реализация основывается на том же обмене местами сравниваемых структур.

***Пример 4.*** Модифицируепм функцию сравнения точек по полю `x`. В случае равенстива полей `x` следует сравнить поля `y`.

```
public int CompareTo(Point p)
{
    if (this.x.CompareTo(p.x) == 0) return this.y.CompareTo(p.y);
    return this.x.CompareTo(p.x);
}
```

Структуру Point можно описать классом Point. Для этого следует ключевое слово `struct` заменить ключевым словом `class`.

**Задача 1.** Разработать программу получения массива с заданной степенью упорядоченности элементов. Для этого создать упорядоченный массив и разупорядочить его путём обмена местами в каждом витке цикла двух элементов, занимающих случайные позиции. Какая часть элементов не участвовала в обмене – такова степень упорядоченности массива, например, если 90 % элементов упорядоченного массива не участвовали в обмене, то степень упорядоченности равна 0,9. Такой подход даёт приближённую оценку упорядоченности, но является самым простым.

**Задача 2.** Исследовать зависимость времени сортировки `QuickSort` от степени упорядоченности входного массива `m`. Построить график зависимости в MS Excel. Сделать вывод о естественности поведения встроенной сортировки.

**Контрольные вопросы**

1. Дайте определение понятию сортировки.
2. Что такое ключевое поле?
3. В чём суть устойчивой сортировки?
4. Каково отличие внутренней и внешней сортировки?
5. В чём заключается естественное поведение сортировки?
6. Для чего используется опорный элемент в быстрой сортировке?
7. Каковы способы задания опорного элемента?
8. Каково условие останова рекурсии в быстрой сортировке?
9. Какова оценка вычислительной сложности быстрой сортировки в среднем и худшем случае?
10. В чём заключается опасность быстрой сортировки?

# 5 Основные алгоритмы сортировки

## 5.1 Пирамидальная сортировка

*Пирамидальная сортировка* использует бинарное сортирующее дерево, которое ещё называют *пирамидой* или *двоичной кучей*. Алгоритм пирамидальной сортировки предложен Дж. Уильямсом в 1964 году.

*Двоичная куча* – это дерево, имеющее свойства:

– упорядоченность: значение в любой вершине не меньше (не больше), чем значения её потомков;

– глубина всех листьев отличается не более чем на 1 уровень;

– последний уровень заполняется слева направо без пропусков.

Если выполняется условие «не меньше», то куча называется максимальной (англ. max-heap), если выполняется условие «не больше» – минимальной (англ. min-heap). Высота кучи равна $\lfloor \log_2 N \rfloor$ рёбер, $N$ – количество вершин дерева, например, двоичная куча (рис. 5.1) имеет $N = 10$ вершин и высоту $\lfloor \log_2 10 \rfloor = 3$ ребра.

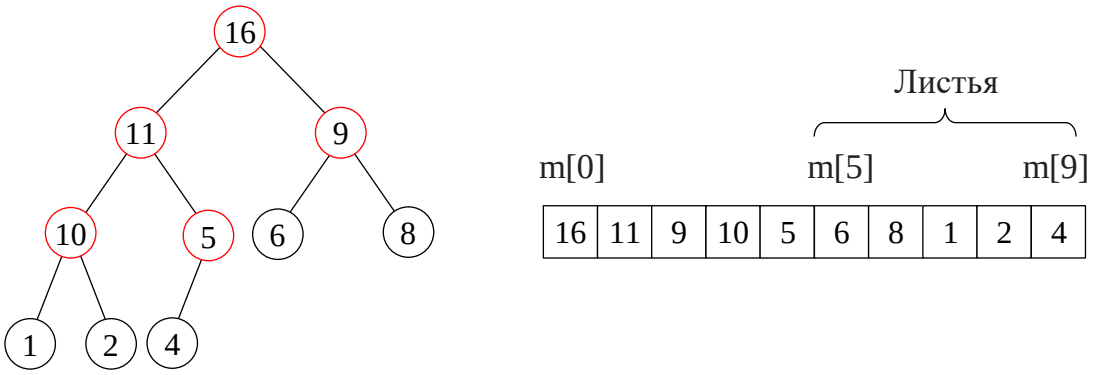



Рис. 5.1. Двоичная куча и массив для её хранения

Удобной структурой данных для хранения двоичной кучи является такой массив `m`, что `m[0]` – корень, а потомками элемента `m[i]` являются элементы `m[2i+1]` и `m[2i+2]`, например, вершина 9 имеет индекс `i=2` (рис. 5.1), следовательно, потомками этой вершины являются элементы массива с индексами 5 и 6, то есть элементы 6 и 8.

Вершины с потомками (красные) имеют индексы от 0 по $\left\lfloor \frac{N}{2} \right\rfloor - 1$, например, при $N = 10$ последней вершиной, имеющей листья (рис. 5.1), является вершина с индексом $i = \left\lfloor \frac{N}{2} \right\rfloor - 1 = \left\lfloor \frac{10}{2} \right\rfloor - 1 = 4$. Действительно,

`m[4]=5`. Листья дерева начинаются с индекса $\left\lfloor \frac{N}{2} \right\rfloor$. Первым листом, является элемент `m[5]=6` с индексом $i = \left\lfloor \frac{N}{2} \right\rfloor = \left\lfloor \frac{10}{2} \right\rfloor = 5$.

Над двоичной кучей можно выполнять следующие операции:

а) добавить элемент в кучу. Сложность $O(\log_2 n)$;

б) исключить максимальный элемент из кучи. Сложность $O(\log_2 n)$;

в) изменить значение любого элемента. Сложность $O(\log_2 n)$.

На основе этих операций допустимы следующие действия:

а) преобразовать неупорядоченный массив элементов в кучу. Сложность $O(n)$;

б) отсортировать массив путём превращения его бинарной кучи в сортированный массив. Сложность $O(n \log_2 n)$.

Ёмкостная сложность для всех вышеперечисленных операций и действий $O(1)$. Стоит отметить, что кроме двоичной кучи существуют биномиальные и фибоначчиевы кучи.

**Восстановление упорядоченности кучи**

В случае модификации кучи необходимо восстановить её упорядоченность. Для этого служит процедура `Heapify`. Она принимает на вход массив `m` и индекс `i` элемента, нарушившего упорядоченность. Процедура восстанавливает упорядоченность во всём поддереве, корнем которого является элемент `m[i]`. Левое и правое поддеревья этого элемента должны быть упорядочены.

Если элемент `m[i]` не меньше своих сыновей, то всё поддерево уже упорядочено. В противном случае меняем местами корень поддерева `m[i]` и наибольшего сына `m[j]`, после чего выполняем `Heapify` для индекса `j`.

```
static void Heapify(int[] m, int i, int size)
{
    int left = 2 * i + 1;  // индекс возможного левого потомка
    int right = 2 * i + 2; // индекс возможного правого потомка
    int max = i;           // i-й элемент – текущий максимальный
    if (left <= size && m[left] > m[max]) max = left;
    if (right <= size && m[right] > m[max]) max = right;
    if (max != i)          // если один из потомков больше
    {
        int temp = m[i];   // то обмен значений m[i] и m[max]
        m[i] = m[max];
        m[max] = temp;
        Heapify(m, max, size);  // упорядочиваем поддерево потомка
    }
}
```

Параметр `size` является максимальным индексом и равен `m.Length-1`. Однако дальше вы увидите, что пирамидальная сортировка

искусственно уменьшает `size`, оставляя `m.Length` неизменным. Поэтому удалить параметр `size` в процедуре `Heapify` нельзя.

Обмен значений можно определить на основе кортежей `(m[i],m[max]) = (m[max],m[i])`. Однако компилятор C# в обоих случаях генерирует код с использованием вспомогательной переменной.

Процедура `Heapify` выполняется за время $O(\log_2 N)$, так как она проходит вниз по дереву от элемента `m[i]` в худшем случае до его листа-потомка за $\lfloor \log_2(N-i) \rfloor$ шагов. Для корня дерева $i = 0$, следовательно, число шагов равно $\lfloor \log_2 N \rfloor$.

**Построение кучи**

Процедура `BuildHeap` предназначена для создания кучи из неупорядоченного массива за время $O(N)$. Если выполнить `Heapify` для всех вершин, имеющих потомков, т.е. для элементов массива с индексами от $i = \left\lfloor \frac{N}{2} \right\rfloor - 1$ по $i = 0$, то массив станет кучей. Стоит учесть, что листья дерева, т.е. вершины с индексами $i \geq \left\lfloor \frac{N}{2} \right\rfloor$, обрабатывать не надо, так как у них нет потомков, и они сами по себе являются кучами. Следовательно, процедура `Heapify` выполняется $\left\lfloor \frac{N}{2} \right\rfloor$ раз.

```
static void BuildHeap(int[] m)
{
    int size = m.Length - 1;    // максимальный индекс
    for (int i = m.Length / 2 - 1; i >= 0; i--)
        Heapify(m, i, size);
}
```

**Алгоритм пирамидальной сортировки**

Рассмотрим вариант алгоритма сортировки по возрастанию.

1. Построить из неупорядоченного массива двоичную кучу с помощью процедуры `BuildHeap`.

2. Поменять местами первый элемент массива (корень кучи является максимальным элементом) и последний элемент. При этом последний элемент массива станет наибольшим, а первый элемент, скорее всего, нарушит свойство упорядоченности двоичной кучи.

3. Исключить последний элемент из кучи путём декремента размера массива `size`.

4. Вызвать метод `Heapify` по отношению к новому корню для восстановления упорядоченности кучи.

5. Повторять пункты 2–4 до тех пор, пока размер массива больше одного элемента. Когда массив сократится до одного элемента, то обмен элементов не выполняется и сортировка считается законченной.

```
static void Heapsort(int[] m)
```

```
{
    BuildHeap(m);              // строим двоичную кучу
    int size = m.Length - 1;   // size - максимальный индекс
    for (int i = size; i > 0; i--)
    {
        int temp = m[0];       // обмен корня с последним листом
        m[0] = m[i];
        m[i] = temp;
        size--;                // «уменьшение» размера массива
        Heapify(m, 0, size);   // восстановление упорядоченности
    }
}
```

Выполним пирамидальную сортировку

```
static void Main()
{
    int[] m = { 6, 1, 7, 5, 4, 3, 9, 8, 2 };
    Heapsort(m);
    Console.WriteLine(string.Join(", ", m));
    Console.ReadLine();
}
```

Результат

```
1, 2, 3, 4, 5, 6, 7, 8, 9
```

**Оценка вычислительной сложности**

Сложность лучшего, среднего и худшего случая одинакова $O(n \log_2 n)$.

**Достоинства пирамидальной сортировки**

а) имеет доказанную оценку худшего случая $O(n \log_2 n)$;

б) сортирует на месте, то есть требует $O(1)$ дополнительной памяти.

**Недостатки пирамидальной сортировки**

а) сложна в реализации;

б) неустойчива, если требуется устойчивость, то приходится расширять ключ;

в) отсутствует естественное поведение;

г) на каждом шаге алгоритм делает выборку хаотично по всей длине массива, из-за чего плохо сочетается с кэшированием памяти;

д) требует произвольный доступ к элементам, поэтому медленно работает на связных списках и других структурах данных последовательного доступа;

е) из-за сложности алгоритма выигрыш получается только при больших значениях $n$.

## 5.2 Сортировка слиянием

*Сортировка слиянием* – упорядочение элементов коллекции путём разделения её на части вплоть до одного элемента и последующее слияние коротких упорядоченных коллекций в коллекцию исходной длины.

Кроме массивов сортировка слиянием упорядочивает списки, потоки и другие структуры данных с последовательным доступ к элементам.

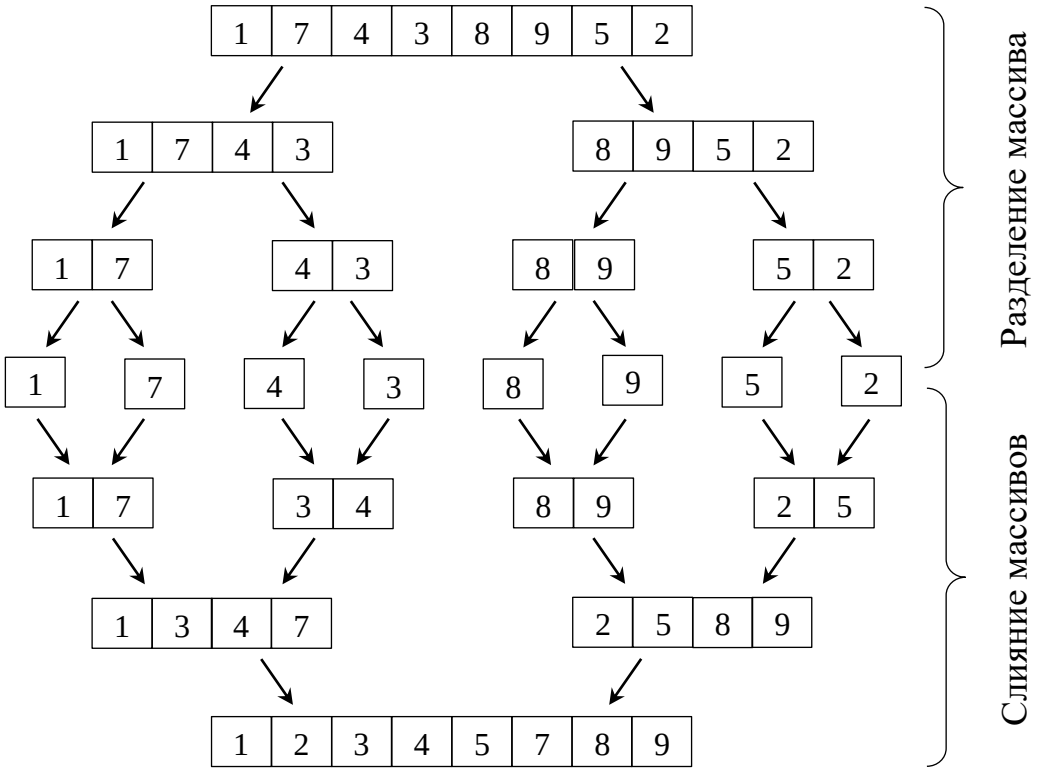



Рис. 5.2. Пример сортировки слиянием

**Алгоритм сортировки слиянием**

1. Найти индекс, разделяющий массив на две части по возможности равной длины (рис. 5.2). Если длина массива чётная, то размеры частей одинаковые, иначе одна часть длиннее другой на один элемент.

2. Применять п. 1 к каждой части массива до тех пор, пока длина части не станет равной единице.

3. Каждую пару частей массива слить в один новый упорядоченный массив. Повторять до тех пор, пока не останется единственный массив.

Перед слиянием двух упорядоченных массивов выделяется место в памяти для общего массива, размер которого равен сумме размеров сливаемых массивов. Общий массив заполняется по одному минимальному

элементу, выбираемому среди первых элементов сливаемых массивов (рис. 5.3). Когда один из двух массивов опустеет – остаток другого копируется в общий массив.

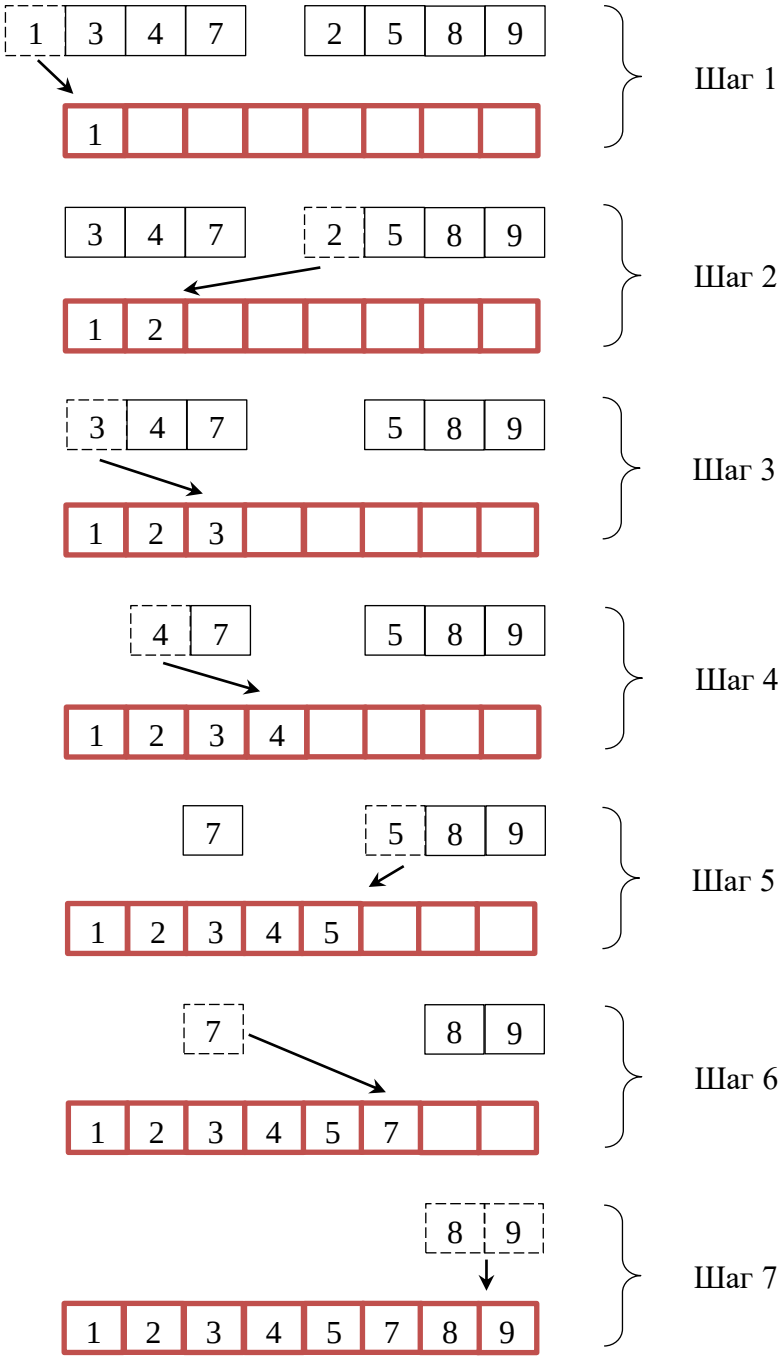



Рис. 5.3. Пример слияния двух упорядоченных массивов

Есть несколько вариантов функции слияния `Merge(a,b)` двух упорядоченных массивов `a` и `b` в общий упорядоченный массив `result`. Наиболее простой её вариант представлен ниже.

```
static int[] Merge(int[] a, int[] b)
{
    int[] result = new int[a.Length + b.Length];  // общий массив
    int i = 0, j = 0, k = 0;
    while (a.Length > i && b.Length > j) // массивы не пустые
        if (a[i] <= b[j])            // минимальный элемент
            result[k++] = a[i++];    // из left в result
        else result[k++] = b[j++];   // из right в result
    while (a.Length > i)        // массив right - пустой
        result[k++] = a[i++];   // остаток left в result
    while (b.Length > j)        // массив left - пустой
        result[k++] = b[j++];   // остаток right в result
    return result;              // возвращаем общий массив
}
```

На основе функции `Merge` строится рекурсивная функция `MergeSort(m,left,right)` сортировки части массива `m`, расположенной от левой границы индексов `left` до правой границы `right`.

```
static int[] MergeSort(int[] m, int left, int right)
{
    if (left >= right)              // реально left == right
    {
        int[] result = { m[left] };  // новый массив
        return result;
    }
        else
    {
        int mid = (left + right) >> 1;    // индекс сред. элемента
        int[] a = MergeSort(m, left, mid);  // сорт. левой части
        int[] b = MergeSort(m, mid+1, right);  // сорт. правой части
        return Merge(a, b);    // слияние массивов a и b
    }
}
```

Выполним сортировку слиянием, указав первый и последний индексы массива `m`.

```
static void Main()
{
    int[] m = { 1, 7, 4, 3, 8, 9, 5, 2 };
    int[] result = MergeSort(m, 0, m.Length-1);
    Console.WriteLine(string.Join(", ", result));
    Console.ReadLine();
}
```

Результат

```
1, 2, 3, 4, 5, 7, 8, 9
```

**Оценка вычислительной сложности**

Сложность лучшего, среднего и худшего случая равна $O(n \log_2 n)$.

**Достоинства сортировки слиянием:**

а) работает на структурах данных последовательного доступа;

б) хорошо сочетается с подкачкой и кэшированием памяти;

в) допускает параллельное выполнение: легко разделить задачи между процессорами, но трудно сделать так, чтобы другие процессоры взяли на себя работу, в случае если один процессор задержится;

г) не имеет «трудных» входных данных;

д) устойчивая: сохраняет порядок равных элементов.

**Недостатки сортировки слиянием:**

а) отсутствует естественное поведение;

б) требует дополнительную память размером, равным размеру исходного массива.

**Задача 1.** Исследовать зависимость времени пирамидальной сортировки от размера массива. Построить график зависимости в MS Excel. Исследовать естесственное поведение пирамидальной сортировки, используя массивы с различной степенью упорядоченности.

**Задача 2.** Исследовать зависимость времени сортировки слиянием от размера массива. Построить график зависимости в MS Excel. Исследовать естесственное поведение сортировки слиянием, используя массивы с различной степенью упорядоченности.

**Задача 3!** Разработать алгоритм и программу выборки *K* максимальных элементов из неупорядоченного массива с использованием минимальной кучи размером *K* для накопления выбираемых элементов. Добавлять в кучу новый элемент в случае, если он больше вершины кучи.

**Контрольные вопросы**

1. Каковы свойства двоичной кучи?
2. В чём отличие максимальной и минимальной двоичных куч?
3. В каком виде хранится двоичная куча в памяти компьютера?
4. Какие операции можно выполнять над двоичной кучей?
5. В чём заключается суть процедуры `Heapify`?
6. Какова вычислительная сложность пирамидальной сортировки?
7. Из каких шагов состоит алгоритм сортировки слиянием?
8. Какова вычислительная сложность сортировки слиянием?
9. Как выполняется слияние двух упорядоченных массивов?
10. Каковы достоинства и недостатки сортировки слиянием?

# 6 Алгоритмы сортировки без сравнений

## 6.1 Сортировка подсчётом

*Сортировка подсчётом* – алгоритм сортировки, в котором используются знания о диапазоне чисел сортируемой коллекции для подсчёта совпадающих элементов.

Рассмотрим алгоритм сортировки подсчётом массива `m`, содержащего целые числа в диапазоне от 0 до `max` (рис. 6.1).

1. Найти значение максимального элемента `max` массива `m`.
2. Создать массив счётчиков `counter` длиной `k=max+1`.
3. Для каждого `m[i]` инкрементировать счётчик `counter[m[i]]`.
4. Для каждого счётчика `counter[m[i]]` записать в массив `m` число `m[i]` столько раз, какова величина `counter[m[i]]`.

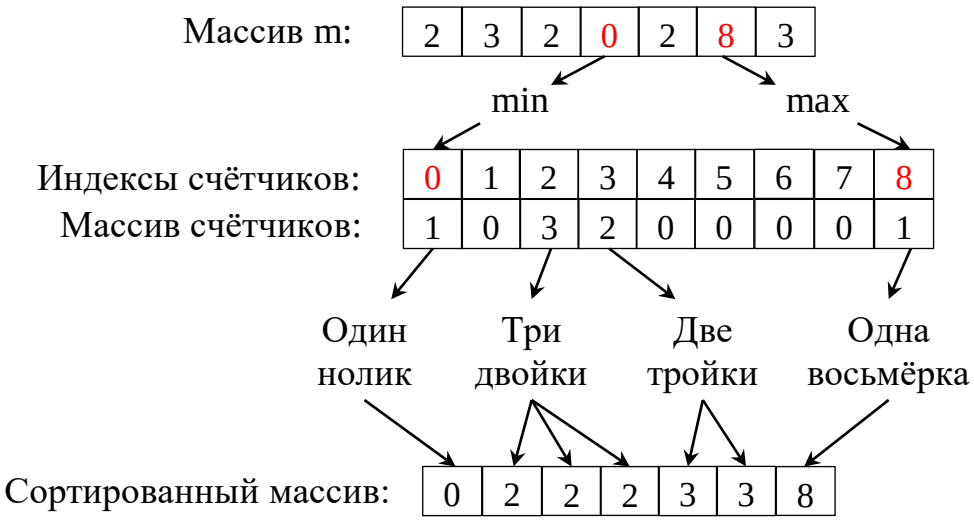



Рис. 6.1. Пример сортировки массива подсчётом элементов

В процедуре `CountingSort`, нет ни одной операции сравнения. Благодаря этому конвейер процессора не использует предсказание переходов и работает без перезагрузки. Сортируемый массив допускает отрицательные числа.

```
static void CountingSort(int[] m)
{
    int max = m.Max();          // макс. элемент
    int min = m.Min();          // мин. элемент
    int k = max - min + 1;      // нужное количество счётчиков
    int[] counter = new int[k];
    foreach (int e in m)
```

```
        counter[e - min]++;   // инкремент счётчика
    int p = 0;                // индекс элементов массива m
    for (int item = 0; item < k; item++)  // цикл по счётчикам
    {
        int e = counter[item];        // читаем счётчик
        for (int j = 0; j < e; j++)  // если счётчик > 0
            m[p++] = item + min;     // модифицируем массив m
    }
}
```

Выполним сортировку подсчётом

```
static void Main()
{
    int[] m = { 2, 3, -5, 2, -1, 8, 3 };
    CountingSort(m);
    Console.WriteLine(string.Join(", ", m));
    Console.ReadLine();
}
```

Результат

```
-5, -1, 2, 2, 3, 3, 8
```

**Оценка вычислительной сложности**

Определение максимального и минимального элементов массива `m` выполняется за $2(N-1)$ операций сравнения, $N$ – длина массива. Создание массива `counter` и инициализация его нулями выполняется за k = `max-min+1` шагов. Заполнение массива `counter` происходит за $N$ шагов. Перезапись массива `m` за $N$ шагов. Итого имеем точную оценку вычислительной сложности $4N + k$ и асимптотическую оценку $O(N + k)$.

**Границы применимости**

Сортировка подсчётом имеет низкую эффективность при $N \ll k$. В этом случае будет создаваться и обрабатываться гигантский массив счётчиков, во много раз превосходящий по размеру сортируемый массив.

**Достоинства сортировки подсчётом:**

а) работает на структурах данных последовательного доступа;

б) хорошо сочетается с подкачкой и кэшированием памяти;

в) не имеет «трудных» входных данных;

г) устойчивая: сохраняет порядок равных элементов.

**Недостатки сортировки подсчётом:**

а) требует дополнительную память размером $K = max - min + 1$;

б) работает только с целочисленными ключами. Однако под эту категорию подходят символы и строки, цвет в любой кодировке (RGB, CMYK и т.п.), дата и время.

## 6.2 Поразрядная сортировка

*Поразрядная сортировка* предназначена для сортировки коллекций, элементы которых имеют разряды. Алгоритм пригоден для сортировки любых объектов, которые можно поделить на «разряды», содержащие сравнимые значения. Например, так можно сортировать строки, являющиеся набором символов, и вообще произвольные значения в памяти, представленные в виде последовательности байтов.

Элементы массива делятся на блоки в соответствии со значениями старшего разряда. Затем блоки делятся на более мелкие блоки в соответствии со значениями следующего разряда. Так продолжается до тех пор, пока не будет произведено разделение блоков на последнем разряде. При этом блоки, содержащие один элемент, не разделяются.

Перед разделением на блоки элементы массива выравниваются по длине так, чтобы у них было одинаковое количество разрядов. Когда выравнивание осуществляется по правому разряду путём добавления к числу нулей слева или к строке пустых символов слева, то такое выравнивание называется «по правой стороне» или в сторону менее значащих разрядов (англ. least significant digit, LSD). Например, числа 9, 82, 253 после LSD выравнивания имеют вид 009, 082, 253. А строки «я», «фа», «вгд» после LSD выравнивания посредством добавления нулевых байтов слева имеют вид «°°я», «°фа», «вгд».

С другой стороны, когда выравнивание делается по левому разряду путём добавления нулей справа к числу или пустых символов справа к строке, то такое выравнивание называется «по левой стороне» или в сторону наиболее значащих разрядов (англ. most significant digit, MSD). Например, числа 9, 82, 253 после MSD выравнивания имеют вид 900, 820, 253. А строки «я», «фа», «вгд» после MSD выравнивания посредством добавления нулевых байтов справа имеют вид «я°°», «фа°», «вгд».

Важно отметить, что в памяти компьютера выравнивание чисел на самом деле не производится, так как они уже выравнены посредством объявления типа данных – каждое число массива занимает то количество байтов, которое соответствует объявленному типу данных. Поэтому все числа имеют LSD выравнивание. Строки также не выравниваются, но при сортировке полагается, что они выровнены по первому символу, т.е. согласно MSD. Поэтому короткие строки считаются меньше длинных, имеющих тот же префикс. Например, строка «я°» больше строки «аа», но меньше строки «яр». Если бы строки выравнивались по правилу LSD, то строка «°я» стала бы меньше строки «аа» и в словаре размещалась выше.

Сортировка чисел делается с применением LSD выравнивания. Поэтому порядок чисел после LSD сортировки уместен для чисел: 9, 82, 253, но не уместен для строк: «а», «я», «аа», «фа», «вгд».

Сортировка строк делается с применением MSD выравнивания. Поэтому порядок строк после MSD сортировки уместен для строк: «а», «аа», «вгд», «фа», «я», но не уместен для чисел: 253, 82, 9.

**Рекурсивный алгоритм поразрядной LSD сортировки чисел**

1. Установить номер текущего разряда элементов массива. Для LSD – старший (левый) разряд.

2. Если массив содержит один элемент, то он не разделяется. Если массив содержит больше одного элемента, то разделить его на два дочерних массива. Для этого получить значения текущих разрядов элементов массива. Элементы с текущим разрядом, равным нулю, поместить в первый дочерний массив, а с текущим разрядом, равным единице, – во второй дочерний массив.

3. Если текущий разряд был последним, то перейти к п.4, иначе – инкрементировать номер текущего разряда и рекурсивно вызвать п.2 для каждого дочернего массива.

4. Собрать элементы дочерних массивов в общий массив.

**LSD сортировка массива чисел по возрастанию**

Возьмём для примера массив трёхбитных чисел {7,3,2,0,1,6,4}. Разделение массива будет проходить за три шага (рис. 6.2), т.к. числа трёхбитные. На четвёртом шаге будет собран сортированный массив.

1. Исходный массив разделяется на две части. Разделяющим признаком является значение старшего разряда. В левый массив попадают числа, имеющие нулевой старший разряд, в правый массив – числа с единичным старшим разрядом.

2. Каждый полученный на первом шаге дочерний массив в свою очередь разделяется на две части. Разделяющим признаком является значение среднего разряда. В левую часть попадают числа, имеющие нулевой средний разряд, в правую часть – числа с единичным средним разрядом.

3. Каждый полученный на втором шаге дочерний массив разделяется на две части. Разделяющим признаком является значение младшего разряда. В левую часть попадают числа, имеющие нулевой младший разряд, в правую часть – числа с единичным младшим разрядом.

4. Собираем все полученные одноэлементные массивы в упорядоченный общий массив.

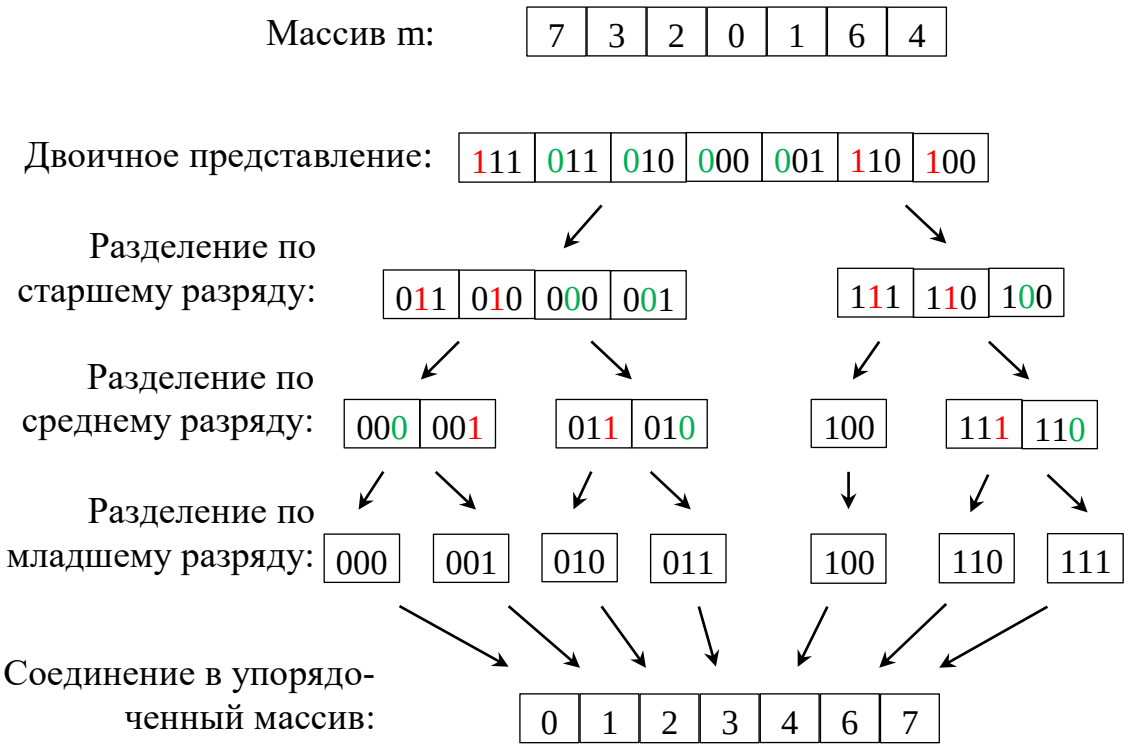



Рис. 6.2. Пример поразрядной LSD сортировки массива чисел

Функция `RadixSort` реализует рекурсивной алгоритм LSD сортировки беззнаковых чисел. Её особенностью является использование списков вместо массивов. Это связано с тем, что мы априори не знаем размеры частей, на которые разделится массив, и потому не можем объявить размеры дочерних подмассивов. Поэтому предлагается использовать списки `List<uint>`, т.к. их размер указывать не надо. Эти списки реализованы динамическими массивами и, следовательно, имеют тот же функционал, что и массивы. Это облегчает разработку программ. В ходе выполнения процедуры список `m` разделяется на два дочерних списка `a` и `b`, которые после своей сортировки сливаются обратно в список `a`. Дном рекурсии является состояние, когда список содержит не более одного элемента или маска, выделяющая разряды, равна нулю.

Первым параметром функция `RadixSort` принимает сортируемый список, вторым параметром – маску для выделения разряда, с которого начнётся разделение массива.

```
static List<uint> RadixSort(List<uint> m, uint p)
{
    if (m.Count <= 1 || p == 0) return m;    // дно рекурсии
    List<uint> a = new List<uint>();   // объявляем левый список
```

```
    List<uint> b = new List<uint>();   // объявляем правый список
    foreach (uint e in m)    //  разделение списка
        if ((e & p) == 0) a.Add(e);  // число в левый список
        else b.Add(e);               // число в правый список
    p >>= 1;          // сдвиг маски
    List<uint> a1 = RadixSort(a, p);  // сортируем левый список
    List<uint> b1 = RadixSort(b, p);  // сортируем правый список
    foreach (uint e in b1)
        a1.Add(e);  // соединение списков
    return a1;
}
```

Выполним поразрядную сортировку

```
static void Main()
{
    List<uint> m = new List<uint> { 7, 3, 2, 0, 1, 6, 4 };
    uint p = 0x80000000; // маска старшего бита 32-разрядных чисел
    List<uint> m1 = RadixSort(m, p); // вызов поразрядной сортировки
    Console.WriteLine(string.Join(", ", m1));
    Console.ReadLine();
}
```

Результат

```
0 1 2 3 4 6 7
```

Для поразрядной сортировки знаковых чисел необходимо вначале разделить числа по знаковому разряду на коллекции отрицательных и положительных чисел. Потом сдвинуть вправо маску разряда и применить к этим коллекциям функцию RadixSort. В конце не забудьте присоединить к коллекции отрицаельных чисел коллекцию положительных чисел.

**MSD сортировка массива строк по возрастанию**

Рассмотрим пример сортировки массива строк {«а», «яга», «я», «ай», «фа», «фу», «фас»} (рис. 6.3). Максимальная длина строки – три символа, поэтому разделение массива будет выполнено за три шага. При тестировании коротких слов, у которых нет второй или третьей буквы, код отсутствующей буквы считается равным нулю.

В отличие от сортировки чисел, в которой массив разделяется всего на два дочерних массива, при сортировке строк, массив может разделяться на 256 дочерних массивов, если используется кодировка ASCII. При использовании кодировки Unicode число дочерних массивов может возрасти многократно. Поэтому массив стоит разделять на массив массивов, а ещё лучше – список массивов.

Сортировка строк может проводиться не по символам, а по битам кодов символов, подобно сортировке чисел. Такой подход исключит обработку массива массивов, но повлечёт увеличение числа витков цикла.

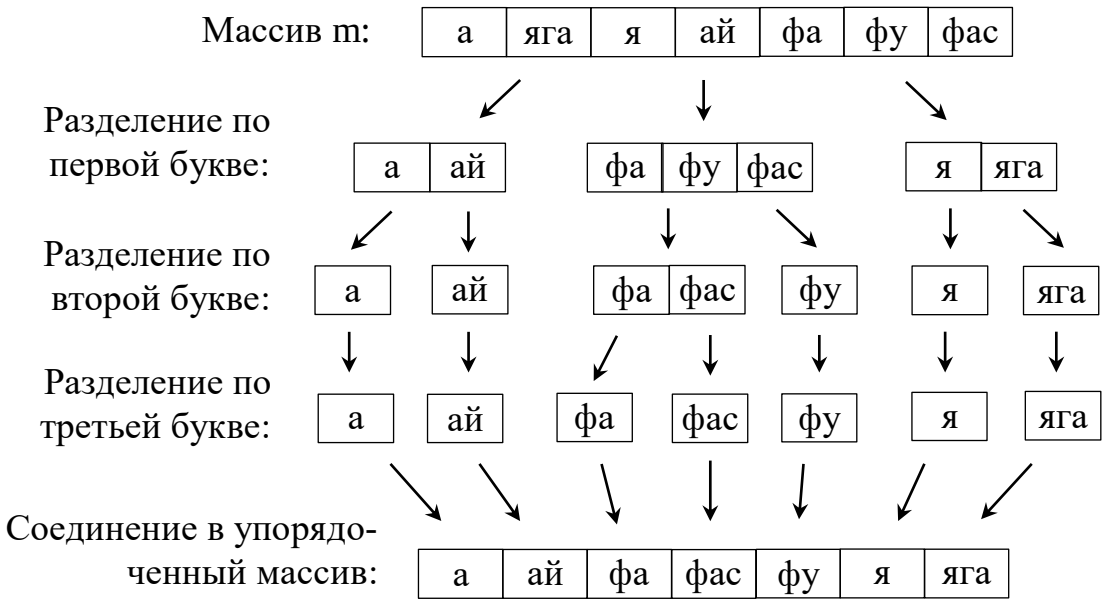



Рис. 6.3. Пример поразрядной сортировки массива строк

**Оценка вычислительной сложности**

Лучший, средний и худший случаи имеют одинаковую сложность $O(N \cdot K)$, $N$ – длина сортируемого массива, $K$ – разрядность элементов.

**Достоинства поразрядной сортировки:**

а) работает на структурах данных последовательного доступа;

б) хорошо сочетается с подкачкой и кэшированием памяти;

в) работает в параллельном варианте – легко разбить задачи между процессорами;

г) не имеет «трудных» входных данных;

д) устойчивая: сохраняет порядок равных элементов.

**Недостатки поразрядной сортировки:**

а) отсутствует естественность поведения;

б) требует дополнительную память размером, равным размеру исходного массива.

### 6.3 Блочная (карманная или корзинная) сортировка

*Блочная сортировка* – алгоритм сортировки, в котором сортируемые элементы помещаются в отдельные блоки (карманы, корзины) так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше, чем в предыдущем. После этого каждый блок сортируется отдельно: либо рекурсивно тем же методом, либо другим. Затем элементы собираются обратно в массив.

Идея алгоритма основана на том, что если значения чисел в массиве распределены примерно равномерно, то в каждую корзину попадёт небольшое количество чисел. Число корзин и их размер определяются на основании количества элементов в массиве, а также минимального и максимального элементов.

**Пример корзинной сортировки по возрастанию**

Массив {29,25,3,49,9,37,21,43} имеет *min* = 3 и *max* = 49. На первом шаге этот массив с диапазоном [3…49] можно разделить, например, на пять корзин с диапазонами: [0…9], [10…19], [20…29], [30…39], [40…49] (рис. 6.4).

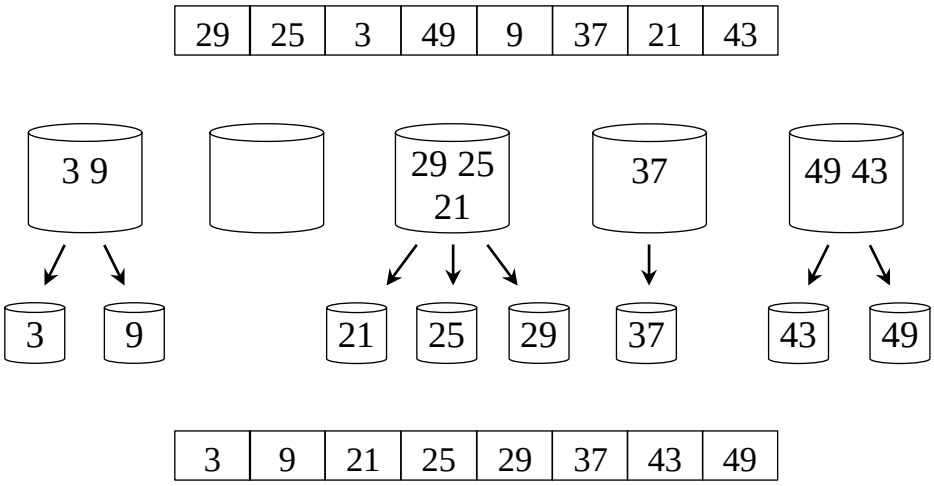



Рис. 6.4. Пример корзинной сортировки массива

На втором шаге каждую корзину можно разделить, например, на 5 маленьких корзинок, каждая с длиной диапазона равной 2, например, корзина с диапазоном [20…29] разделится на корзинки с диапазонами [20…21], [22…23], [24…25], [26…27] и [28…29]. Стоит заметить, что на самом деле диапазон длиной 10 дальше не разделяют, а попавшие в него числа, сортируют каким-нибудь простым алгоритмом, не требующим дополнительной памяти. К таковым можно отнести сортировку вставками или выборкой.

Так как мы априори не знаем количество элементов, которые попадут в ту или иную корзину, то использовать массивы для описания корзин будет неэффективно: неизвестны размеры массивов. Вместо массивов воспользуемся списками `List<int>`.

Входными параметрами метода `BucketSort` служат сортируемый список `m` и количество корзин `d`.

```
static void BucketSort(List<int> m, uint d)
{
    if (m.Count <= 1) return;  // корзина с одним числом или пустая
    if (m.Count == 2)  // корзина с двумя числами, упорядочиваем:
    {
        if (m[0] > m[1])
        { int t = m[0]; m[0] = m[1]; m[1] = t; }  // обмен
        return;
    }
    int min = m.Min(); // минимальный элемент
    int max = m.Max(); // максимальный элемент
    if (min == max) return;  // все числа в корзине одинаковы
    uint delta = (uint)(max - min + 1);  // диапазон корзины
    // если диапазон < числа корзин, то число корзин = диапазону:
    if (delta < d) d = delta;
    uint divisor = delta / d;  // целочисленное деление
    if (delta % d > 0) divisor++; // размер корзин
    List<int>[] buckets = new List<int>[d];  // массив корзин
    for (int i = 0; i < d; i++)
        buckets[i] = new List<int>();  // инициализация корзин
    foreach (int n in m)  // распределяем элементы по корзинам
    {   // индекс корзины для элемента n:
        uint i = (uint)((n - min) / divisor);
        buckets[i].Add(n);  // помещаем элемент в корзину
    }
    for (int i = 0; i < d; i++)
        BucketSort(buckets[i], d); // рекурсивно сортируем
    int k = 0;  // индекс для общей корзины
    for (int i = 0; i < d; i++)
        for (int j = 0; j < buckets[i].Count; j++)
            m[k++] = buckets[i][j];  // соединяем корзины в общую
}
```

На дне рекурсии проверяются условия:

а) если в корзине осталось не более одного числа, то выполняется возврат из рекурсии;

б) если в корзине осталось два числа, то программа их упорядочивает на месте без рекурсивного вызова и выполняет возврат из рекурсии.

После этого определяется *диапазон списка* в виде разности между максимальным и минимальным элементами списка. Размер корзин вычисляется целочисленным отношением диапазона списка к количеству корзин и последующим округлением вверх. Округление вверх выполняется инкрементом частного в случае ненулевого остатка деления. Массив корзин `List<int> buckets` инициализируется пустыми корзинами.

Основной цикл `foreach`, пробегая по всем элементам списка, определяет номер корзины `i`, в которую надо поместить элемент `n`

```
uint i = (uint)((n - min) / divisor);
```

после чего помещает этот элемент в корзину `buckets[i]`

```
buckets[i].Add(n);
```

Обратите внимание, что номер корзины для конкретного элемента определяется не сравнением элемента с границами диапазона корзины, а арифметически: целочисленным делением элемента на размер корзины. После распределения элементов по корзинам для каждой корзины рекурсивно вызывается метод сортировки. Число корзин `d` при углублении в рекурсию не изменяется, что является недостатком предлагаемого варианта программы. В конце программа соединяет корзины в общий список.

Выполним корзинную сортировку, установив постоянное количество корзин, равное 5

```
static void Main()
{
    List<int> m = new List<int> { 29, 25, 3, 49, 9, 37, 21, 43 };
    uint d = 5;  // число корзин
    BucketSort(m, d);
    Console.WriteLine(string.Join(", ", m));
    Console.ReadLine();
}
```

Результат

```
3, 9, 21, 25, 29, 37, 43, 49
```

**Превращения алгоритма**

Рекурсивное использование `BucketSort` превращает данный алгоритм в поразрядную сортировку с заданным числом разрядов `d`. В случае, когда массив делится ровно на две корзины, алгоритм соответствует быстрой сортировке с возможно неудачным выбором опорного элемента.

**Улучшения**

Параметр `d` изначально следует делать как можно больше, но на каждом шаге его надо уменьшать. В целом при большом значении `d` алгоритм эффективнее, чем при малом.

Размер корзины на каждом шаге можно определять выражением $\lceil\sqrt{N}\rceil$, $N$ – длина массива. Эта эвристика на практике показывает хорошие результаты.

**Оценка вычислительной сложности**

*Лучший случай* – равномерное или близкое к нему распределение элементов массива. Оценка сложности $O(N)$.

*Средний и худший случаи* возникают при крайне неравномерном распределении значений элементов массива. Это приводит к деградации

алгоритма: элементы массива распределяются не на `d` корзин, а почти все попадают в одну корзину. Однако асимптотическая оценка сложности остаётся прежней $O(N)$.

**Достоинства корзинной сортировки:**

а) работает на структурах данных последовательного доступа;

б) хорошо сочетается с подкачкой и кэшированием памяти;

в) работает в параллельном варианте: легко разбить задачи между процессорами;

г) устойчивая: сохраняет порядок равных элементов.

**Недостатки корзинной сортировки:**

а) отсутствует естественное поведение;

б) имеет «трудные» входные данные;

в) требует дополнительную память с размером исходного массива.

**Задача 1.** Для заданного количество $N = 10^7$ элементов случайного массива построить в одной координатной системе экспериментальные графики зависимости времени сортировки подсчётом и времени встроенной сортировки `Array.Sort` от восьми равноотстоящих друг от друга значений диапазона этого массива, взятых на отрезке $K = [10^2, 10^8]$.

**Задача 2.** Исследовать зависимость времени поразрядной сортировки массива знаковых случайных чисел от разрядности его элементов: sbyte, short, int и long.

**Контрольные вопросы**

1. Какие существуют алгоритмы сортировки без сравнений?
2. В чём преимущество алгоритмов сортировки без сравнений по отношению к другим алгоритмам?
3. Каковы границы применимости сортировки подсчётом?
4. Какое ограничение накладывается на ключ при сортировке подсчётом?
5. В чём суть поразрядной сортировки?
6. Раскройте понятия LSD и MSD поразрядной сортировки.
7. Какова вычислительная сложность поразрядной сортировки?
8. Каковы превращения алгоритма корзинной сортировки?

# 7 Односвязные списки

*Абстрактная структура данных* (АСД) – способ описания информации и операций над ней. АСД в программе выражаются с помощью абстрактного типа данных (АТД), который виден пользователю, и его реализации, которая скрыта внутри класса, описывающего АСД. *Абстрактный тип данных* – удобный для человека способ описания данных и операций над ними. Примером АТД являются как целые и вещественные числа, так и деревья, графы и списки, а также операции над ними. Алгоритмы выполнения этих операций скрыты от пользователя. Лишь часть программистов знает, как на самом деле представляются вещественные числа в памяти компьютера и выполняются операции умножения и деления таких чисел на низком уровне. В этой главе мы узнаем внутреннее представление односвязных списков и набор операций над ними.

Связные списки относятся к динамическим структурам. *Динамическая структура* – АСД с нелокально расположенными в памяти элементами произвольного размера. Память выделяется динамической структуре по мере необходимости в ходе работы программы.

## 7.1 Односвязные объекты

*Односвязный объект* – объект, содержащий поле со ссылкой на объект этого же типа. Как правило, такой объект содержит ещё хотя бы одно поле данных (рис. 7.1). Эти объекты служат элементами для построения различных односвязных АСД: односвязных списков, стеков, очередей. Ссылкой на АСД является ссылка на её первый элемент. Последний элемент АСД должен иметь пустую ссылку. *Пустая ссылка* обозначается *null* и не ссылается ни на что.

В простейшем варианте односвязный объект описывается классом `Node`.

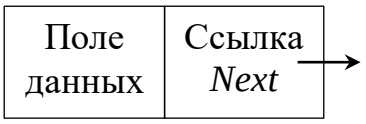



Рис. 7.1. Структура элемента *Node*

```
class Node
{
    public int Value;
    public Node Next;
}
```

Здесь поле данных описывается значением `int Value`. Поле ссылки на следующий элемент описывается значением `Node Next`, где `Node` – тип элемента, а `Next` – имя ссылки. В этом варианте класса `Node` оба поля публичные и имеют доступ по чтению и записи.

Для конструирования одного объекта со значением поля `Value=3` надо воспользоваться конструктором `new`

```
Node n = new Node { Value = 3 };
```

Этот конструктор является конструктором по умолчанию. Переменная `n` получает ссылку на объект типа `Node`, в котором полю `Value` присвоено значение 3, а поле `Next` имеет значение по умолчанию `null` (пустая ссылка).

Односвязные объекты можно соединять, создавая различные АСД, а также удалять их из АСД или вставлять в заданную позицию, например, цепочка из трёх элементов 3, 5, 7 содержит три объекта типа `Node` (рис. 7.2). Над каждым объектом указана воображаемая ссылка на него. Ссылки получают своё значение при загрузке программы в память. Первый элемент имеет поле данных, хранящее число 3, и ссылку на второй элемент *b*. Второй элемент включает поле данных с числом 5 и ссылку на третий элемент *c*. Третий элемент хранит число 7 и пустую ссылку *null*.

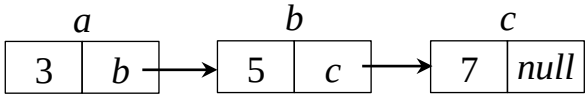



Рис. 7.2. Цепочка из трёх элементов

Ссылкой на всю цепочку элементов является ссылка на её первый элемент *a*. Только посредством этой ссылки мы имеем доступ к первому и всем остальным элементам цепочки.

Для формирования объектов с произвольным типом данных надо использовать обобщённый класс, т.е. класс `Node<T>`, которому в угловых скобках передаётся тип данных `T`, задаваемый программистом при вызове конструктора

```
class Node<T>
{
    public T Value;
    public Node<T> Next;
}
```

Для конструирования объекта `n` со значением `Value="abc"` и пустой ссылкой надо воспользоваться конструктором

```
Node<string> n = new Node<string> { Value = "abc" };
```

Для конструирования объекта `n1` со значением `Value="tr"` и ссылкой на объект `n`, подойдёт конструктор

```
Node<string> n1 = new Node<string> { Value = "tr", Next = n };
```

Сейчас у нас получился список `"tr"`→`"abc"`→`null`. Ссылкой на этот список является значение переменной `n1`.

**Соединение объектов**

Рассмотрим примеры создания цепочки путём соединения «вручную» объектов в различном порядке. При этом ссылки на все объекты будут храниться в программе в виде переменных.

***Пример 1.*** Создадим цепочку `3→5→7→null`, добавляя элементы 3, 5, 7 в естественном порядке: справа.

```
class Node<T>
{
    public T Value;
    public Node<T> Next;
}
static void Main()
{
    Node<int> first = new Node<int> { Value = 3 };   // 3 → null
    Node<int> second = new Node<int> { Value = 5 };  // 5 → null
    first.Next = second;     // 3 → 5 -> null
    Node<int> third = new Node<int> { Value = 7 };   // 7 → null
    second.Next = third;     // 3 → 5 → 7 -> null
    Node<int> node = first;  // ссылка на цепочку
    while (node != null)
    {
        Console.Write(node.Value + " -> ");
        node = node.Next;  // ссылка на следующий объект
    }
    Console.Write("null");
    Console.ReadKey();
}
```

Результат выполнения

```
3 -> 5 -> 7 -> null
```

Цепочка растёт слева направо. Выводим её также слева направо: в порядке просмотра по ссылкам `Next`. Обратите внимание: в угловых скобках конструктора задаётся тип данных `int`.

***Пример 2.*** Создадим цепочку `7→5→3→null`, добавляя элементы 3, 5, 7 слева.

```
class Node<T>
{
    public T Value;
    public Node<T> Next;
}
static void Main()
{
    Node<int> first = new Node<int> { Value = 3 };  // 3 → null
    // 5 → 3 -> null:
    Node<int> second = new Node<int> { Value = 5, Next = first };
    // 7 → 5 -> 3 → null:
    Node<int> third = new Node<int> { Value = 7, Next = second };
    Node<int> node = third;  // ссылка на цепочку
    while (node != null)
```

```
    {
        Console.Write(node.Value + " -> ");
        node = node.Next;  // ссылка на следующий объект
    }
    Console.Write("null");
    Console.ReadKey();
}
```

Результат выполнения

```
7 -> 5 -> 3 -> null
```

Цепочка растёт справа налево, но выводим её слева направо: в порядке просмотра по ссылкам `Next`. При создании нового объекта в конструкторе мы устанавливаем его ссылку `Next` на следующий объект.

***Пример 3.*** Создадим цепочку `5→3→7→null`, добавляя новые элементы к первому с разных сторон.

```
class Node<T>
{
    public T Value;
    public Node<T> Next;
}
static void Main()
{
    Node<int> first = new Node<int> { Value = 3 };  // 3 → null
    // 5 -> 3 -> null:
    Node<int> second = new Node<int> { Value = 5, Next = first };
    Node<int> third = new Node<int> { Value = 7 };  // 7 → null
    first.Next = third;  // 5 → 3 → 7 → null
    Node<int> node = second;  // ссылка на цепочку
    while (node != null)
    {
        Console.Write(node.Value + " -> ");
        node = node.Next;  // ссылка на следующий объект
    }
    Console.Write("null");
    Console.ReadKey();
}
```

Результат выполнения

```
5 -> 3 -> 7 -> null
```

**Вставка объектов**

Чтобы вставить новый объект, надо знать ссылку на предыдущий. Для вставки элемента 7 между элементами 3 и 5 надо перекинуть ссылку с предшествующего объекта на вставляемый, т.е. с элемента 3 на элемент 7 (рис. 7.3). А ссылку вставляемого элемента 7 установить на следующий элемент 5. Жирными стрелками отмечены новые связи, пунктиром – удалённая ссылка.

***Пример 4.*** Создадим цепочку `3→5→null` и вставим элемент 7 внутрь неё, чтобы получилась новая цепочка `3→7→5→null`.

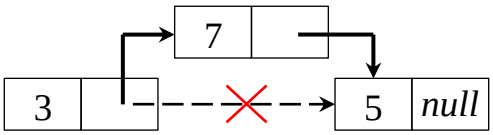



Рис. 7.3. Вставка объекта

```
class Node<T>
{
    public T Value;
    public Node<T> Next;
}
static void Main()
{
    Node<int> first = new Node<int> { Value = 3 };  // 3 → null
    Node<int> second = new Node<int> { Value = 5 }; // 5 → null
    first.Next = second;  // 3 → 5 → null
    Node<int> node = first;  // ссылка на цепочку
    while (node != null)  // вывод исходной цепочки
    {
        Console.Write(node.Value + " -> ");
        node = node.Next;  // ссылка на следующий объект
    }
    Console.WriteLine("null");
    // установка ссылки 7 → 5:
    Node<int> third = new Node<int> { Value = 7, Next = second };
    first.Next = third; // установка ссылки 3 → 7
    node = first;  // ссылка на новую цепочку
    while (node != null)  // вывод новой цепочки
    {
        Console.Write(node.Value + " -> ");
        node = node.Next;  // ссылка на следующий объект
    }
    Console.Write("null");
    Console.ReadKey();
}
```

Результат выполнения

```
3 -> 5 -> null
3 -> 7 -> 5 -> null
```

**Удаление объектов**

Чтобы удалить объект из цепочки, надо знать ссылку предыдущего объекта на удаляемый и ссылку удаляемого объекта на следующий. Для удаления объекта надо перекинуть ссылку с предшествующего объекта на следующий (рис. 7.4). Ссылку от удаляемого объекта на следующий можно не обнулять, так как сборщик мусора сам удалит

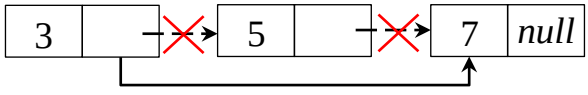



Рис. 7.4. Удаление объекта

объект, на который нет ссылки.

При удалении первого объекта цепочки достаточно знать ссылку на второй объект, если он есть. Ссылка на второй объект и будет являться ссылкой на новую цепочку.

Для удаления последнего объекта цепочки надо знать ссылку на предпоследний объект, чтобы в нём ссылку на последний объект обнулить значением `null`.

***Пример 5.*** Создание цепочки `3→5→7→null` и удаление среднего элемента `5` (рис. 7.4).

```
class Node<T>
{
    public T Value;
    public Node<T> Next;
}
static void Main()
{
    Node<int> first = new Node<int> { Value = 3 };  // 3 → null
    Node<int> second = new Node<int> { Value = 5 };  // 5 → null
    first.Next = second;  // 3 → 5 → null
    Node<int> third = new Node<int> { Value = 7 }; // 7 → null
    second.Next = third;  // 3 → 5 → 7 → null
    Node<int> node = first;  // ссылка на цепочку
    while (node != null)  // вывод исходной цепочки
    {
        Console.Write(node.Value + " -> ");
        node = node.Next;
    }
    Console.WriteLine("null");
    first.Next = third;  // 3 → 7 null (удаление 5)
    second.Next = null;  // 5 → null (необязательно)
    node = first;  // ссылка на цепочку
    while (node != null)  // вывод новой цепочки
    {
        Console.Write(node.Value + " -> ");
        node = node.Next;
    }
    Console.Write("null");
    Console.ReadKey();
}
```

Результат

```
3 -> 5 -> 7 -> null
3 -> 7 -> null
```

На самом деле в программе хранится ссылка на первый объект цепочки, а не на все её объекты, как в рассмотренных примерах. Все опера-

ции над объектами цепочки реализуются в отдельном классе, который обеспечивает доступ к конструктору, методам и функциям, и скрывает все внутренние манипуляции над объектами и служебные свойства цепочки.

### 7.2 Односвязные списки

*Односвязный список* – структура, построенная из односвязных объектов и допускающая добавление элементов в конец списка, а удаление – из произвольной позиции. Список хранит элементы в порядке их записи (рис. 7.5). При необходимости функционал класса, реализующего список, можно расширить функциями добавления элементов в любую позицию, а также чтения элементов из любой позиции.

Добавим в пустой список элементы 8, 1, 5 (рис. 7.5).

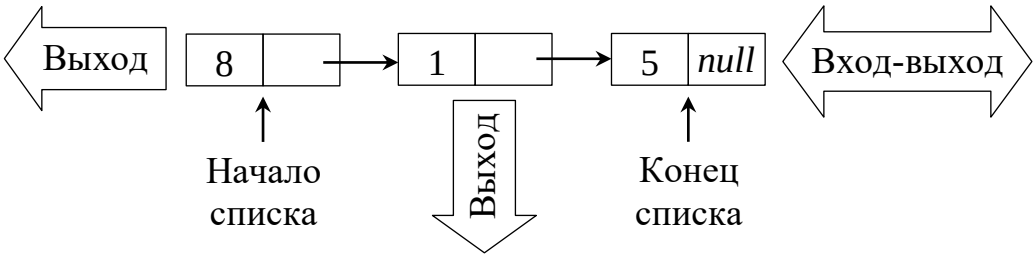



Рис. 7.5. Односвязный список

Добавление в полученный список элемента 6 изменит последний элемент списка (рис. 7.6).

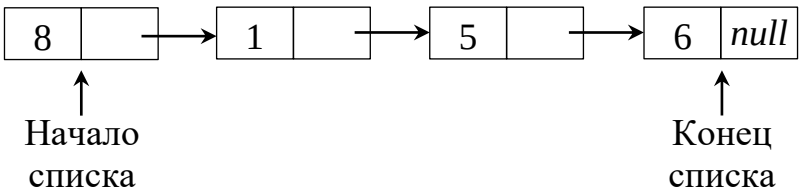



Рис. 7.6. Добавление элемента 6 в односвязный список

Для поддержки описанного функционала в классе списка надо хранить ссылку не только на первый элемент, но и на последний элемент. Ссылка на первый элемент служит для организации доступа по цепочке ссылок к любому элементу списка. Ссылка на последний элемент используется для быстрого добавления нового элемента в конец списка, чтобы не тратить время на прохождение от первого элемента к последнему.

В следующей редакции класса `Node<T>` добавлен полезный конструктор, в котором можно указывать значение `next`. По умолчанию `next = null`.

```
class Node<T>
{
    public T Value;  // поле Value
    public Node<T> Next; // ссылка на следующий элемент
    public Node(T value, Node<T> next = null)
    { Value = value; Next = next; } // конструктор нового элемента
}
```

Этот конструктор упрощает синтаксис его вызова, так как в качестве параметров достаточно указать значения полей без их имён.

В классе `SinglyLinkedList`, описывающем односвязные списки, хранится ссылка на первый элемент `First` и на последний элемент `Last`. Свойство `Size` содержит количество элементов в списке. Добавление элемента `Value` в конец списка выполняется методом `Add(Value)`. Удаление элемента, стоящего в позиции `position`, выполняется функцией `Value=Delete(position)`, где `Value` – значение удалённого элемента. Кроме этого класс снабжён функцией `this[uint position]` для чтения элемента в позиции `position`.

```
class SinglyLinkedList<T>
{
    public Node<T> First;  // ссылка на первый элемент списка
    public Node<T> Last;   // ссылка на последний элемент списка
    public int Size;        // размер списка
    public void Add(T value)
    {
        Node<T> node = new Node<T>(value);
        if (Last == null) { First = node; Last = node; } // первый
        else { Last.Next = node; Last = node; }  // остальные
        Size++;
    }
    public T Delete(uint position)
    {
        Node<T> Temp = First;
        if (position == 0 & Size == 1)  // удаление единственного
        {                               // элемента
            First = null;
            Last = null;
            Size--;
            return Temp.Value;
        }
        if (position == 0 & Size > 1)  // удаление первого элемента
        {
            First = First.Next;
```

```
            Temp.Next = null;
            Size--;
            return Temp.Value;
        }
        if (position < Size & Size > 0)  // удаление со второго
        {
            for (int i = 0; i < position - 1; i++)
            {
                Temp = Temp.Next; // бежим по ссылкам к предыдущему
            }
            Node<T> DelNode = Temp.Next;  // это удаляемый
            Temp.Next = DelNode.Next;  // перенаправили ссылку
            DelNode.Next = null;       // обнулили ссылку
            if (position == Size - 1) Last = Temp;  // если удаляем
            Size--;                                 // последний
            return DelNode.Value; // возврат удалённого элемента
        }
        else { throw new System.IndexOutOfRangeException("Индекс " +
position + " вне пределов диапазона списка"); }
    }
    public T this[uint position]
    {
        get
        {
            if (position < Size)
            {
                Node<T> Temp = First;
                for (int i = 0; i < position; ++i)
                    Temp = Temp.Next; // бежим по ссылкам к позиции
                return Temp.Value;
            }
            else { throw new System.IndexOutOfRangeException("Индекс
" + position + " вне пределов диапазона списка"); }
        }
    }
    public void PrintNodes(string Title)
    {
        Console.Write(Title + " ");
        Node<T> Temp = First;
        while (Temp != null)
        {
            Console.Write(" " + Temp.Value);
            Temp = Temp.Next;
        }
        Console.WriteLine();
    }
}
```

***Пример*** использования списка

```
static void Main(string[] args)
{
    SinglyLinkedList<int> list = new SinglyLinkedList<int>();
    list.Add(1);
    list.Add(2);
    list.Add(4);
    list.Add(5);
    list.Add(9);
    list.PrintNodes("Исходный список:");
    Console.WriteLine("Size = " + list.Size);
    try
    {
        Console.WriteLine("node[3] = " + list[3]);
        int V1 = list.Delete(1);
        Console.WriteLine("Удалили 1-й элемент: " + V1);
        int V2 = list.Delete(3);
        Console.WriteLine("Удалили 3-й элемент: " + V2);
    }
    catch (Exception error) { Console.WriteLine(error.Message); }
    list.PrintNodes("Модифицированный список:");
    Console.WriteLine("Size = " + list.Size);
    Console.ReadKey();
}
```

Результат

```
Исходный список:  1 2 4 5 9
Size = 5
node[3] = 5
Удалили 1-й элемент: 2
Удалили 3-й элемент: 9
Модифицированный список:  1 4 5
Size = 3
```

Если в список добавить всего два элемента, то при удалении третьего будет вызвано исключение

```
static void Main()
{
    SinglyLinkedList<int> node = new SinglyLinkedList<int>();
    node.Add(1);
    node.Add(2);
    node.PrintNodes("Исходный список:");
    Console.WriteLine("Size = " + node.Size);
    try
    {
        Console.WriteLine("node[1] = " + node[1]);
        int V1 = node.Delete(1);
        Console.WriteLine("Удалили 1-й элемент: " + V1);
        int V2 = node.Delete(3);
```

```
            Console.WriteLine("Удалили 3-й элемент: " + V2);
        }
        catch (Exception error) { Console.WriteLine(error.Message); }
        node.PrintNodes("Модифицированный список:");
        Console.WriteLine("Size = " + node.Size);
        Console.ReadKey();
}
```

Результат

```
Исходный список:  1 2
Size = 2
node[1] = 2
Удалили 1-й элемент: 2
Индекс 3 вне пределов диапазона списка
Модифицированный список:  1
Size = 1
```

### 7.3 Стек

*Стек* – абстрактная структура, построенная из односвязных объектов и действующая по принципу «первым пришёл – последним вышел» (англ. first input – last output, FILO).

Работа стека подобна функционированию магазина автомата Калашникова: патрон, задавленный в магазин последним, выстрелит первым. Стек хранит элементы в обратном порядке, т.е. при продвижении по цепочке ссылок с помощью поля `Next`, первым элементом стека будет последний записанный в стек, вторым – предпоследний элемент, а последним элементом будет тот, который записан в стек первым.

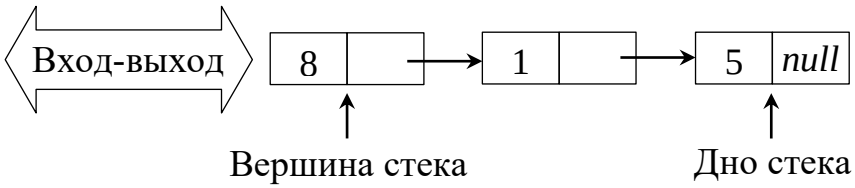



Рис. 7.7. Стек

Задавим в пустой стек элементы 5, 1, 8 (рис. 7.7). Последний добавленный элемент является вершиной стека. При добавлении в стек нового элемента 4, он становится вершиной стека (рис. 7.8), продавливая элемент 8 дальше.

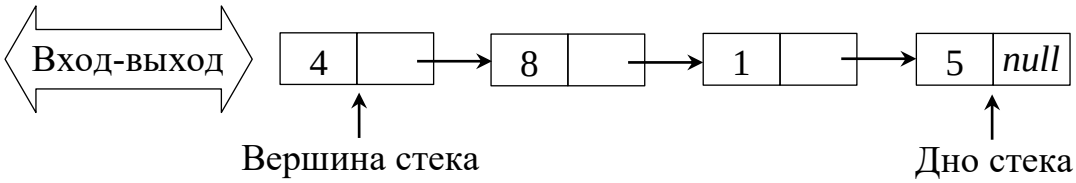



Рис. 7.8. Стек с добавленной вершиной

Стек растёт влево, а дно у него справа. При выталкивании элемента из стека всегда удаляется его вершина, в нашем случае число 4. Новой вершиной становится элемент с числом 8. Когда стек пустой, то есть является ссылкой `null`, удалить из него ничего нельзя.

Обобщённый класс `Stack<T>`, описывающий стек, хранит ссылку на вершину стека `Head`. Этого достаточно, так как в вершине хранится ссылка на второй элемент, во втором – на третий и так до дна стека. Также в переменной `Size` хранится количество элементов в стеке. Добавление элемента в стек обычно называется методом `Push(Value)`: задавить в стек значение `Value`. Удаление из стека называют функцией `Value=Pop()`: вытолкнуть из стека значение `Value`.

Реализация класса `Stack<T>` основана на структуре `Node<T>`.

```
class Node<T>
{
    public T Value;
    public Node<T> Next;
}
class Stack<T>
{
    public int Size { get; internal set; }  // размер стека
    public Node<T> Head { get; set; }       // вершина стека
    public void Push(T value)               // задавить в стек
    {
        Node<T> node = new Node<T>() { Value = value, Next = Head};
        Head = node;  // ссылка на новую вершину стека
        Size++;       // инкремент размера стека
    }
    public T Pop()    // вытолкнуть из стека
    {
        if (Head != null)
        {
            Node<T> Temp = Head; // ссылка на выталкиваемый элемент
            Head = Head.Next;    // ссылка на новую вершину стека
            Size--;              // декремент размера стека
            return Temp.Value;   // возврат вытолкнутого элемента
        }
```

```
        else { throw new System.Exception("Попытка вытолкнуть из пу-
стого стека"); }
    }
    public void PrintNodes(string Title)
    {
        Node<T> Temp = Head;
        Console.Write(Title + " ");
        while (Temp != null)
        {
            Console.Write(" " + Temp.Value);
            Temp = Temp.Next;
        }
        Console.WriteLine();
    }
}
```

Пример использования стека

```
static void Main()
{
    Stack<int> st = new Stack<int>(); // создание пустого стека
    st.Push(1);       // задавить 1 в стек
    st.Push(2);
    st.Push(4);
    st.Push(5);
    st.Push(9);
    st.PrintNodes("Исходный стек:");
    Console.WriteLine("Size = " + st.Size);  // размер стека
    try
    {
        int v1 = st.Pop();  // попытка вытолкнуть вершину 9
        Console.WriteLine("Вытолкнули элемент: " + v1);
        int v2 = st.Pop();  // попытка вытолкнуть вершину 5
        Console.WriteLine("Вытолкнули элемент: " + v2);
    }
    catch (Exception error) { Console.WriteLine(error.Message); }
    st.PrintNodes("Модифицированный стек:");
    Console.WriteLine("Size = " + st.Size);  // размер стека
    Console.ReadKey();
}
```

Результат

```
Исходный стек:  9 5 4 2 1
Size = 5
Вытолкнули элемент: 9
Вытолкнули элемент: 5
Модифицированный стек:  4 2 1
Size = 3
```

Если в стек задавить всего одно число, то при второй попытке вытолкнуть элемент из стека будет вызвано исключение

```
Исходный стек:  1
Size = 1
Вытолкнули элемент: 1
Попытка вытолкнуть элемент из пустого стека
Модифицированный стек:
Size = 0
```

Бывает, что стек изображают вертикально, тогда его вершину располагают вверху, а дно – внизу. В этом случае говорят, что стек растёт вверх, а его размер называют глубиной. Иногда размер стека ограничивают некоторой величиной, называемой *максимальным размером* (максимальной глубиной). Когда стек заполнен до максимума, то попытка задавить в него ещё один элемент вызывает исключительную ситуацию. Например, при попытке записи элемента в заполненный стек, выделенный операционной системой некоторому процессу, возникает исключительная ситуация под названием *переполнение стека* (англ. stack overflow).

В C# стек реализован классом `Stack<T>`.

### 7.4 Очередь

*Очередь* – абстрактная структура, построенная из односвязных объектов и действующая по принципу «первым пришёл – первым вышел» (англ. first input – first output, FIFO). Очередь строится на основе односвязных списков с операциями добавления элементов в конец очереди и удаления элемента, стоящего в начале очереди (рис. 7.9). Очередь хранит элементы в порядке их записи подобно списку.

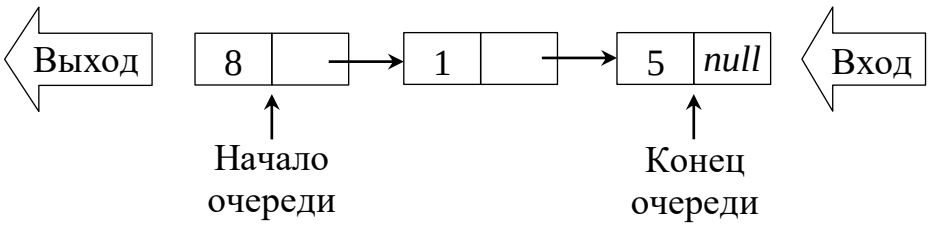



Рис. 7.9. Очередь

Длина очереди не ограничена. При необходимости можно зафиксировать максимальную длину очереди и модифицировать операцию добавления так, чтобы попытки добавления элемента в очередь максимальной длины были неуспешны.

В C# очередь реализована классом `Queue<T>` и хранит ссылки на первый `First` и последний `Last` элементы. Ссылка `First` служит для

удаления элемента из очереди. Ссылка `Last` используется для быстрого добавления элемента, чтобы не тратить время на прохождение от первого элемента к последнему по ссылкам. Добавление элемента `Value` в очередь выполняется методом `Enqueue(Value)`, удаление – с помощью функции `value = Dequeue()`, `value` – значение удалённого элемента.

```
class Node<T>
{
    public Node(T value, Node<T> next = null)
    { Value = value; Next = next; }  // конструктор нового элемента
    public T Value;      // поле Value
    public Node<T> Next; // ссылка на следующий элемент
}
class Queue<T>
{
    public int Size;        // размер очереди
    private Node<T> First;  // ссылка на первый элемент очереди
    private Node<T> Last;   // ссылка на последний элемент очереди
    public void Enqueue(T value)
    {
        Node<T> node = new Node<T>(value);
        if (Last == null) { First = node; Last = node; }
        else { Last.Next = node; Last = node; }
        Size++;
    }
    public T Dequeue()
    {
        Node<T> Temp = First;
        if (Size == 1)  // удаление единственного элемента
        {
            First = null;
            Last = null;
            Size--;
            return Temp.Value;
        }
        if (Size > 1)   // удаление первого элемента
        {
            First = First.Next;
            Temp.Next = null;
            Size--;
            return Temp.Value;
        }
        else { throw new System.Exception("Попытка удаления из пу-
стой очереди"); }
    }
public void PrintNodes(string Title)
    {
        Node<T> Temp = First;
```

```
        Console.Write(Title + " ");
        while (Temp != null)
        {
            Console.Write(" " + Temp.Value);
            Temp = Temp.Next;
        }
        Console.WriteLine();
    }
}
```

Пример использования очереди

```
static void Main(string[] args)
{
    Queue<int> que = new Queue<int>();
    que.Enqueue(1);
    que.Enqueue(2);
    que.Enqueue(4);
    que.Enqueue(5);
    que.Enqueue(9);
    que.PrintNodes("Исходная очередь:");
    Console.WriteLine("Size = " + que.Size);
    try
    {
        int v1 = que.Dequeue();
        Console.WriteLine("Удалили элемент: " + v1);
        int v2 = que.Dequeue();
        Console.WriteLine("Удалили элемент: " + v2);
    }
    catch (Exception error) { Console.WriteLine(error.Message); }
    que.PrintNodes("Модифицированная очередь:");
    Console.WriteLine("Size = " + que.Size);
    Console.ReadKey();
}
```

Результат

```
Исходная очередь:  1 2 4 5 9
Size = 5
Удалили элемент: 1
Удалили элемент: 2
Модифицированная очередь:  4 5 9
Size = 3
```

**Задача 1.** Разработать алгоритм и программу выборки $K$ наибольших элементов из односвязного списка `SinglyLinkedList<T>` с использованием пирамиды. Передвижение по списку реализовать по ссылкам `Next`. Список модифицировать запрещено. Доступ к элементам списка по индексу запрещён.

**Задача 2.** Разработать алгоритм и программу сортировки слиянием односвязных списков. Передвижение по списку реализовать по ссылкам `Next`. Доступ к элементам списка по индексу запрещён.

**Задача 3!** Разработать алгоритм и функцию проверки баланса скобочных последовательностей, например “([]<{}>)”, на основе использования стека. Функция должна возвращать булево значение `True` или `False`.

**Контрольные вопросы**

1. Раскройте понятие абстрактной структуры данных.
2. Раскройте понятие абстрактного типа данных.
3. Что такое динамическая структура данных?
4. Чем отличается динамическая структура данных от статической?
5. Как в языке C# реализуются динамические структуры данных?
6. Какие поля хранятся в односвязном объекте?
7. Что произойдёт при попытке удаления значения из пустого списка?
8. В чём отличие односвязного списка от стека?
9. С какой целью в классе односвязных списков хранятся ссылки на первый и последний элементы?
10. В чём заключается суть очереди?
11. Какой функционал имеет очередь?
12. В каком классе языка C# реализована очередь?
13. Возможно ли зафиксировать максимальную длину очереди?

# 8 Двусвязные списки

## 8.1 Общие сведения о двусвязных списках

*Двусвязный (двунаправленный) список* – структура, построенная из двусвязных объектов и допускающая добавление/удаление элементов в начало и в конец списка, а при необходимости – внутрь списка. *Двусвязный объект* – объект, содержащий две ссылки на объекты этого же типа. Такие объекты служат элементами для построения двусвязных структур данных: двусвязных списков, деков, барабанов.

Каждый элемент двусвязного списка представляет собой объект типа `Node` с полем данных, ссылкой на предыдущий элемент `Previous` и на следующий элемент `Next` (рис. 8.1). Первый элемент двусвязного списка должен иметь пустую ссылку `Previous`, а последний элемент – пустую ссылку `Next` (рис. 8.2).

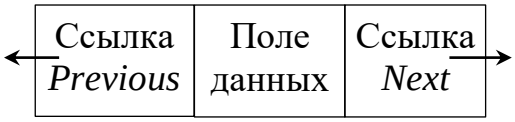



Рис. 8.1. Структура двусвязного объекта *Node*

Названия элементов «первый», «последний» или «начало», «конец» достаточно условны для двунаправленного списка, так как у него, по сути, все стороны равноправны. Несмотря на это договоримся левый край двунаправленного списка называть началом и его элемент – первым, а правый край – концом и его элемент – последним, чтобы в программе как-то отличать стороны двунаправленного списка.

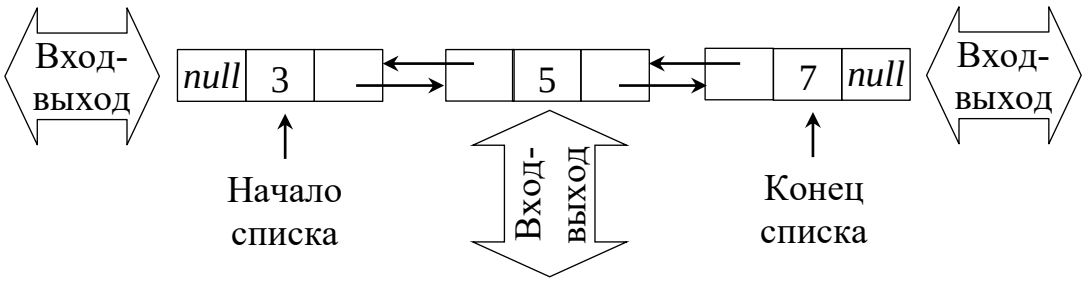



Рис. 8.2. Двунаправленный список из трёх элементов

В варианте с собственным конструктором двусвязный объект описывается обобщённым классом `Node<T>`.

```
public class Node<T>
{
    public T Value;           // поле Value
    public Node<T> Previous; // ссылка на предыдущий элемент
```

```
    public Node<T> Next;     // ссылка на следующий элемент
    public Node(T value, Node<T> previous = null, Node<T> next =
null)  // конструктор нового элемента
    { Value = value; Previous = previous; Next = next; }
}
```

Класс `DoublyLinkedList<T>` двунаправленных списков хранит ссылки на первый `First` и последний `Last` элементы, а также количество элементов `Size`. Кроме этого он содержит методы добавления элемента `value` в начало `Add2Begin(value)` и в конец списка `Add2End(value)`. Также этот класс снабжён тремя итераторами. `Iterator1()` пробегает все элементы списка слева направо. `Iterator2(init,next,cond)` просматривает элементы, стоящие на позициях от `init` с шагом `next` пока истинно условие `cond`. `Iterator3(list1,list2)` возвращает все пары элементов двух списков, стоящие на одинаковых позициях пока не закончится наикратчайший список.

```
class DoublyLinkedList<T>
{
    public uint Size;       // размер списка
    private Node<T> First;  // первый элемент списка
    private Node<T> Last;   // последний элемент списка
    public void Add2Begin(T value)
    {
        Node<T> node = new Node<T>(value, null, First);
        if (First == null) { First = node; Last = node; }
        else { First.Previous = node; First = node; }
        Size++;
    }
    public void Add2End(T value)
    {
        Node<T> node = new Node<T>(value, Last, null);
        if (Last == null) { First = node; Last = node; }
        else { Last.Next = node; Last = node; }
        Size++;
    }
    public IEnumerable<Node<T>> Iterator1()
    {
        Node<T> Temp = First;
        while (Temp != null)
        {
            yield return Temp;
            Temp = Temp.Next;
        }
    }
    public IEnumerable<Node<T>> Iterator2(int init,
           Func<int, int> next, Predicate<int> cond)
    {
```

```
        Node<T> Temp = First;
        int count = 0;    // счётчик
        if (Size > init)
            for (; count < init; count++)
                Temp = Temp.Next;  // идём к позиции init
        else
            yield break;
        while (cond(count))
        {
            yield return Temp;   // возвращаем от init с шагом next
            int d = next(count); // пока истинно условие cond
            for (; count < d; count++)
                if (Temp == null) yield break;
                else Temp = Temp.Next;  // идём к следующей позиции
        }
    }
    public static IEnumerable<(Node<T>, Node<T>)> Iterator3
           (DoublyLinkedList<T> list1, DoublyLinkedList<T> list2)
    {
        Node<T> Temp1 = list1.First;
        Node<T> Temp2 = list2.First;
        while(Temp1 != null & Temp2 != null)  // оба элемента есть
        {
            yield return (Temp1, Temp2);      // возвращаем пару
            Temp1 = Temp1.Next;
            Temp2 = Temp2.Next;
        }
    }
}
```

***Пример*** использования двунаправленных списков и их итераторов

```
static void Main(string[] args)
{
    DoublyLinkedList<int> M = new DoublyLinkedList<int>();
    M.Add2Begin(1);  // добавление в начало
    M.Add2Begin(9);
    M.Add2Begin(4);
    M.Add2End(5);    // добавление в конец
    M.Add2End(2);    // 4 9 1 5 2
    foreach (Node<int> m in M.Iterator1()) // вывод всего списка M
        Console.Write(m.Value + " ");
    Console.WriteLine("\nSize1 = " + M.Size);
    foreach (Node<int> m in M.Iterator2(0, i => i+2, i => i<5)) //
первый способ вызова итератора по чёт.позициям
        Console.Write(m.Value + " ");
    Console.WriteLine();
    IEnumerable<Node<int>> iter = M.Iterator2(0, i => i+2, i =>
i<5); // экземпляр итератора
    foreach (Node<int> m in iter) // второй способ вызова итератора
```

```
            Console.Write(m.Value + " ");
        Console.WriteLine();
        DoublyLinkedList<int> Q = new DoublyLinkedList<int>();
        Q.Add2End(1);  // добавление в начало
        Q.Add2End(2);
        Q.Add2End(4);
        Q.Add2End(5);
        foreach (Node<int> q in Q.Iterator1()) // вывод всего списка Q
            Console.Write(q.Value + " ");
        Console.WriteLine("\nSize2 = " + Q.Size);
        foreach ((Node<int>, Node<int>) t in  // 1 способ вывода пар
                            DoublyLinkedList<int>.Iterator3(M, Q))
            Console.Write("({0},{1}) ",t.Item1.Value,t.Item2.Value);
        Console.WriteLine();
        foreach ((Node<int> t1, Node<int> t2) in  // 2 способ
                            DoublyLinkedList<int>.Iterator3(M, Q))
            Console.Write("({0},{1}) ", t1.Value, t2.Value);
        Console.ReadLine();
}
```

Результат

```
4 9 1 5 2     // это список M
Size1 = 5
4 1 2         // элементы от 1 до 3
1 2 4 5       // это список Q
Size2 = 4
(4,1) (9,2) (1,4) (5,5)     // это пары элементов двух списков
```

**Достоинства связных списков:**

а) эффективная вставка и удаление элементов за время $O(N)$;

б) размер списка ограничен только объёмом памяти компьютера и разрядностью указателей;

в) динамическое добавление и удаление элементов.

**Недостатки связных списков:**

а) медленный доступ к элементу по его индексу за время $O(N)$;

б) расход дополнительной памяти на поля ссылок;

в) чтение элементов списка не кэшируется;

г) над связными списками, по сравнению с массивами, гораздо труднее проводить параллельные векторные операции, т.к. накладные расходы на перебор элементов снижают эффективность распараллеливания.

## 8.2 Деки

*Дек* – абстрактная структура, построенная из двусвязных объектов (рис. 8.3) и допускающая добавление/удаление элементов с обеих сторон. Добавлять объекты внутрь дека и удалять изнутри нельзя. Дек легко вооб-

разить в виде железнодорожного состава, который можно формировать и с начала, и с конца.

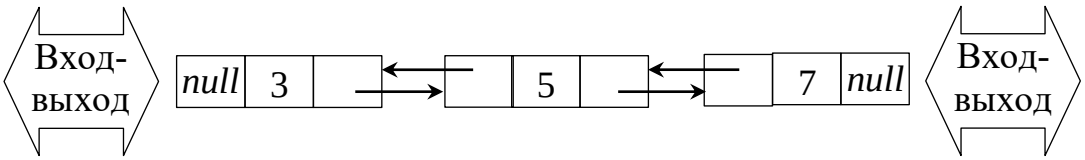



Рис. 8.3. Дек

На самом деле начала и конца у дека нет, ибо все стороны равноправны. Однако для различения сторон в программе будем называть левую сторону началом, а правую – концом.

Пример реализации класса `Deque<T>`, описывающего дек, использует вышеописанный класс `Node<T>`, методы добавления элемента в начало `PushFront(value)` и в конец дека `PushBack(value)`, а также функции удаления из начала `value=PopFront()` и из конца дека `value=PopBack()`.

```
class Deque<T>
{
    public uint Size;        // размер очереди
    private Node<T> First;   // первый элемент дека
    private Node<T> Last;    // последний элемент дека
    public void PushFront(T value)
    {
        Node<T> node = new Node<T>(value, null, First);
        if (First == null) { First = node; Last = node; }
        else { First.Previous = node; First = node; }
        Size++;
    }
    public void PushBack(T value)
    {
        Node<T> node = new Node<T>(value, Last, null);
        if (Last == null) { First = node; Last = node; }
        else { Last.Next = node; Last = node; }
        Size++;
    }
    public T PopFront()
    {
        Node<T> Temp = First;
        if (Size > 1)  // удаление левого элемента
        {
            First = Temp.Next;
            First.Previous = null;
            Temp.Next = null;
            Size--;
```

```
            return Temp.Value;
        }
        if (Size == 1)  // удаление единственного элемента
        {
            First = null;
            Last = null;
            Size--;
            return Temp.Value;
        }
        else { throw new System.Exception("Попытка удаления из пу-
стого дека"); }
    }
    public T PopBack()
    {
        Node<T> Temp = Last;
        if (Size > 1)  // удаление правого элемента
        {
            Last = Temp.Previous;
            Last.Next = null;
            Temp.Previous = null;
            Size--;
            return Temp.Value;
        }
        if (Size == 1)  // удаление единственного элемента
        {
            First = null;
            Last = null;
            Size--;
            return Temp.Value;
        }
        else { throw new System.Exception("Попытка удаления из пу-
стого дека"); }
    }
    public void PrintNodes(string Title)
    {
        Console.WriteLine(Title);
        Console.Write("Слева направо: ");
        Node<T> Temp = First;
        while (Temp != null)
        {
            Console.Write(" " + Temp.Value);
            Temp = Temp.Next;
        }
        Console.WriteLine();
        Console.Write("Справа налево: ");
        Temp = Last;
        while (Temp != null)
        {
```

```
            Console.Write(" " + Temp.Value);
            Temp = Temp.Previous;
        }
        Console.WriteLine();
    }
}
```

***Пример*** использования дека

```
static void Main(string[] args)
{
    Deque<int> deq = new Deque<int>();
    deq.PushFront(1);
    deq.PushFront(2);
    deq.PushFront(4);
    deq.PushBack(5);
    deq.PushBack(9);
    deq.PrintNodes("Исходный дек");
    Console.WriteLine("Size = " + deq.Size);
    try
    {
        int V1 = deq.PopFront();
        Console.WriteLine("Удалили первый элемент: " + V1);
        int V2 = deq.PopBack();
        Console.WriteLine("Удалили последний элемент: " + V2);
    }
    catch (Exception error) { Console.WriteLine(error.Message); }
    deq.PrintNodes("Модифицированный дек");
    Console.WriteLine("Size = " + deq.Size);
    Console.ReadKey();
}
```

Результат

```
Исходный дек
Слева направо:  4 2 1 5 9
Справа налево:  9 5 1 2 4
Size = 5
Удалили первый элемент: 4
Удалили последний элемент: 9
Модифицированный дек
Слева направо:  2 1 5
Справа налево:  5 1 2
Size = 3
```

## 8.3 Списки с циклами

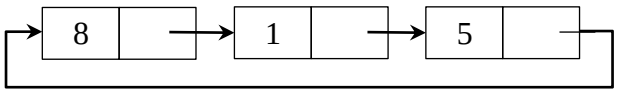



Рис. 8.4. Круговой список

Списки могут иметь циклы или быть целиком зациклены. Такие списки ещё называют *круговыми* (рис. 8.4). В круговых списках последний элемент ссылается на первый. Списки без циклов называются *линейными*

или *открытыми*. К линейным структурам данных кроме списков можно отнести стеки, очереди и деки.

Круговые списки могут иметь фиксированную длину и быть двунаправленными. В этом случае последний элемент ссылается на первый по ссылке `Next`, а первый элемент ссылается на последний по ссылке `Previous`.

**Барабан**

*Барабан* – круговой двунаправленный список фиксированной длины, доступ к которому по чтению и записи может осуществляться только через один элемент, называемый вершиной (англ. `Top`). Программист может вращать барабан по часовой или против часовой стрелки, подгоняя в вершину нужный элемент (рис. 8.5). Например, для чтения элемента, в котором хранится число 5, следует дважды повернуть барабан против часовой стрелки и выполнить операцию чтения. Своё название барабан получил от барабана револьвера, имеющего тот же принцип работы. В описании класса барабана следует предусмотреть переменную для хранения ссылки на элемент `Top`. При вращении барабана вершина `Top` должна обновлять хранимую ссылку. Барабан заполняется последовательным повтором операций записи и вращения. Чтение происходит посредством операций чтения и вращения в ту или иную сторону. Барабан обладает свойствами и стека, и очереди фиксированной длины.

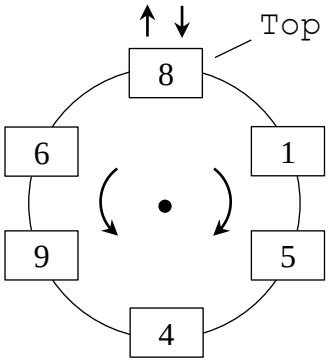


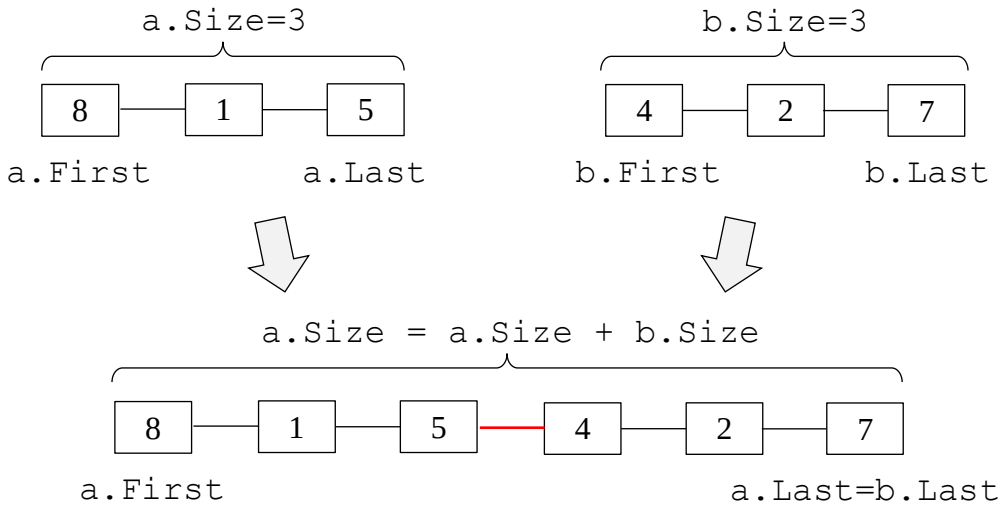
Рис. 8.5. Барабан

## 8.4 Соединение и разделение связных списков

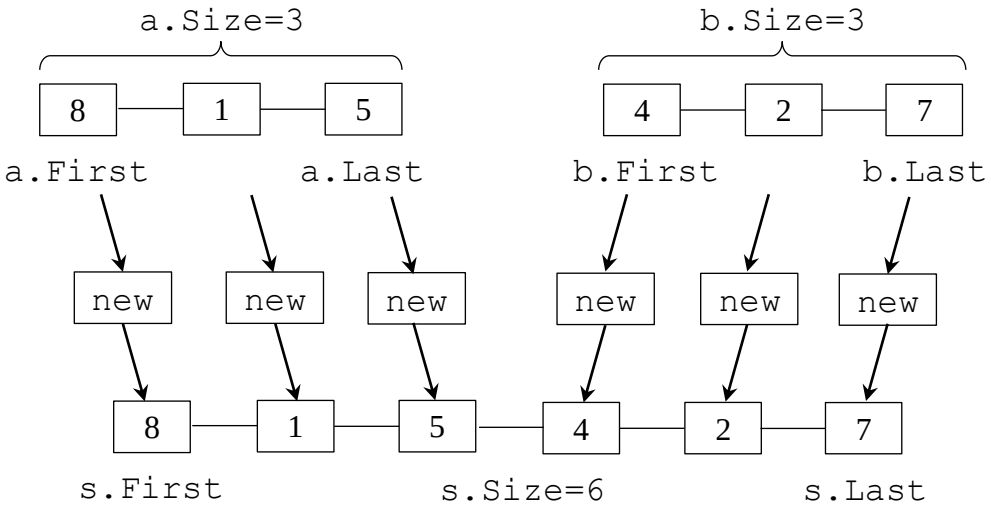
*Соединение* двух связных списков `a` и `b` представляет собой список, который содержит элементы списка `a`, к которым справа присоединены элементы списка `b`. Существуют два способа соединения списков.

Первый способ заключается в установлении связи между последним элементом списка `a` и первым элементом списка `b` (рис. 8.6 а). Такой способ является опасным, так как изменение списка `a` или `b` изменит и соединённый список и наоборот: изменение результата соединения приведёт к изменению списка `a` или `b`. Для частичного недопущения опасных последствий соединённую коллекцию можно сделать доступной только по чтению. Достоинство этого способа – мгновенное соединение списков, недостаток – результат соединения доступен только по чтению и остаётся зависимым от списков `a` или `b`.

Второй способ – безопасный и заключается в поэлементном построении *нового* списка `s` сначала из элементов первого списка, потом – из элементов второго списка (рис. 8.6 б). Достоинство – взаимная независимость списков `a`, `b` и списка `s`, недостаток – длительное время построения нового списка `s`, так как необходимо вызывать конструктор для создания копии каждого элемента исходных списков.




а)




б)

Рис. 8.6. Соединение связных списков

Список можно *разделить* на две части либо указав количество элементов, помещаемых в первый список, либо – значение поля `Value` элемента, с которого должна начаться вторая часть. Для разделения списка можно использовать вышеописанные способы соединения, которые сле-

дует выполнять в обратном порядке. При быстром разделении списка `s` достаточно удалить связь между последним элементом первого списка `a` и первым элементом второго списка `b`. При этом исходный и полученные списки останутся взаимозависимыми между собой. Для частичного недопущения опасных последствий списки `a` и `b` можно сделать доступными только по чтению. Согласно безопасному способу разделение списка выполняется поэлементным построением списков `a` и `b` по отдельности.

Ниже приведён фрагмент класса односвязных списков `SinglyLinkedList<T>`. Метод `Append_modif(list)` соединяет список `this` со списком `list` по первому способу. Соединяемый список `this` теряется как отдельная коллекция и становится результатом. Функция `Append_persist(list)` возвращает результат соединения списка `this` со списком `list` вторым способом. Метод `Split_modif(n, out list)` разделяет список `this` первым способом. Результатом разделения является список `this` длиной $n$, $0 < n \leq Size$ и список `list` длиной $Size - n$. Разделяемый список теряется как целая коллекция. Метод `Split_persist(n, out list1, out list2)` разделяет список `this` вторым способом.

```
class SinglyLinkedList<T>
{
    // this + list -> this:
    public void Append_modif(SinglyLinkedList<T> list)
    {
        this.Last.Next = list.First;
        this.Last = list.Last;
        this.Size = this.Size + list.Size;
    }
    // this + list -> persist:
    public SinglyLinkedList<T> Append_persist(SinglyLinkedList<T>
list)
    {
        SinglyLinkedList<T> persist = new SinglyLinkedList<T>();
        Node<T> Temp = this.First;
        while (Temp != null)
        {
            persist.Add(Temp.Value);
            Temp = Temp.Next;
        }
        Temp = list.First;
        while (Temp != null)
        {
            persist.Add(Temp.Value);
            Temp = Temp.Next;
        }
```

```
        return persist;
    }
    // this -> this + list:
    public void Split_modif(int n, out SinglyLinkedList<T> list)
    {
        if (n < 1 || n > this.Size)
            throw new System.IndexOutOfRangeException("Позиция раз-
деления должна быть в разрешённом диапазоне");
        Node<T> Temp = this.First;
        for (int i = 1; i < n; i++)
        {
            Temp = Temp.Next;   // Temp – посл. эл-т списка this
        }
        list = new SinglyLinkedList<T>();
        list.First = Temp.Next;      // первый элемент втор. списка
        list.Last = this.Last;       // посл. элемент втор. списка
        list.Size = this.Size - n;   // размер второго списка
        Temp.Next = null;   // удал. связь первого списка со вторым
        this.Last = Temp;   // последний элемент первого списка
        this.Size = n;      // размер первого списка
    }
    // this -> list1 + list2:
    public void Split_persist(int n, out SinglyLinkedList<T> list1,
out SinglyLinkedList<T> list2)
    {
        if (n < 1 || n > this.Size)
            throw new System.IndexOutOfRangeException("Позиция раз-
деления должна быть в разрешённом диапазоне");
        list1 = new SinglyLinkedList<T>();
        Node<T> Temp = this.First;
        for (int i = 0; i < n; i++)
        {
            list1.Add(Temp.Value);   // создаём list1
            Temp = Temp.Next;
        }
        list2 = new SinglyLinkedList<T>();
        while (Temp != null)
        {
            list2.Add(Temp.Value);   // создаём list2
            Temp = Temp.Next;
        }
    }
    public void PrintNodes(string Name)
    {
        Node<T> Temp = First;
        Console.Write(Name + " ");
        while (Temp != null)
        {
```

```
            Console.Write(" " + Temp.Value);
            Temp = Temp.Next;
        }
        Console.WriteLine();
    }
}
```

***Пример*** соединения односвязных списков с модификацией.

```
static void Main()
{
    SinglyLinkedList<int> list = new SinglyLinkedList<int>();
    list.Add(34);
    list.Add(35);
    list.PrintNodes("list:");     // 1-й список
    SinglyLinkedList<int> list1 = new SinglyLinkedList<int>();
    list1.Add(7);
    list1.Add(8);
    list1.Add(9);
    list1.PrintNodes("list1:");   // 2-й список
    list.Append_modif(list1);
    list.PrintNodes("list + list1 -> list:");
    Console.WriteLine("Size = " + list.Size);
    list1.First.Value = -99;
    list.PrintNodes("   list:");
    list.Split_modif(3, out SinglyLinkedList<int> list2);
    list.PrintNodes("list -> list + list2:");
    Console.WriteLine("Size = " + list.Size);
    list2.PrintNodes("list2:");
    Console.WriteLine("Size2 = " + list2.Size);
    Console.ReadKey();
}
```

Результат

```
list:  34 35
list1:  7 8 9
list + list1 -> list:  34 35 7 8 9
Size = 5
   list:  34 35 -99 8 9            // есть изменение
list -> list + list2:  34 35 -99
Size = 3
list2:  8 9
Size2 = 2
```

***Пример*** персистентного соединения односвязных списков.

```
static void Main()
{
    SinglyLinkedList<int> list = new SinglyLinkedList<int>();
    list.Add(34);
    list.Add(35);
```

```
    list.PrintNodes("list:");      // 1-й список
    SinglyLinkedList<int> list1 = new SinglyLinkedList<int>();
    list1.Add(7);
    list1.Add(8);
    list1.Add(9);
    list1.PrintNodes("list1:");      // 2-й список
    SinglyLinkedList<int> persist = list.Append_persist(list1);
    persist.PrintNodes("list + list1 -> persist:");
    Console.WriteLine("Size = " + persist.Size);
    list1.First.Value = -99;   // изменяем первый элемент
    persist.PrintNodes("   persist:");
    persist.Split_persist(3, out SinglyLinkedList<int> list2, out
SinglyLinkedList<int> list3);
    list2.PrintNodes("list2:");
    Console.WriteLine("Size = " + list2.Size);
    list3.PrintNodes("list3:");
    Console.WriteLine("Size = " + list3.Size);
    Console.ReadKey();
}
```

Результат

```
list:  34 35
list1:  7 8 9
list + list1 -> persist:  34 35 7 8 9
Size = 5
   persist:  34 35 7 8 9              // нет изменений
list2:  34 35 7
Size = 3
list3:  8 9
Size = 2
```

***Пример*** получения итератора `en` двух двусвязных списков с помощью функции `Concat` встроенного класса `LinkedList<T>` и создания персистентного соединения `z`.

```
static void Main()
{
    LinkedList<int> x = new LinkedList<int>();
    x.AddFirst(10);                              // 1-й список
    Console.WriteLine("x = " + string.Join(",", x));
    LinkedList<int> y = new LinkedList<int>();
    for (int i = 0; i < 5; i++) y.AddLast(i);   // 2-й список
    Console.WriteLine("y = " + string.Join(",", y));
    IEnumerable<int> en = x.Concat(y);  // итератор двух списков
    Console.WriteLine("en = " + string.Join(",", en));
    LinkedList<int> z = new LinkedList<int>(en);   // новый z=en
    Console.WriteLine("z = " + string.Join(",",z));
    Console.WriteLine("Size = " + z.Count);
    y.First.Value = -99;   // изменяем первый элемент
```

```
    Console.WriteLine("   en = " + string.Join(",", en));
    Console.WriteLine("   z = " + string.Join(",", z));
    Console.ReadKey();
}
```

Результат

```
x = 10
y = 0,1,2,3,4
en = 10,0,1,2,3,4
z = 10,0,1,2,3,4
Size = 6
   en = 10,-99,1,2,3,4      // есть изменение
   z = 10,0,1,2,3,4         // нет изменений
```

Сравним время персистентного соединения списков с помощью встроенного класса `LinkedList<T>` и рукописного класса `SinglyLinkedList<T>`

```
static void Main()
{
    LinkedList<int> x = new LinkedList<int>();
    x.AddFirst(10);
    LinkedList<int> y = new LinkedList<int>();
    int N = 5000000;
    for (int i = 0; i < N; i++) y.AddLast(i);
    DateTime t1 = DateTime.Now;  // засекаем начало
    IEnumerable<int> en = x.Concat(y);  // итератор двух списков
    LinkedList<int> z = new LinkedList<int>(en);   // новый z=en
    DateTime t2 = DateTime.Now;  // засекаем конец
    Console.WriteLine("Время LinkedList: " + t2.Subtract(t1));
    Console.WriteLine("Size = " + z.Count);

    SinglyLinkedList<int> list = new SinglyLinkedList<int>();
    list.Add(10);
    SinglyLinkedList<int> list1 = new SinglyLinkedList<int>();
    for (int i = 0; i < N; i++) list1.Add(i);
    DateTime t5 = DateTime.Now;  // засекаем начало
    SinglyLinkedList<int> persist = list.Append_persist(list1);
    DateTime t6 = DateTime.Now;  // засекаем конец
    Console.WriteLine("Время Append_persist: {0}", t6.Subtract(t5));
    Console.WriteLine("Size = " + persist.Size);
    Console.ReadKey();
}
```

Результат

```
Время LinkedList: 00:00:00.9068486
Size = 5000001
Время Append_persist: 00:00:00.4271047
Size = 5000001
```

### 8.5 Методы и свойства класса LinkedList языка C#

Ниже перечисляются конструкторы и основные свойства и методы класса двусвязных списков `LinkedList`.

Конструктор пустого связного списка

```
LinkedList<string> ll = new LinkedList<string>();
```

Конструктор связного списка из массива

```
string[] words = { "fox", "jumps", "over", "dog" };
LinkedList<string> sentence = new LinkedList<string>(words);
```

Свойства:

```
int count = sentence.Count();  // количество элементов
LinkedListNode<string> x = sentence.First;  // первый узел
LinkedListNode<string> y = sentence.Last;  // последний узел
string x = sentence.First();  // первое слово
string y = sentence.Last();   // последнее слово
var a = sentence.First.Next.Next.Next; // четвёртый узел
var b = sentence.First.Next.Next.Next.Value; // 4-е слово
```

Методы:

а) добавить после/перед текущим узлом новый узел

```
sentence.AddAfter(LinkedListNode<T>, LinkedListNode<T>)
sentence.AddBefore(LinkedListNode<T>, LinkedListNode<T>)
```

б) добавить после/перед текущим узлом новый узел, содержащий значение `T`

```
sentence.AddAfter(LinkedListNode<T>, T)
sentence.AddBefore(LinkedListNode<T>, T)
```

в) добавить в начало/конец новый узел

```
sentence.AddFirst(LinkedListNode<T>)
sentence.AddLast(LinkedListNode<T>)
```

г) добавить в начало/конец новый узел, содержащий значение `T`

```
sentence.AddFirst(T)
sentence.AddLast(T)
```

д) определить принадлежность значения `T` списку

```
bool sentence.Contains(T)
```

е) найти первый/последний узел, содержащий указанное значение

```
LinkedListNode<string> current = sentence.Find("the");
LinkedListNode<string> current = sentence.FindLast("the");
```

ж) удалить заданный узел

```
sentence.Remove(LinkedListNode<T>)
```

з) удалить первое вхождение значения T

```
sentence.Remove(T)
```

и) удалить первый/последний узел

```
sentence.RemoveFirst()
sentence.RemoveLast()
```

**Задача 1.** Разработать программу быстрой сортировки двусвязных списков на основе встречного поиска по ссылкам `Next` и `Previous`. Сравнить с функцией быстрой сортировки массивов по времени работы в зависимости от длины коллекции.

**Задача 2.** Разработать программу сортировки слиянием двусвязных списков и сравнить с программой сортировки слиянием односвязных списков по времени работы в зависимости от длины списков.

**Задача 3!** В класс `DoublyLinkedList<T>` добавить:

– итератор, проходящий двусвязный список справа налево,

а также методы:

– персистентного соединения двух двусвязных списков в новый двусвязный список;

– персистентного разделения двусвязного списка на два списка по заданному количеству элементов первого списка;

– персистентного разделения двусвязного списка по первому встреченному элементу с заданным значением поля `Value`. С этого элемента должен начаться второй список.

Сравнить время соединения двух двусвязных списков в классах `LinkedList<T>` и `DoublyLinkedList<T>`.

**Контрольные вопросы**

1. Какие поля хранятся в двусвязном объекте?
2. В чём отличие двусвязного списка от дека?
3. Для чего служат итераторы списков?
4. В чём отличие динамического и статического итератора списков?
5. Каков отличительный признак круговых списков?
6. Каковы достоинства и недостатки связных списков?
7. Почему барабан имеет фиксированную длину?
8. Возможна ли корректная работа барабана с вращением только в одну сторону?
9. Зависит ли время соединения связных списков от их длины в случае модификации списков?
10. Зависит ли время соединения связных списков от их длины в случае персистентных списков?
11. От чего зависит время разделения связного списка?
12. Какой класс C# поддерживает односвязные списки?
13. Какой класс C# поддерживает двусвязные списки?

# 9 Несвязные списки

## 9.1 Представление в памяти

Списки могут быть представлены не коллекцией ссылочных объектов, а динамическими массивами. В этом случае динамические массивы, как правило, поддерживают совместный функционал списков и массивов. Элементами таких списков могут быть как значения, так и ссылки.

Список значений `3, 5, 7` фиксированного размера можно представить в памяти одномерным динамическим массивом `{3, 5, 7}` этих значений (рис. 9.1).

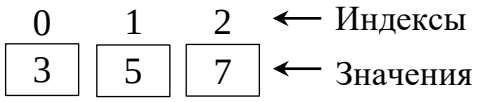



Рис. 9.1. Массив значений

Список объектов, например строк `"qwe", "r", "ty"` произвольного размера, является одномерным динамическим массивом ссылок на эти строки (рис. 9.2). Ссылки получают своё значение 210, 390, 160 только по запуску программы и не являются смежными в отличие от элементов массива значений. Сами строки хранятся в куче.

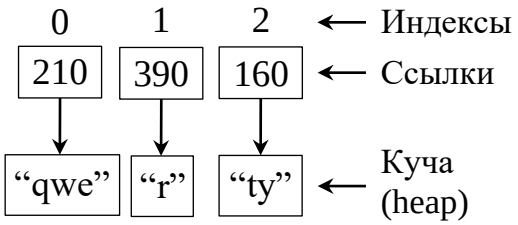



Рис. 9.2. Массив ссылок

При таком способе организации списков, прочитав из массива ссылку на элемент, следует прочитать по этой ссылке сам элемент. Время выполнения операций увеличивается из-за такого двухступенчатого доступа и плохой сочетаемости с кэшированием. Двухступенчатый доступ – плата за произвольный размер элементов списка.

## 9.2 Операции над списками

Списки в виде одномерных массивов выполняют операции чтения/записи по известному индексу за константное время $O(1)$. Однако операции поиска по значению или операции вставки/удаления выполняются за линейное время $O(N)$. Это связано с тем, что массив необходимо раздвинуть (разомкнуть), чтобы освободить место для вставляемого элемента, или сдвинуть (сомкнуть) в позиции удаляемого элемента. Следует иметь в виду, что раздвигать массив надо от последнего элемента к позиции вставляемого. Смыкать массив надо от позиции удаляемого элемента к последнему элементу, иначе будет потеря данных.

При создании массива его *ёмкость* (`Capacity`) задаётся больше, чем изначально нужно. Часть ёмкости, занятую элементами массива, называют *размером* (`Size`). Разность `Capacity-Size` показывает текущий резерв ёмкости массива, т.е. число элементов, которые можно добавить в список не копируя все его элементы для создания нового массива по команде `Resize`.

Когда нужно раздвинуть массив для вставки элемента, достаточно передвинуть его правую границу на размер одного элемента и сместить последующие элементы вправо. Когда правая граница `Size` достигнет предельной ёмкости `Capacity`, то следует установить бóльшую ёмкость.

Рассмотрим обобщённый класс `DynArr<T>`, содержащий два свойства `Capacity` и `Size`, а также конструктор с параметром `capacity` и размером по умолчанию `size=0`. Методы `Add` и `Delete` имеют доступ к концу списка. Метод `Insert(value, position)` вставляет элемент `value` в позицию `position`. Геттер и сеттер дают доступ к элементам массива по индексу. В учебных целях для просмотра значений `Capacity` и `Size` эти свойства объявлены публичными `public`. На самом деле их следует объявить `private`.

```
class DynArr<T>
{
    public int Capacity;  // ёмкость массива
    public int Size;      // используемая ёмкость массива
    public T[] A;         // массив
    public DynArr(int capacity)  // конструктор
    { Capacity = capacity; A = new T[capacity]; Size = 0; }
    public void Add(T value)
    {
        if (Size < Capacity)  // в пределах ёмкости
            A[Size] = value;  // добавили элемент
        else  // вышли за предел ёмкости
        {
            Array.Resize(ref A, Capacity + 4);  // ёмкость + 4
            Capacity += 4;
            A[Size] = value;  // добавили элемент
        }
        Size++;               // размер + 1
    }
    public void Insert(T value, int position)
    {
        if (position >= 0 && position <= Size) // индекс в границах
        {
            if (Size >= Capacity)  // вышли за предел ёмкости
            {
                Array.Resize(ref A, Capacity + 4);  // ёмкость + 4
```

```
                Capacity += 4;
            }
            if (position == Size)     // вставка в конец
            { A[Size] = value; Size++; }
            else
            {
                for (int i = Size - 1; i >= position; i--)
                    A[i + 1] = A[i];  // раздвинули массив
                A[position] = value;  // вставили элемент
                Size++;               // размер + 1
            }
        }
        else { throw new System.Exception("Выход индекса" + position
+ "за границы массива"); }
    }
    public T Delete()
    {
        if (Size > 0)  // удаление из непустого массива
        {
            Size--;    // удаление последнего элемента
            return A[Size];  // возвращаем удалённый элемент
        }
        else { throw new System.Exception("Попытка удаления из пу-
стого списка"); }
    }
    public T this[int position]
    {
        get
        {
            if (position < Size) return A[position];
            else { throw new System.IndexOutOfRangeException("Выход
индекса" + position + "за границы массива"); }
        }
        set
        {
            if (position < Size) A[position] = value;
            else { throw new System.IndexOutOfRangeException("Выход
индекса" + position + "за границы массива"); }
        }
    }
    public void PrintList(string Title)
    {
        Console.Write(Title + " ");
        for (uint i = 0; i < Size; i++)
            Console.Write(A[i] + " ");
        Console.WriteLine();
    }
}
```

***Пример*** использования списка целых чисел

```
static void Main()
{
    var list = new DynArr<int>(4);
    list.Add(1);
    list.Add(2);
    list.Add(4);
    list.Add(5);
    try
    {
        list.Insert(9, 0);
        list.PrintList("Массив:");
        Console.WriteLine("Capacity = " + list.Capacity);
        Console.WriteLine("Size = " + list.Size);
        int V1 = list.Delete();
        Console.WriteLine("Удалили элемент: " + V1);
        int V2 = list.Delete();
        Console.WriteLine("Удалили элемент: " + V2);
    }
    catch (Exception error) { Console.WriteLine(error.Message); }
    list.PrintList("Результат:");
    Console.WriteLine("Capacity = " + list.Capacity);
    Console.WriteLine("Size = " + list.Size);
    Console.ReadKey();
}
```

Результат

```
Массив: 9 1 2 4 5
Capacity = 8
Size = 5
Удалили элемент: 5
Удалили элемент: 4
Результат: 9 1 2
Capacity = 8
Size = 3
```

Обратите внимание, что значение свойства `Capacity` автоматически увеличилось до восьми, когда значение `Size` превысило заданную ёмкость списка, равную четырём.

***Пример*** использования списка строк

```
static void Main()
{
    var list = new DynArr<string>(4);
    list.Add("a");
    list.Add("bb");
    list.Add("ccc");
    list.Add("xxxxx");
```

```
    try
    {
        list.Insert("zzz", 0);
        list.PrintList("Массив:");
        Console.WriteLine("Capacity = " + list.Capacity);
        Console.WriteLine("Size = " + list.Size);
        string v1 = list.Delete();
        Console.WriteLine("Удалили элемент: " + v1);
        string v2 = list.Delete();
        Console.WriteLine("Удалили элемент: " + v2);
    }
    catch (Exception error) { Console.WriteLine(error.Message); }
    list.PrintList("Результат:");
    Console.WriteLine("Capacity = " + list.Capacity);
    Console.WriteLine("Size = " + list.Size);
    Console.ReadKey();
}
```

Результат

```
Масив: zzz a bb ccc xxxxx
Capacity = 8
Size = 5
Удалили элемент: xxxxx
Удалили элемент: ccc
Результат: zzz a bb
Capacity = 8
Size = 3
```

### 9.3 Использование анонимных функций и методов для обработки списков

*Анонимная функция* – это функция, не имеющая имени. При написании анонимных функций в C# используют делегаты. *Делегат* описывает сигнатуру множества функций или методов. *Сигнатура* – перечень типов входных выходного аргументов. *Анонимный метод* – это метод, не имеющий имени. Сигнатура анонимного метода не содержит тип возвращаемого значения.

**Использование анонимных функций и предикатов**

Запишем функцию инкремента переменной в математической нотации $f(x) = x + 1$. Для того чтобы сделать её анонимной избавимся от имени $(x) = x + 1$. Синтаксис C# требует использовать в анонимных функциях составной символ отображения $=>$, являющийся синтаксическим признаком анонимной функции $(x) => x + 1$. Анонимную функцию можно передать сторонней функции/методу или вставить в вычисляемое выражение с помощью ссылки на эту функцию. Для этого следует при-

своить некоторой переменной *inc* ссылку на анонимную функцию, например, $inc = (x) => x + 1$. Но перед этим надо объявить, что ссылка имеет тип унарной функции, т.е. является делегатом, имеющим один входной аргумент типа `int` и выходное значение также типа `int`. Для этого в классе объявим делегат с именем `F` и указанной сигнатурой

```
delegate int F(int x);  // delegate T1 F<T,T1>(T x);
```

после чего можем смело определить нашу анонимную функцию

```
F inc = (x) => x + 1;  // F<int,int> inc = (a) => a + 1;
```

В комметратиях показан обобщённый вариант функции, в котором входной `T` и выходной `T1` типы могут не совпадать.

Делегат `F` может использоваться для определения любой анонимной унарной функции, обрабатывающей числа типа `int`. Для анонимных бинарных функций можно воспользоваться делегатом

```
delegate int F(int x, int y);  // delegate T2 F<T,T1,T2>(T x, T1 y);
```

и определить анонимную функцию, например, суммы

```
F sum = (x,y) => x + y;  // F<int,int,int> sum = (x,y) => x + y;
```

Условные выражения (предикаты) от одной переменной могут быть описаны делегатом

```
delegate bool P(int x);  // delegate bool P<T>(T x)
```

и определены в виде

```
P gr3 = (x) => x > 3;    // P<int> gr3 = (x) => x > 3;
```

Рассмотрим пример вызова анонимных функций

```
public class Program
{
    delegate T1 F<T, T1>(T x);
    delegate T2 F<T, T1, T2>(T x, T1 y);
    delegate bool P<T>(T x);
    static void Main(string[] args)
    {
        F<int, int> inc = (a) => a + 1;
        Console.WriteLine(inc(5));   // 6
        F<int, int> square = (a) => a * a;
        Console.WriteLine(square(inc(5)));   // 36
        F<int, int, int> sum = (x, y) => x + y;
        Console.WriteLine(sum(inc(5), square(2)));   // 10
        P<int> gr3 = (x) => x > 3;
        Console.WriteLine(gr3(5));   // True
        Console.Read();
    }
}
```

В языке C# используемые в библиотечных классах делегаты имеют предопределённые имена:

`Func` – функции с арностью от нуля до 16;

`Predicate` – унарные предикаты;

`Action` – методы с арностью от нуля до 16.

Основная цель использования анонимных функций заключается в организации гибкой обработки коллекций. Для этого ссылки на анонимные функции передаются функциям обработки коллекций в качестве аргументов. Такие функции обработки коллекций называются функциями *второго порядка* или *высокоуровневыми* функциями.

Опишем функцию фильтрации `Filter` списка `list` и вызовем её дважды с различными условиями $x > 3$ и $x \,\%\, 2 == 1 \,\&\, x < 7$, которые будем передавать в функцию в качестве второго аргумента. Ссылка на анонимную функцию, проверяющую первое условие, – это `gr3`. Ссылку `fi` на второе условие определим дополнительно

```
public class Program
{
    delegate bool P<T>(T x);
    static List<int> Filter(List<int> x, P<int> usl)
    {
        List<int> y = new List<int>();
        foreach (int i in x)
            if (usl(i)) y.Add(i);
        return y;
    }
    static void Main(string[] args)
    {
        P<int> gr3 = (x) => x > 3;
        P<int> fi = (x) => x % 2 == 1 & x < 7;
        List<int> list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
        Console.WriteLine(string.Join(" ", Filter(list, gr3)));
        Console.WriteLine(string.Join(" ", Filter(list, fi)));
        Console.Read();
    }
}
```

Результат

```
4 5 6 7 8
1 3 5
```

Допускается передавать анонимные функции непосредственно в качестве аргументов, без использования ссылок. В случае унарной функции скобки можно опустить

```
public class Program
{
```

```
    delegate bool P(int x);
    static List<int> Filter(List<int> x, P usl)
    {
        List<int> y = new List<int>();
        foreach (int i in x)
            if (usl(i)) y.Add(i);
        return y;
    }
    static void Main(string[] args)
    {
        List<int> list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
        Console.WriteLine(string.Join(" ",
                Filter(list,x => x > 3)));
        Console.WriteLine(string.Join(" ",
                Filter(list, x => x % 2 == 1 & x < 7)));
        Console.Read();
    }
}
```

Так как в нашем примере используются анонимные функция, возвращающие булево значение, то они, по сути, является предикатами. Поэтому вместо вручную написанного делегата `P` можно воспользоваться встроенным делегатом `Predicate<int>`.

```
static List<int> Filter(List<int> x, Predicate<int> usl)
{
    List<int> y = new List<int>();
    foreach (int i in x)
        if (usl(i)) y.Add(i);
    return y;
}
static void Main(string[] args)
{
    List<int> list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
    Console.WriteLine(string.Join(" ",
        Filter(list, x => x > 3)));    // 4 5 6 7 8
    Console.WriteLine(string.Join(" ",
        Filter(list, x => x % 2 == 1 & x < 7)));    // 1 3 5
    Console.Read();
}
```

Определим функцию `Map` отображения входного списка в выходной, посредством применения анонимной функции к каждому элементу входного списка для получения нового значения элементов выходного списка. Второй аргумент функции `Map` содержит встроенный делегат `Func<int,int>`. В качестве анонимных функций используем инкремент и возведение в квадрат.

```
static List<int> Map(List<int> x, Func<int,int> f)
```

```
{
    List<int> y = new List<int>();
    foreach (int i in x) y.Add(f(i));
    return y;
}
static void Main(string[] args)
{
    List<int> list = new List<int> { 1, 2, 3, 4 };
    Console.WriteLine(string.Join(" ",
        Map(list, (x) => x + 1)));        // 2 3 4 5
    Console.WriteLine(string.Join(" ",
        Map(list, (x) => x * x)));        // 1 4 9 16
    Console.Read();
}
```

Обратите внимание, что запись `Func<int,int>` на самом деле выражает сигнатуру унарных функций `int Func<int>`. Последний аргумент в паре `<int,int>` является типом выходного значения функции.

Рассмотрим пример использования делегата бинарных функций. Определим функцию фильтрации списка `Filter1`, которая сравнивает каждый элемент со своим предшественником. В качестве делегата условия фильтрации используем `Func<T1,T2,T3>`. Здесь `T1,T2` – типы двух аргументов анонимной функции, `T3` – тип возвращаемого значения. Первый вызов `Filter1` пропускает те элементы, которые больше своего предшественника. Второй вызов пропускает те элементы, которые меньше декремента своего предшественника.

```
static List<int> Filter(List<int> x, Func<int, int, Boolean> usl)
{
    List<int> list1 = new List<int>();
    int y = x.First();
    foreach (int i in x)
    {
        if (usl(i, y)) list1.Add(i);
        y = i;
    }
    return list1;
}
static void Main(string[] args)
{
    List<int> list = new List<int> { 1, 2, 3, 4, 2, 6, 5, 8 };
    Console.WriteLine(string.Join(" ",
        Filter(list, (x, y) => x > y)));   // 2 3 4 6 8
    Console.WriteLine(string.Join(" ",
        Filter(list, (x, y) => x < y - 1)));   // 2
    Console.Read();
}
```

**Использование анонимных методов**

Определим делегат `Act` для анонимного унарного метода, выполняющего некоторое действие над своим аргументом

```
delegate void Act(int x);
```

и используем этот делегат в методе `ForEach` обработки каждого элемента списка для вывода на экран, например, квадратов элементов списка через пробел

```
delegate void Act(int x);
static void ForEach(List<int> x, Act t)
{
    foreach (int i in x) t(i);
}
static void Main(string[] args)
{
    List<int> list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
    ForEach(list, x => {Console.Write(x * x + " ");});
    Console.Read();
}
```

Определим делегат `Act2` для анонимного бинарного метода, выполняющего некоторое действие над каждым элементом списка и результатом сравнения этого элемента со средним арифметическим значением по условию «больше»

```
delegate void Act(int x, Boolean y);
```

и используем этот делегат для маркировки элементов списка по признаку «меньше среднего арифметического»

```
delegate void Act(int x, Boolean y);
static void Each2Aver(List<int> x, Act t)
{
    double a = x.Average();
    foreach (int i in x) t(i, i > a);
}
static void Main(string[] args)
{
    List<int> list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
    Each2Aver(list, (x,y) => {Console.Write(x + "-" + y + " ");});
    Console.Read();
}
```

Если элемент меньше среднего значения, то признаком является значение `True`, иначе – `False`.

Результат

```
1-False 2-False 3-False 4-False 5-True 6-True 7-True 8-True
```

Вышеприведённый пример можно переписать без объявления делегата

```
static void Each2Aver(List<int> x, Action<int, Boolean> t)
{
    double a = x.Average();
    foreach (int i in x) t(i, i > a);
}
static void Main(string[] args)
{
    List<int> list = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
    Each2Aver(list, (x,y) => {Console.Write(x + "-" + y + " ");});
    Console.Read();
}
```

Для подавления вывода на экран элементов, которые меньше среднего значения, следует указать другой анонимный метод

```
    Each2Aver(list, (x, y) => { if (y) Console.Write(x + " "); });
```

Результат

```
5 6 7 8
```

### 9.4 Методы и свойства класса `List` языка C#

Рассмотрим основные свойства, функции и методы обработки списков. Следующие примеры используют список целых чисел `List<int>`.

1. Создание пустого списка

```
var list = new List<int>();  // list: (empty)
```

2. Создание и инициализация списка заданными значениями

```
var list = new List<int>() { 8, 3, 2 }; // list: 8 3 2
```

3. Создание и инициализация списка другим списком

```
var listA = new List<int>() { 8, 3, 2 };
var listB = new List<int>(listA); // listB: 8 3 2
```

4. Создание списка указанной ёмкости

```
var list = new List<int>(16); // Count=0, Capacity=16
```

5. Чтение/запись элемента в указанной позиции

```
// list: 8 3 2
int item = list[1]; // item=3
list[1] = 4; // list: 8 4 2
```

6. Добавление элемента в конец списка

```
// list: 8 3 2
list.Add(5); // list: 8 3 2 5
```

7. Добавление элементов другого списка (или IEnumerable коллекции) в конец списка

```
// listA: 8 3 2
// listB: 5 7
```

```
listA.AddRange(listB); // listA: 8 3 2 5 7
```

8. Бинарный поиск элемента в сортированном списке

```
// list: 1 3 4 6 7 9
int index = list.BinarySearch(6); // index: 3
```

9. Очистка списка

```
// list: 8 3 2
list.Clear(); // list: (empty)
```

10. Проверка наличия элемента в списке

```
// list: 8 3 2
bool result = list.Contains(3);  // result = true
```

11. Копирование элементов в указанный массив

```
// list: 8 3 2
// array: 0 0 0 0 0
list.CopyTo(array); // array: 8 3 2 0 0
```

12. Проверка существования элемента, связанного предикатом

```
// list: 8 3 2
bool result = list.Exists(x => x > 10);  // result = false
```

13. Поиск первого вхождения элемента, связанного предикатом

```
// list: 8 3 2
int item = list.Find(x => x > 2);  // item = 8
```

14. Возвращение списка элементов, связанных предикатом

```
// listA: 8 3 2
var listB = listA.FindAll(x => x > 2);  // listB: 8 3
```

15. Возвращение индекса первого вхождения элемента, связанного предикатом

```
// list: 8 3 6 4 2
int index = list.FindIndex(x => x < 5);  // index: 1
```

16. Возвращение последнего вхождения элемента, связанного предикатом

```
// list: 8 3 2
int item = list.FindLast(x => x < 5);  // item: 2
```

17. Возвращение индекса последнего вхождения элемента, связанного предикатом

```
// list: 2 4 6 3 8
int index = list.FindLastIndex(x => x < 5);  // index: 3
```

18. Выполнение указанного действия для каждого элемента списка

```
// list: 8 3 2
list.ForEach(x => {Console.Write(x);});  // output: 832
```

19. Возвращение индекса первого вхождения элемента в список

```
// list: 8 3 2 6 8
int index = list.IndexOf(8);  // index: 0
```

20. Вставка элемента в указанную позицию списка:

```
// list: 8 3 2
list.Insert(1, item: 5);  // list: 8 5 3 2
```

21. Возвращение индекса последнего вхождения элемента в список

```
// list: 8 3 2 6 8
int index = list.LastIndexOf(8);  // index: 4
```

22. Удаление первого вхождения элемента из списка

```
// list: 8 4 2 4
list.Remove(4);  // list: 8 2 4
```

23. Удаление всех элементов, связанных предикатом

```
// list: 8 3 6 2
list.RemoveAll(x => x < 4);  // list: 8 6
```

24. Удаление элемента из указанной позиции

```
// list: 8 3 6 2
list.RemoveAt(2);  // list: 8 3 2
```

25. Удаление элементов из указанного диапазона

```
// list: 8 3 6 2 4 5
list.RemoveRange(index: 2, count: 3);  // list: 8 3 5
```

26. Реверс списка

```
// list: 8 3 2
list.Reverse();  // list: 2 3 8
```

27. Сортировка списка

```
// list: 8 3 6 2
list.Sort();  // list: 2 3 6 8
```

28. Создание массива из списка

```
// list: 8 3 2
int[] array = list.ToArray();  // array: 8 3 2
```

29. Проверка всех элементов списка указанным предикатом

```
// list: 8 3 2
bool result = list.TrueForAll(x => x < 10);  // result: true
```

### 9.5 Сравнение связных и несвязных списков

Сравнение проведено на основе оценки вычислительной сложности основного функционала класса двусвязных списков `LinkedList` и класса списков `List`, построенных на динамических массивах.

Сравнение связных списков и динамичесих массивов

| Операция | Двусвязный список `LinkedList` | | Динамический массив ссылок `List` | |
|---|---|---|---|---|
| | Число переходов по ссылке | Число чтений/записей в память | Число переходов по ссылке | Число чтений/записей в память |
| Чтение/запись по индексу | $O(N)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| Поиск по значению | $O(N)$ | $O(N)$ | $O(N)$ | $O(N)$ |
| Добавле- | $O(1)$ | $O(1)$ | $O(N)$ | $O(N)$ |

| | | | | |
|---|---|---|---|---|
| ние/удаление в начало/конец | | | | |
| Добавление/удаление внутрь | $O(N)$ | $O(1)$ | $O(N)$ | $O(N)$ |

Несвязные списки (класс `List`) выгодно использовать в алгоритмах, выполняющих много операций чтения/записи по индексу, но мало операций вставки/удаления.

Двусвязные списки (класс `LinkedList`) выгодно использовать в алгоритмах, выполняющих много операций вставки/удаления, но мало операций чтения/записи по индексу.

**Задача 1.** Расширить класс списков `MyList` (динамических массивов) следующими компонентами:

а) конструктором списка из статического массива

```
var list = new MyList<int> (new int[] { 8, 3, 2 });
```

б) функцией вырезки элемента из указанной позиции списка

```
//  list: 8 3 2
value = list.Cut(1);  // list: 8 2,   value: 3
```

в) функцией удаления первого вхождения элемента

```
//  list: 8 4 2 4
index = list.Remove(4);  // list: 8 2 4, index = 1
```

**Задача 2.** Оценить вычислительную сложность последовательного поиска элемента по его значению `value` с использованием классов `List` и `LinkedList` языка C#. Поиск следует выполнять функцией `Contains(value)`. Искать рекомендуется отсутствующий элемент коллекции, чтобы поиск пробежал по всем элементам. Построить в MS Excel графики зависимости времени поиска от размера коллекции.

**Контрольные вопросы**

1. В чём суть списков, созданных на основе динамических массивов?
2. В чём отличие массива значений и массива ссылок?
3. Когда происходит двухступенчатый доступ к элементам массива?
4. Что такое ёмкость массива?
5. В чём отличие ёмкости массива от его размера?
6. Как сочетается массив значений и массив ссылок с кэшированием памяти?
7. В каких случаях выгодно использовать массивы, основанные на классе `List`?
8. В каких случаях выгодно использовать связные списки?

# 10 Корневые деревья

## 10.1 Основные термины и определения

*Корневое дерево* – рекурсивная структура данных, содержащая родительскую вершину дерева (корень) и связи (ссылки) с дочерними вершинами. Вершина с пустыми ссылками называется *листом*, или *терминальной вершиной*, или *терминалом*.

*Ребро* – связь родительской вершины с дочерней. Иногда рёбра называют ветвями или дугами, а вершины – узлами.

*Степень вершины* – количество рёбер, инцидентных вершине. Дерево называется *бинарным*, если каждая вершина имеет не более двух наследников. Оно широко используется в программировании, так как одного сравнения достаточно для выбора одной из двух дочерних вершин. Однако ничто не мешает строить деревья любой арности. Кстати, односвязные списки реализуются унарными деревьями.

*Глубина вершины* – количество рёбер на пути от корня до неё. *Высота вершины* – количество рёбер от неё до глубочайшего потомка. *Высота дерева* – максимальная высота корня. Вершины, имеющие одинаковую глубину *n*, называются вершинами *n*-го *уровня*. Корень дерева лежит на нулевом уровне. К вершинам второго уровня (рис. 10.1) относятся вершины 2, 6, 3. *Поддерево* – вершина дерева со всеми её потомками.

При изображении деревьев корень обычно располагают вверху, а терминальные вершины – внизу (рис. 10.1).

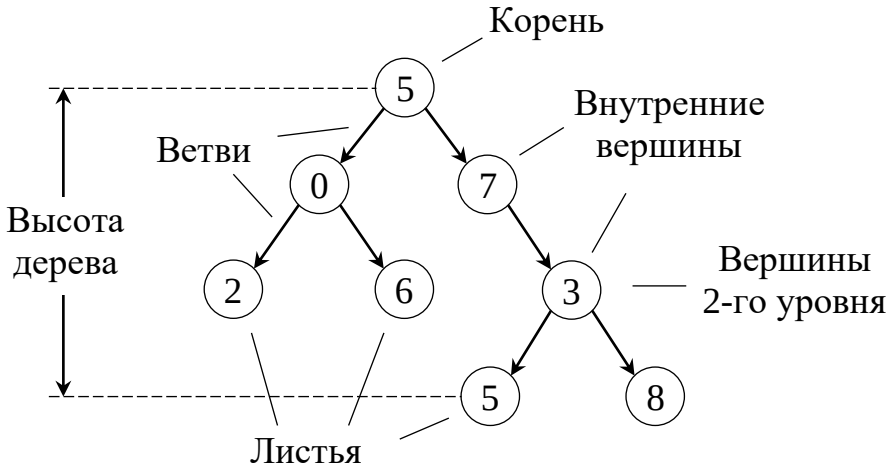



Рис. 10.1. Элементы бинарного дерева

Деревья могут быть уникальными и неуникальными. В *уникальных* деревьях значения всех вершин уникальные, т.е. не повторяются. *Неуни-*

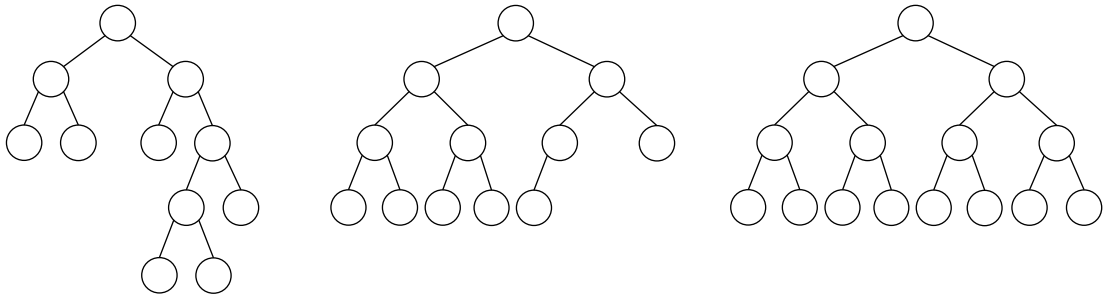
*кальные* деревья могут иметь повторяющиеся значения в вершинах. Неуникальное дерево (рис. 10.1) имеет дубликат вершины с числом 5.

*Полное* дерево – дерево, в котором все вершины, кроме листьев, имеют по две дочерние вершины (рис. 10.2, а).

*Завершённое* дерево – дерево, в котором все терминальные вершины прижаты влево (рис. 10.2, б).

*Идеальное* дерево – полное дерево, в котором все листья расположены на одинаковой глубине (рис. 10.2, в).

*Балансированное* дерево – дерево, у которого максимальная и минимальная глубины терминальных вершин отличаются не более чем на единицу. Завершённые и идеальные деревья считаются балансированными. В полном дереве (рис. 10.2, а) минимальная глубина терминала равна двум, а максимальная глубина равна пяти рёбрам.



а) б) в)

Рис. 10.2. Виды бинарных деревьев:
а) полное дерево;
б) завершённое дерево;
в) идеальное дерево

**Свойства идеальных бинарных деревьев:**

1. Количество вершин $N$ в идеальном бинарном дереве высотой $H$ уровней рассчитывается по формуле

$$N = 2^H - 1.$$

Количество вершин дерева (рис. 10.2, в) равна $N = 2^4 - 1 = 15$.

2. Высота $H$ идеального бинарного дерева с $N$ вершинами определяется по формуле

$$H = \log_2(N + 1).$$

Высота дерева (рис. 10.2, в) $H = \log_2(15 + 1) = 4$.

3. Количество терминальных вершин $L$ в идеальном бинарном дереве высотой $H$ вычисляется по формуле $L = 2^{H-1}$, а количество внутренних вершин $I = L - 1$. Число терминальных и внутренних вершин дерева (рис. 10.2, в) равно $L = 8$ и $I = 7$ соответственно.

**Свойства завершённых и балансированных бинарных деревьев:**

1. Количество вершин $N$ в завершённом бинарном дереве высотой $H$ лежит в диапазоне

$$2^{H-1} \le N \le 2^H - 1.$$

2. Высота завершённого бинарного дерева, имеющего $N$ вершин

$$H = \lceil \log_2(N) \rceil.$$

3. Количество терминальных вершин $L$ в завершённом бинарном дереве высотой $H$ лежит в диапазоне

$$2^{H-2} \le L \le 2^{H-1}.$$

**Следствия свойств:**

1. Поиск/добавление вершины в бинарном дереве имеет сложность, равную максимальной глубине терминала $\log_2(N)$.

2. Построение завершённого бинарного дерева, содержащего $N$ вершин, выполняется за время $O(N \cdot \log N)$.

**Представление двоичных деревьев**

Для создания вершин двоичного дерева используется обобщённый класс `BinNode<T>`.

```
public class BinNode<T>
{
    public T Value;  // Ключевое поле
    public BinNode<T> Left;   // Ссылка на левую ветвь или null
    public BinNode<T> Right;  // Ссылка на правую ветвь или null
// Конструктор нового элемента со значениями Value и Left, Right:
    public BinNode(T value, BinNode<T> left = null, BinNode<T> right
= null) { Value = value; Left = left; Right = right; }
}
```

Приведённый ниже код строит дерево (рис. 10.3).

```
var root = new BinNode<int>(4);
var node1 = new BinNode<int>(1);
var node2 = new BinNode<int>(2);
var node3 = new BinNode<int>(3);
var node5 = new BinNode<int>(5);
var node6 = new BinNode<int>(6);
var node7 = new BinNode<int>(7);
var node8 = new BinNode<int>(8);
root.Left = node2;
root.Right = node5;
node2.Left = node1;
node2.Right = node3;
node5.Right = node7;
node7.Left = node6;
node7.Right = node8;
```

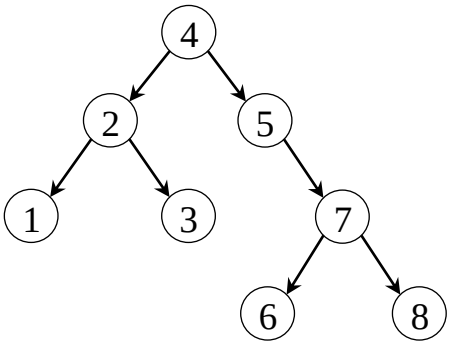



Рис. 10.3. Двоичное дерево

Иногда удобно хранить ссылку на родителя `Parent`, а также дополнительную информацию `G` о вершинах и ветвях. Такой класс `BinNode<T,G>` можно описать в виде

```
public class BinNode<T,G>
{
    // Ссылка на родителя, для корня дерева эта ссылка – null:
    public BinNode<T, G> Parent;
    public T Value;  // Ключевое поле
    public G NodeInfo;  // Дополнительная информация
    // Ссылка на левую ветвь (для листа null):
    public BinNode<T, G> Left;
    // Ссылка на правую ветвь (для листа null):
    public BinNode<T, G> Right;
// Конструктор нового элемента со значениями Value и Left, Right:
    public BinNode(BinNode<T, G> Parent = null, T value,
           BinNode<T, G> left = null, BinNode<T, G> right = null)
                  { Value = value; Left = left; Right = right; }
}
```

Для создания вершин с произвольной степенью следует использовать класс `TreeNode<T>` со списком ссылок на дочерние вершины.

```
public class TreeNode<T>
{
    public T Value { get; internal set; }  // Ключевое поле
    public List<TreeNode<T>> Children; // Список ссылок на дочек
    public TreeNode(T value, List<TreeNode<T>> children = null)
           { Value = value; Children = children; }
}
```

Неуникальные деревья могут иметь счётчики в каждой вершине, которые инкрементируются при повторном добавлении вершины и декрементируются при её удалении.

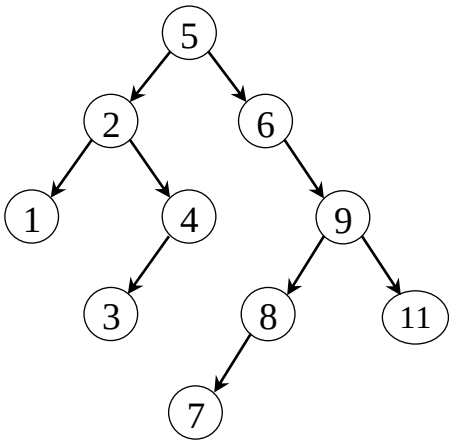


Рис. 10.4. Упорядоченное двоичное дерево

## 10.2 Упорядоченные двоичные деревья

В *упорядоченных* двоичных деревьях ключевое поле вершины больше такового в левом потомке и меньше такового в правом потомке (рис. 10.4).

**Добавление вершины**

В практике программирования используются различные стратегии добавления вершин в уникальные и неуникальные деревья. Добавление в

уникальное дерево может быть реализовано функцией, возвращающей булево значение в зависимости от фактического добавления вершины в дерево. Если добавляемая вершина существует, то добавление новой вершины не происходит и функция возвращает значение `false`, иначе – добавляемая вершина замещает лист дерева так, чтобы сохранить свойство упорядоченности, и возвращается значение `true`. В дерево (рис. 10.4) добавлена новая вершина 10 (рис. 10.5).

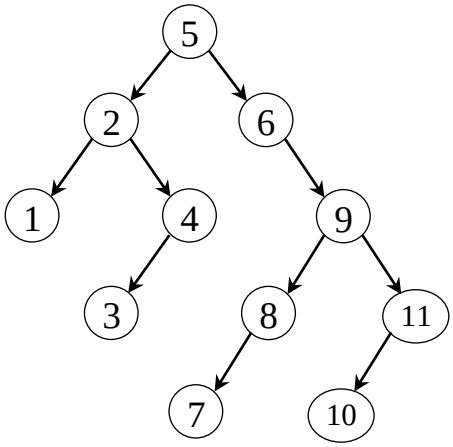



Рис. 10.5. Упорядоченное дерево с добавленной вершиной 10

Если дерево балансированное или близкое к нему, то его глубина оценивается функцией $O(\log_2 N)$. Добавление одной вершины в такое дерево выполняется за $O(\log_2 N)$ шагов.

Если дерево небалансированное, то его глубина может составить $O(N)$, как в связном списке. Добавление одной вершины в такое дерево выполняется за $O(N)$ шагов.

Если вершины описаны обобщённым классом с типом `T`, то сравнение значений этого типа выполняется посредством функции `CompareTo` интерфейса `IComparable<T>`, например, сравнение значения вершины `node.Value` с ключом `Key`:

```
node.Value.CompareTo(Key) == 0  //  Value == Key
node.Value.CompareTo(Key) > 0   //  Value > Key
node.Value.CompareTo(Key) < 0   //  Value < Key
```

**Поиск вершин**

Поиск в балансированном или близком к нему дереве выполняется за $O(\log_2 N)$ шагов. Поиск в дереве, близком к связному списку, выполнится за $O(N)$ шагов.

**Удаление вершин**

Удаление терминальных вершин или вершин, имеющих одно дочернее поддерево, тривиально. Для удаления вершины, имеющей два дочерних поддерева, следует выбрать одну из двух равноправных стратегий. Согласно первой стратегии удаляемую вершину надо заменить вершиной с максимальным ключом, вырезанной из левого поддерева (рис. 10.6). Согласно второй стратегии удаляемую вершину надо заменить вершиной с минимальным ключом, вырезанной из правого поддерева.

Удалим из дерева (рис. 10.6) вершину 32 по первой стратегии. Поиск максимальной вершины в левом поддереве вершины 32 ведётся, начиная с

вершины 21, и следуя только по правым веткам до тех пор, пока не обнаружится либо терминальная вершина, либо вершина, имеющая только левое поддерево. Такой вершиной является вершина 28, имеющая левую дочку 24. При удалении вершины 28 её место занимает вершина 24, а сама вершина 28 занимает место вершины 32. На этом удаление вершины 32 завершается.

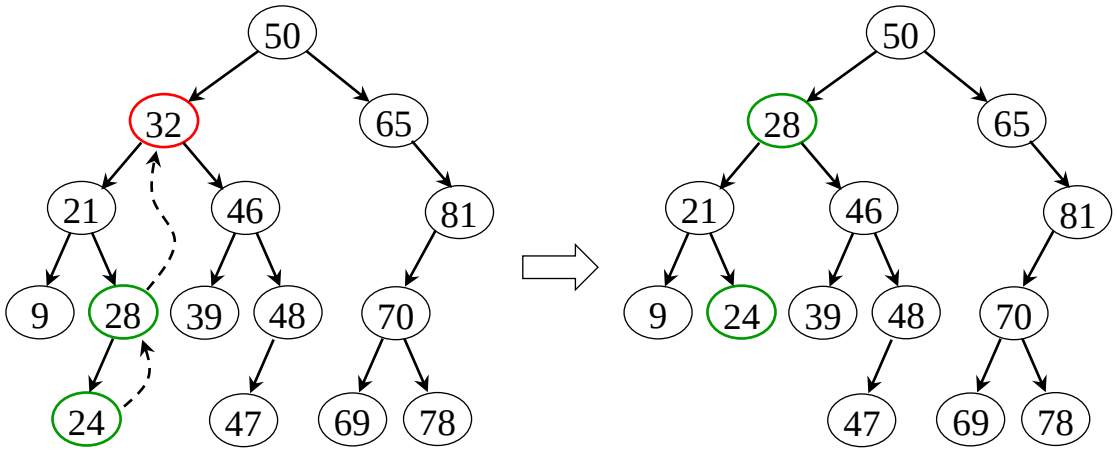



Рис. 10.6. Удаление вершины 32 из бинарного упорядоченного дерева

Для ускорения процесса удаления вершины 32 допускается заменить в ней только поле `Value` новым числом 28, а ссылки `Left` и `Right` оставить прежними.

В случае удаления вершин из неуникальных деревьев со счётчиками производится только декремент счётчика удаляемой вершины. Когда счётчик обнуляется, то возможны две стратегии поведения: либо полностью удалить из дерева такую вершину, либо оставить её с обнулённым счётчиком, но в дальнейшем корректно обрабатывать такого вида вершины при поиске, чтении и удалении.

### 10.3 Способы обхода вершин дерева

Цель обхода – пройти по всем вершинам дерева в определённом порядке и выполнить некоторые действия, в самом простом случае прочитать поле `Value`. Время выполнения обхода $O(N)$, где $N$ – количество вершин. Рассмотрим способы обхода дерева (рис. 10.7).

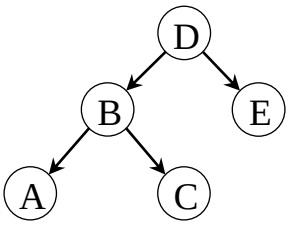



Рис. 10.7. Исходное дерево для обхода

**Обход в прямом порядке**

Алгоритм вначале обрабатывает вершину, затем её левую дочку, а после – правую. По

сути, это обход сначала вглубь. Англоязычное название `TraversePreorder`. Последовательность обхода D→B→A→C→E.

**Симметричный обход**

Алгоритм обрабатывает левый дочерний узел, затем саму вершину и только после этого правый дочерний узел. Англоязычное название `TraverseInorder`. Последовательность обхода A→B→C→D→E. Если дерево упорядоченное, то симметричный обход возвращает упорядоченный список значений вершин. На этом основан один из алгоритмов сортировки массивов: помещение элементов массива в упорядоченное дерево с последующим симметричным обходом. В среднем случае время такой сортировки $O(N \log N)$. Однако если массив уже упорядочен, то время увеличивается до $O(N^2)$.

*Прошивка* упорядоченного дерева – присвоение левым пустым наследникам ссылки на предыдущий элемент, а правым – на следующий (рис. 10.8). В прошитом дереве можно быстро выполнять нерекурсивный симметричный обход. Прошивка актуальна до первой модификации дерева.

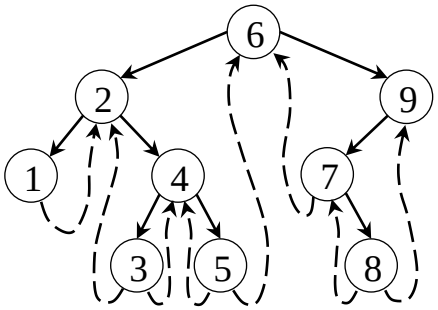



Рис. 10.8. Прошитое дерево

**Обход в обратном порядке**

Алгоритм обрабатывает сначала левый дочерний узел, затем правый узел и только после этого саму вершину. Англоязычное название `TraversePostorder`. Последовательность обхода A→C→B→E→D.

**Обход в ширину**

Алгоритм обрабатывает все вершины дерева на текущем уровне в порядке слева направо, а затем переходит к вершинам следующего уровня. По сути, это обход сначала вширь. Англоязычное название `TraverseBreadthFirst`. Последовательность обхода D→B→E→A→C.

**Пример реализации бинарного упорядоченного дерева**

Рассмотрим классы `BinNode<T>` и `BinTree<T>`, описывающие вершины двоичного упорядоченного дерева и операции добавления, поиска, удаления вершин, а также различные способы обходов дерева.

Метод `Add(T value)` добавляет вершины в неуникальное дерево. Добавление в неуникальное дерево со счётчиками вершин не реализовано.

Метод `Remove(T value)` удаляет вершину со значением `value`. Этот метод использует функцию `CutMax` для поиска максимальной (заменяющей) вершины в левом поддереве.

Метод обхода дерева вширь содержит операции поуровневого вывода вершин на экран таким образом, что вершины одного уровня выводятся в одну строку, вершины следующего уровня – во вторую и т.д.

```
public class BinNode<T>
{
    public T Value;  // Ключевое поле
    public BinNode<T> Left;
    public BinNode<T> Right;
    public BinNode(T value, BinNode<T> left = null, BinNode<T> right
= null)
    { Value = value; Left = left; Right = right; }
}
public class BinTree<T> where T : IComparable<T>
{
    private BinNode<T> Root; // корень дерева
    public BinTree() { }
    public BinTree(T root) { Add(root); }
    public BinTree(T[] items) { foreach (T item in items) Add(item);
}
    public void Add(T value)
    {
        Root = Add(Root, value);
    }
    public BinNode<T> Add(BinNode<T> node, T value)
    {
        if (node == null) node = new BinNode<T> (value);
        else
        {
            if (node.Value.CompareTo(value) > 0)
                node.Left = Add(node.Left, value); // Влево
            else node.Right = Add(node.Right, value); // Вправо
        }
        return node;
    }
    public bool Contains(T value)
    {
        return Contains(Root, value);
    }
    private bool Contains(BinNode<T> node, T value)
    {
        if (node == null) return false;
        if (node.Value.CompareTo(value) == 0) return true;
        if (node.Value.CompareTo(value) > 0)
return Contains(node.Left, value); // Влево
        else return Contains(node.Right, value); // Вправо
    }
    public void Remove(T value)
```

```
{
    Root = Remove(Root, value);
}
private BinNode<T> Remove(BinNode<T> node, T value)
{
    if (node == null) return node;
    if (node.Value.CompareTo(value) == 0) // Узел найден
    {
        if (node.Right == null) // Правой нет, левая - любая
        { node = node.Left; return node; }
        if (node.Left == null)  // Левой нет, правая - любая
        { node = node.Right; return node; }
        BinNode<T> item = CutMax(node, node.Left); // Обе ветви
        item.Left = node.Left;   // Замена левой ссылки
        item.Right = node.Right; // Замена правой ссылки
        node = item;             // Вставка поднятой вершины
        return node;
    }
    if (node.Value.CompareTo(value) > 0)
        node.Left = Remove(node.Left, value); // Идём влево
    else node.Right = Remove(node.Right, value); // Идём вправо
    return node;
}
private BinNode<T> CutMax(BinNode<T> parent, BinNode<T> node)
{   // Ищем самую правую вершину
    if (node.Right == null)  // правой нет
    {
        parent.Left = node.Left;  // вырезаем
        return node;
    }
    while (node.Right != null)
    {
        parent = node;
        node = node.Right;  // идём вправо
    }
    parent.Right = node.Left;  // вырезаем
    return node;
}
public void TraversePreorder() // Обход в прямом порядке
{
    TraversePreorder(Root);
}
private void TraversePreorder(BinNode<T> node)
{
    if (node != null)
    {
        Console.Write(node.Value + " ");
        TraversePreorder(node.Left);
```

```
            TraversePreorder(node.Right);
        }
    }
    public void TraverseInorder() // Симметричный обход
    {
        TraverseInorder(Root);
    }
    private void TraverseInorder(BinNode<T> node)
    {
        if (node != null)
        {
            TraverseInorder(node.Left);
            Console.Write(node.Value + " ");
            TraverseInorder(node.Right);
        }
    }
    public void TraversePostorder() // Обход в обратном порядке
    {
        TraversePostorder(Root);
    }
    private void TraversePostorder(BinNode<T> node)
    {
        if (node != null)
        {
            TraversePostorder(node.Left);
            TraversePostorder(node.Right);
            Console.Write(node.Value + " ");
        }
    }
    public void TraverseBreadthFirst() // Обход в ширину
    {
        var list = new List<BinNode<T>> { Root };
        TraverseBreadthFirst(list);
    }
    private void TraverseBreadthFirst(List<BinNode<T>> list)
    {
        if (list.Count == 0) return;
        var children = new List<BinNode<T>>();
        foreach (BinNode<T> node in list)
        {
            if (node != null)
            {
                Console.Write(node.Value + " ");
                children.Add(node.Left);
                children.Add(node.Right);
            }
        }
        Console.WriteLine();
```

```
            TraverseBreadthFirst(children);
        }
    }
```

***Пример*** использования

```
private static void Main(string[] args)
{
var tree = new BinTree<int>(new int[] { 9, 4, 7, 14, 13, 17 });
    tree.Add(6);
    tree.Add(12);
    tree.Add(1);
    tree.Add(5);
    tree.Add(8);
    tree.TraverseBreadthFirst();
    tree.Remove(14);
    Console.WriteLine("Удалили число 14:");
    tree.TraverseBreadthFirst();
    Console.WriteLine("Дерево содержит 7: " + tree.Contains(7));
    Console.WriteLine("Дерево содержит 15: " + tree.Contains(15));
    Console.Write("Симметричный обход: ");
    tree.TraverseInorder();
    Console.ReadKey();
}
```

Результат

```
9
4 14
1 7 13 17
6 8 12
5
Удалили число 14:
9
4 13
1 7 12 17
6 8
5
Дерево содержит 7: True
Дерево содержит 15: False
Симметричный обход: 1 4 5 6 7 8 9 12 13 17
```

### 10.4 Очередь с приоритетом

*Очередь с приоритетом* (англ. priority queue) – абстрактный тип данных, поддерживающий две обязательные операции:

а) `insert(ключ,значение)` – добавление пары `(ключ,значение)` в очередь, ключ играет роль приоритета;

б) `extract_maximum()` – извлечение из очереди пары `(ключ,значение)` с максимальным значением ключа.

Интерфейс очереди с приоритетом нередко расширяют операциями:

а) вернуть максимальный элемент без удаления из очереди;

б) изменить приоритет произвольного элемента;

в) удалить произвольный элемент;

г) слить две очереди в одну.

Эффективным способом реализации очереди с приоритетом является бинарная куча (пирамида), описанная в гл. 5. В бинарной куче обязательные операции можно выполнить в худшем случае за время $\log N$. Кроме бинарной кучи стоит упомянуть биномиальную кучу (англ. binomial heap), Фибоначчиеву кучу (англ. Fibonacci heap) и сливаемую кучу (англ. Mergable heap).

### 10.5 Дерамиды

Балансированное или близкое к нему дерево можно построить из неупорядоченного массива элементов. Если массив упорядочен, то перед построением балансированного дерева массив всегда можно перемешать. Однако когда дерево строится из элементов, получаемых по одному, и мы не можем перемешать получаемые элементы, то может получиться несбалансированное дерево. Для построения балансированного или почти балансированного дерева из потока элементов используются дерамиды.

*Дерамида* (англ. treap) – упорядоченное бинарное дерево с дополнительным параметром упорядочивания, роль которого играет приоритет каждой вершины, например, генерируемое случайным образом число (рис. 10.9). Ключи (символы) дерамиды подчиняются свойству упорядоченных бинарных деревьев, а приоритеты (числа) подчиняются свойству неубывающей кучи (пирамиды). Своё название дерамида получила от слияния слов дерево и пирамида. Также есть названия *дуча* (дерево и куча) и *декартово дерево*. Англоязычное название *treap* явилось результатом слияния tree и heap. Дерамида поддерживает свойство упорядоченного бинарного дерева и свойство неубывающей/невозрастающей кучи одновременно.

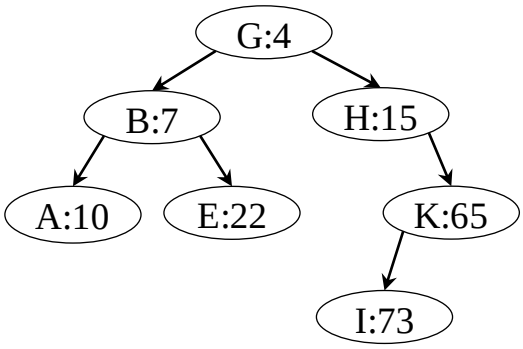


Рис. 10.9. Дерамида

Обязательной для дерамиды является операция добавления пары `(key, priority)`. Добавление выполняется по правилам упорядоченного дерева с полем `key` и последующим восстановлением свойства кучи по полю `priority`. По полю `key` выполняется выбор между левой и правой

ветвями. По полю `priority` выбирается направление вверх или вниз для восстановления свойства кучи.

Рассмотрим простой пример добавления вершины `C:25` в дерамиду (рис. 10.10). На первом этапе мы используем только поле `C` и игнорируем поле `25`. Рассматривая поле C как ключ упорядоченного дерева, мы помещаем вершину `C:25` левее корня `G`, правее вершины `B` и левее вершины `E`. После этого надо проверить и восстановить свойство неубывающей кучи для вершины `C:25` по полю `25`. Проверка успешна, так как поле `25` больше поле `22` родительской вершины. Восстанавливать свойство неубывающей кучи не требуется.

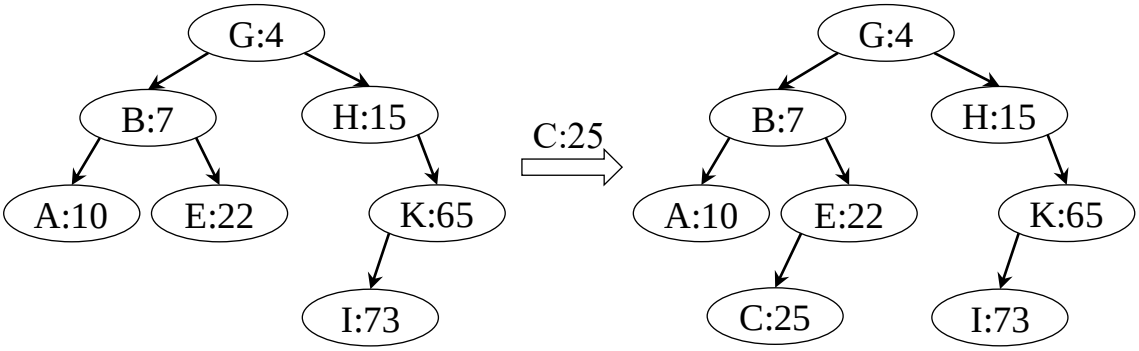



Рис. 10.10. Добавление пары C:25 без восстановления свойств дерамиды

Посмотрим пример добавления вершины с восстановлением свойств дерамиды. Добавим в полученную дерамиду пару `D:9` (рис. 10.11). Вначале, используя поле `D`, пройдём по цепочке вершин `G→B→E→C` и поместим добавляемую пару `D:9` в правую ветвь вершины `C:25`. Сейчас свойство неубывающей кучи нарушено, так как вершина с ключом 9 расположена ниже вершины с ключом 25.

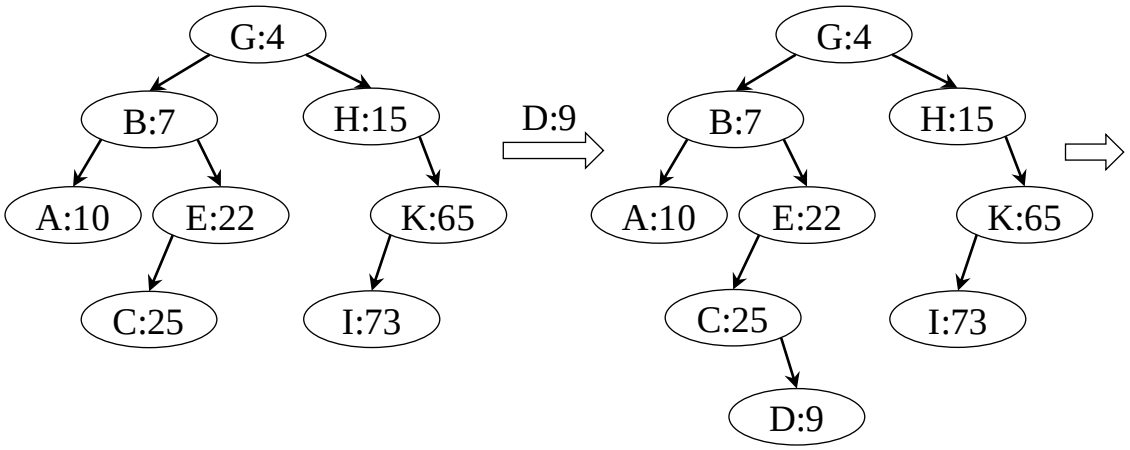



Рис. 10.11. Добавление пары D:9

Для восстановления свойства кучи надо поднимать вершину `D:9` до тех пор, пока её поле 9 станет не меньше поля приоритета своих потомков (рис. 10.12).

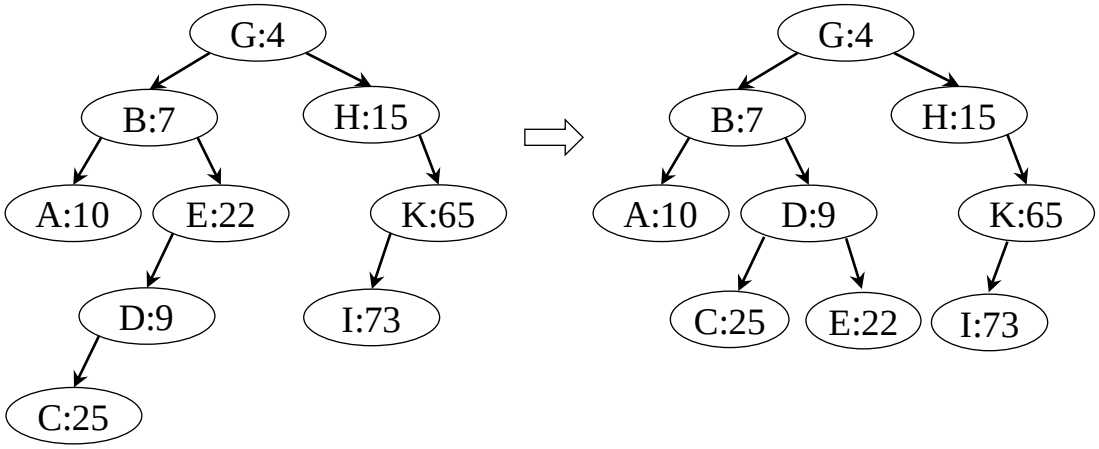



Рис. 10.12. Восстановление свойства кучи

**Задача 1.** Разработать рекурсивную функцию проверки зеркальной симметрии левого и правого поддеревьев корня бинарного дерева. Функция должна возвращать булево значение `True` или `False`.

**Задача 2.** Разработать рекурсивную функцию проверки зеркальной симметрии дочерних поддеревьев всех родительских вершин. Функция должна возвращать булево значение `True` или `False`.

**Задача 3.** Разработать рекурсивную функцию определения высоты бинарного дерева.

**Задача 4.** Разработать рекурсивную функцию проверки полноты дерева. Функция должна возвращать булево значение `True` или `False`.

**Задача 5.** Разработать рекурсивную функцию проверки идеальности дерева. Функция должна возвращать булево значение `True` или `False`.

**Контрольные вопросы**

1. Дайте определение дерева.
2. В чём отличие высоты и глубины узла дерева?
3. Чем определяется степень дерева?
4. Как называется последняя вершина в ветви дерева?
5. Какой признак имеет завершённое дерево?
6. Сколько листьев в идеальном дереве известной глубины?
7. В чём отличие уникальных и неуникальных деревьев?
8. Какие существуют способы обхода вершин дерева?
9. Каков способ реализации очереди с приоритетом?
10. Для чего предназначены дерамиды?

# 11 Балансированные деревья

Время поиска или добавления вершины в упорядоченное дерево зависит только от его высоты, а наименьшую высоту при том же количестве вершин имеют балансированные деревья. Поэтому важно поддерживать свойство сбалансированности деревьев.

При добавлении/удалении вершин баланс дерева может быть нарушен. Поэтому, после таких операций желательно выполнять балансировку. Однако процедура балансировки требует дополнительного времени. Выходом из положения является введение менее строгого критерия сбалансированности. Это ведёт к упрощению алгоритма балансировки за счёт незначительного увеличения высоты дерева и, следовательно, времени поиска в нём.

Различные способы ослабления требований к балансу дерева приводят к двоичным балансированным деревьям специального вида. Здесь рассматриваются два таких вида: АВЛ-деревья и красно-чёрные деревья.

## 11.1 АВЛ-деревья

Георгий Максимович Адельсон-Вельский и Евгений Михайлович Ландис в 1962 г. предложили облегчённый критерий балансировки: дерево называется сбалансированным, если высоты двух поддеревьев каждой вершины отличаются не более чем на единицу. АВЛ-деревья стали самыми первыми балансированными структурами данных. При добавлении или удалении вершины дерево повторно балансируют по необходимости, чтобы сохранить данное свойство.

Каждая вершина АВЛ-дерева, хранит *показатель сбалансированности* в виде разности между высотами левого и правого поддеревьев. Значение -1 говорит о том, что вершина утяжелена справа, 0 – поддеревья равной высоты, +1 – вершина утяжелена слева. На рисунке 11.1 показатели сбалансированности расположены над вершинами, высоты поддеревьев – над рёбрами.

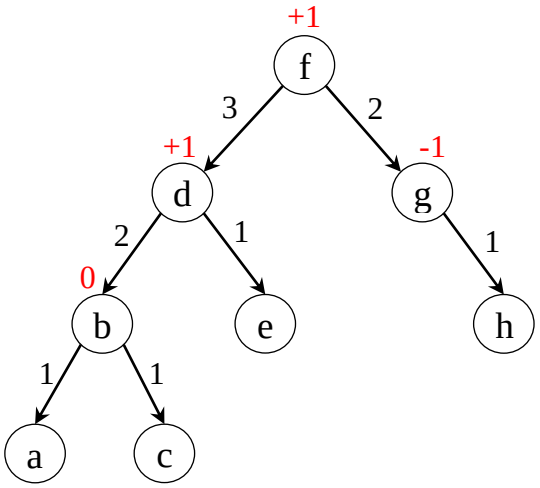



Рис. 11.1. АВЛ-дерево с показателями баланса

Такой ослабленный критерий АВЛ-баланса допускает разбалансированность (рис. 11.1). После операций добавления/удаления

надо проверять показатель сбалансированности вершин дерева. Если он находится в разрешённых пределах от -1 до +1, то механизм балансировки не запускается, иначе – запускается.

Вершины АВЛ-дерева описываются классом `AVLNode<T>`.

```
public class AVLNode<T>
{
    public T Value; // ключ
    public sbyte Balance; // показатель баланса
    public AVLNode<T> Left;
    public AVLNode<T> Right;
    public AVLNode(T value, sbyte balance = 0, AVLNode<T> left =
null, AVLNode<T> right = null)
    {Value = value; Balance = balance; Left = left; Right = right;}
}
```

Преобразования дерева *левым или правым вращением* восстанавливают баланс бинарного упорядоченного дерева (рис. 11.2).

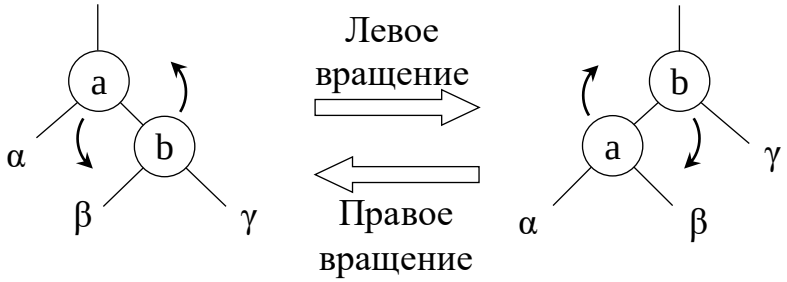



Рис. 11.2. Левое и правое вращения

Вращения влево или вправо укорачивают/удлиняют боковые ветви α и γ, но высоту средней ветви β оставляют без изменений. Возможны четыре случая поведения механизма балансировки в зависимости от того, куда добавляется новая вершина:

1) левое поддерево левой дочерней вершины → правое вращение;

2) левое поддерево правой дочерней вершины → левое + правое вращение;

3) правое поддерево левой дочерней вершины → правое + левое вращение;

4) правое поддерево правой дочерней вершины → левое вращение.

***Пример 1.*** Одно правое вращение (рис. 11.3) устраняет дисбаланс в вершине `e`, если самая длинная ветвь от вершины `e` уходит влево-влево. Вершина `d` сохраняет свою глубину, так как она передаётся уходящей вниз вершине `e`.

Таким образом, при правом вращении происходит «перетекание» вершины из левой в правую ветвь с сохранением глубины средних ветвей.

Следовательно, правое/левое вращение применяется для укорачивания длинной боковой ветви.

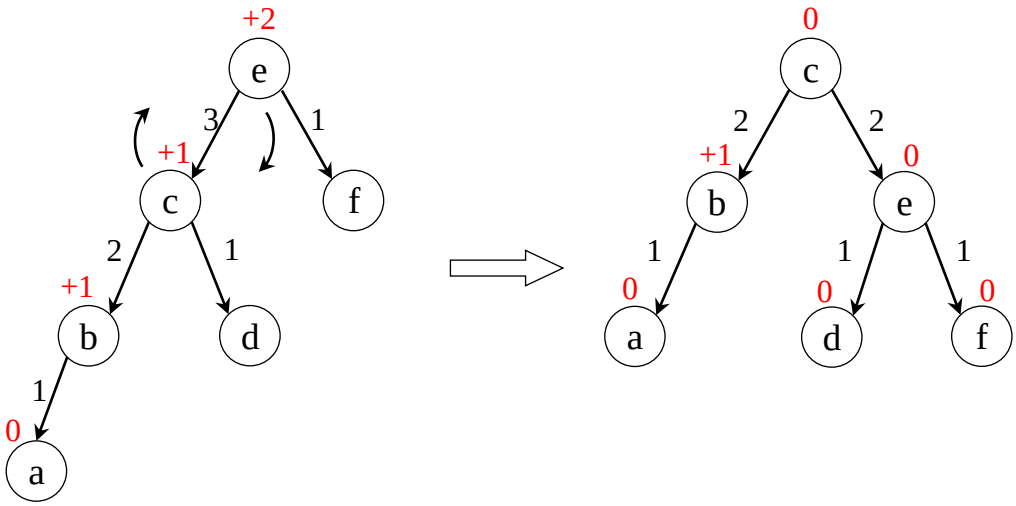



Рис. 11.3. Устранение дисбаланса правым вращением

Для устранения дисбаланса типа левый-правый или правый-левый вначале надо укоротить среднюю ветвь, удлиняя при этом боковую ветвь. Тем самым мы выходим на случай длинной боковой ветви.

***Пример 2.*** Пусть правое поддерево левой дочерней вершины корня дерева `e` самое длинное (рис. 11.4). Одно правое вращение не устраняет дисбаланс в вершине `e`, т.к. вершина `c` сохраняет свою глубину, ибо передаётся уходящей вниз вершине `e`.

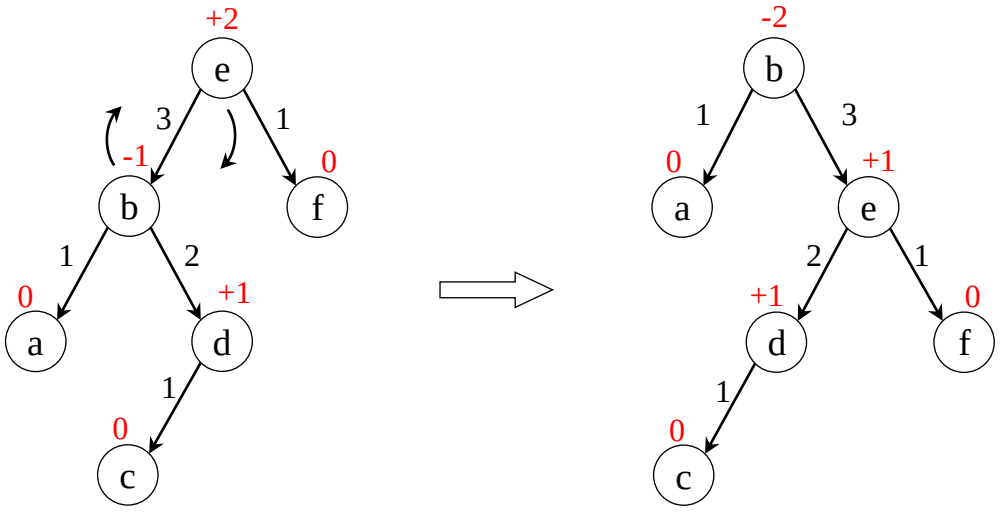



Рис. 11.4. Правое вращение не устраняет дисбаланс

Поэтому вначале надо сделать левое вращение в левой дочке для того, чтобы средняя ветвь стала боковой (рис. 11.5). И после совершить правый поворот, опускающий вершину `e` в правую ветвь.

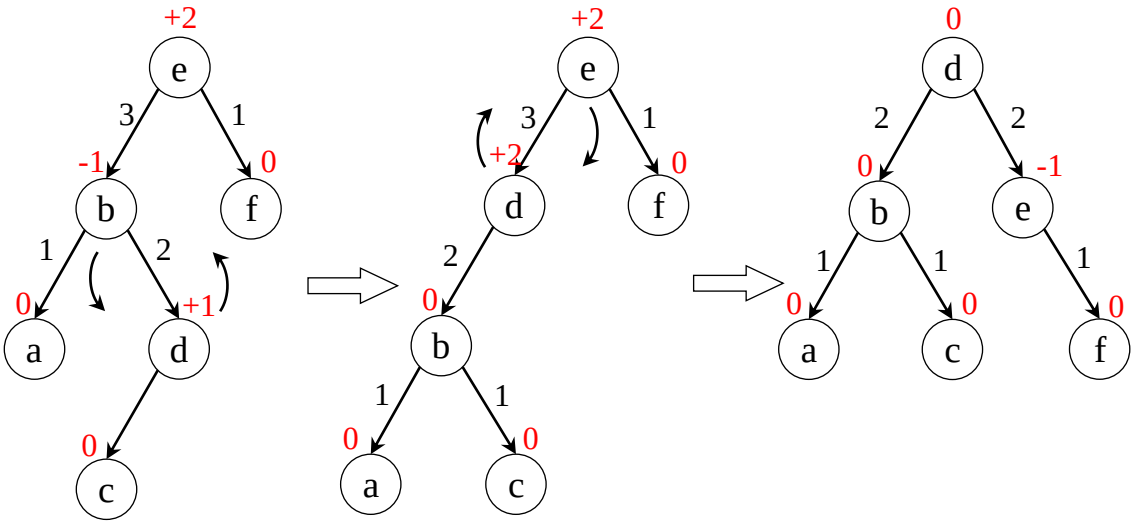



Рис. 11.5. Устранение дисбаланса левым и правым вращением

**Балансировка вращениями в общем случае**

Случай глубокой левой боковой ветви «влево-влево». Когда добавленная вершина содержится в поддереве `A1` и оно ниже поддерева `B2` на два ребра, надо поднять вершину `a`, сделав её корнем, и опустить вершину `b`, передав ей поддерево `A2` (рис. 11.6). Эта операция выполняется правым вращением. В случае с глубокой правой боковой ветвью балансировка выполняется левым вращением.

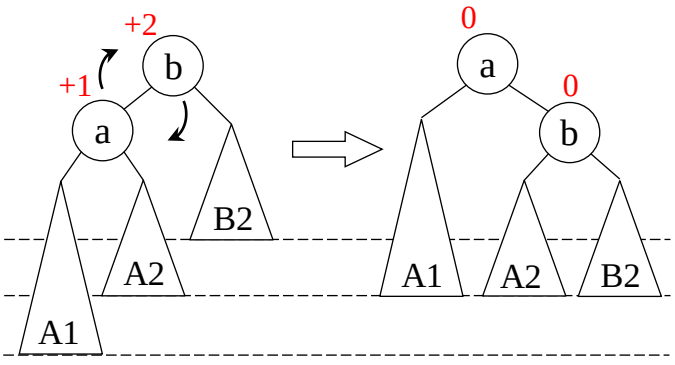



Рис. 11.6. Балансировка боковой ветви

Случай глубокой средней ветви «влево-вправо». Пусть добавленная вершина содержится в правом поддереве левой дочерней вершины (рис. 11.7), которое в свою очередь состоит из вершины `a` и двух поддеревьев `A1` и `A2`. Абсолютно неважно, в каком из этих поддеревьев находится добавленная вершина, так как поддерево с корнем `a` все равно на два уровня ниже, чем поддерево `B2`. Это значит, что поддерево с корнем `c` на два уровня ниже, чем поддерево `B2`, и получается, что дерево с вершиной `b` не сбалансировано.

Чтобы убедиться в правильности наших рассуждений, достаточно во-первых, увидеть, что высота всех поддеревьев `C1`, `A1`, `A2` и `B2` одинакова, во-вторых, подсчитать число вершин от корня `b` до вершины поддерева `A1` (оно равно трём) и до вершины поддерева `B2` (оно равно одному).

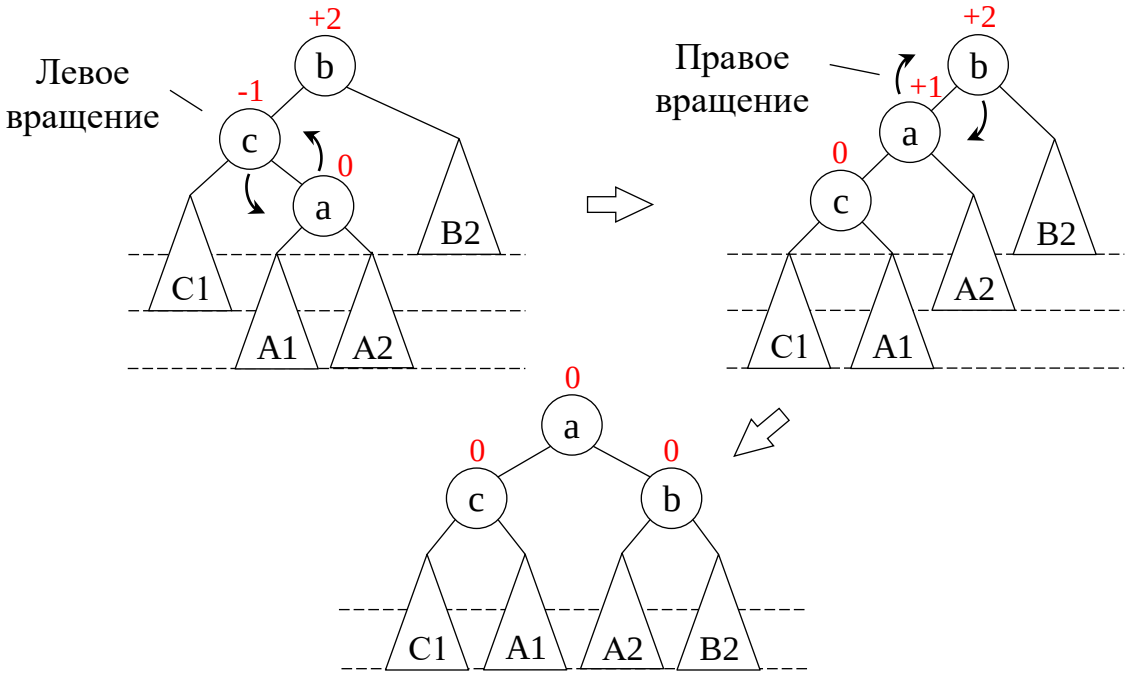



Рис. 11.7. Балансировка средней ветви

Для балансировки дерева (рис. 11.7), нужно использовать вначале левое, а затем правое вращение. После левого вращения образуется случай глубокой левой боковой ветви, которая становится на два уровня ниже правого поддерева корня `b`. Теперь можно сбалансировать всю структуру с помощью правого вращения.

Случай глубокой средней ветви «вправо-влево» аналогичен рассмотренному и балансируется правым, а затем левым вращением.

**Большие вращения**

Последовательно выполненные левое и правое вращения можно заменить *большим правым вращением* (рис. 11.8).

*Большое левое вращение* заменяет последовательно выполненные правое и левое вращения (рис. 11.9).

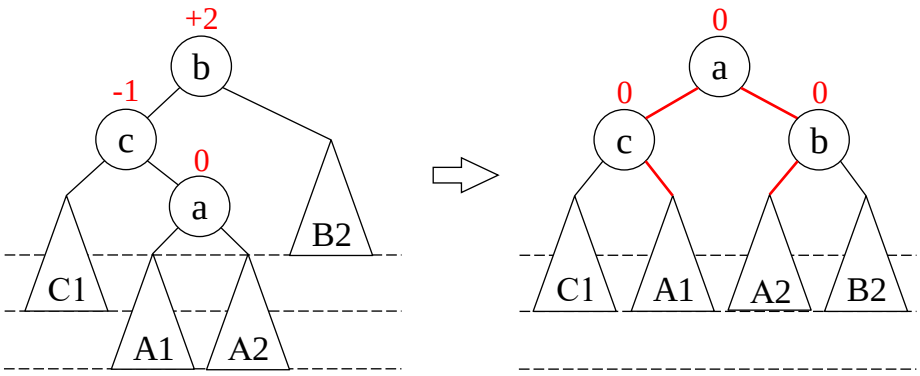



Рис. 11.8. Большое правое вращение

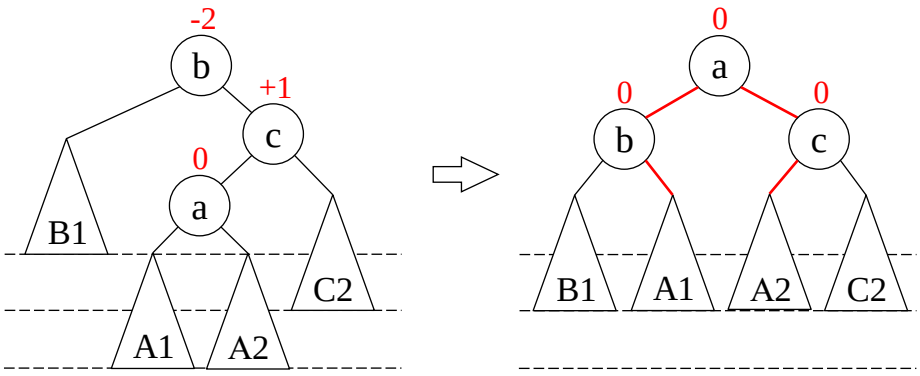



Рис. 11.9. Большое левое вращение

**Алгоритм добавления вершины в АВЛ-дерево:**

1. Пройти путь поиска от корня вниз и убедиться, что ключа нет.

2. Добавить новую вершину вместо листа и присвоить ей нулевой показатель баланса.

3. При возврате из рекурсии к корню модифицировать и проверить баланс каждого предка. При возврате из левого поддерева баланс следует инкрементировать, иначе – декрементировать.

4. Если модифицированный баланс равен нулю, то балансировку при возврате к корню следует прекратить. Если баланс вышел за пределы разрешённого диапазона, то его следует восстановить путём вращений.

Например, при возврате из левого потомка выполняется псевдокод:

```
balance++;
if (balance > 1)
    if (left_balance > 0) малое_правое_вращение;
    else большое_правое_вращение;
```

**Алгоритм удаления вершины из АВЛ-дерева:**

1. Если ключ не найден, то алгоритм завершён. Балансировка не выполняется.

2. Если удаляемая вершина терминальная, то заменить её пустой ссылкой. Если удаляемая вершина имеет одно поддерево, то исключить её. Иначе применить алгоритм удаления в упорядоченном двоичном дереве, используя для замены подходящую вершину из наиглубокого поддерева.

3. При возврате из рекурсии к корню модифицировать и проверить баланс каждого предка. При возврате из левого поддерева баланс следует декрементировать, иначе – инкрементировать.

4. Если модифицированный баланс неравен нулю, то балансировку при возврате к корню следует прекратить. Если баланс вышел за пределы разрешённого диапазона, то его следует восстановить путём вращений.

**Оценка вычислительной сложности**

Операции поиска, добавления и удаления выполняются за время $O(\log_2 N)$, $N$ – число вершин АВЛ-дерева. После добавления/удаления вершины потребуется в лучшем случае ноль поворотов, в худшем – $O(\log_2 N)$ поворотов. Высота АВЛ-дерева при заданном числе вершин $N$ лежит в диапазоне

$$\log_2(N+1) \le H \le \lfloor 1{,}4404 \cdot \log_2(N+1{,}17) - 0{,}328 \rfloor.$$

Максимальная высота балансированного АВЛ-дерева при 32-битной адресации равна 45, при 64-битной – 91.

## 11.2 Красно-чёрные деревья

Рудольф Байер в 1978 г. предложил критерий балансировки, при котором максимальная и минимальная высоты каждой вершины отличаются не более чем в два раза. Такая балансировка достигается введением цветового атрибута вершин дерева и особых свойств дерева. Цвет вершин может быть красным или чёрным. Такие деревья назвали *красно-чёрными* (англ. red-black tree, RBT).

Свойства RBT:

1. Вершина либо красная, либо чёрная.
2. Корень чёрный (необязательное свойство).
3. Все листья чёрные.
4. Красная вершина всегда имеет два листа или две дочерние чёрные вершины. Чёрная вершина может иметь произвольную комбинацию потомков.
5. Всякий путь от данной вершины до её листа-потомка содержит одинаковое число чёрных вершин.

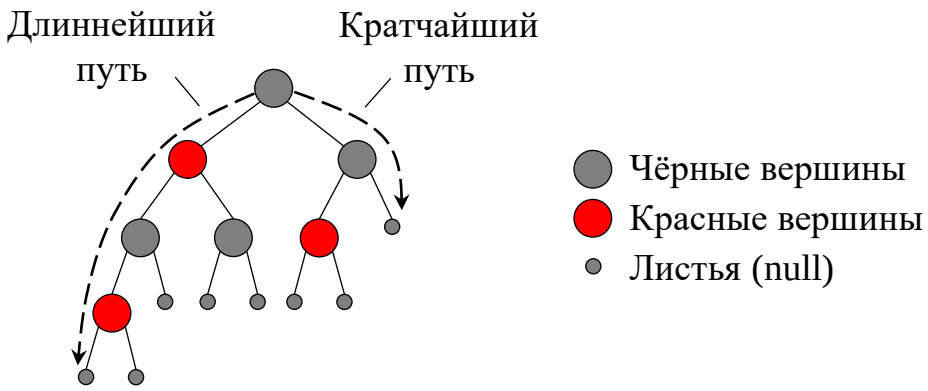



Рис. 11.10. Пути одинаковой чёрной длины, равной трём

Листьями в RBT считают пустые ссылки `null`, которые всегда чёрные. Длину пути определяют по количеству только чёрных вершин. Такую длину будем называть *чёрной длиной*, чтобы отличать её от реальной длины. Пути с фактически разной длиной могут иметь равные чёрные длины (рис. 11.10).

Операция чтения в красно-чёрном дереве ничем не отличается от чтения в бинарном дереве, потому что любое красно-чёрное дерево является частным случаем бинарного дерева. Однако результат вставки или удаления может привести к нарушению свойств RBT. Восстановление свойств осуществляется поворотами, подобными в АВЛ-деревьях.

**Вставка**

В красно-чёрном дереве новый элемент вставляется вместо чёрного листа в виде красной внутренней вершины с двумя чёрными листьями. Дальше происходит анализ структуры изменившегося дерева и при необходимости перекраска/балансировка дерева. Здесь возможно пять случаев.

Примем обозначения:

– *N* – текущая вершина (красная);
– *P* – родитель *N*;
– *G* – дедушка *N*;
– *U* – дядя *N*.

*Случай 1*. В пустое дерево добавили вершину. Красная текущая вершина *N*, являющаяся корнем дерева, перекрашивается в чёрный цвет.

*Случай 2*. Родитель *P* текущего узла чёрный, свойства 4 и 5 не нарушаются. Перекраска/балансировка не нужна.

*Случай 3*. Если родитель *P* и дядя *U* красные, то они перекрашиваются в чёрный, а дедушка *G* – в красный цвет (рис. 11.11). Однако красный дедушка *G* теперь может нарушить свойства 2 (корень чёрный) или 4

(потомки красной вершины чёрные). Чтобы это проверить и исправить, процедура перекраски/балансировки выполняется рекурсивно для *G*.

*Случай 4 «левый-правый».* Родитель *P* красный, дядя *U* и дедушка *G* чёрные (рис. 11.12). Текущий узел *N* – правый потомок *P*, а *P* – левый потомок *G*. У красной вершины *P* разноцветные потомки, свойство 4 (потомки красной вершины чёрные) нарушено. Поворот влево сводит задачу к случаю 5 «левый-левый». Но свойство 4 всё ещё нарушено.

*Случай 5 «левый-левый».* Родитель *P* красный, дедушка *G* и дядя *U* чёрные, текущий узел *N* – левый потомок *P* и *P* – левый потомок *G* (рис. 11.13). В этом случае выполняется правый поворот. В результате родитель *P* становится корнем дерева, а дедушка G – его потомком. Обмен цвета *P* и *G* завершают балансировку.

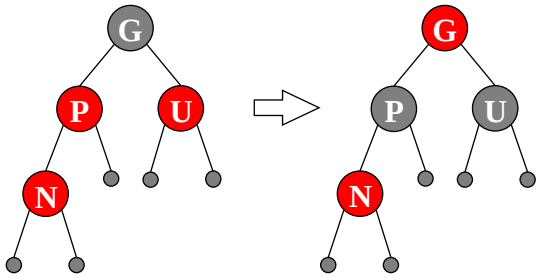



Рис. 11.11. Перекраска вершин

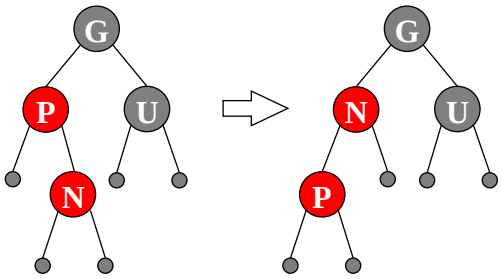



Рис. 11.12. Левое вращение

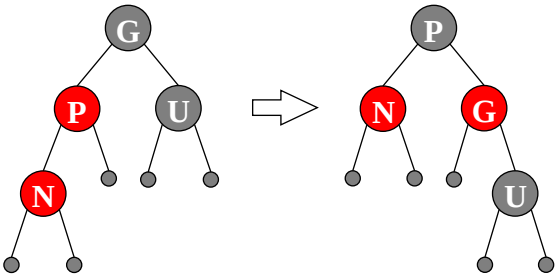



Рис. 11.13. Правое вращение

Сейчас можно выразить главную идею RBT-балансировки. Нежелательно перекрашивать вновь добавленную вершину в чёрный цвет, надо попытаться оставить её красной, даже за счёт дополнительных поворотов и перекрашиваний. Эти затраты не будут напрасны, так как дерево улучшит свою балансировку.

**Удаление**

При удалении вершины с двумя внутренними потомками мы вырезаем либо наибольший элемент в левом поддереве, либо наименьший элемент в правом поддереве и перемещаем его значение в удаляемую вершину. Вырезанная вершина не может иметь сразу двух потомков, так как в противном случае она не являлась бы наибольшим/наименьшим элементом. Таким образом, случай удаления вершины, имеющей два потомка, сводится к случаю удаления вершины, имеющей не более одного потомка.

Возможные варианты:

1. Удаляемая красная вершина *C* не имеет потомков (рис. 11.14). Заменяем вершину *C* листом. Чёрная высота родителя *A* не изменилась.

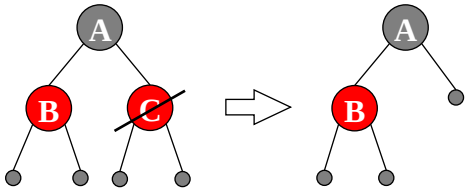



Рис. 11.14. Замена листом

2. Удаляемая чёрная вершина *A* имеет одного красного потомка *B* (рис. 11.15). Заменяем значение вершины *A* значением потомка *B*. Чёрная высота не изменилась.

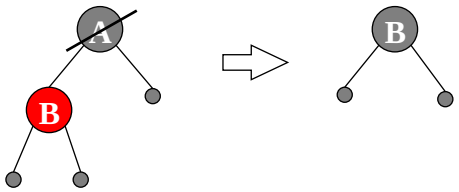



Рис. 11.15. Замена значением потомка

3. Удаляемая чёрная вершина *C* имеет красного предка *A*, чёрного брата *B* и не имеет потомков. Здесь возможны четыре варианта:

а) если брат не имеет потомков (рис. 11.16, а), то выполняем правый поворот;

б) если брат имеет двух красных потомков (рис. 11.16, б), то совершаем правый поворот и перекрашиваем;

в) если брат имеет одного левого красного потомка (рис. 11.16, в), то делаем правый поворот;

г) если брат имеет одного правого красного потомка (рис. 11.16, г), то выполняем левый поворот и получаем вариант (рис. 11.16, в).

4. Удаляемая вершина, её брат и родитель чёрные. Используем правила удаления, принятые для бинарного дерева.

5. Удаляемая вершина и её родитель чёрные, а брат красный. Согласно RB-свойствам у брата должно быть два чёрных потомка. Используем правила удаления, принятые для бинарного дерева.

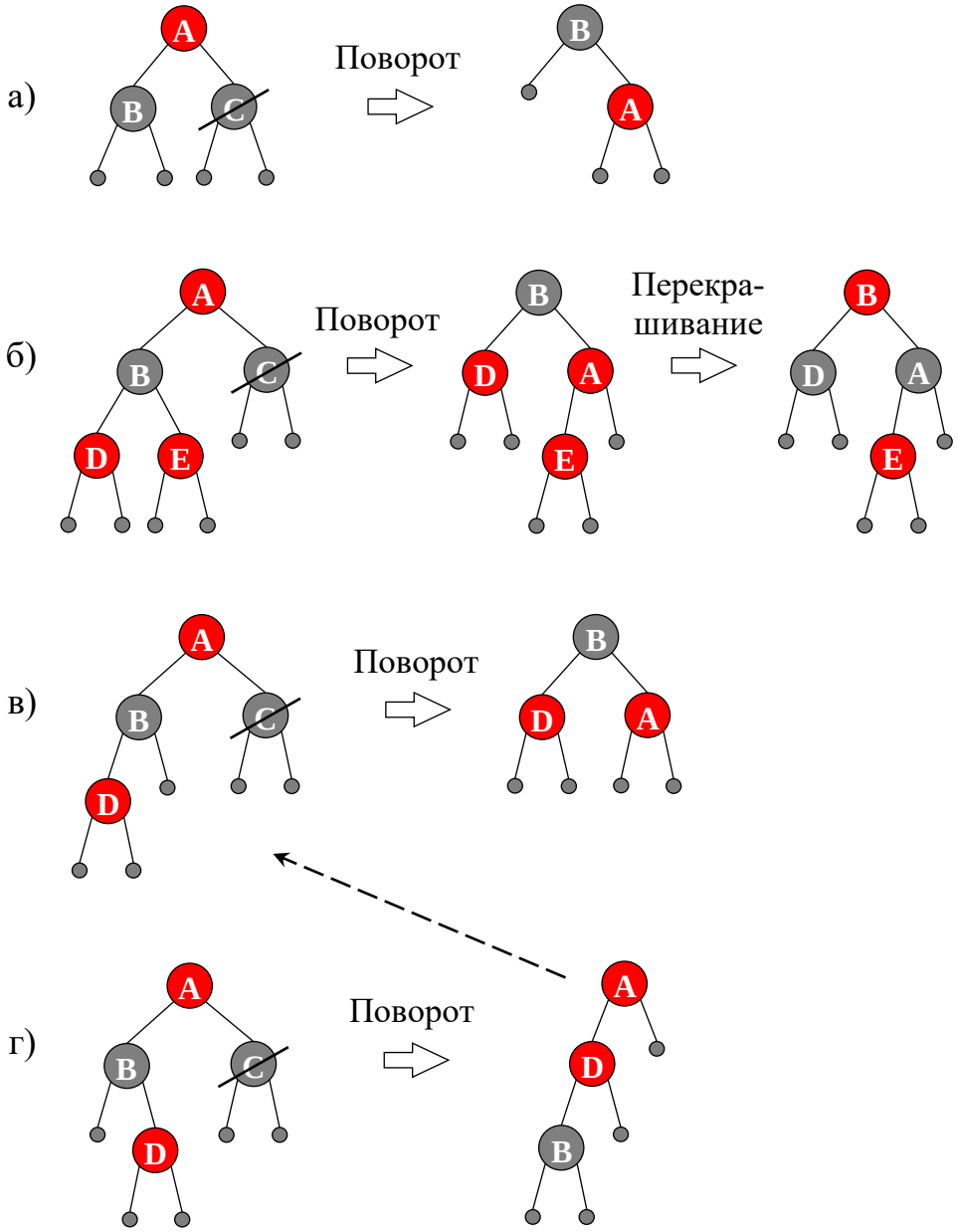



Рис. 11.16. Варианты удаления чёрной вершины, имеющей брата

### 11.3 Сравнение балансированных деревьев

Минимальное число вершин $N$ АВЛ-дерева при заданной высоте $H$

| $H$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | 1 | 2 | 4 | 7 | 12 | 20 | 33 | 54 | 88 | 143 |

Для АВЛ-дерева $N$ растёт как ряд Фибоначчи

$$N(H) = N(H-1) + N(H-2) + 1,$$

для которого известна формула $i$-го члена геометрической прогрессии

$$N(H) = \Theta(\lambda^H) = \Theta(1{,}62^H),$$

где $\lambda = \frac{\sqrt{5}+1}{2} \approx 1{,}62$ – знаменатель геометрической прогрессии.

Минимальное число вершин $N$ красно-чёрного дерева

| $H$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | 1 | 2 | 4 | 6 | 10 | 14 | 22 | 30 | 46 | 62 |

для чётной высоты $H$:

$$N(H) = 2^{\frac{H}{2}+1} - 2 = \Theta\left(\sqrt{2}^H\right) = \Theta(1{,}41^H);$$

для нечётной высоты $H > 1$:

$$N(H) = 3 \cdot 2^{\frac{H-1}{2}} - 2 = \Theta\left(\sqrt{2}^H\right) = \Theta(1{,}41^H).$$

Следовательно, при том же количестве листьев красно-чёрное дерево может быть выше АВЛ-дерева не более чем в $\frac{\log 1{,}62}{\log 1{,}41} \approx 1{,}388$ раза. Это означает, что поиск в красно-чёрном дереве дольше на 39%, чем в АВЛ-дереве.

Зная высоту балансированного дерева $H = \lfloor \log_2(N) \rfloor$ легко определить, что при том же количестве листьев красно-чёрное дерево может быть выше балансированного дерева в $\frac{\log 2}{\log 1{,}41} \approx 2{,}017$ раза, а АВЛ-дерево выше балансированного дерева в $\frac{\log 2}{\log 1{,}62} \approx 1{,}437$ раза. Из приведённых характеристик можно сделать вывод, что самый быстрый поиск выполняется в балансированном дереве, второе место по этому показателю занимают АВЛ-деревья, на третьем месте находятся красно-чёрные деревья.

Балансировка после вставки в обоих видах деревьев выполняется не более чем за два поворота, но в RB-дереве дольше, так как оно выше.

Балансировка после удаления в RB-дереве выполняется не более чем за три поворота. В АВЛ-дереве она может затребовать количество поворотов равное глубине удаляемой вершины. Поэтому теоретически удаление в RB-дереве быстрее. Однако эксперимент показывает, что АВЛ-деревья быстрее красно-чёрных во всех операциях.

Максимальная высота красно-чёрного дерева при 32-битной адресации равна 64 (против 45 АВЛ-дерева), при 64-битной – 128 (против 91 АВЛ-дерева).

Известно множество балансированных деревьев со своими критериями балансировки. Их упрощённая классификация представлена ниже.

а) АВЛ-дерево;

– дерево Фибоначчи.

б) B-дерево;
– 2-3-дерево;
– 2-3-4-дерево;
– B+-дерево;
– B*-дерево.
в) красно-чёрное дерево;
г) расширяющееся дерево.

**Задача 1.** Разработать функцию проверки сбалансированности двоичного дерева. Функция должна возвращать значение `True` или `False`.

**Задача 2.** Разработать функцию проверки АВЛ-сбалансированности двоичного дерева. Функция должна возвращать значение `True` или `False`.

**Задача 3.** Разработать функцию проверки RB-сбалансированности двоичного дерева. Функция должна возвращать значение `True` или `False`.

**Контрольные вопросы**

1. От чего зависит время поиска информации в упорядоченном бинарном дереве?
2. Какие деревья имеют наименьшую высоту при равном количестве листьев?
3. В чём заключается процедура балансировки дерева?
4. Когда запускается процедура балансировки дерева?
5. Для чего введены ослабленные критерии балансировки?
6. Каков критерий АВЛ-балансировки?
7. Для чего предназначены вращения деревьев вправо и влево?
8. В чём заключается критерий RB-балансировки?
9. Каковы свойства АВЛ-деревьев?
10. Каковы свойства RBT?
11. Во сколько раз могут отличаться реальные длины путей в RBT, когда эти пути имеют одинаковую чёрную длину?
12. Во сколько раз высота АВЛ-дерева больше высоты балансированного дерева при том же количестве листьев?
13. Во сколько раз высота RBT больше высоты балансированного дерева при том же количестве листьев?

# 12 Способы представления графов

## 12.1 Матрица смежности

*Матрица смежности* – матрица, в каждой $i$-й строке, $j$-м столбце которой записывается число, определяющее наличие связи от $i$-й к $j$-й вершине графа. Единица показывает наличие ребра/дуги между вершинами, а ноль – отсутствие. Это наиболее удобный способ представления плотных неизменяемых графов.

Достоинства:

а) простота и скорость поиска смежных вершин;

б) простота и скорость модификации множества рёбер.

Недостатки:

а) если граф разрежён, то бóльшая часть памяти будет напрасно тратиться на хранение нулей;

б) изменение размера графа влечёт создание новой матрицы смежности;

в) большие затраты памяти, прямо пропорциональные квадрату количества вершин $|V|^2$, $V$ – множество вершин.

Матрица смежности неориентированного графа симметрична. Петли записываются в главную диагональ (рис. 12.1, а). Наличие кратных рёбер требует создания матрицы списков рёбер.

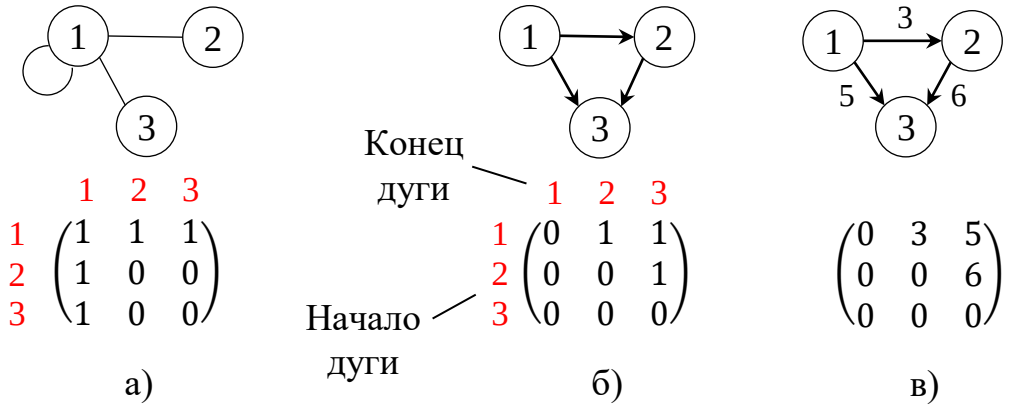



Рис. 12.1. Матрица смежности:
а) неориентированного графа;
б) орграфа;
в) взвешенного орграфа

Матрица смежности орграфа несимметрична (рис. 12.1, б) и представляется в программе двумерным массивом

```
int[,] orgraph = { {0, 1, 1 }, { 0, 0, 1 }, { 0, 0, 0 } };
```

У взвешенного орграфа дуги имеют веса (рис. 12.1, в) и именно эти веса содержит матрица смежности

```
int[,] orgraph = { {0, 3, 5 }, { 0, 0, 6 }, { 0, 0, 0 } };
```

### 12.2 Матрица инцидентности

*Матрица инцидентности* – матрица, в которой строки соответствуют вершинам графа, а столбцы – дугам графа. Элемент матрицы ицидентности орграфа на пересечении строки *i* со столбцом *j* содержит:

а) 1 в случае, если дуга *j* выходит из вершины *i*;

б) -1 если дуга *j* входит в вершину *i*;

в) 1 и -1, если дуга *j* является петлёй, инцидентной вершине *i*;

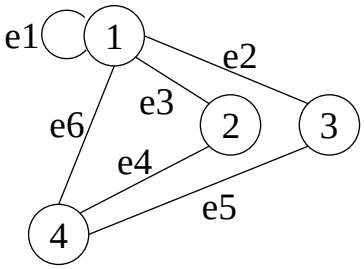
г) 0 если дуга не инцидентна вершине.

Элемент матрицы инцидентности неориентированного графа на пересечении строки *i* со столбцом *j* содержит:

а) 1 в случае, если ребро *j* инцидентно вершине *i*;

б) 0 если ребро не инцидентно вершине.

Столбец, описывающий ребро, должен иметь не более двух единиц. В неориентированном графе (рис. 12.2) петля e1 привязана к вершине 1.




| | e1 | e2 | e3 | e4 | e5 | e6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 1 | 1 | 1 |

Рис. 12.2. Матрица инцидентности

Достоинства:

а) быстрое нахождение циклов в графе;

б) может использоваться для представления мультиграфов и гиперграфов, в последнем случае столбец может содержать больше двух единиц.

Недостатки:

а) большие затраты памяти, пропорциональные |V|×|E|, V – множество вершин, E – множество рёбер/дуг;

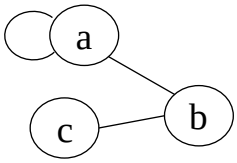
б) если граф разрежён, то бóльшая часть памяти напрасно тратится на хранение нулей;

в) изменение размера графа или количества дуг влечёт создание новой матрицы инцидентности.

### 12.3 Коллекции кустов

*Коллекция кустов* – коллекция элементов, содержащих вершины графа и множества смежных им вершин. Размер занимаемой памяти $|V| + |E|$

Как сама коллекция, так и множества смежных вершин могут представляться массивами, списками и связными списками (рис. 12.3).




| Вершина | Множество смежных вершин |
|---|---|
| a | {a, b} |
| b | {a, c} |
| c | {b} |

$$\{ (a,\{a,b\}),\ (b,\{a,c\}),\ (c,\{b\}) \}$$

Рис. 12.3. Граф и коллекция его кустов

Достоинство – экономное расходование памяти, что выгодно для представления разряжённых графов.

Недостаток – большое время модификации размера графа и множества рёбер.

Рассмотрим разновидности графов, построенных из кустов.

1. *Список структур*. Куст представлен структурой `Bush`, содержащей вершину с именем `Name` и список смежных ей вершин `Neigs`. Переменная `graph` является списком структур `Bush`. В примере (рис. 12.3) приведён способ построения пути из вершины `a`.

```
struct Bush   // структура куста
{
    public char Name;         // вершина
    public List<char> Neigs;  // соседи
}
static void Main()
{
    List<Bush> graph = new List<Bush>
    {
        new Bush { Name = 'a', Neigs = new List<char>() { 'a', 'b' }
},   // куст из вершины 'a'
        new Bush { Name = 'b', Neigs = new List<char>() { 'a', 'c' }
},   // куст из вершины 'b'
        new Bush { Name = 'c', Neigs = new List<char>() { 'b' } }
    };
    Bush x = graph.Find(z => z.Name == 'a'); // вершина 'a'
    char x1 = x.Neigs[1];   // x1 - вторая вершина, смежная к 'a'
    Bush y = graph.Find(z => z.Name == x1); // вершины, смежные x1
```

```
    char y1 = y.Neigs[1]; // вторая вершина, смежная к x1
    Console.Write($"a -> {x1} -> {y1}"); // вывод пути
    Console.ReadKey();
}
```

Результат

```
a -> b -> c
```

2. *Список объектов*. Куст представлен классом `Bush` и подобен структуре из предыдущего примера, имеет поле с именем `Name` и список смежных вершин `Neigs`.

```
class Bush   // класс куста
{
    public char Name;         // вершина
    public List<char> Neigs;  // соседи
}
```

Граф `graph` представляет собой список объектов типа `Bush`.

```
List<Bush> graph = new List<Bush>();
```

3. *Массив кустов*. Для этого способа вершины графа необходимо представлять числами, так как они служат индексами массива. Либо одно из полей вершины должно является её целочисленным идентификатором.

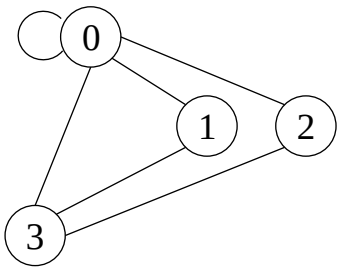



Рис. 12.4. Псевдограф

Граф (рис. 12.4) можно представить массивом

| индекс/номер вершины: | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| список смежных вершин: | {0, 1, 2, 3} | {0, 3} | {0, 3} | {0, 1, 2} |

Ниже приведена программная реализация графа (рис. 12.4) и найден путь длиной 2 из вершины 0.

```
static void Main()
{
    List<int>[] m = new List<int>[4];
    m[0] = new List<int>() { 0, 1, 2, 3 };
    m[1] = new List<int>() { 0, 3 };
    m[2] = new List<int>() { 0, 3 };
    m[3] = new List<int>() { 0, 1, 2 };
    int x = m[0][1]; // номер вершины x, смежной к нулевой
    int y = m[x][1]; // номер вершины y, смежной к x
    Console.WriteLine($"0 -> {x} -> {y}"); // вывод пути
    Console.ReadKey();
}
```

Результат

```
0 -> 1 -> 3
```

4. *Связный список кустов*. Класс `Bush<T>` описывает кусты графа, содержащие вершину `Name` и связный список ссылок на её соседей `Neighbors`.

Достоинства:

а) быстрая модификация связных списков вершин и дуг;

б) подходит для представления псевдографов и мультиграфов.

```
public class Bush<T>
{
    public T Name;  // идентификатор вершины Value
    // список ссылок на смежные вершины (если пустой, то null):
    public LinkedList<Bush<T>> Neighbors = new
LinkedList<Bush<T>>();
    public Bush(T name) { Name = name; }
}
```

Класс `Graph<T>` хранит граф в виде связного списка вершин `LLN` (аббр. фразы Linked List Node). Этот класс позволяет добавлять новую уникальную вершину с помощью метода `AddNode(name)`, а также добавлять одну или несколько дуг с помощью методов `AddEdge(FromName,ToName)` и `AddEdges(FromName,ToNames)` соответственно. Добавлен метод вывода на экран `PrintNeighbors(name)` всех вершин, смежных заданной вершине `name`.

```
class Graph<T>
{
    public LinkedList<Bush<T>> LLN;
    public Graph() { LLN = new LinkedList<Bush<T>>(); }
    public void AddNode(T name)  // добавление уникальной вершины
    {
        foreach (Bush<T> n in LLN) if (n.Name.Equals(name)) return;
        var bush = new Bush<T>(name);
        LLN.AddLast(bush);
    }
    public void AddEdge(T FromName, T ToName)
    {
        foreach (Bush<T> a in LLN)
        {
            if (a.Name.Equals(FromName))
            {
                foreach (Bush<T> b in LLN)
                {
                    if (b.Name.Equals(ToName))
a.Neighbors.AddLast(b);
                }
                return;
```

```
            }
        }
    }
    public void AddEdges(T FromName, T[] ToNames)
    {
        foreach (Bush<T> a in LLN)
        {
            if (a.Name.Equals(FromName))
            {
                foreach (T ToName in ToNames)
                {
                    foreach (Bush<T> b in LLN)
                    {
                        if (b.Name.Equals(ToName))
a.Neighbors.AddLast(b);
                    }
                }
                return;
            }
        }
    }
    public void PrintNeighbors(T name)
    {
        foreach (Bush<T> a in LLN)
        {
            if (a.Name.Equals(name))
            {
                foreach (Bush<T> m in a.Neighbors)
                    Console.Write(m.Name + ", ");
                return;
            }
        }
    }
}
```

Построим орграф с тремя вершинами (рис. 12.5). Вначале создадим все три вершины, потом соединим их дугами. В конце примера выведем на экран имя вершины, добавленной первой, потом все вершины, смежные вершине `a`, а также путь длиной 2 из вершины, которая была добавлена первой, по первым ссылкам списков смежных вершин. В этом примере с целью упрощения не выполняется проверка пустоты списка смежных вершин.

```
static void Main()
{
    var G = new Graph<char>();
    G.AddNode('a');
    G.AddNode('b');
    G.AddNode('c');
```

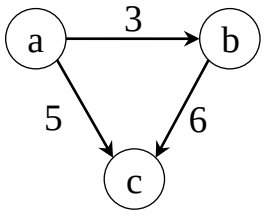



Рис. 12.5. Орграф

```
    G.AddEdges('a', new char[] { 'b', 'c' });
    G.AddEdge('b', 'c');
    Console.WriteLine(G.LLN.First.Value.Name);
    G.PrintNeighbors('a');
    Console.WriteLine();
    Bush<char> A = G.LLN.First.Value;
    Console.Write(A.Name + " -> ");
    Bush<char> B = A.Neighbors.First.Value;
    Console.Write(B.Name + " -> ");
    Bush<char> C = B.Neighbors.First.Value;
    Console.WriteLine(C.Name);
    Console.ReadKey();
}
```

Результат

```
a
b, c,
a -> b -> c
```

### 12.4 Список рёбер

*Список рёбер* – список, в котором каждому ребру графа соответствует структура/объект из двух вершин, инцидентных этому ребру. Размер занимаемой памяти $O(|E|)$.

Достоинство – наиболее компактный способ представления графов, поэтому часто применяется для внешнего хранения или обмена данными.

Недостатки:

а) трудность поиска смежных вершин;

б) для представления неориентированных графов нужно либо удваивать список рёбер, либо делать функцию симметричного замыкания, что увеличит время поиска смежных вершин.

В следующем коде представлена структура `Edge<T>`, описывающая дугу, `T` – тип данных вершины. Эту структуру можно описать и в виде класса с аналогичными полями.

```
struct Edge<T>
{
    public T NodeA;
    public T NodeB;
}
```

Рассмотрим описания орграфа (рис. 12.6) в виде списка дуг и найдём путь длиной 2 из вершины `a`.

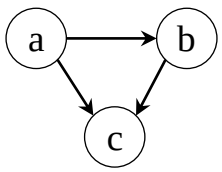



Рис. 12.6. Орграф

```
static void Main()
{
    List<Edge<char>> graph = new List<Edge<char>>();
    graph.Add(new Edge<char> { NodeA = 'a', NodeB = 'b' }); // a-b
```

```
    graph.Add(new Edge<char> { NodeA = 'a', NodeB = 'c' }); // a-c
    graph.Add(new Edge<char> { NodeA = 'b', NodeB = 'c' }); // b-c
    Edge<char> e1 = graph.Find(x => x.NodeA == 'a'); // поиск дуги
e1, с началом в вершине 'a'
    char x1 = e1.NodeB; // имя вершины, инцидентной концу дуги e1
    Edge<char> e2 = graph.Find(x => x.NodeA == x1); // поиск дуги
e2, с началом в вершине x1
    char y1 = e2.NodeB; // имя вершины, инцидентной концу дуги e2
    Console.WriteLine($"a -> {x1} -> {y1}");
    Console.ReadKey();
}
```

Результат

```
a -> b -> c
```

Пример функции симметричного замыкания `SymClose` при использовании неориентированного графа (рис. 12.7), описанного списком дуг, а также способа поиска пути длиной 2 из вершины `c`. Симметричное замыкание восстанавливает ребро, имея одну дугу.

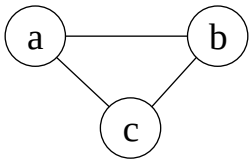



Рис. 12.7. Неориентированный граф

```
private static char SymClose(List<Edge<char>> graph, char node)
{
    // получение ребра, инцидентного вершине node
    Edge<char> e = graph.Find(x => (x.NodeA == node | x.NodeB ==
node));
    // поиск вершины, инцидентной вершине node
    if (e.NodeA == node) return e.NodeB;
    else return e.NodeA;
}
private static void Main(string[] args)
{
    List<Edge<char>> graph = new List<Edge<char>>();
    graph.Add(new Edge<char> { NodeA = 'a', NodeB = 'b' }); // a-b
    graph.Add(new Edge<char> { NodeA = 'a', NodeB = 'c' }); // a-c
    graph.Add(new Edge<char> { NodeA = 'b', NodeB = 'c' }); // b-c
    char x = SymClose(graph, 'c'); // вершина x, смежная 'c'
    char y = SymClose(graph, x); // вершина y, смежная x
    Console.WriteLine($"c -> {x} -> {y}");
    Console.ReadKey();
}
```

Результат

```
c -> a -> b
```

## 12.5 Граф на координатной сетке с подвижными вершинами

Кусты графа на координатной сетке (рис. 12.8) описываются классом с двумя дополнительными полями для координат `X` и `Y` вершины

```
public class Bush<T>
{
    public T Name;
    public int X;  // Координата x
    public int Y;  // Координата y
    // Список смежных вершин (если список пустой, то null)
    public LinkedList<Bush<T>> Neighbors = new
LinkedList<Bush<T>>();
    public Bush(T name, int x, int y) { Name = name; X = x; Y = y; }
}
```

Класс `Graph<T>` хранит граф в виде списка вершин `LLN`. Этот класс позволяет добавлять новую уникальную вершину с помощью метода `AddNode`, а также добавлять одну или несколько дуг с помощью методов `AddEdge` и `AddEdges` соответственно. Метод `MoveNode` перемещает вершину в новые координаты. Кроме этого имеется метод `PrintNeighbors` вывода на экран всех вершин, смежных заданной вершине `name`.

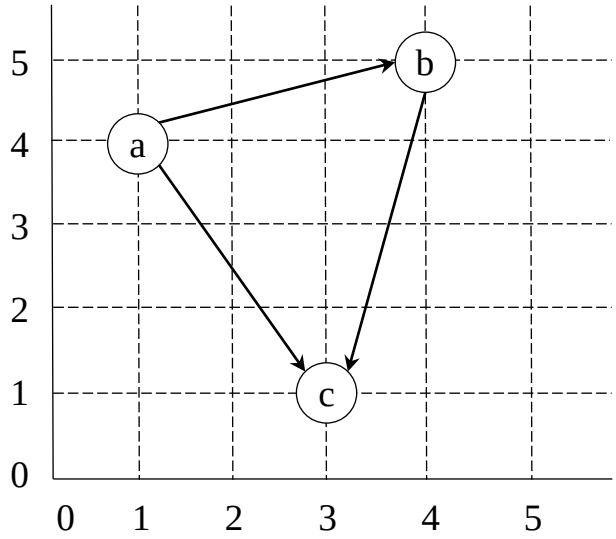



Рис. 12.8. Орграф на координатной сетке

```
class Graph<T>
{
    public LinkedList<Bush<T>> LLN;
    public Graph() { LLN = new LinkedList<Bush<T>>(); }
    public void AddNode(T name, int x, int y)
    {
        foreach (Bush<T> n in LLN) if (n.Name.Equals(name)) return;
        var bush = new Bush<T>(name, x, y);
        LLN.AddLast(bush);
    }
    public void AddEdge(T FromName, T ToName)
    {
        foreach (Bush<T> a in LLN)
```

```
        {
            if (a.Name.Equals(FromName))
            {
                foreach (Bush<T> b in LLN)
                {
                    if (b.Name.Equals(ToName))
a.Neighbors.AddLast(b);
                }
                return;
            }
        }
    }
    public void AddEdges(T FromName, T[] ToNames)
    {
        foreach (Bush<T> a in LLN)
        {
            if (a.Name.Equals(FromName))
            {
                foreach (T ToName in ToNames)
                {
                    foreach (Bush<T> b in LLN)
                    {
                        if (b.Name.Equals(ToName))
a.Neighbors.AddLast(b);
                    }
                }
                return;
            }
        }
    }
    public void MoveNode(T name, int x, int y)
    {
        foreach (Bush<T> a in LLN)
        {
            if (a.Name.Equals(name)) { a.X = x; a.Y = y; }
        }
    }
    public void PrintNeighbors(T name)
    {
        foreach (Bush<T> a in LLN)
        {
            if (a.Name.Equals(name))
            {
                foreach (Bush<T> m in a.Neighbors)
                    Console.Write(m.Name + ", ");
                return;
            }
        }
```

```
    }
}
```

Пример построения графа (рис. 12.8).

```
static void Main()
{
    var G = new Graph<char>();
    G.AddNode('a', 1, 4);
    G.AddNode('b', 4, 5);
    G.AddNode('c', 3, 1);
    G.AddEdges('a', new char[] { 'b', 'c' });
    G.AddEdge('b', 'c');
    Console.WriteLine("Первая вершина графа: " +
G.LLN.First.Value.Name);
    Console.Write("Её соседи: ");
    G.PrintNeighbors(G.LLN.First.Value.Name);
    Console.ReadKey();
}
```

Результат

```
Первая вершина графа: a
Её соседи: b, c,
```

## 12.6 Другие разновидности графов

### Ad-hoc сети

*Ad-hoc сеть* – граф с подвижными вершинами и ограничением на расстояние между ними. При превышении этого расстояния связь между вершинами теряется и граф разделяется на компоненты. Используются для моделирования сети беспроводной мобильной связи.

### Ориентированный ациклический граф

*Ориентированный ациклический граф* (англ. directed acyclic graph, DAG) – граф, имеющий четыре составляющих:

а) узлы, хранящие данные;

б) дуги;

в) одна «великая» вершина без входящих дуг;

г) листья.

Ориентированные ациклические графы широко используются в компиляторах для представления модифицированного дерева синтаксического разбора выражений, содержащих дубликаты подвыражений; в искусственном интеллекте для представления нейронных

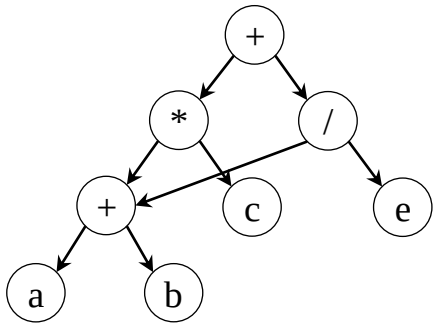



Рис. 12.9. DAG выражения

сетей без обратной связи; в машинном обучении для представления байесовской сети доверия.

Ориентированный ациклический граф (рис. 12.9) синтаксического разбора выражения $(a + b) \cdot c + (a + b)/e$ содержит общую вершину для двух подвыражений $(a + b)$. Именно это отличает его от дерева синтаксического разбора.

**Задача 1.** Разработать метод вывода на экран информации обо всех вершинах ориентированного графа, заданного на координатной сетке: имя вершины, её координаты, имена вершин, в которые можно перейти из данной вершины и расстояние до каждой из них.

**Задача 2.** Разработать генератор связного неориентированного графа по заданному числу вершин и рёбер. Результат вывести в виде матрицы смежности.

**Контрольные вопросы**

1. Что представляет собой матрица смежности?
2. Каковы достоинства и недостатки матрицы смежности?
3. В чём заключается отличие матриц смежности неориентированного графа, орграфа и взвешенного графа?
4. Какие значения может содержать ячейка матрицы инцидентности?
5. Каковы достоинства и недостатки матрицы инцидентности?
6. Какие бывают разновидности коллекций списков смежности?
7. Что является элементом списка структур при представлении графа?
8. Возможно ли заменить список структур списком объектов при представлении графов?
9. Как хранятся вершины графа при представлении его массивом списков смежных вершин?
10. Каковы достоинства и недостатки представления графа в виде связного списка объектов со ссылками между ними?
11. Как можно хранить вершины при представлении графа списком рёбер?
12. Что должен содержать класс, описывающий вершины графа на координатной сетке?
13. Какие графы могут представлять ad-hoc сеть?

# 13 Алгоритмы на графах

Существует гигантское количество алгоритмов на графах. Воспользуемся классификацией основных алгоритмов обхода и поиска:

1) неинформированные алгоритмы;
   а) обход вершин графа;
      – сначала вглубь;
      – сначала вширь;
   б) поиск на графе по целевой функции;
      – поиск с ограничением глубины;
      – поиск по критерию стоимости (веса);
      – двунаправленный поиск;
   в) поиск компонент связности;
   г) поиск минимальных остовных деревьев;
      – жадный алгоритм Ярника (Прима, Дейкстры);
      – жадный алгоритм Крускала;
   д) поиск кратчайших путей;
      – жадный алгоритм Дейкстры;
      – алгоритм Беллмана-Форда;
      – волновой алгоритм;
2) информированные алгоритмы;
   а) алгоритм A*;
   б) альфа-бета-отсечение;
   в) метод ветвей и границ;
   г) поиск по первому наилучшему совпадению.

Неинформированный поиск называют методом грубой силы (англ. brute-force search, BFS), а информированный – эвристическим поиском (англ. informed search, heuristic search).

## 13.1 Методы обхода графов

Есть два метода обхода вершин графов – *вглубь* и *вширь*. Реализация этих методов зависит от способа представления графов. Наилучший для обхода способ представления графов – это коллекция кустов. Кусты позволяют быстро получать все вершины, смежные заданной.

Для обхода графа надо задать стартовую вершину, из которой этот обход начнётся. При обходе графа используется *список посещённых вершин*, который ещё называют *списком табу*. Повторно посещать вершины из этого списка нельзя, чтобы не попасть в бесконечный цикл. Список табу можно заменить *метками посещения* в самих вершинах. Роль такой метки играет дополнительное булево поле в классе `Bush<T>`.

**Обход сначала вглубь**

Обход сначала вглубь проходит по всем доступным вершинам, начиная со стартовой, до тех пор, пока не зайдёт либо в тупиковую вершину, либо в вершину, все соседи которой находятся в списке табу. В этом случае алгоритм должен вернуться назад на одну вершину и попытаться найти альтернативный путь.

При выборе вершины для посещения необходимо проверить её отсутствие в списке табу. Если вершина отсутствует, то её можно посетить, предварительно записав в список табу.

Рассмотрим пример обхода ориентированного графа, вершины которого описываются классом `Bush<T>`. В экземпляре класса хранится имя вершины `Name` и связный список смежных ей вершин `Neighbors`. Конструктор класса создаёт вершину графа по заданному имени и с пустым списком смежных вершин.

```
public class Bush<T>
{
    public T Name;  // имя вершины
    public LinkedList<Bush<T>> Neighbors = new
LinkedList<Bush<T>>();  // соседи
    public Bush(T name) { Name = name; }
}
```

Класс `Graph<T>` описывает граф в виде связного списка вершин `LLN`. Конструктор `Graph()` создаёт пустой список `LLN`. Для конструирования графа и его обхода сначала вглубь предназначены методы:

`AddNode(name)` – добавление в граф вершины с именем `name`;

`AddEdge(FromName, ToName)` – добавление дуги, исходящей из вершины `FromName` и ведущей к вершине `ToName`;

`AddEdges(FromName, ToNames)` – добавление нескольких дуг, исходящих из вершины `FromName` и ведущих к вершинам `ToNames`;

`PrintNeighbors(name)` – вывод всех вершин, смежных заданной вершине `name`;

`DepthFirstTraversal(name)` – обёртка метода обхода графа сначала вглубь, начиная с вершины `name`;

`DepthFirstTraversal(node, tabu)` – обход графа сначала вглубь из вершины `node` с учётом списка табу `tabu`, выводит на экран все пути, получаемые при обходе и откатах назад.

**Обход сначала вширь**

Обход сначала вширь подобен распространению круговой волны от брошенного в воду камня. Волна начинается от стартовой вершины и идёт по смежным вершинам графа во все стороны. Метод обхода сначала вширь использует список внешних/фронтальных вершин волны

externals и список внутренних/посещённых вершин internals. Объединение списков internals и externals является списком табу.

Волна за один виток цикла расходится во все вершины, смежные вершинам волны, и не принадлежащие списку табу. Выполняется это следующим образом. Вначале фронт волны externals присоединяется к списку табу internals. Создаётся пустой список нового фронта newexternals. Для каждой вершины волны externals находятся смежные ей вершины, не принадлежащие списку табу. Такие вершины помещаются в новый фронт волны newexternals. Когда новый фронт заполнен всеми возможными вершинами, выполняется рекурсивный вызов BreadthFirstTraversal(internals, newexternals). Описанная рекурсия останавливается при пустом списке externals.

Назначение методов обхода вширь

BreadthFirstTraversal(name) – обёртка метода обхода графа сначала вширь из вершины с именем name.

BreadthFirstTraversal(internals, externals) – обход графа сначала вширь из вершин фронта externals с учётом списка внутренних вершин internals.

Класс Graph<T> содержит реализацию методов обхода сначала вглубь и сначала вширь.

```
class Graph<T>
{
    public LinkedList<Bush<T>> LLN;
    public Graph() { LLN = new LinkedList<Bush<T>>(); }
    public void AddNode(T name)  // добавление уникальной вершины
    {
        foreach (Bush<T> n in LLN) if (n.Name.Equals(name)) return;
        var bush = new Bush<T>(name);
        LLN.AddLast(bush);
    }
    public void AddEdge(T FromName, T ToName)
    {
        foreach (Bush<T> a in LLN)
        {
            if (a.Name.Equals(FromName))
            {
                foreach (Bush<T> b in LLN)
                {
                    if (b.Name.Equals(ToName)) {
a.Neighbors.AddLast(b); break; }
                }
                return;
            }
        }
```

```
    }
    public void AddEdges(T FromName, T[] ToNames)
    {
        foreach (Bush<T> a in LLN)
        {
            if (a.Name.Equals(FromName))
            {
                foreach (T ToName in ToNames)
                {
                    foreach (Bush<T> b in LLN)
                    {
                        if (b.Name.Equals(ToName)) {
a.Neighbors.AddLast(b); break; }
                    }
                }
                return;
            }
        }
    }
    public void PrintNeighbors(T name)
    {
        foreach (Bush<T> a in LLN)
        {
            if (a.Name.Equals(name))
            {
                foreach (Bush<T> m in a.Neighbors)
                    Console.Write(m.Name + ", ");
                return;
            }
        }
    }
    public void DepthFirstTraversal(T name)  // обход в глубину
    {
        var tabu = new List<T> { name };  // список табу + name
        foreach (Bush<T> bush in LLN)  // поиск стартового куста
        {
            if (bush.Name.Equals(name))
            {
                DepthFirstTraversal(bush, tabu);
                break;
            }
        }
        Console.WriteLine(string.Join(" ", tabu));
    }
    public void DepthFirstTraversal(Bush<T> bush, List<T> tabu)
    {
        foreach (Bush<T> Neighbor in bush.Neighbors) // все соседи
        {
```

```
            if (!tabu.Contains(Neighbor.Name))  // не в списке табу
            {
                tabu.Add(Neighbor.Name);  // добавить в табу
                DepthFirstTraversal(Neighbor, tabu);
            }
        }
    }
    public void BreadthFirstTraversal(T name)  // обход в ширину
    {
        var internals = new List<Bush<T>>();  // список внут. вершин
        var externals = new List<Bush<T>>();  // список фронта
        foreach (Bush<T> bush in LLN)  // поиск стартового куста
        {
            if (bush.Name.Equals(name))
            {
                Console.WriteLine(bush.Name);
                externals.Add(bush);
                BreadthFirstTraversal(internals, externals);
                break;
            }
        }
    }
    public void BreadthFirstTraversal(List<Bush<T>> internals,
List<Bush<T>> externals)
    {
        if (externals.Count == 0) return;
        internals.AddRange(externals);  // список табу
        var newexternals = new List<Bush<T>>();  // новый фронт
        foreach (Bush<T> bush in externals)
        {
            foreach (Bush<T> Neighbor in bush.Neighbors)
            {
                if (!internals.Contains(Neighbor) &&
!newexternals.Contains(Neighbor))  // не табу и не новый фронт
                {
                    Console.Write(Neighbor.Name + " ");
                    newexternals.Add(Neighbor);  // в новый фронт
                }
            }
        }
        Console.WriteLine();
        BreadthFirstTraversal(internals, newexternals);
    }
}
```

Воспользуемся разработанным классом `Graph<T>` для построения орграфа (рис. 13.1) и его обхода двумя способами.

```
static void Main()
```

```
{
    var G = new Graph<char>();
    G.AddNode('a');
    G.AddNode('b');
    G.AddNode('c');
    G.AddNode('d');
    G.AddNode('e');
    G.AddNode('f');
    G.AddEdges('a', new char[] { 'b', 'c', 'e' });
    G.AddEdge('b', 'c');
    G.AddEdges('c', new char[] { 'b', 'd' });
    G.AddEdge('d', 'b');
    G.AddEdge('e', 'f');
    Console.WriteLine("Обход вглубь из вершины 'a':");
    G.DepthFirstTraversal('a');
    Console.WriteLine();
    Console.WriteLine("Обход вширь из вершины 'a':");
    G.BreadthFirstTraversal('a');
    Console.ReadKey();
}
```

Результат

```
Обход вглубь из вершины 'a':
a b c d e f

Обход вширь из вершины 'a':
a
b c e
d f
```

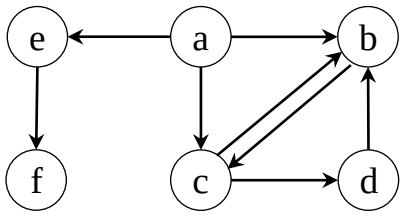



Рис. 13.1. Орграф

Обход графа (рис. 13.1) сначала вглубь вернул список `tabu`, в котором находятся все вершины. Обход сначала вширь вывел каждый фронт волны с новой строки. Всего получилось три фронта.

Обход графа сначала вглубь легко модифицировать так, чтобы получать не список всех вершин, а список всех путей из стартовой вершины. Для этого достаточно после возврата из рекурсии удалять помещённую в список `tabu` вершину.

```
public void DepthFirstTraversal(T name)  // обход в глубину
{
    var tabu = new List<T> { name };  // список табу + name
    foreach (Bush<T> bush in LLN)  // поиск стартового куста
    {
        if (bush.Name.Equals(name))
        {
            DepthFirstTraversal(bush, tabu);
            break;
        }
    }
```

```
}
public void DepthFirstTraversal(Bush<T> bush, List<T> tabu)
{
    foreach (Bush<T> Neighbor in bush.Neighbors) // все соседи
    {
        if (!tabu.Contains(Neighbor.Name))  // не в списке табу
        {
            tabu.Add(Neighbor.Name);  // в список табу
            Console.WriteLine(string.Join(" ", tabu));
            DepthFirstTraversal(Neighbor, tabu);
            tabu.Remove(Neighbor.Name);  // удаление после возврата
        }
    }
}
```

Такой модифицированный обход вглубь развернул граф в дерево обхода (рис. 13.2). Вывод на экран показывает пошаговое удлинение ветвей с последующими откатами назад.

```
Обход путей вглубь из вершины 'a':
a b
a b c
a b c d
a c
a c b
a c d
a c d b
a e
a e f
```

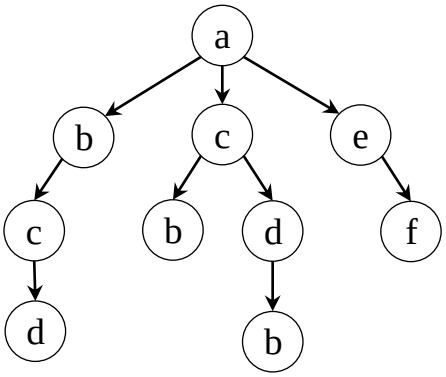



Рис. 13.2. Дерево обхода путей орграфа вглубь

### 13.2 Компоненты связности и остовные деревья

*Компонента связности* – подмножество графа, любая вершина которого связана путём только с вершинами этого подмножества.

**Алгоритм поиска компонент связности:**

1. Создать пустой список списков $S$. В нём мы получим список всех компонент связности графа.

2. Получить коллекцию всех вершин графа $L$.

3. Для одной вершины из $L$ построить обход графа, занося все посещённые вершины в список $S_i$. Поместить $S_i$ в $S$. Удалить вершины $S_i$ из $L$.

4. Если коллекция $L$ пуста, то конец алгоритма. Иначе перейти к п. 3.

Когда список $S$ содержит один элемент, то граф связный, иначе он состоит из стольких компонент связности, сколько элементов в списке $S$.

*Шарнир/разделяющая вершина/точка сочленения* – вершина, например, `a` (рис. 13.1), удаление которой увеличивает число компонент связно-

сти. Граф, не имеющий шарниров, называется *блоком*. *Мост* – ребро, например, `a-e`, удаление которого увеличивает количество компонент связности в графе. *Вершинный сепаратор* – множество вершин, например, {`c`, `b`}, удаление которых приводит к увеличению числа компонент связности.

**Остовные деревья**

*Остовное дерево* неориентированного графа – это дерево с корнем в произвольной вершине, покрывающее все вершины графа. Остовное дерево можно построить, применив алгоритм обхода графа сначала вширь, и запоминая все ветви дерева обхода.

**Минимальные остовные деревья**

Если неориентированный граф является невзвешенным или веса всех рёбер одинаковы, то любое остовное дерево является *минимальным*. Если веса рёбер различаются, то *минимальное остовное дерево* – это дерево с минимальной суммой весов своих рёбер. Граф может иметь несколько различных минимальных остовных деревьев.

## 13.3 Алгоритмы поиска минимальных остовных деревьев

**Алгоритм Войцеха Ярника**

Алгоритм был открыт чешским математиком Ярником в 1930 году. Однако впоследствии он был повторно описан Р. Примом в 1957 г. и Э. Дейкстрой в 1959 г. В настоящее время в западной научной литературе алгоритм Ярника упоминается под именем Прима или Дейкстры.

1. Выбрать произвольную вершину. Считать её деревом.

2. Присоединить к дереву ту вершину, которая пока не входит в дерево и связана с ним ребром наименьшего веса (рис. 13.2).

3. Повторять п. 2 до тех пор, пока к дереву не будут присоединены все вершины.

Этот алгоритм является жадным, т.к. на каждом этапе выбирает наилучшее локальное решение. Выгодным способом представления графа для данного алгоритма являются списки смежности. При этом наилучшую оценку вычислительной сложности алгоритм Ярника имеет в случае использования фибоначчиевой пирамиды для хранения вершин графа, не входящих в остовное дерево.

Оценка вычислительной сложности алгоритма Ярника

| Представления графа | Невыбранные вершины | Оценка сложности |
|---|---|---|
| Список кустов (смежности) | Массив | $O(V^2)$ |
| Список кустов (смежности) | Бинарная пирамида | $O((V+E)\log V)$ |
| Список кустов (смежности) | Фибоначчиева пирамида | $O(E+V\log V)$ |

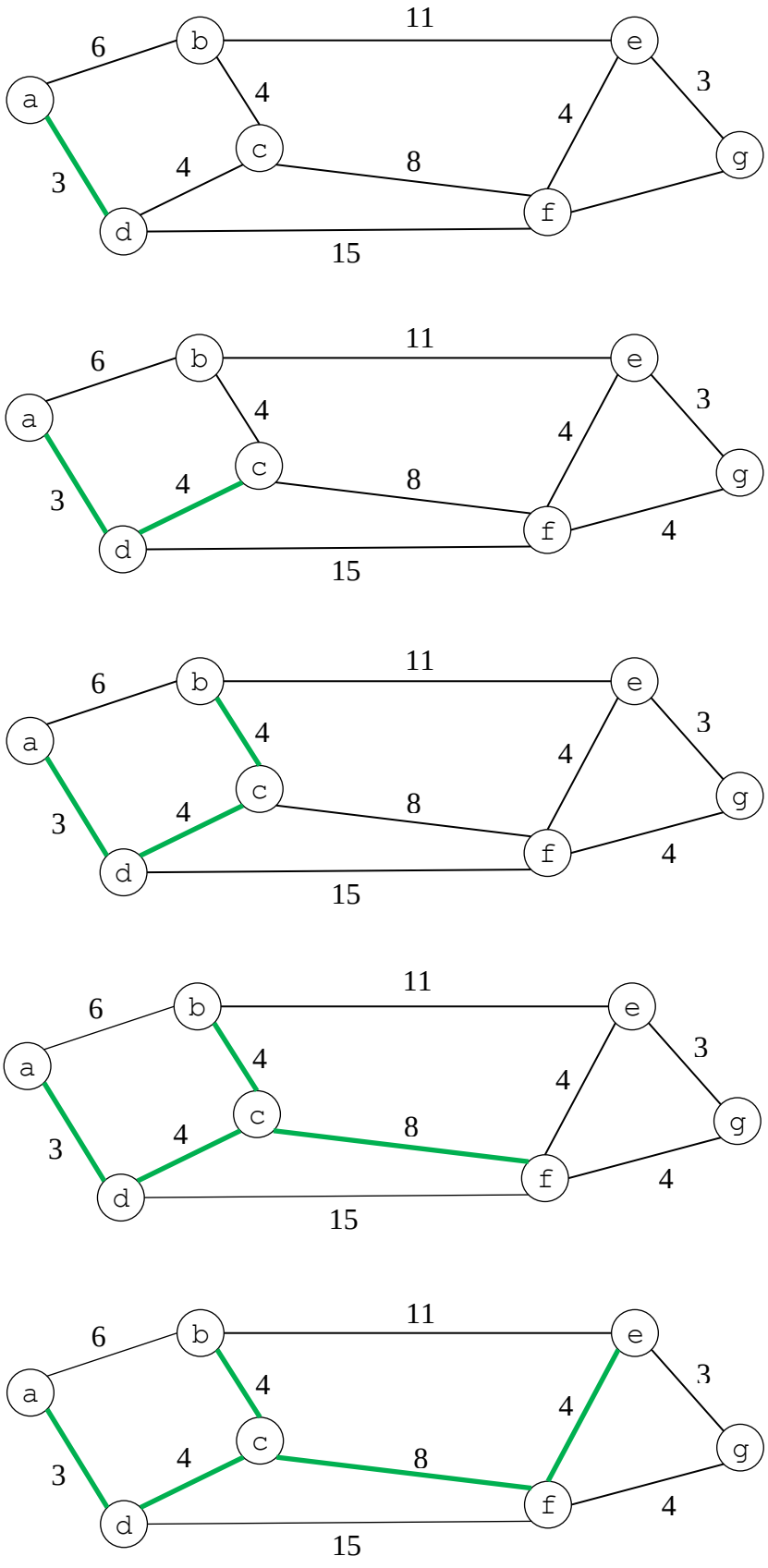



Рис. 13.2. Шаги алгоритма Ярника (начало)

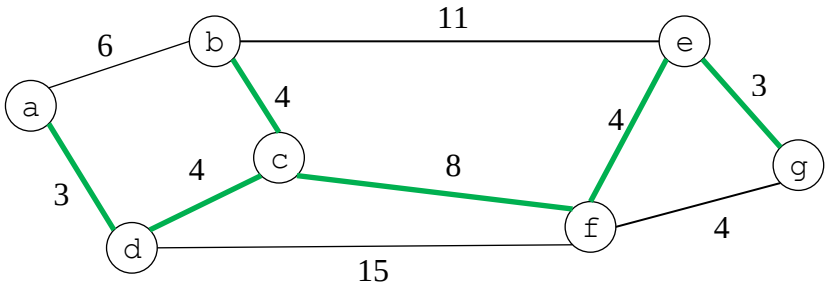



Рис. 13.2. Шаги алгоритма Ярника (окончание)

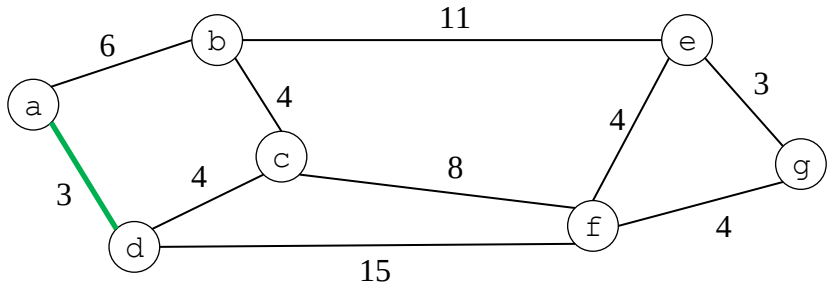
**Алгоритм Краскала**

Алгоритм был открыт американским математиком Джозефом Краскалом в 1956 году. Однако ещё в 1926 году чешский математик Отакар Борувка открыл практически этот же алгоритм. В отечественной литературе фамилию Краскала можно встретить в прочтении «Крускал».

1. Установить множество рёбер остовного дерева пустым.

2. Выбрать на графе ребро минимального веса, не делающее цикл в дереве, и добавить его к множеству рёбер (рис. 13.3).

3. Если таких рёбер больше нет, то завершить алгоритм, иначе перейти к п. 2.



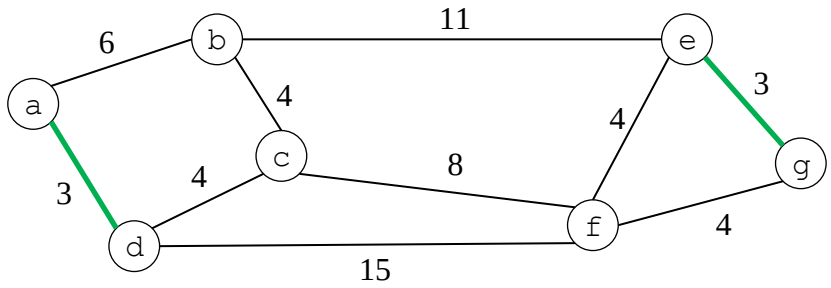




Рис. 13.3. Шаги алгоритма Краскала (начало)

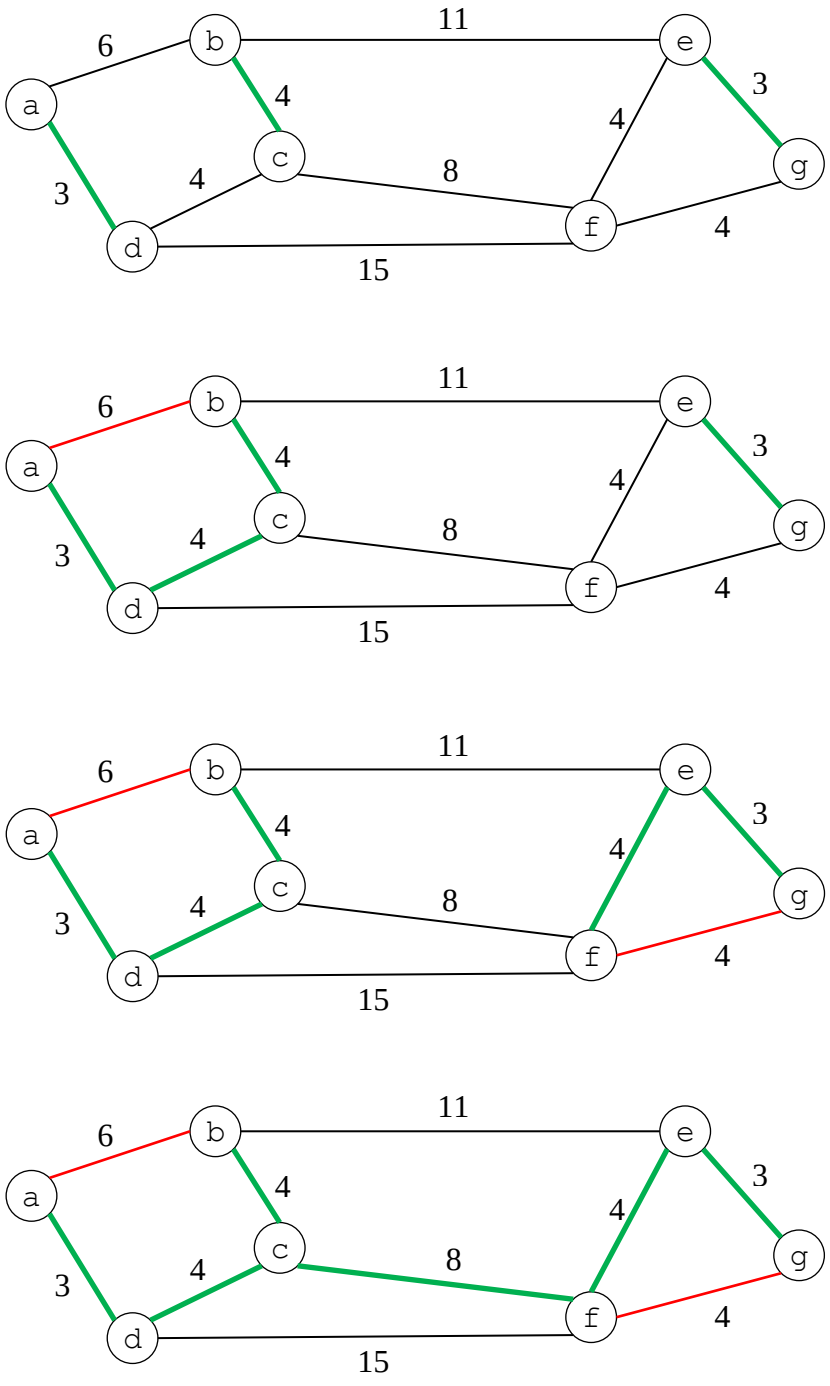



Рис. 13.3. Шаги алгоритма Краскала (окончание)

Этот алгоритм также является жадным, т.к. на каждом этапе выбирает наилучшее локальное решение. Оценка вычислительной сложности алгоритма Краскала $O(E \log E)$.

**Задача Штейнера**

На задачу нахождения минимального остовного дерева похожа задача о *дереве Штейнера*. В ней задано несколько точек на плоскости и тре-

буется проложить между ними граф путей, так, чтобы минимизировать сумму длин путей, связывающих точки. Главное отличие от задачи о минимальном остовном дереве заключается в том, что разрешается добавлять дополнительные точки ветвления рёбер с целью минимизации суммы длин рёбер. Задача о дереве Штейнера является NP-полной.

**Прикладное значение задачи Штейнера**

Проектирование путей между пунктами на местности в случае, когда стоимость прокладки пути высока, а доставка грузов по пути практически бесплатна. Например, прокладка оптоволокна между городами для передачи информации. Оптоволокно и его прокладка имеют большую стоимость. Стоимость передачи информации определяется только затратой электроэнергии. Строительство дорог или газопроводов также имеет высокую стоимость, особенно в пересечённой местности. При этом стоимость доставки грузов по этим дорогам значительно ниже и определяется стоимостью топлива, которое пропорционально длине пути.

**Задача 1.** Разработать программу обхода графа сначала вглубь с выводом полных ветвей дерева обхода, без промежуточных ветвей.

**Задача 2.** Разработать программу определения того, что неориентированный граф является связным и не имеет циклов, т.е. является деревом. Граф представить одним из трёх способов:

а) матрицей смежности;

б) списком списков смежных вершин;

в) списком рёбер.

**Контрольные вопросы**

1. В чём отличие неинформированных и информированных алгоритмов на графах?
2. В чём заключается суть алгоритма обхода графа сначала вглубь?
3. Каков программный способ возврата назад в алгоритме обхода сначала вглубь?
4. Для чего выполняется возврат назад в алгоритме обхода сначала вглубь?
5. В чём заключается суть алгоритма обхода графа сначала вширь?
6. Что такое список табу?
7. Что такое компонента связности графа?
8. Что такое минимальное остовное дерево?
9. Каким образом ищется минимальное остовное дерево?

# 14 Поиск путей на графах

Виды задач о кратчайшем пути:
а) поиск кратчайшего пути из исходной вершины во все остальные;
б) поиск кратчайшего пути из вершины $v$ в вершину $u$;
в) поиск кратчайшего пути между всеми парами вершин графа.

## 14.1 Алгоритмы поиска кратчайших путей на графах

Для оценки сложности алгоритмов воспользуемся обозначениями: $V$ (англ. Vertex) – множество вершин графа, $E$ (англ. Edge) – множество рёбер (дуг). Оценка сложности алгоритмов поиска путей зависит от способа представления графа. Наиболее популярные алгоритмы перечислены ниже.

1. Алгоритм Дейкстры находит кратчайшие пути из исходной вершины графа до всех остальных. При использовании матрицы смежности оценка сложности $O(|V|^2)$. Алгоритм является *жадным*.

2. Алгоритм Беллмана – Форда находит кратчайшие пути из исходной вершины графа до всех остальных. Оценка сложности $O(|V| \cdot |E|)$. Алгоритм относится к методам динамического программирования.

3. Алгоритм A* (читается «а звезда») находит маршрут наименьшей стоимости от начальной вершины до конечной.

4. Алгоритм Флойда – Уоршелла находит кратчайшие пути между всеми парами вершин взвешенного графа. Оценка сложности $\theta(|V|^3)$.

5. Алгоритм Джонсона находит кратчайшие пути между всеми парами вершин взвешенного ориентированного графа с оценкой сложности $\theta(|V||E|\log|V|)$.

6. Алгоритм Ли (волновой алгоритм) основан на методе поиска в ширину, имеет оценку сложности $O(|E|)$. Находит путь между двумя несовпадающими вершинами графа, содержащий минимальное количество промежуточных вершин. Основное применение – трассировка электрических соединений в микросхемах и на печатных платах. Также используется для поиска кратчайшего расстояния на карте в компьютерных играх.

**Применение алгоритмов поиска кратчайших путей**

1. В картографических сервисах Yandex, Google и OpenStreetMap используют алгоритмы нахождения кратчайшего пути на графе для нахождения путей между физическими объектами.

2. Для поиска оптимальной последовательности решений на графе работы недетерминированной абстрактной вычислительной машины, например, если вершинами являются состояния кубика Рубика, а дуга

представляет собой одно действие над кубиком, тогда алгоритм может быть применён для сборки кубика с минимальным количеством ходов.

### 14.2 Алгоритм Дейкстры

Автор алгоритма – голландский учёный Эдсгер Дейкстра (1959 г). Алгоритм находит кратчайшие пути от заданной вершины графа до всех остальных и работает с графами, имеющими рёбра только положительного веса. Однако, любой граф с рёбрами отрицательного веса можно преобразовать в граф с рёбрами положительного веса путём сложения весов всех рёбер с модулем наименьшего веса.

Суть алгоритма заключается в расстановке расстояний от исходной вершины $a$ до всех смежных, образующих фронт волны, расходящейся от исходной вершины. На каждом шаге алгоритма фронт волны расширяется путём замены самой близкой к исходной вершине $a$ фронтальной вершины смежными ей вершинами, которые алгоритм ещё не посетил. Заменённая вершина заносится в список посещённых. Все посещённые вершины снабжаются расстоянием от начальной вершины до неё. Алгоритм останавливается, когда все вершины посещены, либо когда посещена искомая вершина. После завершения алгоритма каждая вершина будет содержать минимальное расстояние от исходной вершины до неё и собственно путь в виде списка вершин.

Постановка задачи. Дан взвешенный ориентированный граф $G(V,E)$ без дуг отрицательного веса. Необходимо найти кратчайшие пути от некоторой вершины $a$ графа $G$ до всех остальных вершин этого графа.

Пусть граф $G(V,E)$ задан списком кустов, хранящих имя вершины $v$ и список смежных вершин $u$ с весами рёбер $w_{v,u}$. Обозначим:

- $w_{v,u}$ – вес ребра, соединяющего вершины $v$ и $u$;
- $a$ – начальная вершина;
- $U$ – список посещённых вершин (список табу);
- $F$ – сортированный список вершин фронта (в идеале – пирамида);
- $d_u$ – длина кратчайшего пути из $a$ в $u$;
- $p_u$ – кратчайший путь из $a$ в $u$ в виде списка вершин;
- $v$, $u$ – вершины графа;
- $max$ – максимальное число выбранного типа данных для веса.

**Описание алгоритма**

1. Снабдим каждую вершину $u$ меткой, содержащей расстояние $d_u$ и путь к ней $p_u$. Исходная вершина имеет метку 0. Остальные вершины имеют метку бесконечности, которая при программировании может быть представлена максимальным числом $max$.

2. Помещаем исходную вершину в список вершин фронта $F$.

3. Вырезаем первую вершину $v$ из списка $F$, помещаем её в $U$.

4. Устанавливаем меткам вершин $u \notin U$, смежных $v$, минимальные значения расстояний $d_u$ и путь $p_u$

если $d_u > d_v + w_{v,u}$
то $d_u = d_v + w_{v,u}$;
$p_u = (p_v, u)$;

5. Помещаем в список $F$ вершины $u$, таким образом, чтобы список $F$ сохранил упорядоченность по увеличению значений меток расстояния.

6. Если список $F$ пуст, то завершаем алгоритм. Иначе переходим к п. 3.

Рассмотрим алгоритм Дейкстры в действии.

**Шаг 1.** Выполняем п. 1, 2 алгоритма. Расставляем метки $max$ расстояний на исходном графе (рис. 14.1). Помещаем исходную вершину $a$ в список вершин фронта $F$. Вершины фронта волны – синие.

$U = ();$
$F = (a);$
$d_a = 0; p_a = (a);$
$d_b = max; p_b = ();$
$d_c = max; p_c = ();$
$d_d = max; p_d = ();$
$d_e = max; p_e = ();$
$d_f = max; p_f = ().$

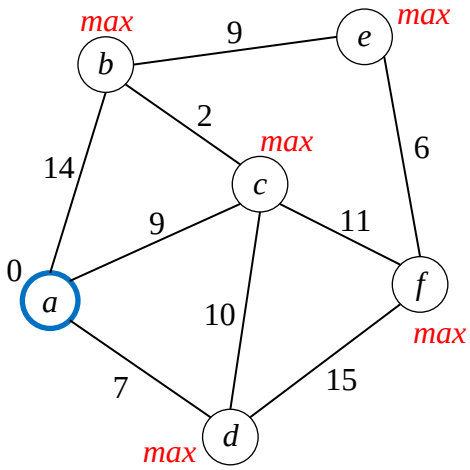



Рис. 14.1. Исходный размеченный граф

**Шаг 2.** Выполняем п. 3, посещённые вершины обозначены оранжевым цветом

$U = (a);$
$F = ().$

Выполняем п. 4

$d_b = 14;\ p_b = (a, b);$
$d_c = 9;\ p_c = (a, c);$
$d_d = 7;\ \mathrm{p_d} = (a, d).$

Выполняем п. 5 (рис. 14.2)

$F = (d, c, b).$

Так как список $F$ не пуст, переходим к п.3.

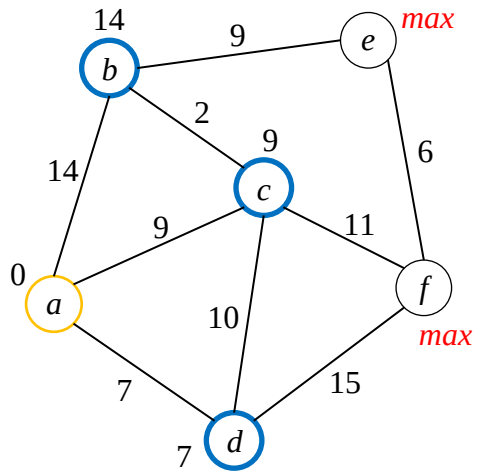



Рис. 14.2

**Шаг 3.** Выполняем п. 3 для вершины $d$

$F = (c, b);$
$U = (a, d).$

Выполняем п. 4. Расширяем фронт волны из вершины $d$. Метка вершины $c$ не изменяется, так как 9<17. Метка вершины $f$ изменяется

$d_f = 7 + 15 = 22; p_f = (a, d, f).$

Выполняем п. 5 (рис. 14.3)

$F = (c, b, f).$

Так как список $F$ не пуст, переходим к п. 3.

**Шаг 4.** Выполняем п. 3 для вершины $c$

$F = (b, f);$
$U = (a, d, \mathrm{c}).$

Выполняем п. 4 (рис. 14.4). Фронт волны из вершины $c$ мы расширить не можем, так как все смежные вершины уже есть во фронте. Метки вершин $b$ и $f$ изменяются, так как путь к ним через вершину $c$ короче

$d_b = 9 + 2 = 11; p_b = (a, c, b);$
$d_f = 9 + 11 = 20; p_f = (a, c, f).$

Пункт 5 выполнять не надо, ибо нет новых вершин, смежных $c$, которые бы расширили фронт волны. Так как список $F$ не пуст, переходим к п. 3.

**Шаг 5.** Выполняем п. 3 алгоритма для вершины $b$

$F = (f);$
$U = (a, d, \mathrm{c}, b).$

Выполняем п. 4 (рис. 14.5). Расширяем фронт волны из вершины $b$. Метка вершины $e$ изменяется

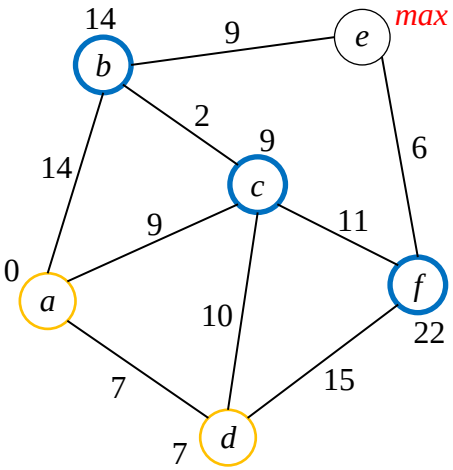



Рис. 14.3

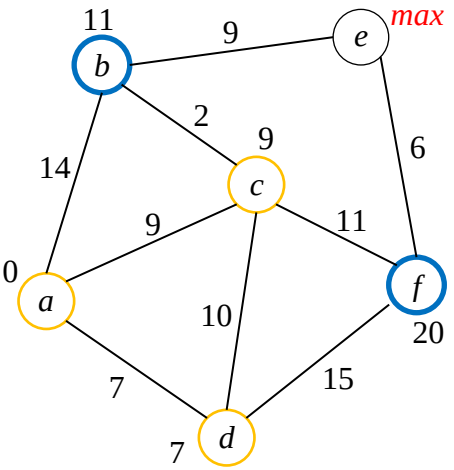



Рис. 14.4

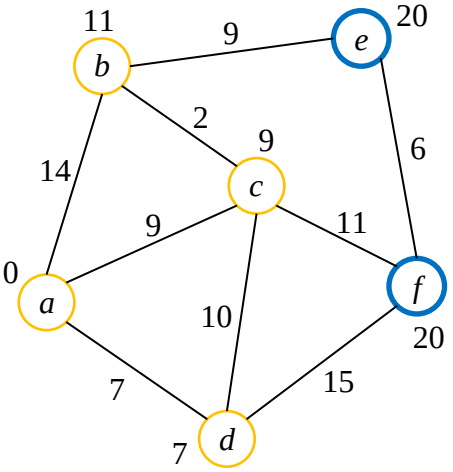



Рис. 14.5

$d_e = 11 + 9 = 20; p_e = (a, c, b, e)$.

Выполняем п. 5

$F = (e, f)$.

Так как список $F$ не пуст, переходим к п. 3.

**Шаг 6.** Выполняем п. 3 для вершины $e$

$F = (f)$;
$U = (a, d, \mathrm{c}, b, e)$.

Выполняем п. 4 (рис. 14.6). Фронт волны из вершины $e$ мы расширить не можем – нет новых вершин. Метка вершины $f$ не изменяется, так как путь в $f$ из вершины $c$ короче, чем из вершины $e$.

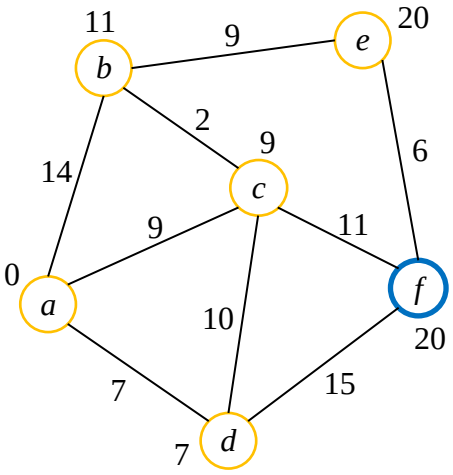



Рис. 14.6

Выполняем п. 5

$F = (f)$.

Так как список $F$ не пуст, переходим к п.3.

**Шаг 7.** Выполняем п. 3 для вершины $f$

$F = ()$;
$U = (a, d, \mathrm{c}, b, e, f)$.

Выполняем п. 4 (рис. 14.7). Фронт волны из вершины $f$ мы расширить не можем – нет новых вершин. Пункт 5 выполнять не надо, ибо нет новых вершин, смежных $f$, которые бы расширили фронт волны. Список $F$ пуст, завершаем алгоритм.

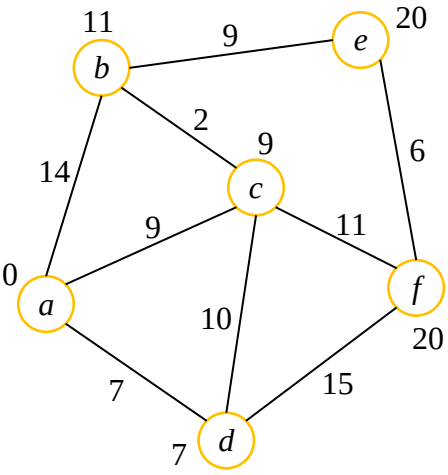



Рис. 14.7

В результате мы получили минимальные расстояния и пути из вершины $a$ до всех остальных вершин графа

$d_a = 0; p_a = (a)$;
$d_b = 11; p_b = (a, c, b)$;
$d_c = 9;\ p_c = (a, c)$;
$d_d = 7;\ p_d = (a, d)$;
$d_e = 20; p_e = (a, c, b, e)$;
$d_f = 20; p_f = (a, c, f)$.

### 14.3 Алгоритм Беллмана – Форда

Алгоритм Беллмана – Форда решает ту же задачу, что и алгоритм Дейкстры, но может работать на графах с отрицательными весами и позволяет обнаруживать отрицательные циклы. *Отрицательный цикл* – это цикл с суммарным отрицательным весом. Алгоритм проще в реализации по сравнению с алгоритмом Дейкстры, но имеет бóльшую вычислительную сложность $O(|V| \cdot |E|)$, так как количество дуг в графе обычно больше чем вершин: $|E| > |V|$.

Постановка задачи. Дан взвешенный ориентированный граф $G(V,E)$, возможно с дугами отрицательного веса. Найти кратчайшие пути от некоторой вершины графа $G$ до всех остальных вершин этого графа.

Пусть граф $G(V,E)$ задан матрицей смежности M, хранящей веса дуг $w_{v,u}$. Обозначим:

- $w_{v,u}$ – вес ребра, соединяющего вершины $v$ и $u$;
- $s$ – начальная вершина;
- $d$ – массив расстояний от вершины $s$ до любой другой вершины;
- $v$, $u$ – вершины графа;
- $max$ – максимальное число выбранного типа данных для веса.

**Описание алгоритма**

1. Создадим одномерный массив $d$, присвоим всем элементам максимальное число $max$, а элементу с индексом $s$ присвоим ноль, так как путь из вершины $s$ до неё же самой равен нулю.

2. Выполним $|V| - 1$ раз действия: для каждой дуги $(u, v)$, имеющей вычисленное расстояние до исходящей вершины $d[u] < max$, обновить расстояние $d[v]$ до вершины $v$ в случае, если $d[v] > d[u] + w(u, v)$

```
for i=1 to |V|-1
    foreach (u,v)∈E и d[u] < max
        if d[v] > d[u] + w(u, v)
        then d[v] = d[u] + w(u, v)  // релаксация дуги
```

3. Проверим наличие отрицательных циклов. Для этого попытаемся провести релаксацию хотя бы одной дуги. Если релаксация получится, то граф содержит отрицательный цикл

```
foreach (u,v)∈E
    do if d[v] > d[u] + w(u, v)  // релаксация возможна?
        then return false  // есть отрицательный цикл
```

4. Если отрицательный цикл не найден, то возвращаем массив $d$, в котором находятся длины кратчайших путей от вершины $a$ до остальных вершин

```
return d
```

Алгоритм выполняет $|V|-1$ шагов. Этого достаточно, так как самый длинный путь содержит не более чем $|V|-1$ дуг.

Рассмотрим алгоритм Беллмана – Форда в действии.

**Шаг 1.** Начальная вершина $s$ исходного графа (рис. 14.8) сверху помечена расстоянием 0. До остальных вершин расстояние *max*. Алгоритм сделает ещё 4 шага, так как граф содержит 5 вершин.

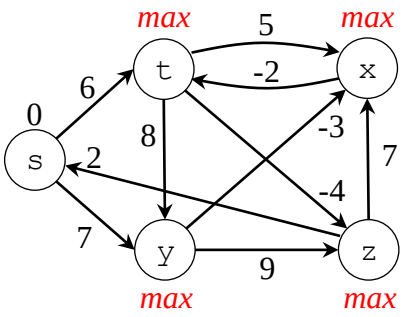



Рис. 14.8

**Шаг 2.** На графе (рис. 14.9) найдено две дуги $(s,t)$ и $(s,y)$, у которых исходящая вершина имеет расстояние, меньшее *max*. Поэтому вершины $t$ и $y$ помечаются новыми расстояниями 6 и 7 соответственно.

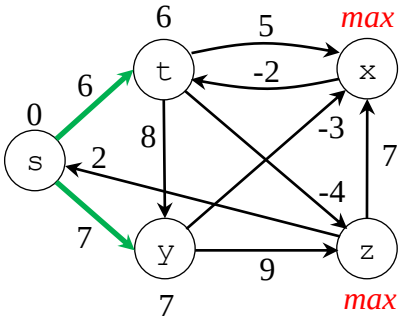



Рис. 14.9

**Шаг 3.** Кроме дуг $(s,t)$ и $(s,y)$ на графе найдено ещё пять дуг (рис. 14.10), исходящих из вершин $t$ и $y$, расстояние до которых нам уже известно. Поэтому вершины $x$ и $z$ помечаются новыми расстояниями 4 и 2 соответственно.

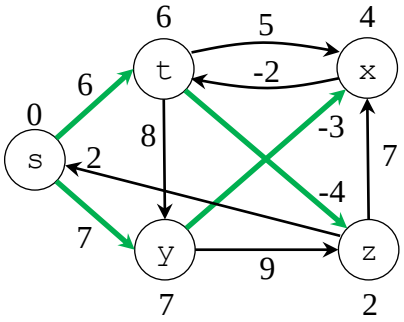



Рис. 14.10

**Шаг 4.** Обновлено расстояние до вершины $t$ (рис. 14.11) за счёт отрицательного веса дуги $(x,t)$.

**Шаг 5.** Обновлено расстояние до вершины $z$ (рис. 14.12) за счёт отрицательного веса дуги $(t,z)$.

Если алгоритм Беллмана – Форда не сошёлся после $|V|-1$ итераций (т.е. дальнейшие релаксации возможны), то в графе есть циклы с отрицательным весом.

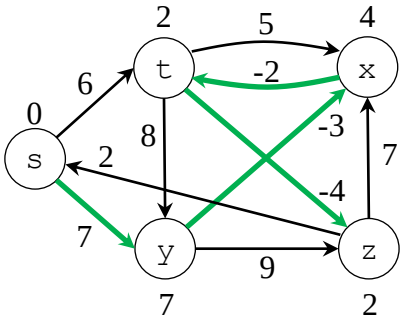



Рис. 14.11

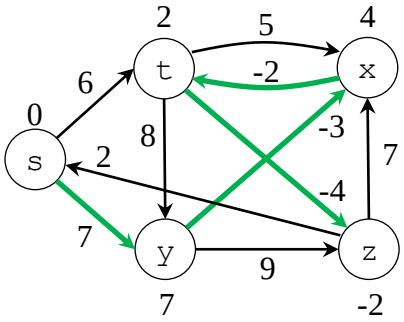



Рис. 14.12

Описанный алгоритм вычисляет только длины путей, но не сами пути. Для сохранения минимальных путей предлагается в дополнительный одномерный массив записывать предшественника той вершины, расстояние до которой было сохранено. На рисунках 14.9 – 14.12 выделены дуги, по которым можно определить предшественника любой вершины. Обратите внимание: при переходе рис. 14.10 → рис. 14.11 предшественник вершины *t* был заменён новым предшественником: вершиной *x*. До этой замены предшественником была вершина *s*.

Алгоритма Беллмана – Форда допускает параллельную релаксацию каждой дуги на всех вершинах графа.

Веса всех рёбер можно увеличить на абсолютную величину самого маленького веса. Тогда граф не будет иметь отрицательных рёбер и к нему можно применить алгоритм Дейкстры. При выводе результата надо не забыть выполнить обратное действие: уменьшить веса всех рёбер на величину, которую прибавили для устранения отрицательных рёбер.

### 14.4 Алгоритм Ли

Волновой алгоритм Ли (англ. Lee) находит кратчайший путь между двумя вершинами на планарном графе. Кратчайший путь содержит минимальное количество рёбер (вершин) между двумя заданными вершинами. Веса рёбер, если они есть, при поиске не учитываются. Прообразом такого графа может являться координатная сетка или дискретное поле, состоящее из одинаковых геометрических примитивов без перекрытий и пропусков: квадратов, прямоугольников, треугольников или шестиугольников (рис. 14.13).

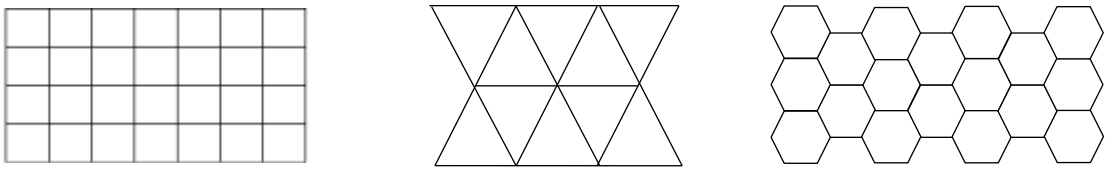


Рис. 14.13

Недоступные/непроходимые ячейки дискретного поля должны иметь соответствующие метки. Вершины графа, соответствующие непроходимым ячейкам, могут быть удалены.

В основу алгоритма Ли положен обход вершин графа в ширину. Каждая вершина графа наделяется атрибутом расстояния от стартовой вершины до неё. Алгоритм также использует коллекцию *F* вершин волны. Алгоритм содержит следующие шаги.

**Шаг 1.** Во всех ячейках дискретного поля инициализируются атрибуты расстояния значением *max*, описанном выше. Атрибут начальной (синей) ячейки сбрасывается в ноль (рис. 14.14). Указывается конечная ячейка (жёлтая). Помечаются непроходимые ячейки (штриховкой). Стартовая ячейка помещается во фронт волны *F*. В случае планарного графа непроходимые ячейки не представляются вершинами (рис. 14.15).

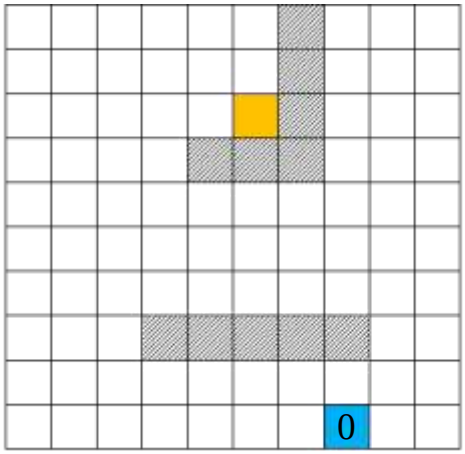



Рис. 14.14

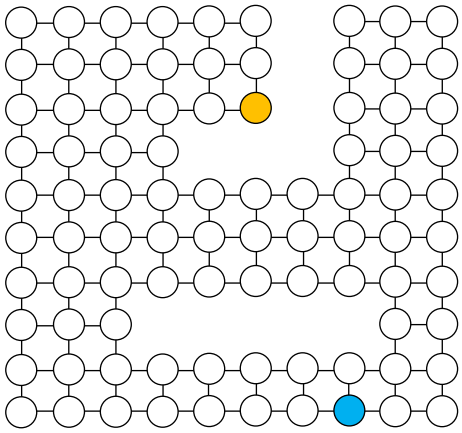


Рис. 14.15

**Шаг 2.** Создаётся новая пустая коллекция *NF* для новой волны.

**Шаг 3.** На этом шаге выполняется распространение очередной волны (рис. 14.16). Если коллекция *F* не пуста, то из неё удаляется ячейка. Её атрибут расстояния содержит некоторое значение $p$. Атрибуты всех смежных ей ячеек, содержащих значение *max*, обновляются новым значением $p + 1$. Эти ячейки помещаются в коллекцию *NF*. В случае нахождения конечной вершины алгоритм останавливается. Иначе – повторяется шаг 3.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 15 | 14 | 15 | | | | 11 | 10 | 11 |
| 15 | 14 | 13 | 14 | 15 | | | 10 | 9 | 10 |
| 14 | 13 | 12 | 13 | 14 | 15 | | 9 | 8 | 9 |
| 13 | 12 | 11 | 12 | | | | 8 | 7 | 8 |
| 12 | 11 | 10 | 11 | 10 | 9 | 8 | 7 | 6 | 7 |
| 11 | 10 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 6 |
| 10 | 9 | 8 | 9 | 8 | 7 | 6 | 5 | 4 | 5 |
| 9 | 8 | 7 | | | | | | 3 | 4 |
| 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 1 | 2 |

Рис. 14.16

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | 9 | 9 | 9 |
| | 9 | 9 | 9 | 9 | 9 | | 8 | 8 | 8 |
| | 9 | 8 | 8 | 8 | 9 | | 7 | 7 | 7 |
| 9 | 9 | 8 | 7 | | | | 6 | 6 | 6 |
| 8 | 8 | 8 | 7 | 6 | 5 | 5 | 5 | 5 | 5 |
| 7 | 7 | 7 | 7 | 6 | 5 | 4 | 4 | 4 | 4 |
| 7 | 6 | 6 | 6 | 6 | 5 | 4 | 3 | 3 | 3 |
| 7 | 6 | 5 | | | | | | 2 | 2 |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 1 | 1 | 2 |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 1 | 2 |

Рис. 14.17

**Шаг 4.** Если коллекция *F* пуста, то непустая коллекция *NF* переименовывается в *F* и выполняется шаг 2. Если коллекция *NF* пуста, то считается, что конечная вершина недоступна или отсутствует.

Восстановление кратчайшего пути происходит в обратном направлении: от конечной к стартовой ячейке на каждом шаге выбирается та ячейка, которая имеет атрибут расстояния на единицу меньше текущей ячейки. На рисунке 14.16 путь помечен зелёными ячейками. Очевидно, что кратчайших путей между парой заданных ячеек может существовать несколько. Выбор окончательного пути диктуется другими соображениями, находящимися вне этого алгоритма. Например, при трассировке печатных плат – минимумом линейной длины проложенного проводника.

Окрестностью вершины могут являться не четыре, а все восемь смежных вершин. В этом случае допускается путь по диагонали (рис. 14.17).

**Задача 1.** Разработать программу поиска всех путей на орграфе из заданной вершины с использованием алгоритма Дейкстры. Способ представления графа выбрать самостоятельно.

**Контрольные вопросы**

1. Какого вида бывают задачи о кратчайшем пути?
2. Перечислите популярные алгоритмы поиска кратчайших путей на графах.
3. Какие приложения алгоритмов поиска кратчайших путей на графах Вы знаете?
4. В чём суть алгоритма Дейкстры и какова оценка его вычислительной сложности?
5. В чём суть алгоритма Беллмана – Форда и какова оценка его вычислительной сложности?
6. Какой из этих алгоритмов обрабатывает графы с отрицательными весами дуг?
7. Как следует изменить алгоритм Беллмана – Форда, чтобы он вычислял не только длины путей, но и сами пути?
8. Каков признак наличия циклов с отрицательным весом в орграфе?
9. Какой из рассмотренных алгоритмов допускает параллельное выполнение?
10. В чём измеряется кратчайший путь в алгоритме Ли?

# 15 Информированные методы поиска путей

*Информированный поиск* (эвристический поиск) – стратегия поиска решений в пространстве состояний, в которой используется дополнительная информация (эвристика) о направлении к искомой вершине графа или о минимальном расстоянии до неё. Информированные методы поиска обеспечивают более эффективный поиск по сравнению с неинформированными методами.

*Эвристическая функция* на каждом шаге поиска оценивает нижнюю границу расстояния до конечной вершины, чтобы принять решение о том, из какой вершины следует продолжить движение.

Эвристическая функция должна быть:

а) *допустимой*, т.е. являться нижней оценкой расстояния до цели или даже быть меньше минимальной оценки;

б) *монотонно невозрастающей* вдоль любого исследуемого пути. При движении к цели каждая следующая вершина должна быть не дальше к целевой вершине, чем предыдущая.

Если $h_1(n)$, $h_2(n)$ – допустимые эвристические функции и для любой вершины $n$ верно неравенство $h_1(n) \geq h_2(n)$, то $h_1$ является более информированной эвристикой или доминирует над $h_2$.

## 15.1 Алгоритм A*

Авторы алгоритма – Питер Харт, Нильс Нильсон и Бертрам Рафаэль (1968 г.). *Алгоритм A** находит кратчайший путь от начальной вершины $a$ до конечной вершины $z$. Он является модификацией алгоритма Дейкстры. Модификация заключается в том, что из фронта волны для раскрытия выбирается не самая близкая к началу вершина, а вершина $v$, лежащая на самом коротком пути к цели. Самый короткий путь оценивается суммой уже пройдённого расстояния $g(v)$ от $a$ до $v$ и оценкой расстояния от $v$ до целевой вершины $z$ с помощью эвристической функции $h(v)$. На каждом шаге алгоритма фронт волны расширяется путём замены каждой фронтальной вершины $v$, имеющей минимальное значение $f(v) = g(v) + h(v)$, смежными ей вершинами, которые алгоритм ещё не посетил. Заменённая вершина приобретает статус *посещённой*. Алгоритм останавливается, когда целевая вершина $z$ приобретает статус посещённой. При этом она будет содержать расстояние $g(z)$ от начальной вершины до неё.

Постановка задачи. Дан взвешенный орграф $G(V,E)$, содержащий вершины без дуг отрицательного веса. Найти кратчайший путь от вершины $a$ до вершины $z$ этого графа.

Пусть граф $G(V,E)$ задан списком кустов, хранящих имя вершины $v$ и список смежных вершин $u$ с весами рёбер $w_{v,u}$. Обозначим:

– $w_{v,u}$ – вес ребра, соединяющего вершины $v$ и $u$;

– $a$ – начальная вершина;

– $z$ – конечная вершина;

– $g(v)$ – функция пройдённого расстояния от вершины $a$ до $v$;

– $h(v)$ – допустимая монотонная функция оценки расстояния от вершины $v$ до конечной вершины $z$. Пусть эта функция возвращает расстояние по прямой линии от вершины $v$ до конечной вершины $z$;

– $U$ – список посещённых вершин (список табу);

– $F$ – очередь вершин фронта волны с приоритетом значений $f(v)$;

– $p_v$ – кратчайший путь из $a$ в $v$ в виде списка вершин;

– $max$ – максимальное число выбранного типа данных для веса.

**Описание алгоритма**

1. Снабдим каждую вершину $v$ меткой, содержащей значение $g(v)$ и путь к ней $p_v$. Исходная вершина $a$ имеет метку 0 и путь $p_v = ()$. Остальные вершины имеют метку $max$.

2. Помещаем исходную вершину $a$ в очередь вершин фронта $F$ с приоритетом $f(v) = 0 + h(a)$.

3. Вырезаем из очереди $F$ первую вершину $v$, т.е. вершину, имеющую минимальное значение $f(v)$, и помещаем её в список табу $U$. Если в список $U$ была помещена конечная вершина $z$, то останавливаем алгоритм, иначе переходим к п. 4.

4. Устанавливаем меткам вершин $u \in F$, смежных $v$, минимальное значение $g(u)$

если $g(u) > g(v) + w_{v,u}$, то $g(u) = g(v) + w_{v,u}$

и помещаем их в очередь $F$ с приоритетом $f(u) = g(u) + h(u)$. Устанавливаем меткам непосещённых вершин $u \notin U$, смежных $v$, значение $g(u) = g(v) + w_{v,u}$ и помещаем их в очередь фронта волны $F$ с приоритетом $f(u) = g(u) + h(u)$. Переходим к п. 3.

Рассмотрим алгоритм A* в действии.

**Шаг 1.** Выполняем п.п. 1 и 2: расставим метки расстояний $g(u)$ на исходном графе (рис. 15.1). Кроме этого сразу расставим значения эвристики $h(v)$ для всех вершин. Хотя это действие следует выполнять последовательно по мере прохождения

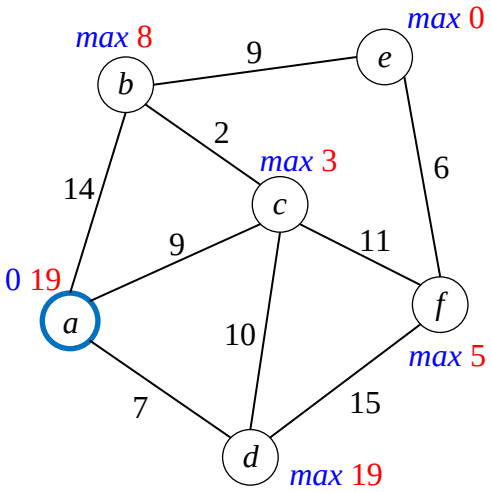



Рис. 15.1. Исходный размеченный граф

по графу. Сейчас рядом с каждой вершиной имеем два числа: левое-синее – пройдённое расстояние $g(u)$, правое-красное – оценка расстояния до конечной вершины $h(v)$.

Помещаем исходную вершину $a$ в очередь вершин фронта $F$ со значением $f(a) = 19$. Будем обрамлять очередь вершин $F$ круглыми скобками с указанием значения $f(v)$. Очередь $F$ упорядочим по увеличению значения $f(v)$. Вершины фронта волны выделим жирными синими обводами.

$F = (a - 19);$
$U = ();$
$g(a) = 0; p_a = (a);$
$g(b) = max; p_b = ();$
$g(c) = max; p_c = ();$
$g(d) = max; p_d = ();$
$g(e) = max; p_e = ();$
$g(f) = max; p_f = ().$

**Шаг 2.** Выполняем п. 3: вырезаем из очереди $F$ единственную вершину $a$ и помещаем её в список $U$. Посещённые вершины обозначены оранжевым цветом (рис. 15.2).

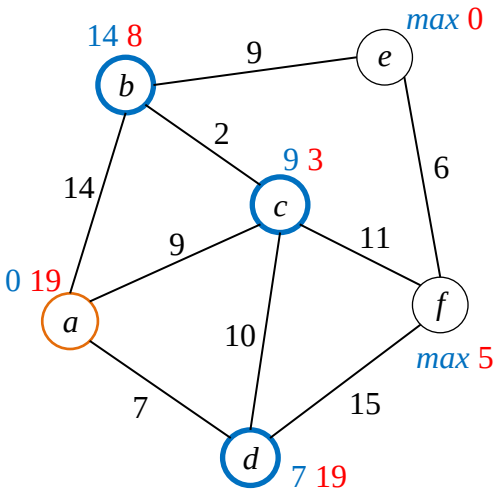


Рис. 15.2

$F = ();$
$U = (a).$

Выполняем п. 4

$g(b) = 14;\ p_b = (a, b);$
$F = (b - 22);$
$g(c) = 9;\ p_c = (a, c);$
$F = (c - 12, b - 22);$
$g(d) = 7;\ p_d = (a, d);$
$F = (c - 12, b - 22, d - 26).$

Переходим к п. 3.

**Шаг 3.** Выполняем п. 3: вырезаем из очереди $F$ первую вершину, т.е. вершину $c$, имеющую минимальное значение $f(c) = 12$, и помещаем её в список табу $U$ (рис. 15.3)

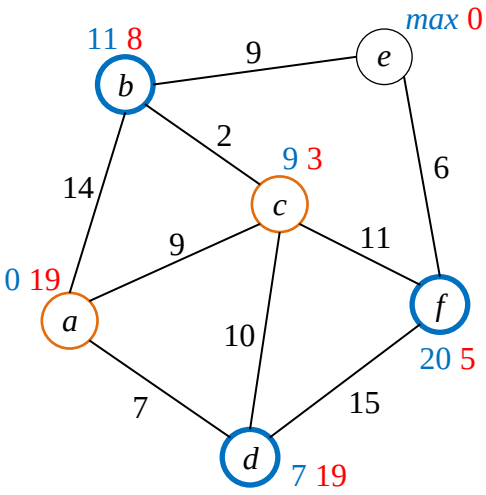


Рис. 15.3

$F = (b - 22, d - 26);$
$U = (a, c).$

Выполняем п. 4

$g(b) = 11;\ p_b = (a, c, b);$
$F = (b - 19, d - 26).$

Метку вершины $d$ не обновляем, так как текущее значение $g(d) = 7$ меньше, чем путь к этой вершине через вершину $c$. Добавляем в очередь $F$ новую вершину фронта $f$ (рис. 15.3)

$g(f) = 20;\ p_f = (a, c, f);$
$F = (b - 19, f - 25, d - 26).$

Переходим к п. 3.

**Шаг 4.** Выполняем п. 3: вырезаем из очереди $F$ первую вершину, т.е. вершину $b$, имеющую минимальное значение $f(b) = 19$, и помещаем её в список $U$ (рис. 15.4)

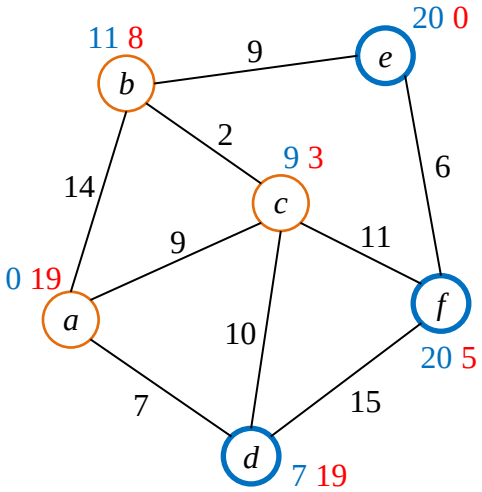


Рис. 15.4

$F = (f - 25, d - 26);$
$U = (a, c, b).$

Выполняем п. 4 (рис. 15.4)

$g(e) = 20;\ p_e = (a, c, b, e);$
$F = (e - 20, f - 25, d - 26).$

Мы добрались до искомой вершины $e$ по пути $p_e = (a, c, b, e)$. Скорее всего этот путь самый короткий. Но во фронте могут быть вершины $v$, у которых $g(v) < f(e)$. Поэтому нам надо найти и раскрыть такие вершины, чтобы выяснить какой путь короче, либо лучше всего продолжать алгоритм до тех пор, пока конечная вершина $e$ будет помещена в список $U$. Изберём второй путь и вновь перейдём к п. 3.

**Шаг 5.** Выполняем п. 3: вырезаем из очереди $F$ первую вершину, т.е. вершину $e$, имеющую минимальное значение $f(e) = 20$, и помещаем её в список табу $U$ (рис. 15.5)

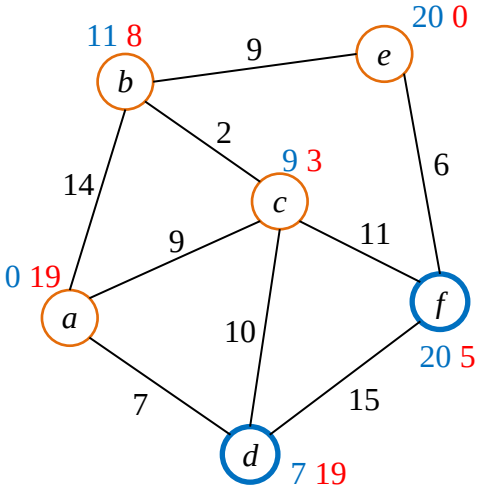


Рис. 15.5

$F = (f - 25, d - 26);$
$U = (a, c, b, e).$

Завершаем алгоритм, так как искомая вершина $e$ была помещена в список табу $U$. В результате имеем самый короткий путь $p_e = (a, c, b, e)$, и его длину $g(e) = 20$.

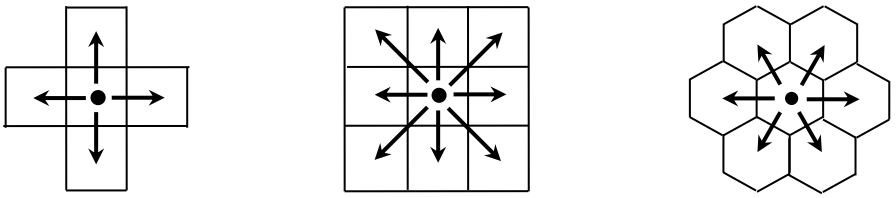
**Примечания к алгоритму A***

1. Пути $p_u$ в каждую вершину $u$ можно не сохранять. Достаточно для каждой вершины $u$ сохранять предыдущую вершину $v$. Тогда всегда можно восстановить путь, двигаясь от конечной вершины к начальной.

2. Если очередь фронта $F$ имеет несколько перспективных вершин с одинаковой оценкой $f(v)$, то выбор порядка раскрытия таких вершин будет влиять на стратегию поиска. Если такие вершины раскрывать от самой первой, помещённой в очередь FIFO, то поиск будет широконаправленным, обтекая возможные препятствия со всех сторон, и только потом устремляться к цели. Если такие вершины раскрывать от самой последней, помещённой в стек FILO, то поиск будет узконаправленным, т.е. вначале будет устремляться к цели и если она не будет достигнута, то совершать обход возможных препятствий.

## 15.2 Эвристики оценки расстояния на графах, привязанных к координатной сетке

Соседние ячейки координатной сетки принято классифицировать в смысле окрестности фон Неймана и в смысле окрестности Мура. В *окрестности фон Неймана* соседними ячейками считаются только четыре смежных ячейки (рис. 15.6, а), в *окрестности Мура* – все восемь (рис. 15.6, б). На гексагональном поле в окрестность входит шесть соседних ячеек (рис. 15.6, в).



а) б) в)

Рис. 15.6. Окрестности ячеек на плоской карте:
а) – окрестность фон Неймана (ход ладьи);
б) – окрестность Мура (ход короля);
в) – гексагональная окрестность (ход героя).

Функции оценки расстояния в евклидовом пространстве:

а) *манхэттенское расстояние* в окрестности фон Неймана – минимальное количество ходов ладьи на шахматной доске (рис. 15.7, а). Названо по количеству кварталов Манхэттена, которые надо проехать такси между стартовым $v$ и конечным $goal$ перекрёстками

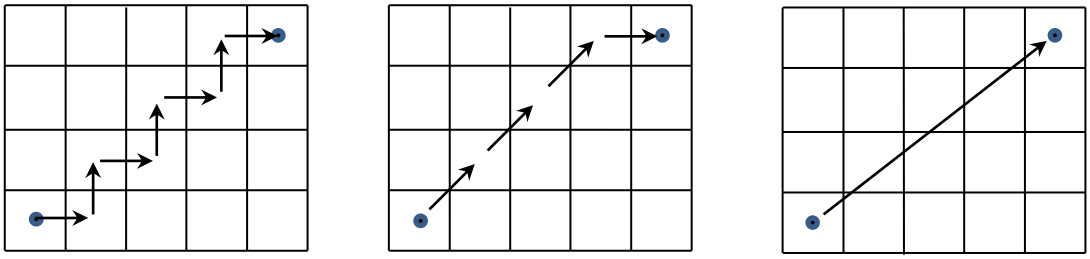
$$h(v) = |v.x - goal.x| + |v.y - goal.y|;$$

б) *расстояние Пафнутия Чебышёва* в окрестности Мура – минимальное количество ходов короля на шахматной доске (рис. 15.7, б). Ход по диагонали равен ходу по вертикали или горизонтали

$$h(v) = max(|v.x - goal.x|, |v.y - goal.y|);$$

в) *расстояние Евклида* на координатной сетке, в которой перемещения не ограничены центрами ячеек (рис. 15.7, в)

$$h(v) = \sqrt{(v.x - goal.x)^2 + (v.y - goal.y)^2};$$



а) б) в)

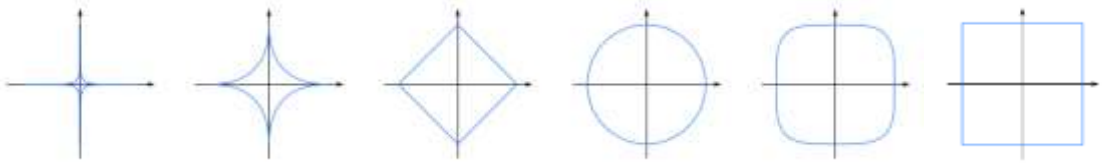
Рис. 15.7. Эвристики определения расстояния на плоской карте:
а) манхэттенское расстояние равно 7;
б) расстояние Чебышёва равно 4;
в) расстояние Евклида $\sqrt{3^2 + 4^2} = 5$

г) *расстояние Германа Минковского* является обобщением функций расстояния в евклидовом пространстве с параметром $p$

$$h(v) = \left( \sum_{i=1}^{n} |v_i - goal_i|^p \right)^{\frac{1}{p}},$$

где $n$ – мерность пространства,

$p$ – параметр, определяющий эквидистантную (равноудалённую от центра) линию (рис. 15.8).

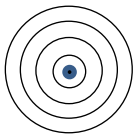


$p = \frac{1}{4}$ $p = \frac{1}{2}$ $p = 1$ $p = 2$ $p = 4$ $p = \infty$

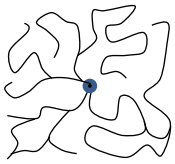
Рис. 15.8. Эквидистантная линия при различных значениях параметра p расстояния Минковского

При $p = 1$ расстояние Минковского является манхэттенским расстоянием, при $p = 2$ –евклидовым расстоянием, при $p = \infty$ обращается в расстояние Чебышёва.

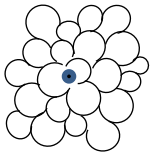
### 15.3 Сравнительный анализ алгоритмов поиска путей



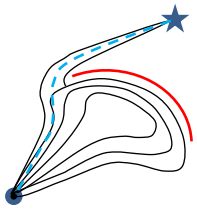
*Поиск сначала вширь*, а также *алгоритм Ли*, выполняют обход графа равномерно во всех направлениях, как круговая волна на воде от брошенного камня. Используется для графов с одинаковыми весами рёбер. Требует дополнительную память для хранения фронта волны и коллекции внутренних посещённых вершин.



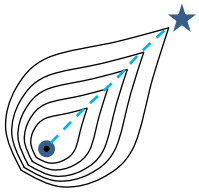
*Поиск сначала вглубь* выполняет последовательный обход всех направлений с помощью откатов назад на основе табу. Использование эвристической функции при выборе направления поиска резко повышает эффективность поиска. Этот поиск прост в реализации. Требует небольшую дополнительную память для хранения списка табу текущего пути.

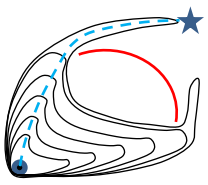


*Алгоритм Дейкстры* раскрывает вершины с наименьшим расстоянием от начала, подобно пенным пузырям, образующимся в самых ближних точках. Алгоритм Дейкстры использует дополнительную память для хранения фронта волны и списка внутренних посещённых вершин. В этом и следующих алгоритмах можно задавать уменьшенные затраты, чтобы алгоритм двигался по автомагистралям, или повышенную стоимость, чтобы он избегал рек, лесов и оврагов, и многое другое.



Среди прочих алгоритмов здесь можно рассмотреть вариант *жадного алгоритма*, использующего только эвристическую оценку $h(v)$ оставшегося до цели пути. Этот алгоритм эффективно решает задачи, удовлетворяющие принцип Беллмана: оптимальные на каждом локальном шаге решения ведут к глобальному оптимальному решению. Такой алгоритм быстро находит путь, но он не всегда будет минимальным. Причиной этого являются возможные препятствия, которые надо обходить, двигаясь в сторону от цели и даже назад, как в лабиринте. Алгоритм требует дополнительную память для хранения списков табу во фронте волны.





*Алгоритм A** использует сумму пройдённого расстояния от начала и оценки расстояния до цели. Такую оценку можно проводить на графах, привязанных к координатной сетке. Этот алгоритм можно

настроить как на широкий поиск, охватывающий возможные препятствия, так и на узконаправленный поиск, устремляющийся строго к цели. Алгоритм A* обладает достоинствами алгоритма Дейкстры и жадного алгоритма. Требует дополнительную память для хранения фронта волны и списка табу.

### 15.4 Рекомендации по выбору алгоритма поиска пути

1. Для нахождения минимального пути между вершинами в неинформированном графе с равноценными рёбрами подходит поиск в ширину и алгоритм Ли. Алгоритм Дейкстры используйте в случае, если стоимость движения по рёбрам различная.

2. Для нахождения минимального пути между вершинами на информированном графе, используйте алгоритм A*. Жадный алгоритм тоже найдёт путь, но не гарантирует, что он оптимальный.

3. Если эвристика алгоритма A* не больше истинного расстояния, то алгоритм A* гарантирует оптимальность решения. Если эвристика уменьшает истинное расстояние до цели, то алгоритм A* в своём поведении смещается в сторону поведения алгоритма Дейкстры, т.е. развивает не самые перспективные пути. Если для всех вершин $h(v) = 0$, то A* превращается в алгоритм Дейкстры. Если эвристика преувеличивает истинное расстояние до цели, то алгоритм A* в своём поведении смещается в сторону поведения жадного алгоритма, т.е. раскрывает только те вершины, которые расположены ближе к цели по прямой линии, рассчитывая, что препятствий на пути не будет.

4. Если используется координатная сетка, то уменьшение размера графа помогает уменьшить вычислительную сложность поиска. Например, граф с 162 вершинами и 263 рёбрами можно преобразовать в граф с 16 вершинами и 22 рёбрами (рис. 15.9). При оценке $O(|V| \cdot |E|)$ вычислительная сложность уменьшится более чем в 100 раз.

5. Если поиск проводится на абстрактном графе, не привязанном к карте, то поиск допустимой эвристики усложнится и может не дать результата вовсе.

6. Если значения $g(v)$ и $h(v)$ измеряются в разных величинах, например, $g(v)$ – это расстояние в километрах, а $h(v)$ – оценка времени пути в часах, то A* может выдать некорректный результат.

7. Поведение алгоритма поиска кратчайшего пути сильно зависит от того, какая эвристика используется. В свою очередь, выбор эвристики зависит от постановки задачи. Часто A* используется для моделирования перемещения по поверхности, покрытой координатной сеткой, так как для неё есть обоснованные эвристики.

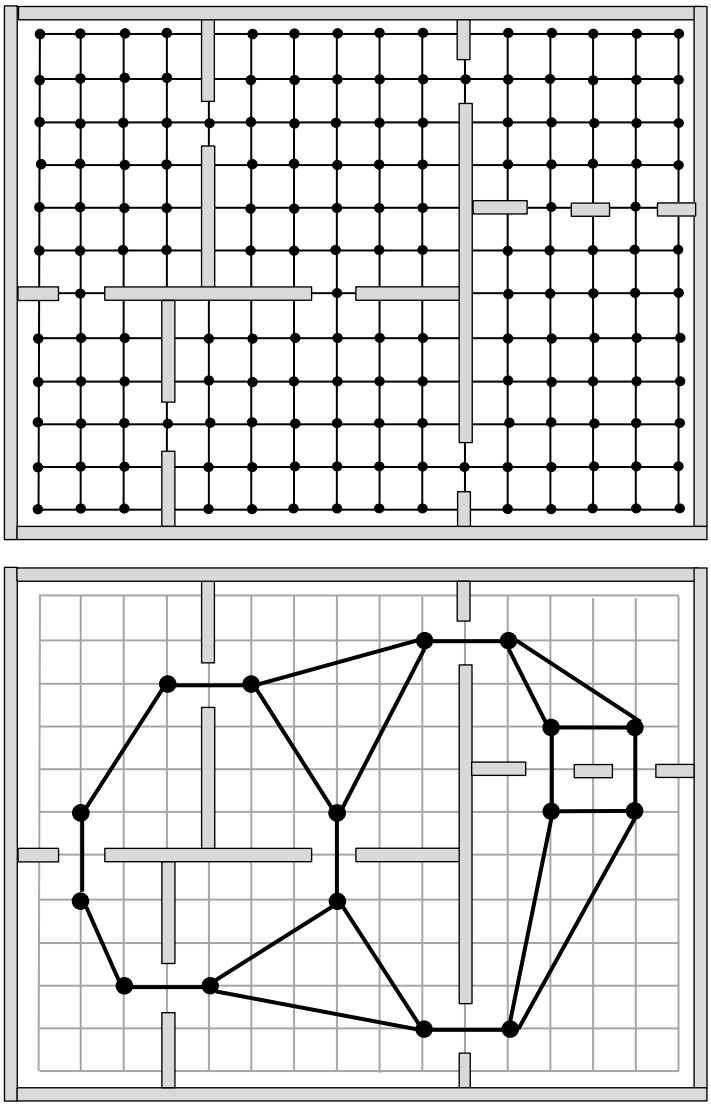


Рис. 15.9. Уменьшение размера графа на координатной сетке

8. В игре пятнашки эвристикой может служить либо количество костяшек, лежащих не на своём месте, либо сумма манхэттенских расстояний от текущих мест костяшек до их целевых мест.

**Задача 1.** Разработать генератор случайной расстановки костей на поле для игры в пятнашки. Учесть неразрешимые расстановки: которые нельзя свести к упорядоченной расстановке.

**Задача 2.** Разработать эвристическую функцию вычисления расстояния до цели в игре пятнашки в виде суммы манхэттенских расстояний каждой костяшки до её места.

**Контрольные вопросы**

1. Чем отличаются информированные методы поиска путей на графах от неинформированных?
2. Какая информация используется в информированных методах поиска?
3. Для чего нужна эвристическая функция в информированных методах поиска?
4. Какие требования накладываются на эвристическую функцию?
5. В чём суть алгоритма A* поиска кратчайшего пути на графе?
6. Как влияет выбор порядка раскрытия вершин фронта в алгоритме A* на ход поиска?
7. Какие бывают окрестности ячеек на плоской карте?
8. Какие бывают эвристические функции оценки расстояния на плоской карте?
9. К какому алгоритму сводится A*, когда выбранная эвристика уменьшает истинное расстояние до цели?
10. К какому алгоритму сводится A*, когда выбранная эвристика преувеличивает истинное расстояние до цели?
11. Для каких графов возможно использование эвристической оценки расстояния до цели?

# 16 Хеширование

## 16.1 Понятие хеш-функции

*Хеширование* – преобразование битовой последовательности произвольной длины в битовую последовательность заданной длины. Функция, выполняющая алгоритм преобразования, называется *хеш-функцией* или *функцией свёртки*. Результат преобразования называется *хешем* или *хеш-значением*.

В значение хеша вносит вклад каждый бит входной последовательности. Если хотя бы один бит исходного сообщения изменится, то хеш тоже изменится.

Некоторые приложения хеширования:

а) контроль целостности при передаче или хранении информации;

б) быстрый поиск в хеш-таблицах;

в) быстрое сравнение строк по признаку равенства;

г) хранение паролей в виде их хешей.

Пусть областью определения хеш-функции $h(x)$ является множество строк $X$ произвольной длины, а её областью значений служат 16-разрядные беззнаковые целые, составляющие множество $H$ с мощностью $|H| = 2^{16}$. Так как множество слов $X$ бесконечно, а множество $H$ ограничено, то будут иметь место множественные коллизии.

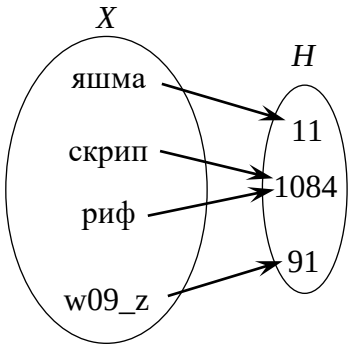



Рис. 16.1. Отображение множества $X$ во множество $H$

*Коллизия* – совпадение хешей двух различных входных строк. Ниже будет показан пример (рис. 16.1) равенства хешей слов «скрип» и «риф»: $h(\text{"скрип"}) = 1084$ и $h(\text{"риф"}) = 1084$. Величина хеша не зависит от длины строки. Для уменьшения числа коллизий мощность области значений хеш-функции необходимо увеличивать.

*Совершенная хеш-функция* – функция, не имеющая коллизий на заданном входном словаре. В практике программирования совершенные хеш-функции не достижимы и являются предметом научного поиска.

Требования к хеш-функции:

а) быстрое вычисление хеш-значения;

б) минимальное число коллизий.

Криптографические хеш-функции не имеют обратной функции либо её вычислительная сложность очень высока. Некриптографические хеш-функции могут иметь обратную функцию.

## 16.2 Примеры хеш-функций, основанных на операциях сдвига и сложения по модулю 2

***Пример 1.*** Пусть хеш-функцией строк является сумма по модулю 2 кодов символов этих строк. Если строки заданы в двухбайтовой кодировке UTF-16LE, то наша хеш-функция будет 16-разрядной. Буквы слова «риф» имеют 16-ричные коды: ‘р’ – 440, ‘и’ – 438, ‘ф’ – 444. Вычислим хеш слова «риф» (рис. 16.1)

$$h(\text{"риф"}) = 440 \oplus 438 \oplus 444 = 43\text{C}_{16} = 1084_{10}.$$

Такое же хеш-значение $43\text{C}_{16}$ имеет множество слов, например,

$$h(\text{"рот"}) = 440 \oplus 43\text{E} \oplus 442 = 43\text{C},$$
$$h(\text{"тор"}) = 440 \oplus 43\text{E} \oplus 442 = 43\text{C},$$
$$h(\text{"скрип"}) = 441 \oplus 43\text{A} \oplus 440 \oplus 438 \oplus 43\text{F} = 43\text{C}.$$

***Пример 2.*** Усовершенствуем нашу хеш-функцию из примера 1. Чётные коды символов будем сдвигать вправо на 1 разряд перед сложением по модулю 2. В слове «риф» все коды букв чётные, поэтому перед сложением эти коды сдвинем вправо на один бит: ‘р’ – 220, ‘и’ – 21C, ‘ф’ – 222. Тогда хеш слова «риф» изменится

$$h(\text{"риф"}) = 220 \oplus 21\text{C} \oplus 222 = 21\text{E}.$$

Аналогично поступим со словом «скрип»: ‘с’ – 441, ‘к’ – 21D, ‘р’ – 220, ‘и’ – 21C, ‘п’ – 43F. Хеш слова «скрип» также изменится

$$h(\text{"скрип"}) = 441 \oplus 21\text{D} \oplus 220 \oplus 21\text{C} \oplus 43\text{F} = 25\text{F}.$$

Предложенная операция сдвига повысила равномерность распределения хеш-значений и посему эти слова получили разные хеш-значения.

***Пример 3.*** Функция MurmurHash основана на сложении по модулю 2 и сдвигах. Дополнительно используется умножение на константу.

***Пример 4.*** Функция Дженкинса основана на арифметическом сложении, сложении по модулю 2 и сдвигах.

## 16.3 Хеширование в конечном поле

Пусть входная последовательность $x$ разделена на битовые слова одинаковой длины $x_1, x_2, \dots, x_m$. Хеш можно вычислить по модулю

$$h(x_1) = (P \cdot x_1)\ mod\ 2^n,$$
$$h(x_2) = (h(x_1) \cdot x_2)\ mod\ 2^n,$$
$$h(x_3) = (h(x_2) \cdot x_3)\ mod\ 2^n,$$
$$\dots$$
$$h(x_m) = (h(x_{m-1}) \cdot x_m)\ mod\ 2^n,$$

где $P$ – число, взаимно простое с $2^n$;

$n$ – разрядность элементов поля.

***Пример 5.*** Хеш-функция FNV (Глен Фаулер, Лондон Нол и Фогн Во) вычисляется в конечном поле, $x_i$ – 32-битное слово

$$h(x_1) = (h_0 \cdot P\ )\ mod\ 2^{32} \oplus x_1,$$
$$h(x_2) = (h(x_1) \cdot P\ )\ mod\ 2^{32} \oplus x_2,$$
$$h(x_3) = (h(x_2) \cdot P\ )\ mod\ 2^{32} \oplus x_3,$$
$$\dots$$
$$h(x_m) = (h(x_{m-1}) \cdot P\ )\ mod\ 2^{32} \oplus x_m,$$

где $h_0 = 2166136261$ – простое число;

$P = 16777619$ – простое число.

Остаток деления на $2^{32}$ быстро определяется приведением к беззнаковому типу `uint`.

## 16.4 Хеширование, основанное на циклических избыточных кодах

*Циклический избыточный код* (англ. cyclic redundancy check, CRC) – остаток деления последовательности битов $X$ на константу $P$

$$CRC = X\ mod\ P.$$

CRC является хешем последовательности $X$. Такой хеш ещё называют *контрольной суммой*, т.к. он вычисляется суммированием по модулю 2 и сдвигами. Роль константы $P$ обычно играет простое число, размер которого в байтах кратен $2^n$.

***Пример 6.*** Вычислим CRC (рис. 16.2) последовательности битов $X$ = 11000100110101 с использованием делителя $P$ = 10011, являющегося простым числом 19. Остаток всегда на один разряд короче делителя. Старший разряд остатка равен нулю, т.к. делимое и делитель имеют единицу в старшем разряде. Поэтому снос разрядов в делимое выполняется до первой единицы.

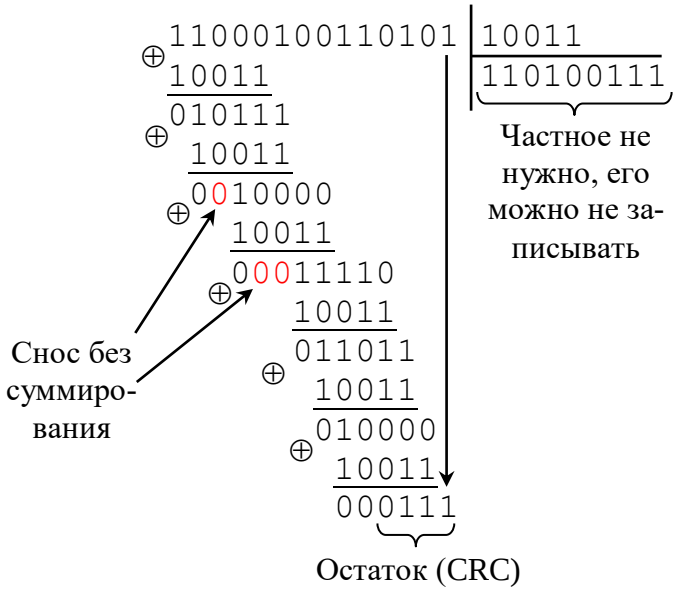



Рис. 16.2. Получение CRC

В теории кодирования делитель $P$ называют *образующим полиномом*

$$P = (1)0011_2 = x^4 + x^1 + x^0.$$

Его старший разряд (в скобках) указывает на старшую степень полинома. Этот старший разряд не пишут, потому что он вынесен за пределы регистра, содержащего CRC (рис. 16.3). Поэтому и длина CRC на один разряд короче длины делителя.

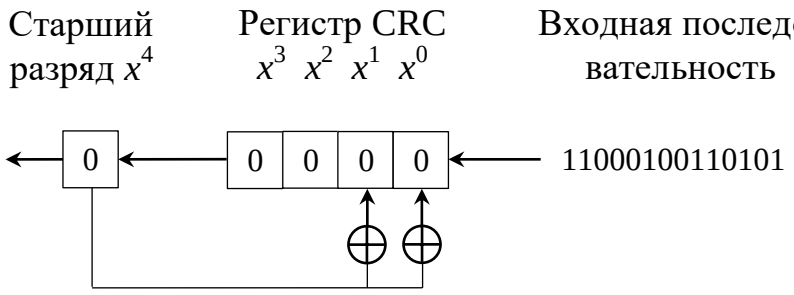



Рис. 16.3. Аппаратная реализация вычислителя CRC

В исходном состоянии регистр CRC содержит нули. При запуске схемы входная последовательность поразрядно сдвигается влево, заполняя регистр CRC. Когда в старший разряд CRC попадает единица, то она складывается по модулю два с двумя младшими разрядами регистра, тем самым инвертируя их. Когда все биты последовательности задвинуты в регистр и выполнено сложение, то алгоритм завершает свою работу, а регистр CRC будет содержать хеш входной последовательности.

Как можно увидеть, образующий полином «зашит» в схему аппаратно, т.к. линии сумматоров подведены строго к разрядам $x^1$ и $x^0$. Для смены образующего полинома надо вносить аппаратные изменения в схему. В компьютерах используется программная реализация хеширования на основе тех же стандартных образующих полиномов.

Стандартные образующие полиномы для CRC

| Название | Полином |
|---|---|
| CRC-8 | $x^8 + x^7 + x^6 + x^4 + x^2 + x^0 = (1)11010101_2 = D5_{16}$ |
| CRC-8-SAE | $x^8 + x^4 + x^3 + x^2 + x^0 = (1)00011101_2 = 1D_{16}$ |
| CRC-16-IBM<br>CRC-16-ANSI | $x^{16} + x^{15} + x^2 + x^0 = (1)1000000000000101_2 = 8005_{16}$ |
| CRC-16-CCITT | $x^{16} + x^{12} + x^5 + x^0 = (1)0001000000100001_2 = 1021_{16}$ |
| CRC-32-IEEE 802.3 | $04C11DB7_{16}$ |

***Пример 7.*** Построим аппаратную реализацию (рис. 16.4) хеширования образующим полиномом CRC-8

$$x^8 + x^7 + x^6 + x^4 + x^2 + x^0 = (1)11010101_2.$$

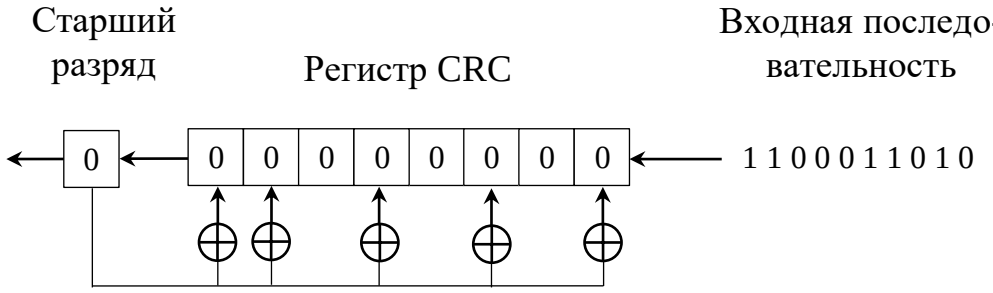



Рис. 16.4. Аппаратная реализация вычислителя CRC-8

**Алгоритм вычисления CRC:**

1. Создать массив (регистр CRC), заполненный нулями, равный по длине разрядности (степени) образующего полинома.
2. Задавить входную последовательность в регистр CRC на один разряд, смещая остальные разряды регистра влево.
3. Если значение, выдвинутое из старшего разряда регистра, равно единице, то инвертировать биты в тех разрядах регистра, которые соответствуют единицам в полиноме.
4. Если входная последовательность пустая, то остановить алгоритм и прочитать содержимое регистра CRC, иначе – перейти к п. 2.

## 16.5 Хеш-таблицы

*Хеш-таблица* – это одномерный массив, хранящий значения по адресам, роль которых играют хеши этих значений. Операциями над хеш-таблицей являются добавление, удаление и поиск данных. Достоинство хеш-таблиц – скорость выполнения операций. Оценка сложности равна $O(1)$. Недостаток – большой размер требуемой памяти, так как количество строк хеш-таблицы равно $2^n$, $n$ – разрядность хеш-функции.

Показатели качества хеш-функций, применяемые к словарю:

а) *равномерность плотности распределения* хеш-значений, определяется по дисперсии плотности распределения;

б) *показатель $\alpha$ заполнения* хеш-таблицы рассчитывался как отношение количества уникальных хеш-значений всех слов словаря к количеству строк хеш-таблицы;

в) *показатель $\beta$ коллизий* в словаре рассчитывался как отношение количества коллизий к общему числу словарных единиц словаря.

**Способы разрешения коллизий**

1. *Двойное хеширование*. Хеш-таблица реализуется в виде двумерного массива. Ключами в примере являются фамилии (рис. 16.5), а записями – фамилия и год рождения. Номер строки массива определяет первая хеш-функция, как правило, многоразрядная, поэтому таблица имеет много строк. Номер столбца определяет вторая хеш-функция – малоразрядная, поэтому столбцов мало. Вторая хеш-функция предназначена для устранения коллизий после первого хеширования. Если второе хеширование не устраняет коллизию, то последняя считается неустранимой в данной хеш-таблице. Выход из такой ситуации заключается в использовании третьего хеширования или изменении второй хеш-функции. Недостаток двойного хеширования – большой размер хеш-таблицы.

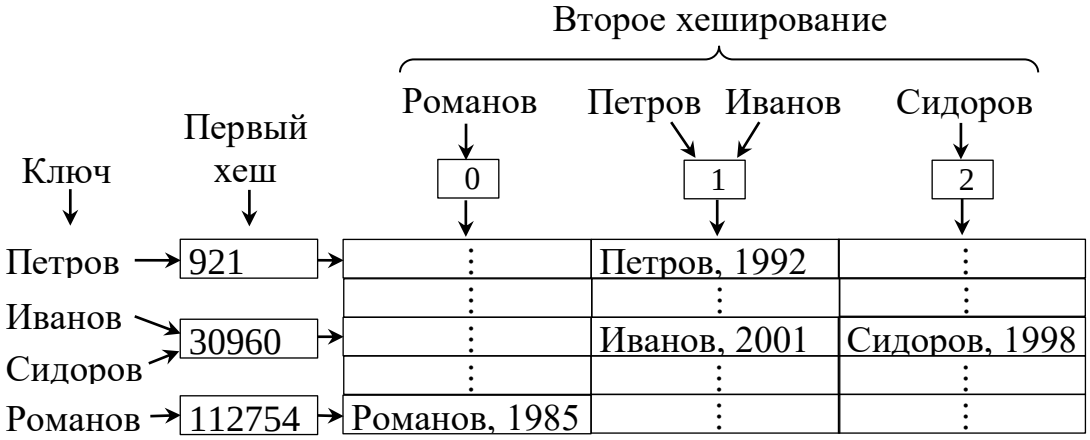



Рис. 16.5. Разрешение коллизий двойным хешированием

2. *Метод цепочек* (открытое хеширование). Хеш-таблица реализуется одномерным массивом цепочек: списков элементов (рис. 16.6). Коллизии приводят к появлению нескольких элементов в цепочке.

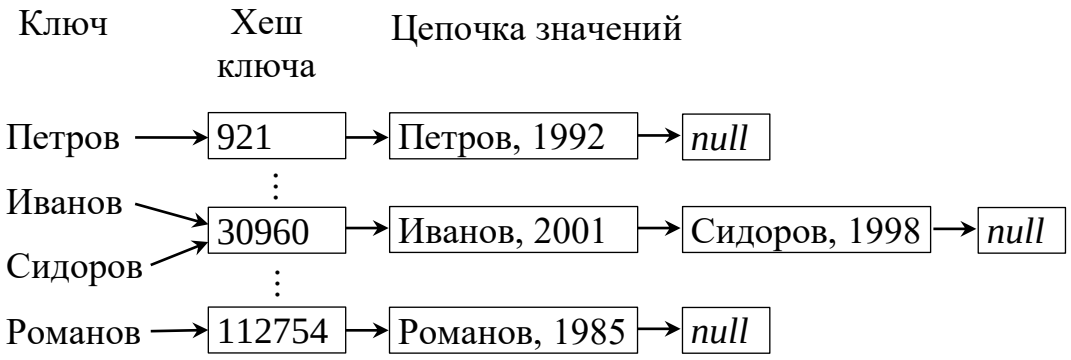



Рис. 16.6. Разрешение коллизий при помощи цепочек

В этом методе может использоваться показатель заполнения хеш-таблицы $\alpha$, который указывает на допустимое число коллизий в строке таблицы. Среднее время доступа к значению хеш-таблицы равно $1+\alpha$. Если в строке цепочка пустая, то искомого значения нет. Если цепочка непустая, то поиск продолжается в цепочке непосредственно по значению. Недостаток метода цепочек заключается в том, что элементы списков не локализованы в памяти из-за использования ссылочных типов данных, поэтому чтение данных плохо кэшируется.

***Пример 8.*** Создадим 16-разрядную хеш-таблицу методом цепочек. Будем использовать младшие 16 разрядов библиотечной хеш-функции `GetHashCode`. В хеш-таблицу `ht` добавим слова из входного массива `w` и продублируем эту операцию выводом на экран слов и их индексов.

```
string[] w = { "пир", "кот", "009", "тир"}; // входной массив слов
List<string>[] ht = new List<string>[65536]; // хеш-таблица
for (int i = 0; i < 65536; i++)
{
    ht[i] = new List<string>(); // инициализация цепочек
}
foreach (string s in w)
{
    int ind = s.GetHashCode() & 0xFFFF; // 16 младших разрядов
    Console.WriteLine($"индекс = {ind}\tстрока: {s}");
    ht[ind].Add(s); // добавление слов в цепочку
}
```

Результат:

```
индекс = 20155  строка: пир
индекс = 14326  строка: кот
индекс = 9965   строка: 009
индекс = 20232  строка: тир
```

3. *Метод открытой адресации* (закрытое хеширование). Хеш-таблица реализуется одномерным массивом значений. Пустые ячейки массива, в которые никогда не записывались данные, имеют метку *null*. Если в ячейках данные были удалены, то в них ставится метка *del*. В случае коллизии при добавлении нового элемента $x$ в позицию $h(x)$ определённый алгоритм просматривает другие ячейки таблицы, лежащие в позициях $h_1(x)$, $h_2(x)$ и т.д., пока не найдёт ячейку с меткой *del* или *null*, в которую и поместит новый элемент. Последовательность просмотра называется *последовательностью проб*. Поиск и удаление используют ту же последовательность проб. При поиске элемента $x$ необходимо просмотреть в таблице все возможные позиции $h(x)$, $h_1(x)$, $h_2(x)$ и т.д., пока не будет найден $x$ или пока не встретится ячейка с меткой *null*. Метка *del* говорит о том, что поиск надо продолжать. Достоинство метода открытой адресации – эко-

номия памяти. Недостаток – увеличенное время операций добавления, удаления и поиска из-за необходимости выполнения проб.

**Типы последовательностей проб**

1. *Линейное пробирование.* Ячейки хеш-таблицы последовательно просматриваются с некоторым фиксированным шагом $k$ между ними, обычно $k = 1$ (рис. 16.7). Пример последовательности пробирования для размера хеш-таблицы равного степени двойки ($N = 2^n$)

$$h(x) = hash(x)\,,$$
$$h_1(x) = (hash(x) + 1 \cdot k)\ mod\ N,$$
$$h_2(x) = (hash(x) + 2 \cdot k)\ mod\ N,$$
$$h_3(x) = (hash(x) + 3 \cdot k)\ mod\ N, \ldots$$

Операция вычисления остатка деления используется для того, чтобы значения хеш-функций $h_i(x)$ не превысили максимальную величину из-за того, что мы к значению базовой хеш-функции $hash(x)$ прибавляем величину $i \cdot k$.

В хеш-таблицу Дуров был добавлен после Иванова (рис. 16.7), но перед Сидоровым. Хеши у Дурова и Иванова равны 30960. Поэтому при пробировании других ячеек для Дурова нашлась свободная «чужая» ячейка, расположенная в следующей позиции 30961. Однако при записи Сидорова его «законная» позиция 30961 уже занята Дуровым, и Сидорова пришлось пробовать записать в другие свободные ячейки. Свободной оказалась следующая ячейка 30962.

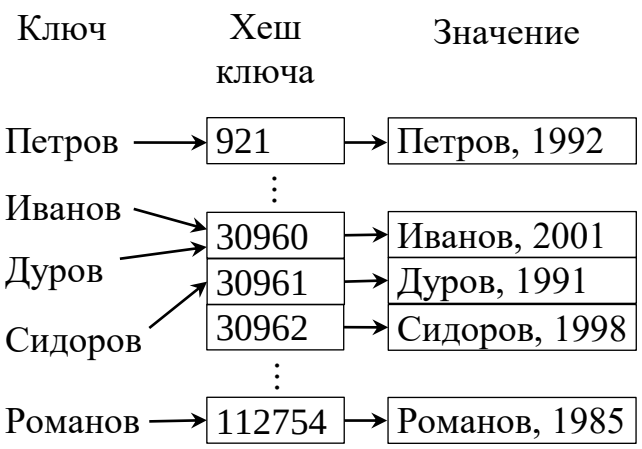



Рис. 16.7. Разрешение коллизий при помощи линейного пробирования

2. *Квадратичное пробирование.* Интервал между ячейками с каждым шагом увеличивается на константу. Если размер хеш-таблицы равен степени двойки ($N = 2^n$), то одним из примеров последовательности, при которой каждый элемент будет просмотрен по одному разу, является:

$$h(x) = hash(x)$$
$$h_1(x) = (hash(x) + 1^2)\ mod\ N,$$
$$h_2(x) = (hash(x) + 2^2)\ mod\ N,$$
$$h_3(x) = (hash(x) + 3^2)\ mod\ N, \ldots$$

3. *Пробирование по второму хешу.* Аналогично линейному пробированию, но шаг просмотра определяется второй малоразрядной хеш-функцией.

Рассмотрим два простых примера использования хеш-таблиц.

***Пример 9.*** Построим букварь текста, содержащий информацию о количестве каждой буквы алфавита в тексте. Пусть текст задан в кодировке ASCII (в Visual Studio ASCII является 7-битовым и потому содержит только первые 128 символов). Будем использовать хеш-таблицу, в которой хешем является код символа.

Решение.

1. Создаём одномерный массив нулевых значений длиной 128 (от 0 до 127 включительно). Он будет являться массивом счётчиков символов.

2. Читаем текст посимвольно. Код очередного символа Code находится в диапазоне от 0 до 127, так как задана однобайтовая кодировка. Этот код используем в качестве хеша символа и, следовательно, индекса.

3. Инкрементируем значение элемента массива с индексом Code.

4. Когда все символы текста прочитаны, останавливаемся и выводим массив на экран.

***Пример 10.*** Сравним буквари двух текстов. Если буквари совпадают, то возвращаем true, иначе возвращаем false.

Решение.

1. Построим хеш-таблицу символов первого текста.

2. При посимвольном чтении второго текста счётчики массива первого текста будем декрементировать.

3. Если в ходе обработки второго текста хотя бы один счётчик станет отрицательным, то буквари различаются, возвращаем false.

4. Если после обработки второго текста все счётчики будут содержать нули, то буквари одинаковые – возвращаем true, иначе – false.

## 16.6 Словари

*Словарь* – это коллекция, хранящая пары (ключ, значение). В языке C# словарь реализован в классе `Dictionary<TKey,TValue>`, где `TKey` – тип ключа, `TValue` – тип хранимого значения. В других языках программирования словари носят название Associative Array или Map.

В языке C# словарь реализован как хеш-таблица с методом цепочек и фиксированной ёмкостью. Время доступа $O(Const)$ определяется скоростью хеширования и длиной цепочки. Ёмкость словаря `Capacity` программно недоступна, но всегда не меньше количества хранимых пар `Count`. При добавлении новой пары в случае `Count > Capacity` ёмкость превентивно увеличивается автоматически.

Пара (ключ, значение) является структурой `KeyValuePair<TKey,TValue>`. Эти структуры можно получить из словаря перечислителем `foreach`

```
var myDictionary = new Dictionary<string, string>();
myDictionary.Add("Иванов","Адрес1");
myDictionary.Add("Петров", "Адрес2");
foreach (KeyValuePair<string, string> tuple in myDictionary)
    Console.WriteLine($"Key = {tuple.Key}, Value = {tuple.Value}");
```

### 16.7 Хеш-деревья

*Хеш-дерево* – это дерево, хранящее пары (хеш_ключа, значение). Достоинство хеш-деревьев – экономия памяти. Время доступа к хранимым элементам $O(\log n)$. Хеш-деревья выгодно использовать там, где объём хранимых данных велик, а время доступа к ним не стоит на первом месте.

**Задача 1.** Разработать алгоритм и программу, вычисляющую $n$-разрядную хеш-функцию FNV из примера 5. Величина $n$ является входным параметром, $8 \leq n \leq 64$.

**Задача 2.** Разработать программу для решения примеров 9 и 10.

### Контрольные вопросы

1. Раскройте понятие «хеширование».
2. Каково назначение хеширования?
3. В чём суть хеш-функции?
4. Имеет ли хеш-функция обратную?
5. Раскройте понятие коллизии.
6. Что такое совершенная хеш-функция?
7. Каковы требования, выдвигаемые к хеш-функциям?
8. Что такое циклический избыточный код?
9. Раскройте понятие образующего полинома.
10. В чём заключается алгоритм вычисления CRC?
11. Какие параметры влияют на значение контрольной суммы?
12. Что такое хеш-таблица?
13. Каковы методы разрешения коллизий в хеш-таблицах?
14. Каковы достоинства и недостатки хеш-деревьев?

# 17 Поиск подстроки в строке

Принадлежность подстроки (образца) строке в программировании называется *вхождением подстроки* (образца) в строку. *Позиция вхождения* – это позиция первого символа подстроки, входящей в строку. Сравнение строк бывает контекстно-зависимым и контекстно-независимым. *Контекстно-зависимое* сравнение строк выполняется путём сравнения кодов символов строк, занимающих одинаковые позиции. *Контекстно-независимое* сравнение сравнивает символы, предварительно приведённые к нижнему регистру. Поэтому при таком сравнении заглавные и строчные буквы неразличимы.

## 17.1 Наивный поиск подстроки в строке

Обозначим символы строки крестиками. Поиск образца `abcba` в строке `××××××××××××` заключается в посимвольном сопоставлении образца и строки. Если образец не найден в нулевой позиции строки, обозначенной верхней скобкой, то его поиск повторяется с первой позиции и так далее (рис. 17.1). Последнее сравнение образца со строкой произойдёт в позиции $n$-$m$, $n$ – длина строки, $m$ – длина образца. Эта позиция указана верхней пунктирной скобкой.

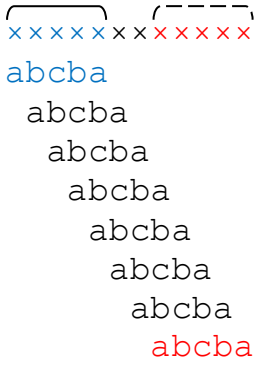



Рис. 17.1. Наивный поиск

Оценка сложности $O(m(n-m))$. В худшем случае $m = \frac{n}{2}$, что приводит к оценке $O(n^2)$. Однако, если $m$ мало по сравнению с $n$, то сложность близка к $O(n)$.

## 17.2 Алгоритм Рабина – Карпа приближённого поиска

*Алгоритм Рабина – Карпа* ищет предвычисленное значение хеш-функции образца в строке. Он был разработан в 1987 г. Майклом Рабином и Ричардом Карпом. Суть алгоритма заключается в сравнении хеш-значений образца и подстроки такой же длины вместо сравнения самих строк. Такой подход существенно уменьшает время поиска, так как сравнение кодов всех пар символов заменяется сравнением двух хешей. Если хеш-значения совпали, то вероятность совпадения самих строк велика, и в этом случае можно сравнить строки посимвольно. При использовании хороших хеш-функций коллизии случаются крайне редко.

Для того чтобы заново не вычислять хеш-значение каждой новой подстроки, сдвинутой на один символ относительно текущей подстроки,

применяют *кольцевые* хеш-функции. Они позволяют быстро вычислить хеш-значение новой подстроки по хеш-значению текущей подстроки.

В качестве примера простейшей кольцевой хеш-функции рассмотрим функцию в виде арифметической суммы кодов символов. Пусть в строке «abracadabra» надо вычислить хеши всех восьми подстрок длиной четыре символа. Хеш первой подстроки «abra»

$$h("abra") = 97_a + 98_b + 114_r + 97_a.$$

Нижним индексом указана буква кода. Для вычисления хеша второй подстроки «brac» достаточно вычесть из хеша первой подстроки $h("abra")$ код первого символа $97_a$ и добавить код нового символа $99_c$

$$h("brac") = h("abra") - 97_a + 99_c.$$

Указанный способ вычисления существенно эффективней по сравнению с суммированием кодов всех символов второй подстроки заново. Поступая так дальше, легко получить хеши остальных подстрок.

В алгоритме Рабина-Карпа используется полиномиальный кольцевой хеш

$$h(a_1 a_2 \dots a_n) = (a_1 x^{n-1} + a_2 x^{n-2} + \cdots a_n x^0) \bmod q,$$

где $a_i$ – коды символов;

$x = 2$;

$q$ – простое число, например, простое число Мерсéнна $2^{31} - 1$ или $2^{61} - 1$.

Для получения хеша новой подстроки необходимо из хеша текущей подстроки вычесть хеш левого символа $a_1 x^{n-1}$, умножить остаток на $x$ и добавить код нового символа. Для ускорения вычитания $a_1 x^{n-1}$ в программе хранят предвычисленное значение величины $x^{n-1} \bmod q$.

Таким образом, вычисление хеша новой подстроки требует вычитания величины $a_1 x^{n-1}$, умножения на 2 посредством сдвига влево и сложения с кодом нового символа

$$h(a_2 a_3 \dots a_{n+1}) = \left((h(a_1 a_2 \dots a_n) - a_1 x^{n-1})x + a_{n+1}\right) \bmod q.$$

В программной реализации операцию $\bmod q$ можно не выполнять, если результат преобразовывать к типу `int` или `long`.

**Алгоритм Рабина – Карпа**

Дана строка $s$ длиной $n$ и образец $p$ длиной $m$ символов. Алгоритм поиска всех вхождений содержит следующие шаги:

1. Вычислить хеш образца

$$hp = h(p[0..m-1]).$$

2. Вычислить хеш подстроки в позиции 0

$$hs(0) = h(s[0..m-1]).$$

3. Если условие $hp = hs(0)$ истинно, то в позиции 0 найдено вхождение образца в строку.

4. Для всех $i = 1..n-m-1$ повторять:

а) вычислить кольцевой хеш подстроки в позиции $i$

$$hs(i) = h(s[i..i+m-1]);$$

б) если условие $hp = hs(i)$ истинно, то в позиции $i$ найдено вхождение образца в строку.

***Пример 1.*** Пусть в строке «abracadabra» надо найти все вхождения образца «bra». Так как длина образца мала, то остатки деления на простое число $q$ вычислять не будем: переполнения разрядной сетки гарантированно не будет. Коды используемых букв: «a» – 97, «b» – 98, «c» – 99, «d» – 100, «r» – 114.

1. Хеш образца «bra»

$$hp = 98 \cdot 2^2 + 114 \cdot 2^1 + 97 = 717.$$

2. $i$ = 0, хеш первой подстроки «abr»

$$hs = 97 \cdot 2^2 + 98 \cdot 2^1 + 114 = 698,$$
$$hp \neq hs.$$

3. $i$ = 1, хеш второй подстроки «bra»

$$hs = (hs - 97 \cdot 2^2) \cdot x + 97 = 717,$$
$$hp = hs.$$

В позиции $i$=1 скорее всего найден образец.

4. $i$ = 2, хеш третьей подстроки «rac»

$$hs = (hs - 98 \cdot 2^2) \cdot x + 99 = 749,$$
$$hp \neq hs.$$

5. $i$ = 3, хеш четвёртой подстроки «aca»

$$hs = (hs - 114 \cdot 2^2) \cdot x + 97 = 683,$$
$$hp \neq hs.$$

6. $i$ = 4, хеш пятой подстроки «cad»

$$hs = (hs - 97 \cdot 2^2) \cdot x + 100 = 690,$$
$$hp \neq hs.$$

Дальнейшие шаги не приводим ввиду их подобия рассмотренным.

Алгоритм Рабина – Карпа может одновременно искать $k$ образцов одинаковой длины за время $O(n+k)$. Для этого хеши всех образцов надо поместить в хеш-таблицу и проверять хеш очередной подстроки в этой хеш-таблице. Так работает система «Антиплагиат».

В среднем случае оценка вычислительной сложности $O(n)$. В худшем случае $O(mn)$.

## 17.3 Алгоритм Кнута – Морриса – Пратта

*Алгоритм Кнута – Морриса – Пратта* (КМП) – эффективный алгоритм поиска подстроки в строке. Оценка сложности алгоритма линейная $O(n+m)$. Разработать асимптотически более эффективный алгоритм невозможно. Алгоритм КМП основан на применении префикс-функции. Собственно эта функция и была открыта для поиска подстроки в строке.

**Префикс-функция**

*Бо́рдер* – непустая подстрока, являющаяся одновременно префиксом и суффиксом строки длиной больше одного символа. Позиции префикса и суффикса не должны совпадать. Максимальная длина бордера на единицу меньше длины строки. Примеры строк и их бордеров:

- «a» – имеет бордер «» длиной 0;
- «aa» – имеет бордер «a» длиной 1;
- «ab» – имеет бордер «» длиной 0;
- «aba» – имеет бордер «a» длиной 1;
- «aaa» – имеет бордер «aa» длиной 2;
- «abab» – имеет бордер «ab» длиной 2.

*Префикс-функция* – массив длин наибольших бордеров каждого префикса строки при просмотре слева направо.

Если задана строка $s[0 \dots n-1]$, то префикс-функция вычисляет массив чисел $\pi[0 \dots n-1]$, в котором $\pi[i]$ – длина наибольшего собственного суффикса подстроки $s[0 \dots i]$, совпадающего с её префиксом.

*Собственный суффикс* – суффикс, не совпадающий со всей строкой. Поэтому всегда $\pi[0] = 0$ и $\pi[1] = 0$.

***Пример 2.*** Найдём префикс-функцию строки «abcabcd».

- «a» – не имеет собственного суффикса;
- «ab» – не имеет префикса, совпадающего с суффиксом;
- «abc» – не имеет префикса, совпадающего с суффиксом;
- «abca» – префикс «a» длины 1 совпадает с суффиксом «a»;
- «abcab» – префикс «ab» длины 2 совпадает с суффиксом «ab»;
- «abcabc» – префикс «abc» длины 3 совпадает с суффиксом «abc»;
- «abcabcd» – не имеет префикса, совпадающего с суффиксом.

Следовательно, префикс-функция равна массиву $\pi = \{0,0,0,1,2,3,0\}$.

Обратите внимание на то, что следующий бордер в примере 2 можно найти по значению текущего бордера, например, строка «abc» имеет бордер длиной ноль символов. Для определения бордера новой строки «abca» достаточно сравнить символ, стоящий в позиции, равной длине бордера прежней строки, и последний символ новой строки. Другими словами, надо сравнить символ, стоящий в нулевой позиции, и последний символ

строки «abca». Сравнение успешно, следовательно, длина нового бордера на единицу больше длины текущего бордера, т.е. равна одному символу.

Рассмотрим в этом же примере 2 строку «abcab». Она имеет бордер «ab» длиной два символа. Поэтому для нахождения бордера новой строки «abcabc» сравним символ, стоящий во второй позиции, с последним символом. Сравнение снова успешно, следовательно, длина нового бордера на единицу больше длины текущего бордера, т.е. равна трём. В случае неуспешного сравнения, бордер становится пустой строкой.

На основе вышеописанного можно определить префикс-функцию.

```
static int[] Prefix_function(string s)
{
    int n = s.Length;
    int[] pi = new int[n];
    for (int i = 1; i < n; ++i)
    {
        int j = pi[i - 1];      // длина предыдущего бордера
        while (j > 0 && s[i] != s[j])
            j = pi[j - 1];      // позиция в подстроке s
        if (s[i] == s[j]) ++j; // проверка совпадения букв
        pi[i] = j;              // длина нового бордера
    }
    return pi;
}
```

Использовать префикс-функцию следует следующим образом. Пусть дан текст $t$ длиной $n$ и строка $s$ длиной $m$. Требуется найти позиции всех вхождений строки $s$ в текст $t$. Построим строку $w = s + \text{"\u0000"} + t$, где символ "\u0000" – это разделитель, который не должен принадлежать строке $s$ и тексту $t$.

Вычислим префикс-функцию `Prefix_function(w)` для строки $w$ и рассмотрим её значения $\pi[i]$ правее разделителя, т.е. для $i > m + 1$, которые, относятся к тексту $t$. Величина $\pi[i]$ показывает максимальную длину суффикса, совпадающего с префиксом. Величина $i$ является позицией последнего символа суффикса в строке $w$. Равенство $\pi[i] = m$ означает, что префикс $s$ найден в $w$. Здесь надо учесть, что позиции в строке $w$ нумеруются с самого начала. Поэтому позиция найденной в тексте $t$ подстроки $s$ определится по формуле

$$pos = i - 2m.$$

***Пример 3.*** Вариант использования префикс-функции для поиска строки "bra" в тексте "abraabra" представлен ниже.

```
static void Main()
{
    string c = "\u0000";    // разделитель, "\a"
    string s = "abraabra";  // текст
```

```
    string p = "bra";       // образец
    Console.WriteLine(string.Join(" ", Prefix_function(p+c+s)));
    Console.ReadKey();
}
```

Результат

```
0 0 0 0 0 1 2 3 0 1 2 3
```

Искомая строка имеет длину 3, следовательно, надо искать тройки в префикс-функции. Первая тройка расположена по индексу 7, значит позиция первого вхождения $7 - 2 \cdot 3 = 1$, и вторая тройка – по индексу 11, значит позиция второго вхождения $11 - 2 \cdot 3 = 5$.

### 17.4 Алгоритм Бойера – Мура

Алгоритм Бойера – Мура считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки в строке. Алгоритм Бойера – Мура используется в стандартных приложениях и в командах Ctrl+F браузеров и текстовых редакторов.

Важной особенностью алгоритма является сравнение символов справа налево. В случае несовпадения символов образец сдвигается вправо. Величина сдвига определяется с помощью одной из двух предварительно вычисленных таблиц: таблицы символов, не совпавших с последним символом образца (плохих символов или *стоп-символов*), и таблицы совпавших суффиксов (*хороших суффиксов*).

**Таблица стоп-символов**

Таблица стоп-символов нужна для определения величины сдвига образца при несовпадении его правого символа с символом строки.

***Пример 4.*** Рассмотрим построение таблицы стоп-символов. Будем искать образец `abcab` в некоторой строке. Символы строки обозначены крестиками, т.к. мы их изначально не знаем. Совмещаем левые края образца и строки и сравниваем крайний правый символ `b` образца с символом строки (пусть это будет символ `z`)

```
××××z××××××××××××××××××××
abcab
     abcab
```

Символ `z` не совпал с символом образца и вообще он не принадлежит образцу, следовательно, мы можем сдвинуть образец на пять символов вправо, т.е. на всю его длину.

Пусть напротив последнего символа образца находится символ `a` строки. Очевидно, что в этом случае образец надо сдвинуть вправо на один символ

```
××××z××××a××××××××××××××××
      abcab
       abcab
```

Если бы напротив последнего символа образца находится символ c, то сдвиг вправо должен быть на два символа

```
××××z××××c××××××××××××××××
      abcab
        abcab
```

Сейчас мы можем заполнить таблицу стоп-символов, в которой определена величина сдвига образца вправо:

| Стоп-символ | a | b | c | Остальные символы |
|---|---|---|---|---|
| Величина сдвига | 1 | - | 2 | 5 |

Обратите внимание, что символ b лишён своего сдвига, т.к. является «хорошим» символом, ибо с ним никогда не будет неудачного сравнения. В программе вместо знака «-» обычно указывают значение 0 или -1, которое является признаком перехода к поиску по таблице хороших суффиксов.

**Таблица хороших суффиксов**

Если при сравнении текста и образца совпал один или больше символов суффикса, то образец сдвигается в зависимости от того, какой суффикс совпал, и какой символ перед этим суффиксом не совпал.

***Пример 5.*** Будем искать образец babcab в некоторой строке. Совмещаем левые края и сравниваем символы справа налево. Пусть совпал суффикс длины 1, т.е. последний символ b образца совпал с символом строки, а предпоследний, обозначенный знаком вопроса, не совпал

```
××××āb××××××××××××××××××
babcab
```

Хороший суффикс – символ b, но в строке перед хорошим суффиксом стоит символ, отличный от a, обозначим его ā (не «a»). Ищем подстроку āb в образце, двигаясь справа налево. Символ ā может совпадать с любым символом, отличным от a, а также он может совпасть с «пустым» символом. В нашем образце подстрока āb длиной 2 отсутствует. Однако есть подстрока āb длиной 1 – это первый символ образца b, которому предшествует «пустой» символ

```
××××āb××××××××××××××××××
babcab
      babcab
```

Для совмещения подстроки āb с образцом мы должны сдвинуть образец вправо на 5 символов.

Рассмотрим следующий случай – совпал суффикс длины 2, а третий символ не совпал

```
×××c̄ab××××××××××××××××××
babcab
```

Хороший суффикс ab, но в строке перед хорошим суффиксом стоит некоторый символ c̄, отличный от c. Ищем подстроку c̄ab в образце, двигаясь справа налево. В нашем образце есть такая подстрока – bab

```
×××c̄ab××××××××××××××××××
babcab
   babcab
```

Для совмещения подстроки $\bar{c}ab$ с образцом мы должны сдвинуть образец вправо на 3 символа.

Рассмотрим третий случай – совпал суффикс длины 3, а четвёртый символ не совпал

```
××b̄cab×××××××××××××××××
babcab
```

Хороший суффикс cab и в строке перед хорошим суффиксом стоит некоторый символ b̄, отличный от b. Ищем подстроку b̄cab в образце, двигаясь справа налево. В нашем образце нет такой подстроки, но есть префикс b

```
××b̄cab×××××××××××××××××
babcab
     babcab
```

Следовательно, сдвиг вправо должен быть на 5 символов.

Рассмотрим четвёртый случай – совпал суффикс длины 4, а пятый символ не совпал

```
×ābcab×××××××××××××××××
babcab
```

Хороший суффикс bcab и в строке перед хорошим суффиксом стоит некоторый символ ā, отличный от a. Ищем подстроку ābcab в образце, двигаясь справа налево. В нашем образце нет такой подстроки, но есть префикс b

```
×ābcab×××××××××××××××××
babcab
     babcab
```

Следовательно, сдвиг вправо должен быть на 5 символов.

Рассмотрим пятый случай – совпал суффикс длины 5, а шестой символ не совпал

```
b̄abcab××××××××××××××××××
babcab
```

Хороший суффикс `abcab` и в строке перед хорошим суффиксом стоит некоторый символ $\bar{b}$, отличный от `b`. Ищем подстроку $\bar{b}$`abcab` в образце, двигаясь справа налево. В нашем образце нет такой подстроки, но есть префикс `b`

```
b̄abcab××××××××××××××××××
babcab
      babcab
```

Следовательно, сдвиг вправо должен быть на 5 символов.

Рассмотрим последний случай – совпал весь образец

```
babcab××××××××××××××××××
babcab
```

Хороший суффикс является образцом `babcab`. Сейчас нам надо найти величину сдвига образца для поиска его второго вхождения в строку. Ищем подстроку `babcab` в образце, двигаясь справа налево

```
babcab××××××××××××××××××
babcab
      babcab
```

Следовательно, сдвиг вправо должен быть на 5 символов.

Теперь можно построить таблицу сдвигов по совпавшему суффиксу:

| Хороший суффикс | babcab | abcab | bcab | cab | ab | b |
|---|---|---|---|---|---|---|
| Величина сдвига | 5 | 5 | 5 | 5 | 3 | 5 |

*Добрый совет* – совпадающие фрагменты после сдвига образца можно не сравнивать.

**Оценка сложности алгоритма Бойера – Мура**

Предварительное вычисление табличных сдвигов образца длины $m$ в алфавите длиной $k$ имеет оценку $O(m^2 + k)$. Для ASCII $k = 256$, для двухбайтового Unicode $k = 65\,536$.

Сложность поиска строки в тексте длиной $n$ в худшем случае $O(n \cdot m)$. Такие случаи возникают, когда строка многократно входит в текст и содержит повторяющийся символ. На практике таких особых случаев не бывает. Поэтому на естественно- и искусственно-языковых текстах оценка вычислительной сложности $O(m + n)$.

**Достоинства:**

1. Алгоритм Бойера – Мура на хороших данных очень быстр, а вероятность появления плохих данных крайне мала. Поэтому он оптимален в большинстве случаев, когда нет возможности провести предварительную обработку текста, в котором проводится поиск.

2. На больших алфавитах (относительно длины образца) алгоритм чрезвычайно быстрый и требует намного меньше памяти, чем алгоритм Ахо-Корасик, который мы рассмотрим в следующем разделе.

3. Позволяет добавить множество модификаций, таких как поиск подстроки, включающей любой символ в заданной позиции.

**Недостатки:**

1. Алгоритмы семейства Бойера – Мура не расширяются до приблизительного поиска и поиска нескольких строк.

2. На больших алфавитах (например, Юникод) может занимать много памяти. В таких случаях обходятся хеш-таблицами или дробят алфавит, рассматривая, например, 4-байтовый символ как пару двухбайтовых.

3. На искусственно подобранных неудачных текстах скорость алгоритма Бойера-Мура серьёзно снижается.

**Задача 1.** Разработать программу поиска подстроки в строке на основе алгоритма Рабина – Карпа.

**Задача 2.** Разработать программу вычисления таблицы стоп-символов для алгоритма Бойера – Мура. Экспериментально оценить вычислительную сложность.

**Контрольные вопросы**

1. Раскройте суть наивного алгоритма поиска подстроки в строке.
2. В каких случаях наивный поиск эффективен?
3. Почему алгоритм Рабина – Карпа является приближённым?
4. Что такое кольцевая хеш-функция?
5. Какие преимущества даёт кольцевая хеш-функция в сравнении с обычной хеш-функцией?
6. Почему максимальная длина искомой подстроки в алгоритме Рабина – Карпа ограничена?
7. Раскройте понятие бордера в алгоритме КМП.
8. Что такое префикс-функция?
9. Какова оценка сложности алгоритма КМП?
10. В чём заключается суть алгоритма Бойера – Мура?
11. Где используется алгоритм Бойера – Мура?
12. Что такое таблица стоп-символов в алгоритме Бойера – Мура?

13. В чём отличие таблицы хороших суффиксов от таблицы стоп-символов в алгоритме Бойера – Мура?

14. Какова оценка сложности алгоритма Бойера – Мура?

# 18 Поиск подстрок в строке

## 18.1 Бор для хранения и поиска подстрок в строке

*Бор* (англ. trie) – структура данных для хранения набора строк, представляющая из себя *префиксное дерево* с символами на рёбрах (рис. 18.1). Вершины такого графа имеют поле, содержащее признак конца строки. Если вершина является концом строки, то она называется *терминальной*. Одна ветвь дерева может содержать несколько терминальных вершин. Это происходит тогда, когда короткое слово является префиксом длинного слова. Строки получаются последовательным соединением всех символов, хранящихся на рёбрах между корнем бора и терминальной вершиной (рис. 18.2). Размер бора линейно зависит от суммы длин всех хранимых строк, а время поиска в боре пропорционально длине искомого образца.

По сути бор является деревом с произвольной степенью вершин. Каждая вершина хранит список ссылок на дочерние вершины и признак того, является ли она терминальной для некоторого слова из словаря.

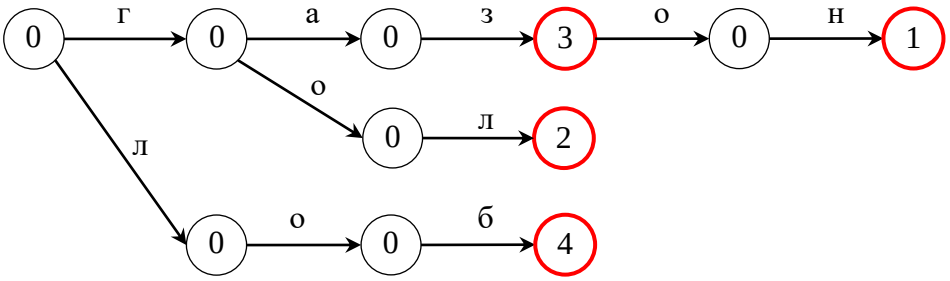



Рис. 18.1. Бор

***Пример 1.*** Множество слов и их номеров {газон – 1, гол – 2, газ – 3, лоб – 4} представлено в префиксном дереве (рис. 18.1), имеющем всего 10 рёбер и столько же вершин, не считая корень. Терминальные вершины выделены красными обводами и содержат номера слов. Нетерминальные вершины хранят ноль. Количество рёбер дерева меньше суммы всех букв во множестве слов, т.к. бор содержит общие префиксы «г», «га» и «газ».

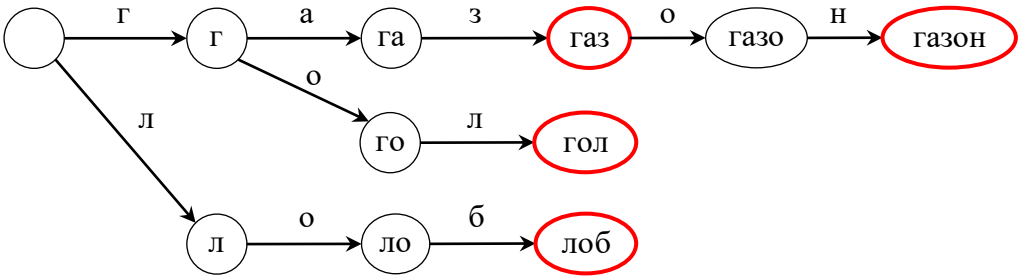



Рис. 18.2. Демонстрация соединения букв в боре

Поиск всех слов бора начинается от корня. Мы не имеем права начинать поиск слова в произвольном месте бора. Например, слово «лоб» нельзя начать с последней буквы слова «гол». Образование слов происходит путём соединения букв на рёбрах одной ветви (рис. 18.2).

Вершины бора определяются классом `Vertex`, которое имеет поле `char Letter`, содержащее символ ребра, идущего от родительской вершины, поле `int Terminal` с номером хранимого слова. Если вершина не является терминальной, то `Terminal=0`. Если вершина терминальная, то `Terminal>0`. Список дочерних вершин хранится в поле `List<Vertex> Next`.

```
class Vertex
{
    public char Letter;
    public int Terminal;
    public List<Vertex> Next;
    public Vertex(char letter, int terminal)
    { Letter = letter; Terminal = terminal; Next = new
List<Vertex>(); }
    public Vertex(char letter, int terminal, List<Vertex> next)
    { Letter = letter; Terminal = terminal; Next = next; }
}
```

Корень бора является фиктивной вершиной. Поле `Letter` корня содержит нулевой код, что означает пустой символ. Именно из корня бора начинается поиск образцов в строке.

Алгоритм поиска образцов, хранящихся в боре, заключается в его обходе из корня по тем рёбрам, символы которых встречаются в строке. При достижении вершины, имеющей ненулевое терминальное поле, образец считается найденным. Однако поиск может продолжаться, если у этой терминальной вершины есть дочки. При отсутствии в боре ребра, соответствующего очередной букве строки, совершается переход в корень.

***Пример 2.*** Создадим бор (рис. 18.1) и найдём в строке `"толоконный лоб"` образцы из этого бора.

```
private static void Main(string[] args)
{
    var root = new Vertex('\0',0); // создание вершин бора
    var c1 = new Vertex('г', 0);
    var c2 = new Vertex('а', 0);
    var c3 = new Vertex('з', 3);   // газ
    var c4 = new Vertex('о', 0);
    var c5 = new Vertex('н', 1);   // газон
    var c6 = new Vertex('о', 0);
    var c7 = new Vertex('л', 2);   // гол
    var c8 = new Vertex('л', 0);
```

```
    var c9 = new Vertex('о', 0);
    var c10 = new Vertex('б', 4);   // лоб
    root.Next = new List<Vertex>() { c1, c8 }; // рёбра бора
    c1.Next = new List<Vertex>() { c2, c6 };
    c2.Next = new List<Vertex>() { c3 };
    c3.Next = new List<Vertex>() { c4 };
    c4.Next = new List<Vertex>() { c5 };
    c6.Next = new List<Vertex>() { c7 };
    c8.Next = new List<Vertex>() { c9 };
    c9.Next = new List<Vertex>() { c10 };
    string s = "толоконный лоб";
    Vertex n = root;                // активная вершина бора
    for (int i = 0; i < s.Length; i++)
    {
        n = n.Next.Find(x => x.Letter == s[i]); // поиск буквы s[i]
        if (n != null)
        {
            if (n.Terminal > 0)
                Console.WriteLine($"Найден образец №{n.Terminal}");
        }
        else n = root;
    }
    Console.ReadKey();
}
```

Результат

```
Найден образец №4
```

Поиск образцов в тексте `"толоконный лоб на газоне"` приведёт к нахождению трёх слов: «лоб», «газ» и «газон»

```
Найден образец №4
Найден образец №3
Найден образец №1
```

Вышеописанный пример показывает факт вхождения подстроки в строку, но не возвращает позицию вхождения.

В профессиональной реализации бора вместо списка дочерних вершин `Next` используют хеш-таблицу ссылок. Хешем может служить код символа. Это позволяет искать элемент по очередной букве строки за время $O(1)$, в отличие от поиска в списке за время $O(n)$, $n$ – длина списка.

**Достоинства бора:**

1. Быстрое добавление нового образца в бор за время $O(m)$, $m$ – длина образца.
2. Поиск всех образцов в тексте длиной $k$ за время $O(k)$.

## 18.2 Алгоритм Ахо – Корасик

*Алгоритм Ахо – Корасик* разработан Альфредом Ахо и Маргарет Корасик в 1975 году и реализует поиск по орграфу, полученному из бора. Бор преобразуется в орграф путём добавления дуг для организации переходов между подстроками образцов. Это важно, когда часть одного слова входит в другое слово.

Орграф, полученный из бора образцов {«газ», «газон», «зона», «нога», «ноль»}, имеет дополнительные дуги, выделенные пунктиром (рис. 18.3). Эти дуги показывают переходы между подстроками различных слов. На орграфе не показаны дуги, осуществляющие переход в корень бора из тех вершин, для которых нет исходящей дуги с очередной буквой строки. Для наглядности в вершинах орграфа показаны собранные последовательности букв. Терминальные вершины выделены красным обводом.

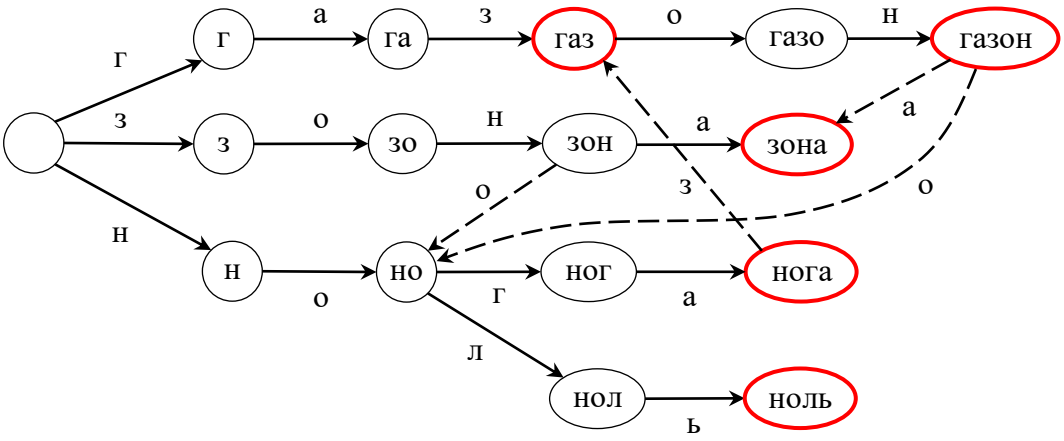



Рис. 18.3. Орграф работы алгоритма Ахо – Корасик

По орграфу (рис. 18.3) можно найти все образцы {«нога», «газ», «газон», «ноль»} в строке «ногазоноль». Это выгодно отличает его от алгоритма поиска в боре.

Для программного преобразования бора в орграф надо знать информацию обо всех ветвях, расположенных выше (левее) текущей вершины, т.е. ближе к корню. В этих ветвях следует искать префиксы максимальной длины, совпадающие с суффиксами текущей вершины.

Если в графе есть ветви без ветвлений – *бамбуки*, то тогда каждый такой бамбук можно заменить одной дугой, имеющей соединённые в суффикс буквы ветвей. Такие дуги называются *сжатыми суффиксными ссылками* (рис. 18.4). Их использование повышает скорость работы алгоритма Ахо – Корасика.

Алгоритм Ахо – Корасик широко применяется в системном программном обеспечении, в построении баз сигнатур вирусов. Одна такая

база может занимать сотни мегабайт, однако разница в скорости поиска вирусов между 2- и 200-мегабайтной базами совсем невелика.

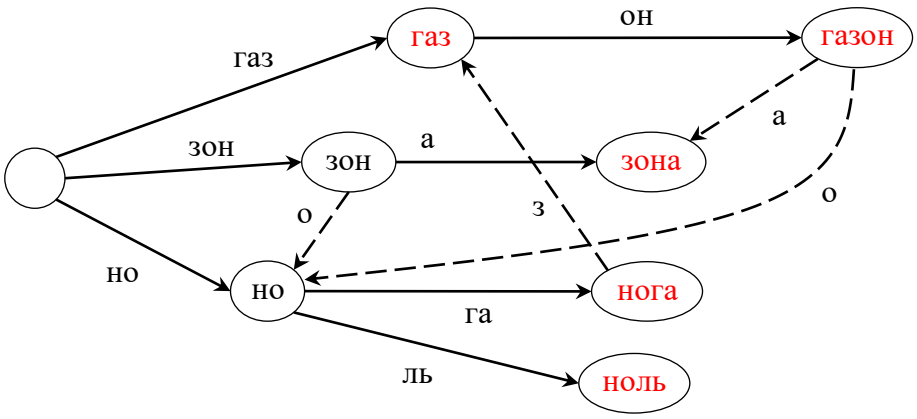



Рис. 18.4. Орграф со сжатыми суффиксными ссылками

Если переходы между вершинами орграфа хранить в виде хеш-таблиц, в которых роль хеша играет код символа, то вычислительная сложность алгоритма Ахо – Корасик равна $O(n)$, $n$ – длина текста.

Если переходы между вершинами орграфа хранить в виде списков ссылок, то вычислительная сложность будет равна $O(n \cdot a)$, $a$ – размер алфавита.

## 18.3 Некоторые задачи обработки текстов

### Редакционное расстояние

*Редакционное расстояние* (расстояние Левенштейна, дистанция редактирования) – метрика, измеряющая разность между двумя строками. Эта разность равна минимальному количеству операций вставки одного символа, удаления одного символа или замены одного символа другим, необходимых для преобразования одной строки в другую. Допускается этим операциям назначать разные стоимости.

*Редакционным предписанием* называется последовательность действий, необходимых для получения из первой строки второй кратчайшим образом.

Найти только расстояние Левенштейна – более простая задача, чем найти редакционное предписание.

*Расстояние Дамерау – Левенштейна* является модификацией редакционного расстояния и отличается от него добавлением операции перестановки. Дамерау показал, что 80 % человеческих ошибок при наборе текстов составляют перестановки соседних символов, пропуск символа, добавление нового символа, и ошибка в символе. Поэтому метрика Даме-

рау – Левенштейна часто используется в редакторских программах для проверки правописания.

**Определение авторства текста**

Методы компьютерного определения авторства художественных текстов имеют вероятностную природу: достаточно субъективны. Для повышения достоверности эти методы используют совместно. Каждый автор имеет определённый стиль изложения, который определяет расстановку знаков пунктуации и частоту их использования. Поэтому для определения авторства кроме лексики используют и пунктуацию.

1. *Пунктуационный портрет* текста получают удалением из текста всех символов, кроме знаков пунктуации

. , ; : … ! ? « » “ ” ‘ ’ ( ) [ ] { } * ~ - # –

Такие портреты выводят в окно текстового редактора малым размером шрифта и визуально сравнивают. Обычно тексты одного автора имеют схожие пунктуационные портреты.

2. *Пунктуационная статистика* – удельное количество знаков пунктуации.

3. *Лексический портрет* текста строится на основе его частотного словаря лексем. Портрет обычно ограничивают частотами первой сотни наиболее употребляемых лексем. Такой портрет можно сравнить с портретом другого текста, чьё авторство известно. Сравнение выполняется программно: путём вычисления редакционного расстояния между портретами, в котором лексемы – это неделимые элементы портретов.

## 18.4 Рекомендации по выбору и использованию структур данных и алгоритмов

К приводимым рекомендациям следует относиться критично, так как реальные задачи содержат букет нюансов, влияющих на выбор подходящих структур данных и алгоритмов их обработки. Идиома «дьявол кроется в деталях» сюда относится в полной мере. Кроме этого аппаратные характеристики компьютеров подвержены прогрессу и непрерывно улучшаются, поэтому тот выбор, который был недопустим вчера, завтра может стать вполне приемлемым.

Какую из коллекций следует выбрать для хранения данных? Это зависит от размера исходных данных и набора операций над ними (рис. 18.5). Набор операций определяется задачей.

Скорость обработки структур данных зависит не только от самой структуры, но и от упорядоченности элементов в ней, от размещения элементов в памяти и/или на внешнем носителе, что влияет на возможность оптимизации доступа к кэш-памяти (табл. 18.1).

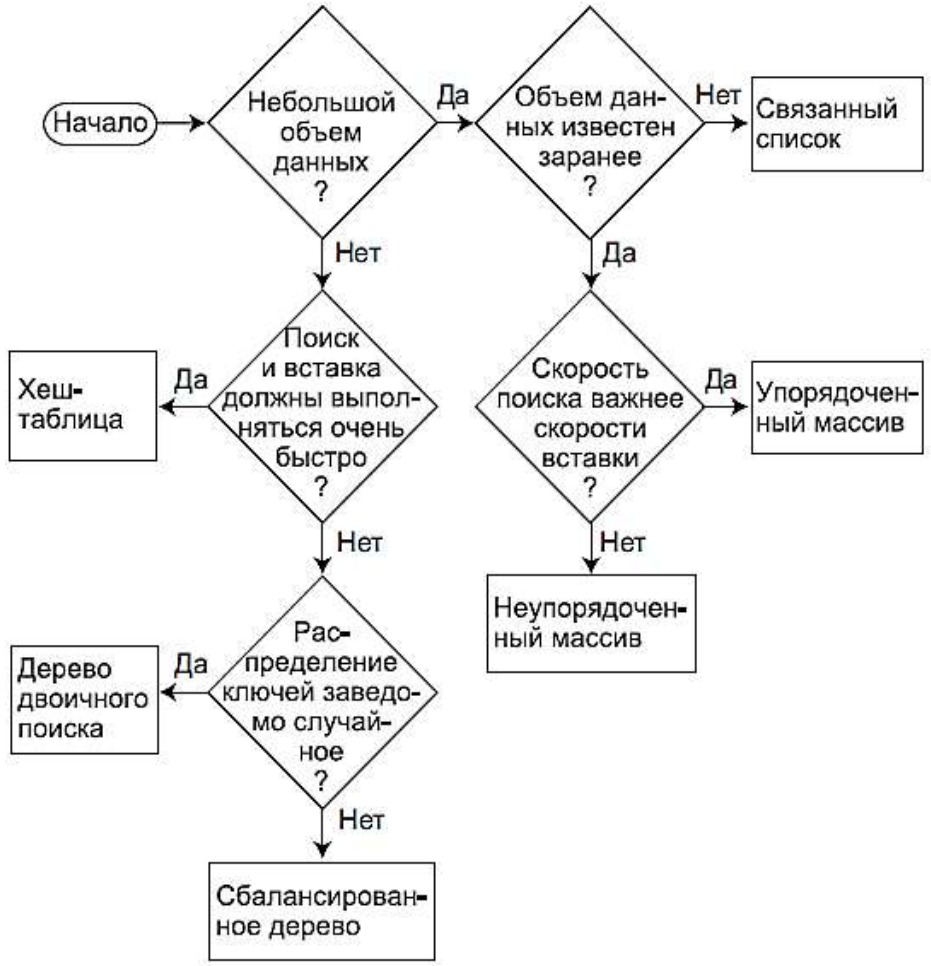



Рис. 18.5. Выбор структуры данных

**Условия выбора статических массивов:**

– объём данных известен заранее и не изменяется в ходе работы;

– требуется быстрый доступ к элементам по чтению/записи;

– не требуются операции вставки/удаления;

– если требуется операция поиска, то следует использовать двоичный поиск на упорядоченном массиве;

– простая программная реализация.

**Условия выбора динамических массивов (несвязных списков):**

– объём данных может изменяться в ходе работы;

– требуется быстрый доступ к элементам по чтению/записи;

– если требуются операции вставки/удаления элементов, то следует использовать неупорядоченный массив;

– если требуется операция поиска, то следует использовать двоичный поиск на упорядоченном массиве;

– программная реализация простая, но несколько сложнее, чем у статических массивов.

Т а б л и ц а 18.1

Вычислительная сложность доступа к структурам данных

| Структура данных | Чтение/ запись | Вставка | Удале-ние | Поиск | Оптимизация доступа к кэш-памяти |
|---|---|---|---|---|---|
| Неупорядоченный статический массив | $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ | есть |
| Упорядоченный статический массив | $O(1)$ | $O(n)$ | $O(n)$ | $O(\log n)$ | есть |
| Неупорядоченный динамический массив (несвязный список) | $O(1)$ | $O(1)$ | $O(n)$ | $O(n)$ | есть |
| Упорядоченный динамический массив (несвязный список) | $O(1)$ | $O(n)$ | $O(n)$ | $O(\log n)$ | есть |
| Неупорядоченный связный список | $O(n)$ | $O(1)$ | $O(n)$ | $O(n)$ | нет |
| Упорядоченный связный список | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | нет |
| Двоичное дерево (средний случай) | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | нет |
| Двоичное дерево (худший случай) | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | нет |
| Балансированное двоичное дерево (худший случай) | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | нет |
| Хеш-таблица | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ | нет |

**Условия выбора связных списков:**

– объём данных может изменяться в ходе работы;

– не требуются или редко требуются операции чтения/записи по индексу и вставки/удаления по индексу;

– часто требуются операции вставки/удаления в начало, в конец или справа/слева от текущего элемента;

– не требуется или редко требуется операция поиска;

– программная реализация сложнее, чем у динамических массивов.

**Условия выбора двоичных деревьев:**

– объём данных может изменяться в ходе работы;

– часто требуются операции чтения/записи, вставки, удаления и поиска;

– операция вставки выполняется на случайных или на неупорядоченных данных;

– если предполагается выполнение вставки упорядоченных данных, то все операции над деревом резко замедляются, так как высота дерева становится не логарифмической, а линейной (см. худший случай в таблице 18.1). В этом случае приемлемый выбор – это дерамида;

– программная реализация сложнее, чем у связных списков.

**Условия выбора балансированных деревьев:**

– объём данных может изменяться в ходе работы;

– часто требуются операции чтения/записи, вставки, удаления и поиска;

– упорядоченность вставляемых данных не влияет на скорость операций чтения/записи, удаления и поиска. Вставка данных замедленна, т.к. периодически будет вызывать балансировку дерева;

– программная реализация сложнее, чем у двоичных деревьев.

**Условия выбора хеш-таблиц:**

– объём данных большой и может изменяться в ходе работы;

– очень часто требуются операции поиска;

– требуются операции чтения/записи, вставки, удаления;

– упорядоченность вставляемых данных не влияет на скорость операций чтения/записи, вставки, удаления и поиска;

– не требуются операции выборки минимального и максимального элементов;

– программная реализация проще, чем у балансированных деревьев.

Дополнительные структуры данных (табл. 18.2), как правило, не имеют самостоятельного значения и применяются при реализации некоторых алгоритмов обработки данных. К таким структурам относят стек, очередь, дек, пирамиду (очередь с приоритетом). Эти структуры реализуются на базе массивов, связных списков и деревьев. Программный интерфейс к ним содержит ограниченный функционал – реализуются только те функции, которые важны для обрабатывающего алгоритма. Реализация на массивах работает быстрее, чем на списках, но требует максимального размера памяти.

**Условия выбора стека на базе статического массива:**

– максимальный объём данных известен заранее и не меняется;

– простая программная реализация.

Т а б л и ц а 18.2

Вычислительная сложность доступа к дополнительным структурам

| Структура данных | Вставка | Удаление | Используемые операции |
|---|---|---|---|
| Стек на базе статического массива | $O(1)$ | $O(1)$ | Извлекает элемент, вставленный последним |
| Стек на базе односвязного списка | $O(1)$ | $O(1)$ | |
| Очередь на базе статического массива | $O(1)$ | $O(1)$ | Извлекает элемент, вставленный первым |
| Очередь на базе односвязного списка | $O(1)$ | $O(1)$ | |
| Дек на базе статического массива | $O(1)$ | $O(1)$ | Извлекает элемент, вставленный первым или последним |
| Дек на базе двусвязного списка | $O(1)$ | $O(1)$ | |
| Очередь с приоритетом на базе упорядоченного статического массива | $O(n)$ | $O(1)$ | Извлекает элемент с наивысшим приоритетом |
| Очередь с приоритетом на базе пирамиды | $O(\log n)$ | $O(\log n)$ | |

**Условия выбора стека на базе односвязного списка:**

– объём данных может изменяться в ходе работы;

– программная реализация сложнее, чем у динамических массивов.

**Условия выбора очереди на базе статического массива:**

– максимальный объём данных известен заранее и не изменяется;

– надо хранить два индекса для вставки и удаления и использовать автоинкрементную индексацию, массив потребует дополнительного программирования циклического перехода индексов вставки и удаления в ноль при достижении конца массива;

– программная реализация сравнима с односвязными списками.

**Условия выбора очереди на базе односвязного списка:**

– объём данных может изменяться в ходе работы;

– программная реализация сложнее, чем у динамических массивов.

**Условия выбора дека на базе статического массива:**

– максимальный объём данных известен заранее и не изменяется;

– надо хранить два индекса для начала и конца дека, массив потребует дополнительного программирования циклического перехода индексов вставки и удаления при достижении конца массива;

– программная реализация сравнима с односвязными списками.

**Условия выбора дека на базе двусвязного списка:**

– объём данных может изменяться в ходе работы;

– программная реализация сложнее, чем у односвязных списков.

**Условия выбора очереди с приоритетом на базе упорядоченного статического массива:**

– максимальный объём данных известен заранее;

– надо хранить индекс последнего элемента очереди;

– относительно простая программная реализация.

**Условия выбора очереди с приоритетом на базе пирамиды:**

– объём данных может изменяться в ходе работы;

– надо хранить два индекса для начала и конца очереди, массив потребует дополнительного программирования циклического перехода индексов вставки и удаления при достижении конца массива;

– программная реализация сложнее, чем у списков.

**Сортировка коллекций**

При выборе алгоритма сортировки коллекции следует учитывать оценку вычислительной сложности алгоритма (табл. 18.3), структуру сортируемых данных (массив или список), тип данных, расположение в памяти: коллекция находится в памяти целиком или её элементы поступают по одному из потока ввода, а также желательно знать сведения об упорядоченности исходной коллекции.

Таблица 18.3

Оценка вычислительной сложности алгоритмов сортировки

| Алгоритм сортировки | Сложность в среднем случае | Сложность в худшем случае |
|---|---|---|
| Быстрая сортировка | $O(N \cdot \log N)$ | $O(N^2)$ |
| Пирамидальная сортировка | $O(N \cdot \log N)$ | $O(N \cdot \log N)$ |
| Сортировка слиянием | $O(N \cdot \log N)$ | $O(N \cdot \log N)$ |
| Сортировка подсчётом | $O(N + K)$ | $O(N + K)$ |
| Поразрядная сортировка | $O(N \cdot K)$ | $O(N \cdot K)$ |
| Блочная сортировка | $O(N)$ | $O(N)$ |

*Быстрая сортировка* в своём классе сложности $O(n \cdot \log n)$ является лучшей в большинстве случаев. Однако при неудачных исходных данных быстрая сортировка может резко замедлиться, перейдя в класс медленных сортировок. Это зависит от выбора опорного элемента. Если выбор удачен, то мы получим время $O(n \cdot \log n)$, иначе – $O(n^2)$. Быстрая сортировка хорошо применима к массивам и хорошо кэшируется. Сортировка происходит на месте хранения массива. Двусвязные списки сортируются медленнее массивов, но всё же достаточно быстро, так как имеют последова-

тельный доступ к элементам с двух сторон списка. Односвязные списки сортируются медленно, так как имеют последовательный доступ только с одной стороны списка.

*Пирамидальная сортировка* имеет гарантированное время сортировки $O(n \cdot \log n)$ и не зависит от исходных данных. Оно эффективно сортирует массивы. На связных списках работает медленно, так как требует доступ к элементам, расположенным в произвольных позициях.

*Сортировка слиянием* несколько быстрей пирамидальной сортировки, имеет гарантированное время сортировки $O(n \cdot \log n)$ и не зависит от исходных данных. Быстро работает как на массивах, так и на любых связных списках и других структурах последовательного доступа. Хорошо распараллеливается.

*Сортировка подсчётом* работает с целочисленными ключами за линейное время. Её выгодно использовать, если диапазон ключей в массиве сравним с числом элементов или несколько больше. Когда диапазон ключей во много раз превосходит число элементов в массиве, то сортировка подсчётом резко теряет свою скорость. Эффективна на связных списках и другим структурах последовательного доступа, хорошо кэшируется.

*Поразрядная сортировка* хорошо применима к массивам, связным спискам и другим структурам последовательного доступа. Имеет линейную сложность в зависимости от разрядности ключа и числа элементов в коллекции, хорошо кэшируется.

*Блочная сортировка* на равномерно распределённых данных имеет линейное время выполнения. На неравномерно распределённых исходных данных, содержащих группы близких по значению элементов, блочная сортировка может деградировать до $O(n^2)$. Хорошо применима к массивам, связным спискам и другим структурам последовательного доступа, хорошо кэшируется.

**Графы**

Графы являются абстрактной структурой данных, которую никакой другой структурой заменить нельзя.

Если граф плотный, то матрица смежности сэкономит память. Если граф разряжённый, то для представления графа лучше использовать списки смежности.

Выбор способа представления графа также зависит от набора операций, которые будут выполняться над графом. Если граф задан матрицей смежности, то время обхода всех вершин, время построения минимального остовного дерева и время поиска минимального пути на графе имеют оценку $O(|V|^2)$. Если же граф задан списками смежности, то обход всех вершин выполняется за время $O(|V| + |E|)$, а время построения минимального остовного дерева и время поиска минимального пути на графе

имеют оценку $O\big((|E| + |V|) \cdot \log|V|\big)$. Зная примерное значение величин $|V|$ и $|E|$, всегда можно выбрать подходящий способ представления графа.

**Задача 1.** Начертить бор для поиска слов {«пар», «ров», «воз», «паровоз», «ровный», «бор», «род», «борода»} и создать бор программно.

**Задача 2.** Разработать программу поиска в текстовом файле слов, хранимых в созданном боре.

**Контрольные вопросы**

1. Что такое бор?
2. Какая абстрактная структура данных легла в основу бора?
3. Какую роль играют в боре терминальные вершины?
4. Что является весом рёбер бора?
5. В чём заключается алгоритм поиска образцов, хранящихся в боре?
6. Перечислите достоинства бора.
7. Для чего служит алгоритм Ахо – Корасик?
8. Как строится орграф из бора для алгоритма Ахо – Корасик?
9. Что такое бамбук в орграфе?
10. Чем заменяются бамбуки в орграфе?
11. Какова вычислительная сложность алгоритма Ахо – Корасик?

## Заключение

Настоящее учебное пособие освещает основные абстрактные структуры данных и способы их представления на языке программирования C# с использованием инструментов объектно-ориентированного программирования, раскрывает способы рекурсивного программирования, излагает алгоритмы обработки коллекций значений и ссылочных типов, а также объясняет асимптотическую оценку вычислительной сложности конкретных алгоритмов и её роль в современной технологии разработки программных средств.

Важной особенностью пособия является детальное описание программ, реализующих описываемые алгоритмы, указание возможных направлений уменьшения сложности программ, способы эффективного использования кэш-памяти, перечисление достоинств и недостатков алгоритмов. Учебный материал пособия завершается подробными рекомендациями по выбору структур данных и алгоритмов их обработки в зависимости от размера коллекции, упорядоченности её элементов и размещения их в памяти и/или на внешнем носителе, а также требуемого набора операций над коллекцией и её элементами.

Вошедший в пособие учебный материал был выбран по критерию гибкости структур данных и эффективности алгоритмов их обработки. К сожалению, в круг излагаемых структур данных не вошли словари, фибоначчиевы, биномиальные и сливаемые кучи, различные виды B-деревьев, а также некоторые интересные алгоритмы сортировки, такие как сортировка Шелла и плавная сортировка. Автором намеренно пропущены все алгоритмы сортировки, имеющие квадратичную оценку сложности: с ними можно ознакомиться самостоятельно.

Область знаний профессии программиста развивается непрерывно и стремительно. Сейчас нельзя останавливаться на полученных в вузе знаниях и умениях. От профессионала требуется постоянно быть в курсе новинок, как в области инструментальных языковых средств, так и в области разработки новых алгоритмов. Для желающих поднять свой уровень и расширить кругозор в этой области автор рекомендует посещать сайты:

http://e-maxx.ru/algo
https://proglib.io/p/awesome-algorithms
https://www.lektorium.tv/medialibrary
http://neerc.ifmo.ru/wiki
https://www.csharp-examples.net/
https://blog.rc21net.ru/коллекции-в-c-sharp/

## Библиографический список

1. Белов В.В., Чистякова В.И. Алгоритмы и структуры данных [Электронный ресурс]: учебник – М: КУРС: ИНФРА-М: 2020. – 240 с. URL: https://znanium.com/bookread2.php?book=1057212 (дата обращения: 6.12.2021)

2. Колдаев В.Д. Структуры и алгоритмы обработки данных [Электронный ресурс]: учеб. пособие – М: РИОР: ИНФРА-М, 2020. – 296 с. URL: https://znanium.com/catalog/document?id=356125 (дата обращения: 6.12.2021)

3. Барский А.Б., Шилов В.В. Теория цифрового компьютера [Электронный ресурс]: учеб. пособие – М: ИНФРА-М, 2018. – 304 с. URL: https://znanium.com/bookread2.php?book=912953 (дата обращения: 6.12.2021)

4. Медведев М.А., Медведев А.Н. Программирование на Си# [Электронный ресурс]: учеб. пособие – 2-е изд., стер. – М: ФЛИНТА: Изд-во Урал. ун-та, 2017. – 64 с. URL: https://znanium.com/bookread2.php?book=948428 (дата обращения: 6.12.2021)

Учебное издание

Марков Виталий Николаевич

# АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Учебное пособие

Редактор Н.А. Колычева

Компьютерная вёрстка В.Н. Марков

| | | |
|---|---|---|
| Подписано в печать 30.12.2021 г. | | Формат 60×84/16 |
| Бумага офсетная | | Офсетная печать |
| Печ. л. | 13,25 | Изд. № 6 |
| Усл. печ. л. | 12,3 | Тираж 70 экз. |
| Уч.-изд. л. | 9,5 | Заказ № |

ФГБОУ ВО «Кубанский государственный технологический университет»
350072, г. Краснодар, ул. Московская, 2, кор. А
Типография ФГБОУ ВО «КубГТУ»: 350058, г. Краснодар, ул. Старокубанская 88/4