МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания

по выполнению лабораторных работ
по дисциплине «Технологии программирования»
Для студентов направления подготовки 09.03.02 Информационные системы и технологии, направленность (профиль) Информационные системы и технологии в бизнесе

(ЭЛЕКТРОННЫЙ ДОКУМЕНТ)

ЧАСТЬ I

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ **5**
1 СТРУКТУРЫ ДАННЫХ **9**
Контрольные вопросы 13
2 ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ **14**
2.1 Общие сведения 14
2.2 Регулярные структуры данных 17
2.3 Комбинированные структуры данных 24
2.4 Структуры данных типа множеств 24
2.5 Списочные структуры данных 32
Контрольные вопросы 41
3 ИЕРАРХИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ **43**
3.1 Общие сведения 43
3.2 Бинарные деревья 46
3.3 Бинарные деревья поиска 53
3.4 Сбалансированные бинарные деревья 60
Контрольные вопросы 68
4 МНОГОСВЯЗНЫЕ СТРУКТУРЫ ДАННЫХ **70**
4.1 Общие сведения 70
4.2 Алгоритмическое представление графа 75
4.3 Поиск в графах 78
4.4 Компоненты связности 82
4.5 Стягивающие деревья 84
4.6 Циклы 89
4.7 Кратчайшие пути 95
Контрольные вопросы 101
5 РЕАЛИЗАЦИЯ СТРУКТУР ДАННЫХ СРЕДСТВАМИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ **103**
5.1 Структуры данных в языке ***Pascal*** 103
5.1.1 Массивы и строки 103
5.1.2 Записи 112
5.1.3 Множества 120

5.1.4 Списки 123
5.1.5 Деревья 125
5.1.6 Графы 127
5.2 Структуры данных в языке ***C++*** 133
5.2.1 Массивы и строки 133
5.2.2 Структуры и объединения 141
5.2.3 Множества 148
5.2.4 Списки 149
5.2.5 Деревья 151
5.2.6 Графы 153
Контрольные вопросы 158
6 ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ 161
6.1 Организация практикума 161
6.2 Работа №1. Обработка массивов 161
6.3 Работа №2. Обработка строк 166
6.4 Работа №3. Обработка данных комбинированного типа 170
6.5 Работа №4. Обработка множеств 174
6.6 Работа №5. Обработка списков 178
6.7 Работа №6. Обработка деревьев 179
6.8 Работа №7. Обработка графов 180
СПИСОК ЛИТЕРАТУРЫ 183
ПРИЛОЖЕНИЕ А. Реализация множественного типа на языке C++ 185
ПРИЛОЖЕНИЕ Б. Массивы 191
ПРИЛОЖЕНИЕ В. Тексты 202
ПРИЛОЖЕНИЕ Г. Геометрия 212
ПРИЛОЖЕНИЕ Д. Множества 221
ПРИЛОЖЕНИЕ Е. Списки 227
ПРИЛОЖЕНИЕ Ж. Деревья 234
ПРИЛОЖЕНИЕ И. Графы 239

ПРЕДИСЛОВИЕ

Основной целью изучения дисциплины «Технологии программирования» является приобретение устойчивых практических навыков программной инженерии и формирование точки зрения на программирование, как на формальный процесс, который можно изучать и совершенствовать.

Для успешного усвоения материала должны быть изучены:

- основные способы кодирования информации;
- основные алгоритмические конструкции;
- основы программирования на одном из языков высокого уровня;
- основы алгебры логики и дискретной математики.

В процессе изучения дисциплины должны решаться следующие задачи:

- развитие логического и алгоритмического мышления;
- выработка умения самостоятельно ставить и решать задачи по созданию программного обеспечения;
- изучение инструментальных средств разработки и отладки программного обеспечения.

Для успешного изучения дисциплины необходимо знать:

- методологические основы проектирования программного обеспечения, жизненный цикл программного изделия;
- принципы структурного и объектно-ориентированного построения программ;
- методики дедуктивного мышления, методы генерации решений и выбора их наилучших вариантов.

Для успешного изучения дисциплины необходимо уметь:

- разрабатывать структурированные алгоритмы в форме процедур разработки функциональных описаний;
- планировать работы по созданию программы;

- разрабатывать программы средней сложности, обеспечивать их самодокументирование, тестирование, отладку.

По итогам изучения дисциплины необходимо овладеть:

- дедуктивным мышлением;
- инструментарием современных сред разработки программных приложений;
- техникой программирования, обеспечивающей необходимую производительность при реализации программных проектов.

Успешное изучения дисциплины обеспечит приобретение следующих компетенций:

- владение базовыми знаниями, необходимыми для решения практических задач;
- умение проводить техническое и рабочее проектирование программного обеспечения;
- умение выбирать исходные данные для тестирования проекта и контролировать их качество;
- умение разрабатывать методические, алгоритмические и программные средства реализации информационных технологий.

Грамотное использование существующих алгоритмов и структур данных в профессиональной практике является одной из основ программирования. Оно повышает качество и увеличивает срок жизни программного обеспечения, а также обеспечивает нужную управляемость процесса проектирования программ. Хороший алгоритм или структура данных позволят в кратчайшее время решить проблему, которая без них решалась бы очень долго.

Базовых алгоритмов, которые применяются практически в каждой программе, немного. Это, прежде всего, поиск и сортировка, и даже они зачастую включены в стандартные библиотеки. Почти все сложные конгломераты данных созданы на основе нескольких фундаментальных структур — массивов, списков, деревьев и т. п. Задача программиста — знать, что имеется у него в наличии, а также понимать, как выбрать то, что ему необходимо. Иначе его время

будет потрачено впустую в попытках плохо сделать то, что уже кем-то было сделано хорошо.

Автор придерживается точки зрения, что программированием нельзя овладеть, прочитав несколько руководств или прослушав курс лекций. Определяющую роль должна играть практическая подготовка, которая основывается на прочных теоретических знаниях. В соответствующем ключе выдержана структура данного пособия. В первой главе рассмотрены общие принципы организации структур данных. Во второй, третьей и четвертой главах рассмотрены различные типы структур с последовательным усложнением принципов их организации (от линейных — к иерархическим и многосвязным). В пятой главе описаны правила реализации основных структур на языках программирования ***Pascal*** и ***C++***. В шестой главе представлено описание практикума по программированию. Предложены также варианты задач для самостоятельного решения, различающиеся уровнем сложности.

Автор не претендует на рассмотрение всех алгоритмов над структурами данных (особенно это касается алгоритмов на графах), предпочитая ограничиться наиболее важными из них.

Для записи алгоритмов использован т. н. «псевдокод», сходный с языком ***Pascal***. В ряде случаев применяются обозначения в виде функций языков программирования (так, $\boldsymbol{Length}(\boldsymbol{X})$ — количество элементов (длина) структуры $\boldsymbol{X}$, $\boldsymbol{Max}(\boldsymbol{X})$ — максимальное значение элемента структуры $\boldsymbol{X}$ и т. п.). В алгоритмах переменные, кроме параметров подпрограмм, считаются глобальными.

Предполагается, что читатель знаком с основами программирования на одном из языков высокого уровня, а материал пособия используется им для повышения уровня подготовки в конкретной области. Все примеры в книге даны на популярных языках ***Pascal*** и ***C++*** (это позволяет сравнить различные программные реализации одних и тех же алгоритмов), однако они легко могут быть «переведены» на любой язык программирования. Разбор этих примеров

должен помочь читателю в написании собственных программ, в освоении некоторых приемов программирования и, наконец, в приобретении собственного стиля программирования. Все фрагменты программ протестированы с использованием компиляторов ***Borland Pascal 7.0*** и ***Borland C++ 3.1***.

Формулировки приведенных в книге задач не ориентированы на конкретную среду разработки программ. Более легкие задания обозначаются символом «↓», более сложные — символом «↑».

Автор выражает глубокую признательность всем тем, кто своей заинтересованностью, поддержкой или советами способствовал изданию данной книги. Автор также благодарен всем студентам, которые на себе испытали достоинства и недостатки предложенной технологии обучения программированию.

1 СТРУКТУРЫ ДАННЫХ

Структурой считается совокупность данных, организованных по заранее установленным правилам. Число компонентов структуры определяет ее **мощность**. *Использование структур как единого целого минимизирует потребность в отслеживании больших наборов данных.*

В процессе разработки программы применяются различные **модели** структур. *Концептуальная модель* на этапе анализа задачи определяет правила формирования абстракций объектов реального мира. *Логическая модель* на этапе проектирования программы определяет правила взаимодействия элементов структуры в соответствии с ограничениями, накладываемыми конкретной системой программирования. *Физическая модель* на этапе выполнения программы определяет способ размещения элементов структуры в машинной памяти.

Любая структура строится на основе некоторого **базового типа** данных. Для него должны оставаться *неизменными*:

- правила *представления* информации (т. е. порядок выделение памяти под данные и интерпретация их двоичного кода);
- множество допустимых *значений*;
- множество допустимых *операций*.

Над структурами данных могут выполняться следующие основные **операции**:

- *создание* — выделение памяти под структуру на этапах компиляции или выполнения программы (противоположная по своему действию операция называется *уничтожением*);
- *выбор* — доступ к данным внутри структуры;
- *модификация* — изменение данных в структуре.

Для каждой структуры должны быть определены:

- **дескриптор** — совокупность правил доступа к структуре и основных ее характеристик;

- **дискриминант** — характеристический признак, позволяющий идентифицировать любой элемент структуры.

Структуры данных бывают статические и динамические. Размеры **статических структур** задаются при проектировании программы и не меняются в процессе ее работы. Память для них выделяется автоматически на этапе компиляции программы или при ее выполнении в момент активизации программных блоков, в которых выполнено описание структур.

Примечание. *Выделение памяти на этапе компиляции является настолько удобным свойством статических структур, что в ряде случаев их используют даже для представления объектов, обладающих изменчивостью. Например, если размер массива неизвестен заранее, для него резервируют максимально возможную область памяти.*

Примечание. *Ряд языков программирования допускают размещение статических структур в памяти на этапе выполнения программы по явному требованию разработчика, но и в этом случае объем выделенной памяти остается неизменным до их уничтожения.*

Элементы статических структур располагаются в памяти естественным образом по принципу *плотной упаковки*, занимая смежные ячейки. Такой подход к представлению данных называется *последовательным распределением памяти*.

Дескриптор статической структуры содержит информацию о ее начальном адресе и размере объекта базового типа. *Дискриминантом* является порядковый номер (индекс) элемента. По этим данным можно получить *прямой доступ* к любому элементу.

Достоинства статических структур:

- компактное размещение в памяти;
- упрощение обработки за счет циклического выполнения над структурой однотипных поэлементных операций;

- высокая скорость прямого доступа к элементам структуры по их индексам (номерам).

Основной *недостаток* статических структур — громоздкость операций по их реорганизации (например, вставки или удаления элементов) из-за их компактного хранения.

Размеры **динамических структур** могут меняться в процессе их обработки. Память для таких структур выделяется на этапе выполнения программы по явному требованию программиста.

Элементы динамических структур размещаются в машинной памяти по произвольным адресам без соблюдения принципа плотной упаковки. Любой элемент состоит из:

- *информационной части*, в которой содержатся данные, для хранения которых создается структура (в общем случае эта часть сама может быть структурой данных);
- *связующей части*, в которой содержатся один или несколько указателей (ссылок) на последующие элементы структуры.

Такой подход к представлению данных называется *связным распределением памяти*.

Дескриптор динамической структуры содержит один или несколько адресов, позволяющих обращаться к ней и выполнять ее просмотр по цепочке указателей. *Дискриминантом* является адрес элемента в памяти. Прямой доступ к произвольному элементу структуры невозможен.

Достоинства динамических структур:

- ограничение размера только доступным объемом машинной памяти;
- обеспечение высокого уровня изменчивости (при изменении последовательности элементов требуется не перемещение данных в памяти, а только коррекция указателей).

Основные *недостатки* динамических структур:

- более высокие требования к квалификации программиста, вынужденного работать с указателями;

- дополнительный расход памяти на хранение связующей части элементов;
- бóльшие (по сравнению с прямым доступом) затраты времени на обращение к элементам.

Примечание. *Последний недостаток является наиболее серьезным, и именно им ограничивается область применения связных структур. Они практически никогда не используются, если данные логически организованы как массивы, адресуемые по номерам элементов, но часто применяются в задачах, где информация представляется в виде таблиц, списков и т. п.*

Важный признак связных структур данных — характер *упорядоченности их элементов*. По этому параметру их делят на линейные, иерархические и многосвязные (см. рисунок 1.1).

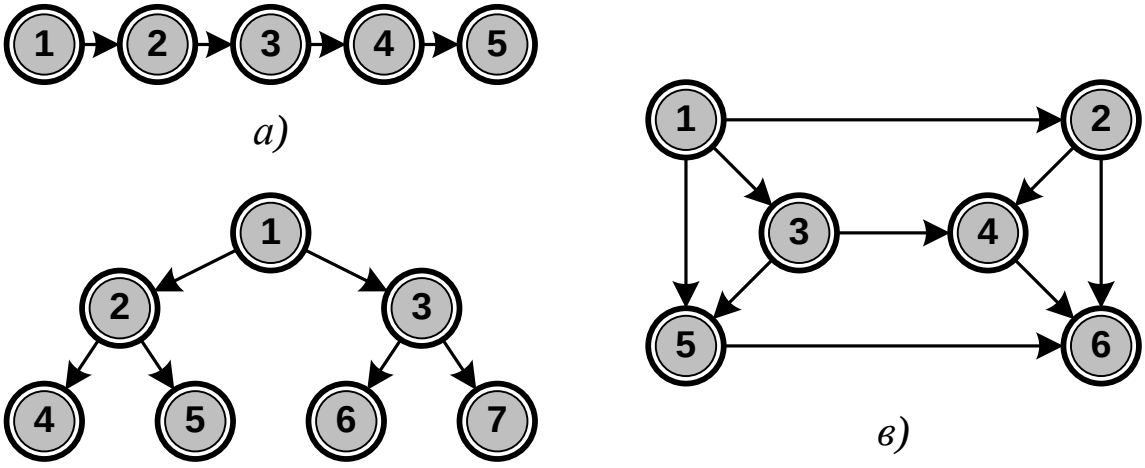


Рисунок 1.1 — Структуры данных: а) линейная; б) иерархическая; в) многосвязная

Линейная структура обладает следующими свойствами:

- все элементы принадлежат одному уровню подчиненности;
- на каждый элемент может быть не более одной ссылки; сам он может иметь связь не более чем с одним элементом;
- каждая связь имеет направление.

Иерархическая структура обладает следующими свойствами:

- все элементы делятся на уровни подчиненности;
- на каждый элемент может быть не более одной ссылки; сам он может иметь связи с произвольным числом элементов нижних уровней;
- каждая связь имеет направление от элементов верхнего уровня к элементам нижних уровней.

Многосвязная структура обладает следующими свойствами:

- все элементы принадлежат одному уровню подчиненности;
- на каждый элемент может быть произвольное число ссылок; сам он может иметь связи с произвольным числом элементов;
- каждая связь может иметь направление и вес.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что понимается под структурой данных? Что определяет мощность структуры? Какие модели структур данных используются при разработке программ?
2. Что считается базовым типом структуры? Какие основные операции выполняются над структурами данных?
3. Какая информация содержится в дескрипторе структуры и дискриминанте ее элемента?
4. Какие структуры данных считаются статическими? Как и когда выполняется их размещение в машинной памяти? Каковы достоинства и недостатки статических структур?
5. Какие структуры данных считаются динамическими? Как и когда выполняется их размещение в машинной памяти? Каковы достоинства и недостатки динамических структур?
6. Каковы основные свойства линейных, иерархических и многосвязных структур?

2 ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ

2.1 Общие сведения

Линейной структурой называется конечное множество, состоящее из $N > 0$ *узлов* (*англ.* ***node***) $node_1, node_2, ..., node_N$, относительное одномерное положение которых определяет следующие структурные свойства:

- узел $node_1$ не имеет предшественников (*от англ.* ***ancestors*** *— предшественники*);
- узел $node_N$ не имеет последователей (*от англ.* ***descendants*** *— последователи, потомки*);
- узлу $node_n$ (где $n \in [2, N-1]$) предшествует единственный узел $node_{n-1}$, и за ним следует единственный узел $node_{n+1}$.

Если структура является статической, то для ее размещения используется последовательное распределение памяти. Любой n-й узел будет состоять только информационной части, хранящей данные $data_n$. Структура считается **регулярной**, если все ее элементы имеют один тип (см. рисунок 2.1-а). Прямой доступ к n-му узлу производится по адресу, который вычисляется по номеру узла, размеру объекта базового типа ***size*** в байтах и начальному адресу структуры $address_1$

$$address_n \leftarrow address_1 + (n-1) \cdot size.$$

Структура считается **комбинированной**, если ее элементы имеют различный тип (см. рисунок 2.1-б). Прямой доступ к n-му узлу производится по адресу, который вычисляется по размерам его предшественников и начальному адресу структуры

$$address_n \leftarrow address_1 + \sum_{i=1}^{n-1} size_i .$$

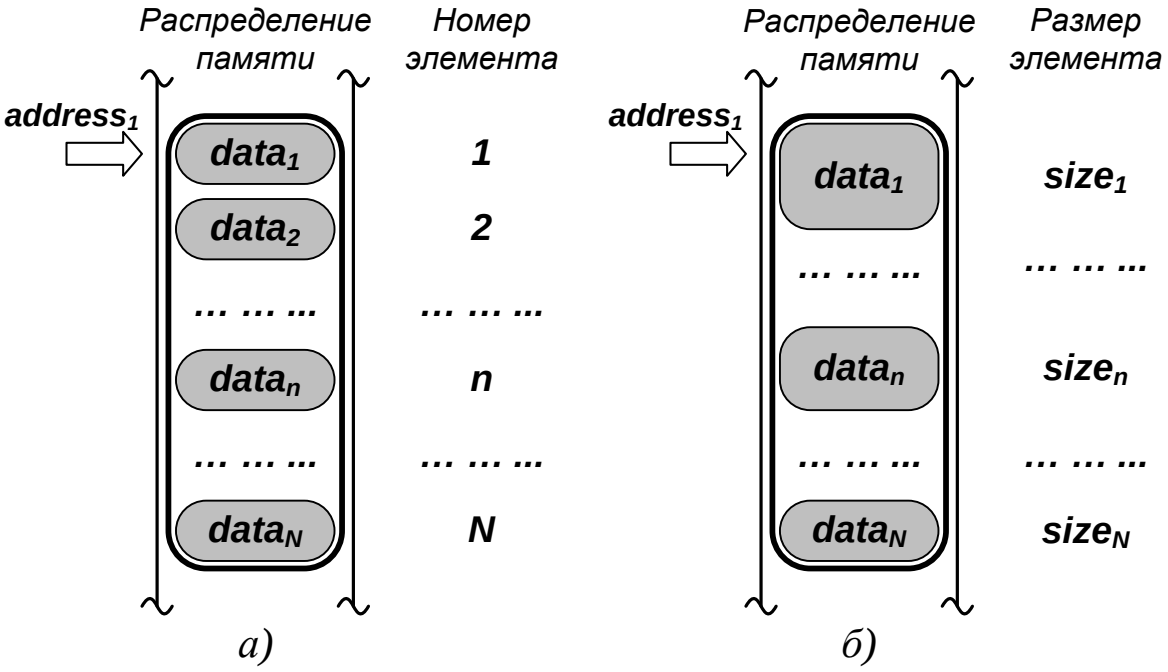


Рисунок 2.1 — Последовательные структуры данных: а) регулярная; б) комбинированная

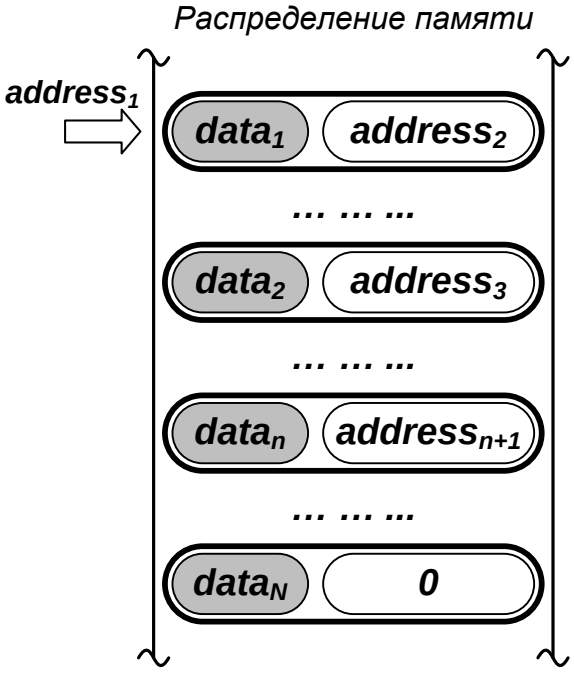


Рисунок 2.2 — Связная структура данных

Если структура является **динамической**, то для ее размещения используется связное распределение памяти. Любой ***n***-й узел состоит из информационной части, хранящей данные $\boldsymbol{data_n}$, и связующей части, содержащей адрес следующего элемента $\boldsymbol{address_{n+1}}$ (см. рисунок 2.2). Доступ к структуре организуется по базовому адресу, который обычно ассоциируется с первым узлом. Доступ к узлам с номерами $\boldsymbol{n = 2,3,..., N}$ по их адресам можно получить в процессе последовательного просмотра структуры. Признаком окончания просмотра является *пустая («нулевая») ссылка* в связующей части последнего узла.

Над линейными структурами выполняются **операции**:

- создание и уничтожение;
- доступ к узлу с анализом или изменением его содержимого;
- включение новых и исключение имеющихся узлов;
- разбиение на несколько подструктур;
- объединение с другими структурами.

Линейные структуры данных, в которых все операции выполняются со стороны первого или последнего узла, имеют специальные названия. Механизм их работы по традиции иллюстрируют предложенными классиком программирования Э. Дейкстрой аналогиями с железнодорожными разъездами (см. рисунок 2.3).

Структура, в которой все операции производятся с одной стороны (с «верхушки»), называется **стеком** (*от англ.* ***stack*** *— стопка*). Стек реализует принцип обслуживания ***LIFO*** (*от англ.* ***last in — first out*** *— «последним вошел — первым вышел»*).

Структура, в которой включение элементов производится с одной стороны («с хвоста»), а исключение (и обычно всякий доступ) — с другой («с головы»), называется **очередью** (*от англ.* ***queue*** *— очередь, хвост*). Очередь реализует принцип обслуживания ***FIFO*** (*от англ.* ***first in — first out*** *— «первым вошел — первым вышел»*).

Структура, в которой включение и исключение элементов (и обычно всякий доступ) производятся с обеих сторон, называется

деком (*от англ. аббревиатуры* ***deque*** *или* ***double-ended queue*** *— двусторонняя очередь*). В ряде случаев используют деки с ограниченным входом (*англ.* ***input restricted deque***) и ограниченным выходом (*англ.* ***output restricted deque***), допускающие только одностороннее включение или исключение элементов.

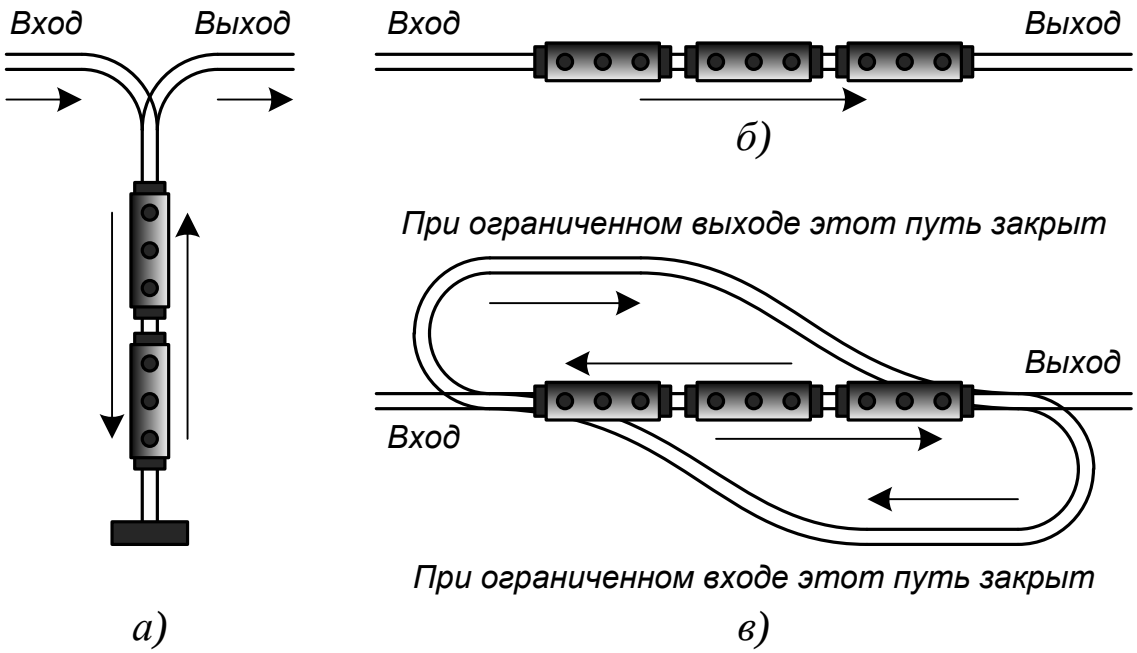


Рисунок 2.3 — Линейные структуры данных с предопределенными правилами доступа: а) стек; б) очередь; в) дек

2.2 Регулярные структуры данных

Статические регулярные структуры или **массивы** используются для представления *логически связанной однородной информации*. Для идентификации их элементов используются *индексы*. Число индексов, определяющее *размерность* структуры, формально не ограничивается. В зависимости от этого числа выделяют *одномерные* и *многомерные* массивы.

Примечание. *На локализацию элемента многомерного массива тратится время, так как необходимо учитывать значение каждого индекса. Поэтому на практике редко применяются массивы*

размерностью более ***3****. Кроме того, с ростом числа измерений резко увеличиваются затраты памяти. Так, в двумерном массиве размерностью* 100×100 *память отводится для* ***10 000*** *элементов, в трехмерном массиве размерностью* $100 \times 100 \times 100$ *— уже для* ***1 000 000****.*

Присваивание массивам различного числа измерений отражает только их логическую организацию. На физическом уровне они всегда приводятся к линейному виду (*векторизуются*). Одномерные массивы размещаются в смежных ячейках памяти (см. рисунок 2.1-а). Элементы многомерных массивов располагаются в памяти с соблюдением лексикографического порядка индексов (по принципу «правый индекс изменяется раньше»). Векторизация позволяет имитировать многомерные массивы одномерными. Это сокращает затраты времени на индексацию.

Доступ к элементам массива может быть *произвольным*. Обращение производится *по индексам* (или *по номерам*). Индекс элемента одномерного массива явно определяет его относительное положение (номер) в структуре. Между индексами элемента многомерного массива и его номером существует взаимосвязь. Номер n элемента с индексами i и j в двумерном массиве A размерностью $M \times N$ определится выражением (см. рисунок 2.4)

$$n \leftarrow j + (i - 1) \cdot N.$$

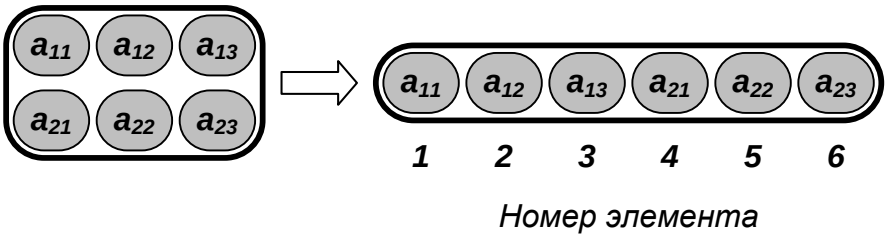


Рисунок 2.4 — Векторизация двумерного массива A

***Пример**: инициализация элементов векторизованной матрицы **A** размерностью $M \times N$ значением **value**.*

```
FOR i ∈ [1, M] DO
  k ← (i − 1) × N
  FOR j ∈ [1, N] DO
    a[j + k] ← value
```

• • •

Номер n элемента с индексами i, j и k в трехмерном массиве A размерностью $L \times M \times N$ определится по формуле

$$n \leftarrow k + (j-1) \cdot N + (i-1) \cdot N \cdot M.$$

Для массивов бóльшей размерности используются аналогичные соотношения.

Если элементы массива закономерно дублируют друг друга, то за счет векторизации можно получить экономию памяти. Например, для *симметричной матрицы* размерностью $N \times N$ достаточно хранить верхнюю или нижнюю треугольную часть (см. рисунок 2.5), что позволит вместо N^2 запоминать только $N \times (N+1)/2$ элементов. В этом случае

$$n \leftarrow j + \frac{(i-1) \cdot i}{2}.$$

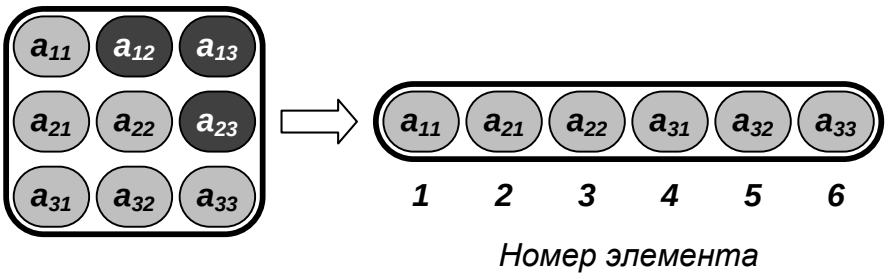


Рисунок 2.5 — Векторизация симметричной матрицы A

***Пример**: инициализация элементов векторизованной симметричной матрицы **A** размерностью $N \times N$ значением **value**.*

```
FOR i ∈ [1, N] DO
  k ← (i − 1) × i / 2
  FOR j ∈ [1, i] DO
    a[j + k] ← value
```

• • •

Для *диагональной матрицы* размерностью $N \times N$ достаточно хранить элементы ее главной диагонали. В этом случае вместо N^2 запоминаются только N элементов.

С помощью массивов можно организовывать структуры со специфическими правилами доступа — стеки и очереди.

На рисунке 2.6 показана реализация **последовательного стека** ***S*** на базе массива из N элементов.

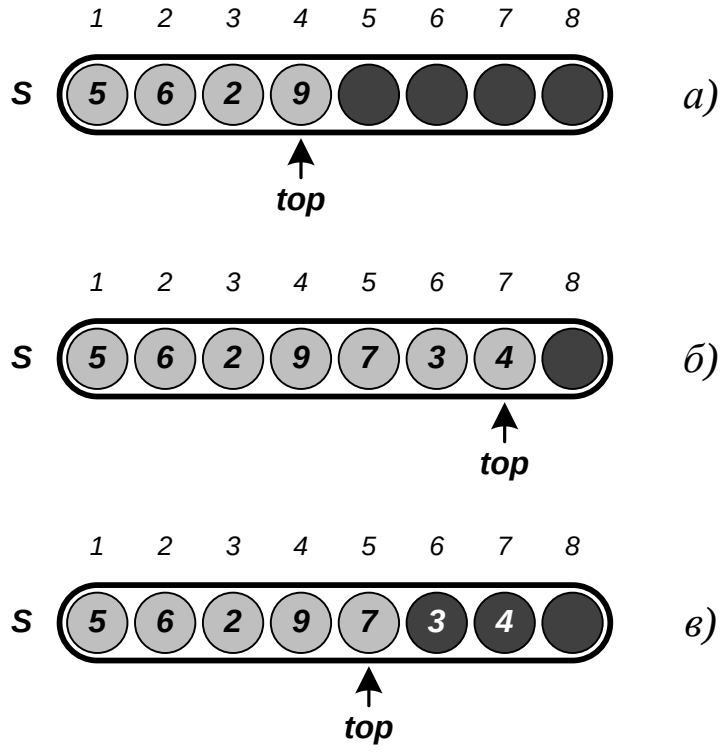


Рисунок 2.6 — Последовательный стек: а) исходное состояние; б) после включения ***7***, ***3*** и ***4***; в) после исключения ***4*** и ***3***

Для работы со стеком необходимо хранить индекс его «верхушки» ***top***. Если $top = 0$, стек пуст. Если $top = N$, стек полон. До начала операций стек считается пустым. *Добавление* и *удаление* данных выполняется с помощью процедур ***Push*** (*от англ.* ***to push*** — *вталкивать*) и ***Pop*** (*от англ.* ***to pop*** — *выскакивать*). Результаты их работы показаны на рисунке 2.6.

```
Push(value)
  IF top = N
    THEN Ошибка: стек заполнен!
    ELSE top ← top + 1
         s[top] ← value

Pop()
  IF top = 0
    THEN Ошибка: стек пуст!
    ELSE top ← top − 1
         RETURN s[top + 1]
```

Величина ***top*** явно определяет *мощность* структуры. *Очистка* стека сводится к обнулению этого индекса.

На рисунке 2.7 показана реализация **последовательной очереди** ***Q*** на базе массива из ***N*** элементов. Подразумевается, что она имеет циклическую организацию, т. е. за элементом с номером ***N*** следует элемент с номером ***1***. Для работы с очередью необходимо хранить индексы ее «головы» ***head*** и «хвоста» ***tail*** — свободной ячейки, в которой будет размещаться новый элемент. Если $head = tail$ при любом их значении, очередь пуста. Если $head = tail + 1$, очередь заполнена.

Примечание. *Последнее условие требует наличие «зазора» из одного свободного элемента между «хвостом» и «головой» очереди. Это позволяет отличать ее от пустой очереди.*

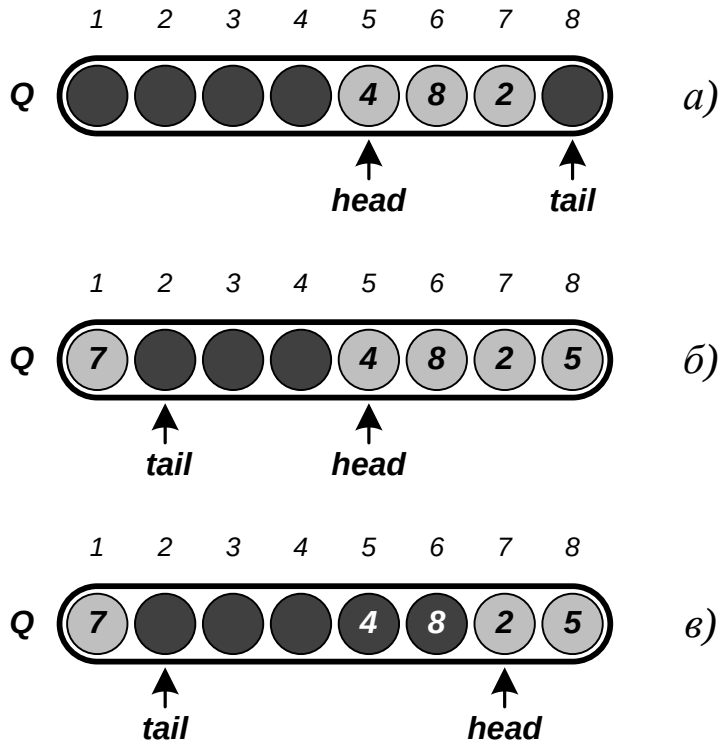


Рисунок 2.7 — Последовательная очередь: а) исходное состояние; б) после включения ***5*** и ***7***; в) после исключения ***4*** и ***8***

До начала операций считается, что очередь пуста, а индексы ***head*** и ***tail*** равны ***1***. *Добавление и удаление* данных выполняется с помощью процедур ***Enqueue*** и ***Dequeue***. Результаты их работы показаны на рисунке 2.7.

```
Enqueue(value)
  IF head = tail + 1
    THEN Ошибка: очередь заполнена!
    ELSE q[tail] ← value
         IF tail = N
           THEN tail ← 1
           ELSE tail ← tail + 1
```

```
Dequeue()
  IF head = tail
    THEN Ошибка: очередь пуста!
    ELSE value ← q[head]
```

```
        IF head = N
          THEN head ← 1
          ELSE head ← head + 1
        RETURN value
```

Мощность очереди определяется разностью индексов ***head*** и ***tail*** (необходимо учитывать цикличность структуры). *Очистка* очереди сводится к приравниванию этих индексов.

Для представления текстовой информации используются одномерные массивы символов — **строки**. Они организуются по особым правилам.

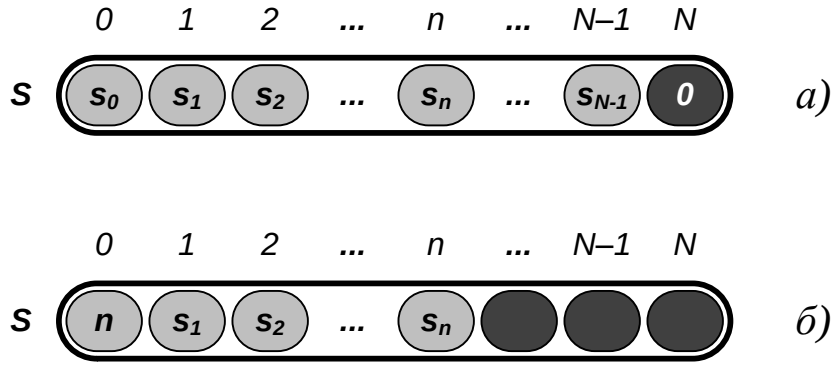


Рисунок 2.8 — Организация строк: а) неопределенной длины; б) фиксированной длины

Если длина строки заранее не задана, то признаком ее окончания считается символ с нулевым кодом (см. рисунок 2.8-а). Размер строки ограничивается только объемом доступной памяти. В некоторых операциях он может меняться.

Ряд языков программирования допускает использование строк, максимальная длина которых N определяется при их создании и остается постоянной в процессе работы. Такая строка состоит из $N+1$ элементов, младший из которых (с нулевым номером) хранит длину фактически занятой части строки n, а остальные (с номерами от 1 до n) — коды символов (см. рисунок 2.8-б). Значение n (но не N) может меняться в процессе работы со строкой.

Массивы, обладающие всеми достоинствами и недостатками статических структур, широко применяются на практике из-за простоты группирования однородной информации. Они также служат базой для построения более сложных типов данных.

2.3 Комбинированные структуры данных

Статические комбинированные структуры или **записи** используются для представления *логически связанной разнородной информации*. Они состоят из фиксированного числа *полей*. Поля могут быть *разного типа* и, соответственно, размера (см. рисунок 2.1-б). Для их идентификации используются *индивидуальные имена*. Порядок доступа к полям может быть произвольным.

Комбинированные структуры, как и массивы, обладают всеми достоинствами и недостатками статических структур. Их использование дает возможность объединять разнородные данные в осмысленные группы, что делает программу более «интеллектуальной» при решении широкого круга задач.

2.4 Структуры данных типа множеств

Статические структуры типа **множеств** (*от англ.* ***set*** *— набор*) используются для представления *возможных сочетаний* элементов базового типа, которые *должны быть отличимыми* друг от друга. Если объект x является элементом множества S, то говорят, что он *принадлежит* S (обозначается $x \in S$). В противном случае он считается *не принадлежащим* S (обозначается $x \notin S$). Множество, не содержащее элементов, называется *пустым* (обозначается «∅»).

Задать множество можно следующими способами:

- перечислением элементов

$$S = \{s_1, s_2, ...s_N\};$$

- характеристическим условием (предикатом) $P(s)$

$$S = \{s | P(s)\};$$

- порождающей процедурой f

$$S = \{s | f : s \to S\}.$$

К любым множествам A и B применимы следующие теоретико-множественные **операции**:

- пересечение

$$A \cap B = \{x | x \in A \,\&\, x \in B\};$$

- объединение

$$A \cup B = \{x | x \in A \vee x \in B\};$$

- разность

$$A \setminus B = \{x | x \in A \,\&\, x \notin B\}.$$

Множества допускают *сравнение*. Говорят, что множество A содержится в B или B *включает* A (обозначается $A \subset B$), если каждый элемент A входит в B. A считается *подмножеством* B, а B — *надмножеством* A. Если $A \subset B$ и $A \neq B$, то A называется *собственным подмножеством* B. Два множества *равны*, если они являются подмножествами друг друга.

Операции над множествами подчиняются **закономерностям**:

- свойство нуля

$$A \cap \varnothing = \varnothing,$$
$$A \cup \varnothing = A;$$

- идемпотентность

$$A \cap A = A,$$
$$A \cup A = A;$$

- коммутативность

$$A \cap B = B \cap A,$$
$$A \cup B = B \cup A;$$

- ассоциативность

$$A \cap (B \cap C) = (A \cap B) \cap C,$$
$$A \cup (B \cup C) = (A \cup B) \cup C;$$

- дистрибутивность

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C),$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C);$$

- поглощение

$$A \cap (A \cup B) = A,$$
$$A \cup (A \cap B) = A;$$

- законы де Моргана

$$A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C),$$
$$A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C).$$

Часто все рассматриваемые множества являются подмножествами некоторого достаточно широкого множества — **универсума** U. Для любых $A \subseteq U$ и $B \subseteq U$ можно определить их *дополнения* $\overline{A} = U \setminus A$ и $\overline{B} = U \setminus B$. Будут справедливы закономерности:

- свойство единицы

$$A \cap U = A,$$
$$A \cup U = U;$$

- свойство дополнения

$$A \cap \overline{A} = \varnothing,$$
$$A \cup \overline{A} = U;$$

- свойство разности

$$A \setminus B = A \cap B;$$

- инволютивность

$$\overline{\overline{A}} = A;$$

- законы де Моргана

$$\overline{A \cap B} = \overline{A} \cup \overline{B},$$
$$\overline{A \cup B} = \overline{A} \cap \overline{B}.$$

Мощность множества S обозначается $|S|$ (например, $|\varnothing| = 0$, однако $|\{\varnothing\}| = 1$). Если $|A| = |B|$, то A и B *равномощны*. Мощность *конечного множества* — натуральное число. Его элементы можно поставить во взаимно однозначное соответствие натуральным числам, поэтому само множество называется *счетным*. Для *бесконечного множества* определение мощности требует аккуратного подхода. Бесконечные множества являются *несчетными*. Для любых двух конечных множеств справедливо соотношение

$$|A \cup B| = |A| + |B| - |A \cap B| \le |A| + |B|.$$

Прямое (декартово) произведение множеств **A** и **B** определится как *множество упорядоченных пар*, образованных их элементами $a \in A$ и $b \in B$ (где $(a,b) \neq (b,a)$)

$$A \times B = \{(a,b) | a \in A \,\&\, b \in B\},$$
$$|A \times B| = |A| \cdot |B|.$$

Пример: *декартово произведение двух множеств*

$$\{a,b\}\times\{a,b,c\}=\{(a,a),(a,b),(a,c),(b,a),(b,b),(b,c)\}.$$

• • •

Можно ввести понятие *декартовой степени* множества

$$A^n = A\times A\times ...\times A,\ \left|A^n\right| = |A|^n.$$

Декартово произведение n множеств определится как

$$A_1\times A_2\times ...\times A_n = \{(a_1,a_2,...a_n)|a_i\in A_i, i\in I\},$$
$$|A_1\times A_2\times ...\times A_n| = |A_1|\cdot|A_2|\cdot ...\cdot|A_n|,$$

где I — некоторое множество целых чисел, используемых для индексации.

Множество всех подмножеств (включая пустое) множества S называется **булеаном** или *множеством-степенью* 2^S. Для конечного S выполняется условие $\left|2^S\right| = 2^{|S|}$.

Пример: *булеан множества двух чисел*

$$2^{\{a,b\}} = \{\varnothing,\{a\},\{b\},\{a,b\}\}.$$

• • •

Семейство $\{S_i\}$ непустых подмножеств множества S образует его **покрытие**, если каждый элемент S принадлежит *хотя бы одному* из подмножеств S_i. Семейство $\{S_i\}$ непустых *непересекающихся* подмножеств S образует его **разбиение**, если каждый элемент S принадлежит *только одному* из подмножеств S_i.

Множественные структуры данных организуются с соблюдением следующих **принципов**:

- базовый тип множества должен быть *дискретным*;

- фиксируется только *нижняя* **lo** *и верхняя* **hi** *границы* диапазона значений элементов, но не их фактическое количество;
- явная упорядоченность элементов *не поддерживается*.

Обычно множества организуют на базе массива ***8***-разрядных целых чисел, размерность ***N*** которого равна

$$N \leftarrow 1 + \left\lfloor \frac{hi - lo}{8} \right\rfloor,$$

где скобки $\lfloor\ \rfloor$ обозначают операцию деления нацело. Значение считается принадлежащим множеству, если соответствующий ему бит установлен в состояние ***1*** (см. рисунок 2.9).

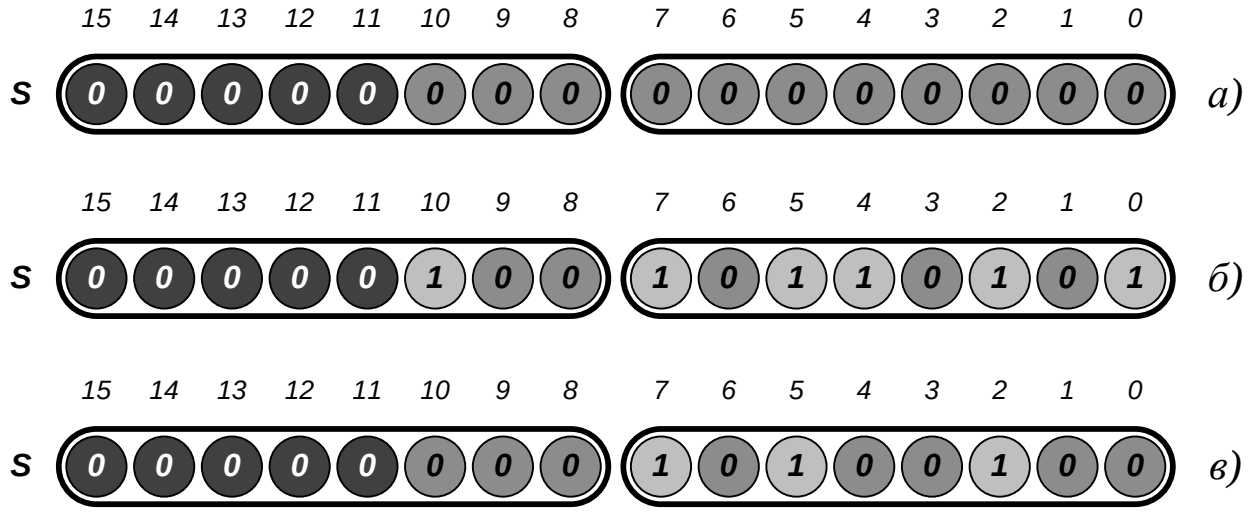


Рисунок 2.9 — Множество целых чисел от ***0*** до ***10*** (с разбивкой на байты): а) исходное состояние (пустое множество); б) после включения ***0, 2, 4, 5, 7, 10***; в) после исключения ***0, 4, 10***

Для *доступа* к отдельному элементу множества $x \in [lo, hi]$ необходимо вычислить:

- номер байта, соответствующего значению ***x***,

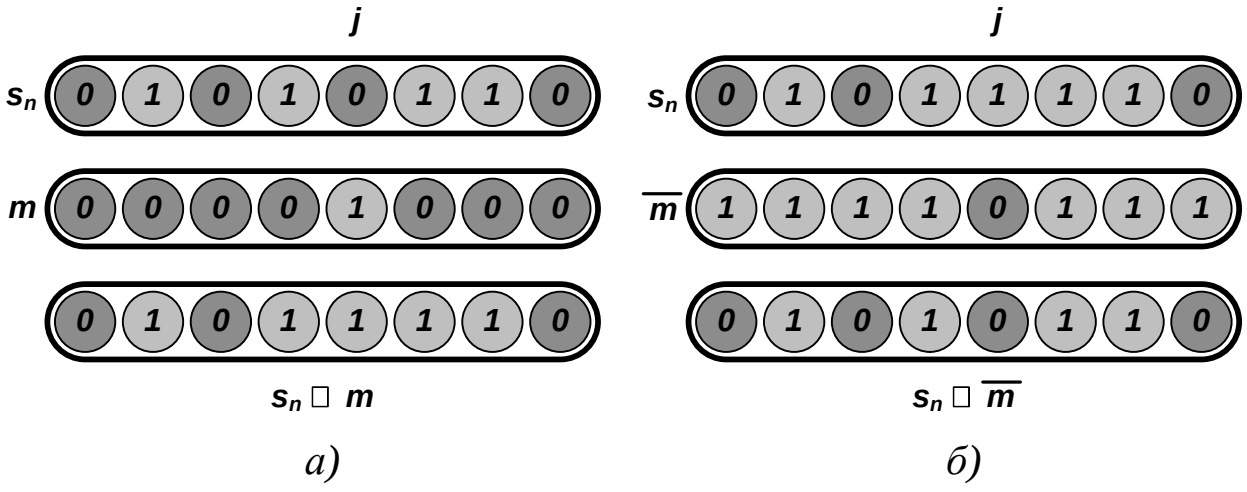
$$n \leftarrow 1 + \left\lfloor \frac{x - lo}{8} \right\rfloor;$$

- номер бита в ***n***-м байте, соответствующего значению ***x***,

$$j \leftarrow 1 + (x - lo) \bmod 8,$$

где ***mod*** — операция вычисления остатка от деления.

Операции *включения* и *исключения* элемента производятся с использованием побитовых дизъюнкции, конъюнкции и инверсии. Вторым операндом служит *маска* (*англ.* ***mask***) — ***8***-разрядное целое число, содержащее ***1*** в разряде ***j***, определяемом значением элемента ***x***, и нули во всех остальных разрядах (см. рисунок 2.10).



а) *б)*

Рисунок 2.10 — Операции над элементом множества: а) включение; б) исключение

В целом над множествами, организованными на базе массивов S_1 и S_2 размерностью N, выполняются следующие операции (см. рисунок 2.11):

- *пересечение*

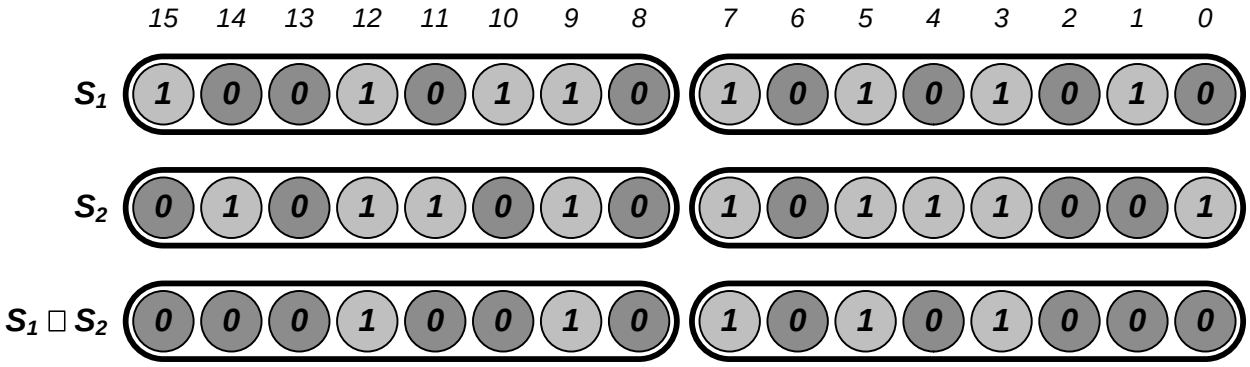
$$S_{1n} \cap S_{2n}, \quad n \in [1, N];$$

- *объединение*

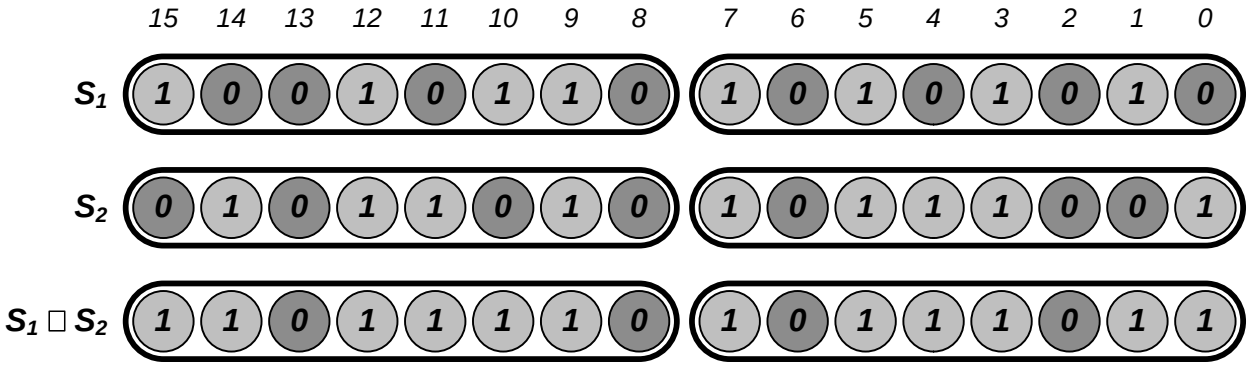
$$S_{1n} \cup S_{2n}, \quad n \in [1, N];$$

- *разность*

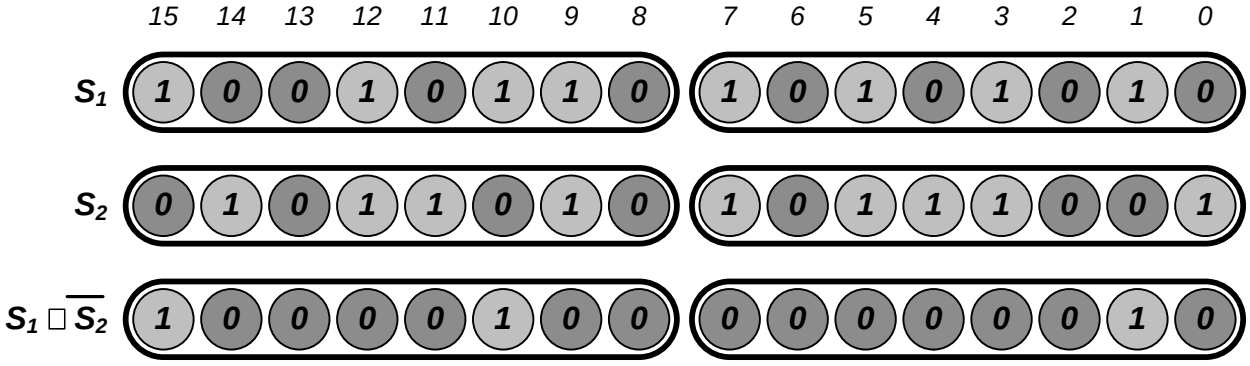
$$S_{1n} \cap \overline{S_{2n}}, \ n \in [1, N].$$



а)



б)



в)

Рисунок 2.11 — Действия над множествами целых чисел от ***0*** до ***15***: а) пересечение; б) объединение; в) разность

Использование множественных типов данных дает возможность придавать программам черты искусственного интеллекта. Однако трудности однозначного определения формальных теоретико-множественных закономерностей, использование которых дает адекватный результат, могут стать причиной возникновения алгоритмической неразрешимости или парадоксов.

2.5 Списочные структуры данных

Динамические списочные структуры или **списки** используются для представления *логически связанной однородной информации*. Для идентификации их элементов используются *адреса* (ссылки). Доступ к отдельным элементам возможен только в процессе последовательного просмотра структуры.

Списки строятся по технологии связного распределения памяти. Каждый элемент списка состоит из информационной части ***Info*** и одной или нескольких связующих частей, по числу которых различают *односвязные* и *многосвязные* списки.

Примечание. *Работа со списками требует выделения и освобождения памяти, отводимой для хранения их элементов. В приведенных алгоритмах эти операции обозначаются* ***Create*** *и* ***Delete***.

Если каждый узел имеет одно связующее поле ***Next*** (с последователем), список называется **односвязным** (*от англ.* ***linked list*** *— связный список*). Он организуется по правилам

$$\begin{cases} Next(Node_n) = Node_{n+1}, n \in [1, N-1], \\ Next(Node_N) = 0. \end{cases}$$

Структура списка показана на рисунке 2.12. Для обращения к нему хранят ссылку на «головной» узел ***Head***. Если в списке нет элементов, то $Head = 0$.

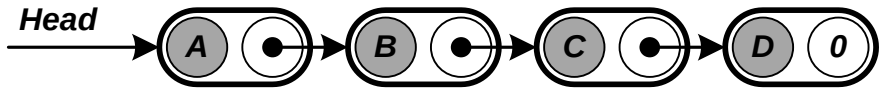


Рисунок 2.12 — Односвязный список

Создание нового узла списка, содержащего в поле ***Info*** значение ***value*** и связанного с узлом ***NextNode***, и возврат ссылки на него выполняет процедура ***CreateNode***.

```
CreateNode(value, NextNode)
    Create(New)
    Info(New) ← value
    Next(New) ← NextNode
    RETURN New
```

До начала работы список считается пустым (***Head = 0***). Включение в него новых элементов (его *расширение*) производится «от конца — к началу», а исключение элементов (или *сокращение*) — «от начала — к концу» (см. рисунок 2.13).

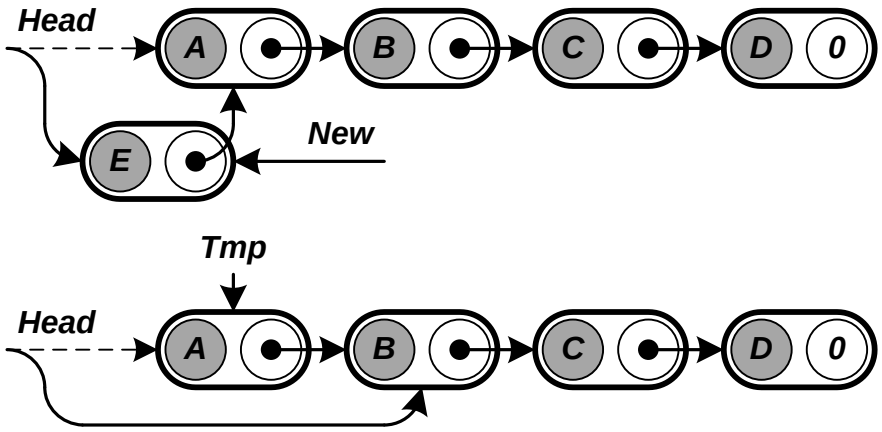


Рисунок 2.13 — Расширение и сокращение односвязного списка

Процедура ***Add*** добавляет новое значение ***value*** в начало списка. Процедура ***Remove*** исключает из списка первый элемент (указатель ***Tmp*** необходим для корректного освобождения памяти).

```
Add(value)
  Head ← CreateNode(value, Head)

Remove()
  IF Head ≠ 0
    THEN Tmp ← Head
         Head ← Next(Head)
         Delete(Tmp)
```

Вставка в список значения ***value*** производится *после* выбранного по какому-либо принципу узла ***Current***, что определяется направленностью связей. По этой же причине при *удалении* любого узла ***Current***, кроме первого, необходимо хранить адрес его предшественника ***Prev*** (см. рисунок 2.14).

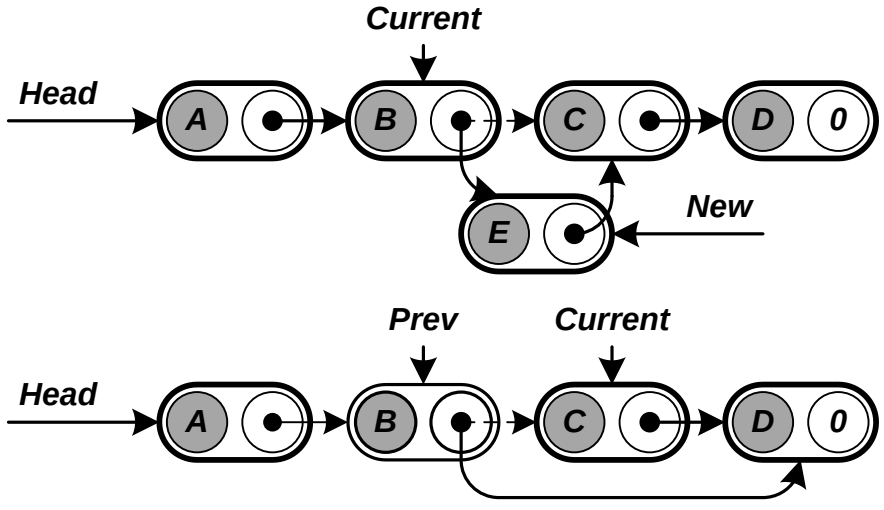


Рисунок 2.14 — Вставка и удаление элемента односвязного списка

Операции вставки и удаления элемента для односвязного списка выполняют процедуры ***InsertAt*** и ***RemoveAt***.

```
InsertAt(value, Current)
  IF Current = 0
    THEN Add(value)
    ELSE Next(Current) ← CreateNode(value, Next(Current))

RemoveAt(Prev, Current)
  IF Current ≠ 0
    THEN IF Prev = 0
           THEN Remove()
           ELSE Next(Prev) ← Next(Current)
                Delete(Current)
```

Действия над отдельными элементами списка выполняются в процессе его последовательного *просмотра*. Процедура ***ListPass*** решает эту задачу итерационно.

```
ListPass()
  Current ← Head
  WHILE Current ≠ 0
    DO Выполнение действия над Current.
       Current ← Next(Current)
```

Рекурсивная процедура ***RecursiveListPass***, вызываемая с параметром ***Head***, решит эту же задачу.

```
RecursiveListPass(Current)
  IF Current ≠ 0
    Выполнение действия над Current.
    ListPass(Next(Current))
```

Пример. *Поиск в списке значения* **value** *выполняет процедура* ***ListSearch***. *Она возвращает ссылку на первый элемент* **Current**, *у*

которого $\boldsymbol{Info(Current) = value}$*, или пустую ссылку, если значение* ***value*** *в списке отсутствует.*

```
ListSearch(value)
  Current ← Head
  WHILE Current ≠ 0 AND Info(Current) ≠ value
    DO Current ← Next(Current)
  RETURN Current
```

• • •

Очистку списка выполняет процедура ***ListClear***.

```
ListClear()
  WHILE Head ≠ 0
    DO Remove()
```

Изменив правила обращения, односвязные списки можно преобразовать в связные стеки или очереди.

У связного стека хранится ссылка не его «верхушку» ***Top*** (у пустого стека $\boldsymbol{Top = 0}$). Операции *записи* и *чтения* для него выполняют определенные выше процедуры ***Add*** и ***Remove*** (вместо ***Head*** используется указатель ***Top***).

У связной очереди хранятся ссылки не ее «голову» ***Head*** и «хвост» ***Tail*** (см. рисунок 2.15; у пустой очереди $\boldsymbol{Head = Tail = 0}$).

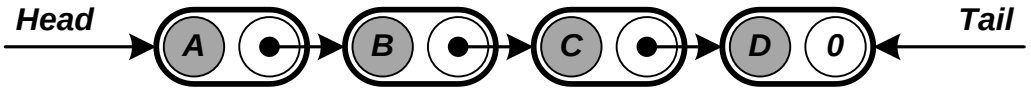


Рисунок 2.15 — Связная очередь

Операции *записи* и *чтения* для связной очереди выполняют процедуры ***Enqueue*** и ***Dequeue***.

```
Enqueue(value)
  New ← CreateNode(value, 0)
  IF Head = 0
    THEN Head ← New
    ELSE Next(Tail) ← New
  Tail ← New

Dequeue()
  Remove()
  IF Head = 0
    THEN Tail ← 0
```

Если содержимым связующего поля последнего элемента является адрес первого узла, то образуется **циклический список** (*англ.* ***circular list***). Правила его организации следующие

$$\begin{cases} Next(Node_n) = Node_{n+1}, n \in [1, N-1], \\ Next(Node_N) = Node_1. \end{cases}$$

Структура такого списка показана на рисунке 2.16. Доступ к нему возможен из любой точки, поэтому выбор начального узла может быть произвольным. Удобным считается хранить ссылку на «хвостовой» узел ***Tail***, так как $Next(Tail) = Head$.

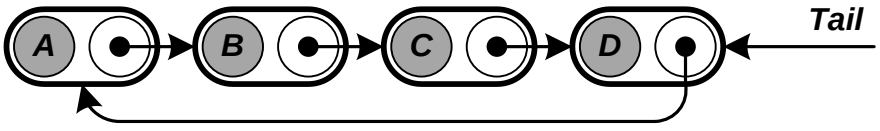


Рисунок 2.16 — Циклический односвязный список

Обработка циклических списков проводится по общим правилам (вместо пустой ссылки используется указатель ***Tail***).

Если каждый узел имеет два связующих поля — ***Prev*** (с предшественником) и ***Next*** (с последователем), список называется **двух-**

связным (*англ.* ***doubly linked list***). Правила его организации следующие

$$\begin{cases} Prev(Node_1) = 0, \\ Prev(Node_n) = Node_{n-1}, n \in [2, N], \\ Next(Node_n) = Node_{n+1}, n \in [1, N-1], \\ Next(Node_N) = 0. \end{cases}$$

Структура двухсвязного списка показана на рисунке 2.17. Для обращения к нему хранят адреса первого и последнего элементов ***Head*** и ***Tail***. Если список пуст, то ***Head*** и ***Tail*** — пустые ссылки.

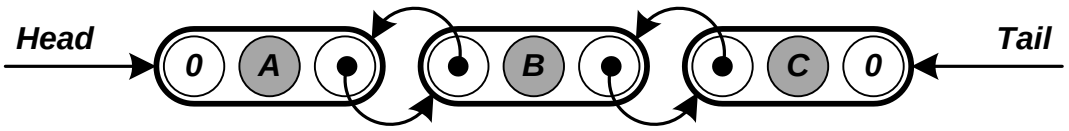


Рисунок 2.17 — Двухсвязный список

Двухсвязные списки обрабатываются по тем же правилам, что и односвязные. Операции над ними более громоздкие, однако решение многих задач упрощается (например, без проблем реализуется проход по структуре в обоих направлениях). В качестве примера на рисунке 2.18 показано выполнение *вставки* в список узла ***New*** (выполняется процедурой ***Insert***) и *удаления* из него узла ***Current*** (выполняется процедурой ***RemoveAt***).

Примечание. *Случай, когда список пустой, предлагается рассмотреть самостоятельно.*

```
Insert(New, Current)
  IF Current = 0
    THEN Next(New) ← Head
         Head ← New
         RETURN
```

```
IF Next(Current) = 0
  THEN Next(Current) ← New
       Tail ← New
       RETURN
Next(New) ← Next(Current)
Prev(New) ← Current
Next(Current) ← New
Prev(Next(Current)) ← New
```

```
RemoveAt(Current)
IF Prev(Current) = 0
  THEN Head ← Next(Current)
       RETURN
IF Next(Current) = 0
  THEN Tail ← Prev(Current)
       RETURN
Next(Prev(Current)) ← Next(Current)
Prev(Next(Current)) ← Prev(Current)
Delete(Current)
```

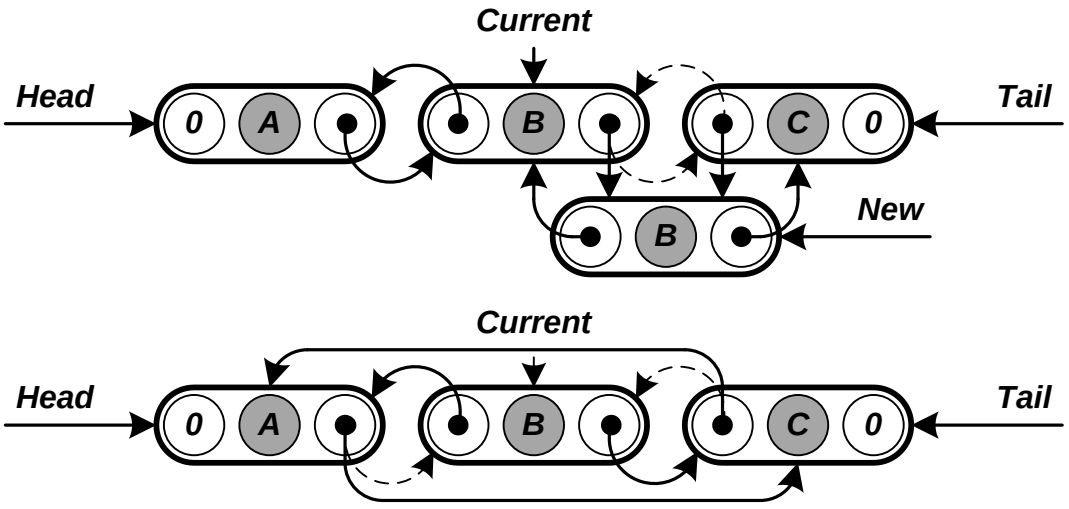


Рисунок 2.18 — Вставка и удаление элемента двухсвязного списка

Примечание. *Выполнение остальных операций над двухсвязными списками (создание, расширение, сокращение, просмотр и т. п.) предлагается рассмотреть самостоятельно.*

Двухсвязный список может быть дополнен фиктивным элементом — *ограничителем* (*англ.* ***sentinel*** *— дозорный, наблюдатель*) ***Nil*** и циклическими связями (см. рисунок 2.19).

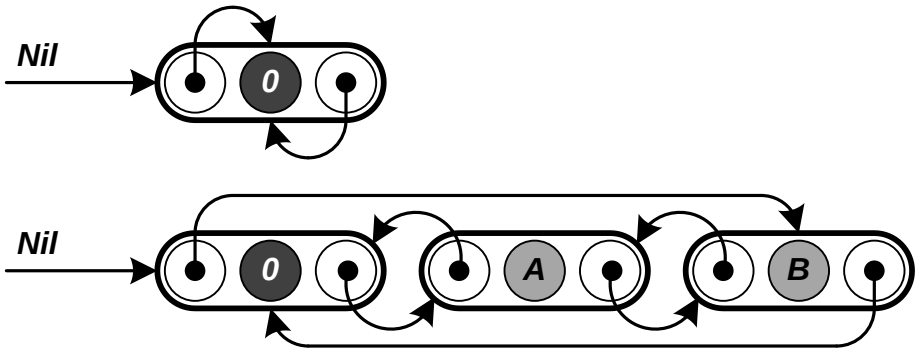


Рисунок 2.19 — Циклический двухсвязный список с ограничителем

При обработке таких списков вместо пустой ссылки используется указатель ***Nil***, вместо ***Head*** используется ***Next(Nil)***, а вместо ***Tail*** — ***Prev(Nil)***. Список с ограничителем *не может быть пустым*, поэтому *учет граничных условий упрощается* (см. приведенную ниже реализацию процедур ***Insert*** и ***RemoveAt***).

```
Insert(New, Current)
    Next(New) ← Next(Current)
    Prev(New) ← Current
    Next(Current) ← New
    Prev(Next(Current)) ← New

RemoveAt(Current)
    Next(Prev(Current)) ← Next(Current)
```

```
Prev(Next(Current)) ← Prev(Current)
Delete(Current)
```

Списки наравне с массивами широко применяются на практике группирования однотипных данных. Они могут также служить базой для построения более сложных связных типов данных.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие структуры данных считаются линейными? Какие основные операции выполняются над ними? Какие существуют линейные структуры с особыми правилами доступа?
2. По каким принципам строятся линейные статические структуры? Что входит в состав их элементов? В чем различаются регулярные и комбинированные структуры?
3. По каким принципам строятся линейные динамические структуры? Что входит в состав их элементов?
4. Каково назначение массивов? По каким принципам они строятся? В чем их достоинства и недостатки?
5. Как массивы размещаются в памяти? Что понимается под векторизацией массивов? Как производится доступ к их отдельному элементу?
6. Как организуются и по каким правилам обрабатываются последовательные стеки и очереди?
7. В чем особенности строковых структур?
8. Каково назначение комбинированных структур? По каким принципам они строятся? В чем их достоинства и недостатки?
9. Как комбинированные структуры размещаются в памяти? Как производится доступ к их отдельному элементу?
10. Каково назначение множеств? По каким принципам они строятся? В чем их достоинства и недостатки? Какие существуют способы задания множеств? Каковы основные закономерности теории множеств?

11. По каким принципам на базе одномерного массива строится множество? Каков порядок выполнения операций включения и исключения его элементов?

12. Какие действия выполняются над множествами как структурами данных? Каков порядок выполнения пересечения, объединения и вычитания множеств?

13. Каково назначение списков? По каким принципам они строятся? В чем их достоинства и недостатки?

14. Как списки размещаются в памяти? Как производится доступ к их отдельному элементу?

15. Как организуются и по каким правилам обрабатываются односвязные списки?

16. Как организуются и по каким правилам обрабатываются связные стеки и очереди?

17. Как организуются и по каким правилам обрабатываются односвязные циклические списки?

18. Как организуются и по каким правилам обрабатываются двухсвязные списки?

19. В чем особенности двухсвязного списка с ограничителем? В чем его преимущества?

3 ИЕРАРХИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

3.1 Общие сведения

Иерархической структурой или **деревом** (*англ.* ***tree***) называется нелинейная структура данных, представляющая собой некоторое множество T из $N > 0$ узлов (или *вершин*), относительное пространственное положение которых определяет следующие структурные свойства:

- специально выделяется одна вершина — *корень* дерева (*англ.* ***root***);
- все вершины, кроме корня, включаются в $m \geq 0$ попарно непересекающихся подмножеств $T_1, ..., T_m$, которые называются *ветвями* или *поддеревьями* (*англ.* ***subtree***). *Любая ветвь обладает свойствами дерева* — каждый ее узел может быть корнем некоторого поддерева.

Вершины и ветви дерева располагаются по уровням следующим образом (см. рисунок 3.1):

- корень имеет уровень 0, все остальные вершины имеют уровень на единицу больше уровня содержащей их ветви;
- к любой вершине уровня $n > 0$ ведет линия связи *только от одной вершины* уровня $n - 1$.

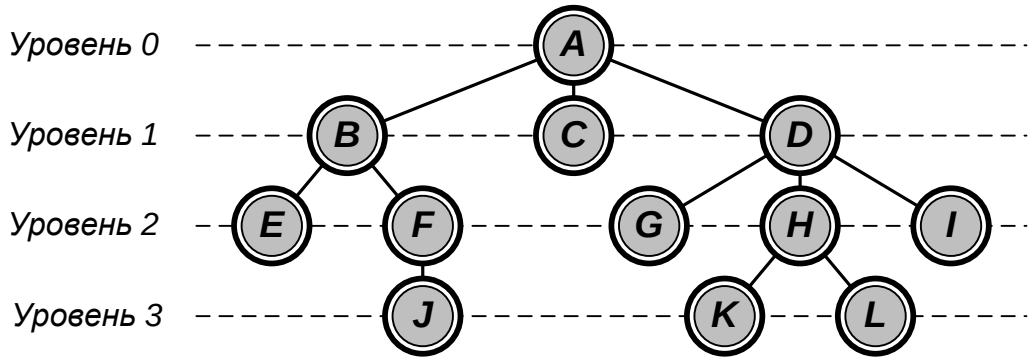


Рисунок 3.1 — Дерево с выделенным корнем в вершине A

В таблице 3.1 показано распределение по уровням вершин и ветвей дерева, изображенного на рисунке 3.1. Такое представление позволяет однозначно определить структуру дерева.

Таблица 3.1 — Распределение по уровням вершин и ветвей дерева с рисунка 3.1

Уровень	***Вершины***	***Ветви***
0	***A***	$\{B, E, F, J\}, \{C\}, \{D, G, H, I, K, L\}$
1	***B, C, D***	$\{E\}, \{F, J\}, \{G\}, \{H, K, L\}, \{I\}$
2	***E, F, G, H, I***	$\{J\}, \{K\}, \{L\}$
3	***J, K, L***	—

Число поддеревьев любого узла называют его *степенью* (*англ.* ***degree***). Узел с нулевой степенью называют *концевым* или внешним (*англ.* ***external node***), а также *листом* (*англ.* ***leaf***). Узел с ненулевой степенью называют *узлом ветвления* или внутренним (*англ.* ***internal node***). Максимальное значение степени узлов называют *степенью* (*порядком*) дерева.

Пусть v — произвольная вершина дерева с выделенным корнем ***root***. Существует *единственный путь* $root \to v$. Все вершины на этом пути считаются предшественниками v, а v считается их последователем. Если (u, v) — последнее ребро на пути $root \to v$, то u считается *«родителем»* v (*англ.* ***parent***; v при этом обозначается термином ***child*** — *«ребенок»*). Единственной вершиной, не имеющей «родителя», является корень.

Длина пути $root \to v$ определяет *глубину* (*англ.* ***depth***) или *уровень* вершины v. Максимальная глубина вершин определяет *ранг* (*англ.* ***rank***) дерева.

Если имеет значение относительный порядок ветвей, то дерево является *упорядоченным* (*англ.* ***ordered tree***). Если имеет значение только распределение вершин по ветвям и уровням, дерево счита-

ется *ориентированным*. Любые два дерева на рисунке 3.2 одинаковы как ориентированные, но различны как упорядоченные.

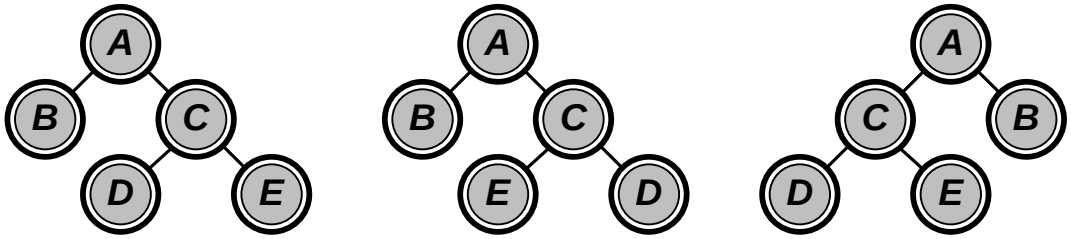


Рисунок 3.2 — Сравнение ориентированных и упорядоченных деревьев

Множество, состоящее из некоторого числа непересекающихся деревьев, называется **лесом** (*англ.* ***forest***). Можно считать, что лесом становится любое дерево после исключения из него корня.

Над деревьями выполняются следующие **операции**:

- создание и уничтожение;
- доступ к вершине с анализом или изменением содержимого;
- включение новых и исключение имеющихся вершин;
- упорядочивание вершин по некоторому признаку;
- разбиение на ветви;
- объединение с другими деревьями.

Все операции над деревьями выполняются в процессе их упорядоченного обхода — **поиска**, при котором каждая вершина посещается произвольное число раз, но обработка относящейся к вершине информации выполняется *только один раз*. Если в процессе этой обработки структура дерева не меняется, то наиболее полезными оказываются следующие способы поиска.

Исчерпывающий поиск в глубину (обход в прямом порядке) определяет следующую процедуру:

- посещается и обрабатывается вершина ***u*** (в общем случае произвольная), которая рассматривается как корень некоторого поддерева;

- если u не является листом, то выполняется *рекурсивное обращение* к процедуре поиска для каждого ее потомка v_j, причем v_j также рассматривается как корень некоторого поддерева; обход всех поддеревьев с корнями v_j выполняется с учетом их упорядоченности, как потомков u.

Обход начинается с корня. Для дерева, показанного на рисунке 3.1, вершины будут проходиться в следующем порядке

$$A \to B \to E \to F \to J \to C \to D \to G \to H \to K \to L \to I.$$

Исчерпывающий поиск в ширину (обход в горизонтальном порядке) определяет процедуру посещения вершин по уровням сверху вниз, начиная с корня. На каждом уровне вершины посещаются с учетом их упорядоченности, как потомков вышележащих элементов (слева направо). Для дерева, показанного на рисунке 3.1, вершины будут проходиться в следующем порядке

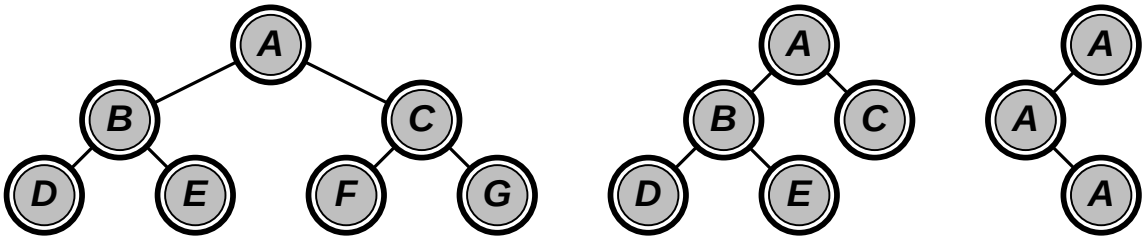
$$A \to B \to C \to D \to E \to F \to G \to H \to I \to J \to K \to L.$$

3.2 Бинарные деревья

Бинарное (или двоичное) дерево (*англ.* ***binary tree***) рекурсивно определяется как конечное множество вершин, которое либо пусто (не содержит вершин), либо разбито на три непересекающихся подмножества — корневую вершину ***Root***, левое и правое бинарные поддеревья (*англ.* ***left subtree*** *и* ***right subtree***). Если узел бинарного дерева содержит обе ветви, он считается *полным*, если только одну (левую или правую) — *неполным*. Если узел вообще не содержит ветвей, его считают *концевым*.

На любом уровне n бинарного дерева может содержаться от 1 до 2^n вершин. Дерево высоты N, имеющее на N-м уровне 2^N вершин, называется *полным* (*англ.* ***full binary tree***; см. рисунок 3.3-а). Дерево высоты N, у которого на уровне $N-1$ имеется полный

набор вершин, и все листы уровня ***N*** расположены слева, называется *завершенным* (*англ.* ***complete binary tree***; см. рисунок 3.3-б). Дерево высоты ***N***, имеющее по одной вершине на каждом уровне, называется *вырожденным* (*англ.* ***degenerate binary tree***; см. рисунок 3.3-в). Чем плотнее структура дерева, тем эффективнее его использование для хранения данных.



а) *б)* *в)*

Рисунок 3.3 — Бинарные деревья: а) полное; б) завершенное; в) вырожденное

Бинарные деревья считаются *подобными*, если существует взаимно однозначное соответствие между их вершинами. Если у подобных деревьев соответствующие вершины содержат одинаковую информацию, они считаются *эквивалентными*.

Бинарные деревья строятся по технологии связного распределения памяти. Каждый их элемент состоит из информационной части ***Info*** и связующих частей ***Left*** и ***Right***, содержащих адреса его левого и правого потомков.

Примечание. *Работа с деревьями требует выделения и освобождения памяти, отводимой для хранения их вершин. В алгоритмах эти операции обозначаются* ***Create*** *и* ***Delete***.

Структура бинарного дерева показана на рисунке 3.4. Для обращения к нему хранят адрес корневой вершины ***Root***. Если дерево не содержит вершин, то $Root = 0$.

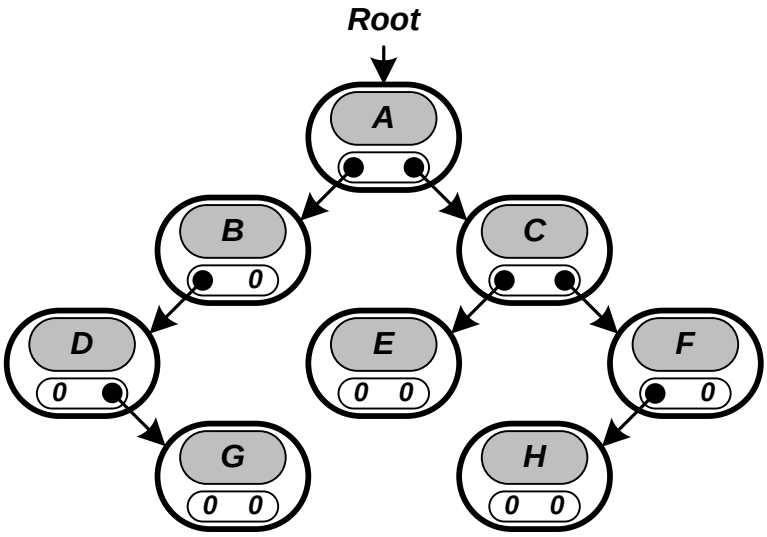
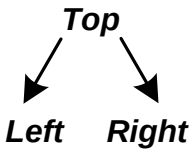


Рисунок 3.4 — Бинарное дерево

Обход бинарных деревьев производится с соблюдением правил, принятых для деревьев с произвольным ветвлением. Для элементарного полного бинарного дерева можно определить три варианта его обхода (поиска по дереву):



- прямой порядок: ***Top → Left → Right;***
- внутренний порядок: ***Left → Top → Right;***
- обратный порядок: ***Left → Right → Top.***

Правила обхода рекурсивно распространяются на бинарные деревья любой сложности. Для дерева, показанного на рисунке 3.4, существуют следующие способы прохождения вершин:

- обход в прямом порядке

$$A \rightarrow B \rightarrow D \rightarrow G \rightarrow C \rightarrow E \rightarrow F \rightarrow H;$$

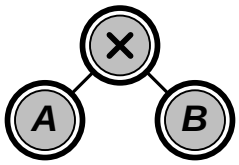
- обход во внутреннем порядке

$$D \rightarrow G \rightarrow B \rightarrow A \rightarrow E \rightarrow C \rightarrow H \rightarrow F;$$

- обход в обратном порядке

$$G \rightarrow D \rightarrow B \rightarrow E \rightarrow H \rightarrow F \rightarrow C \rightarrow A.$$

Примечание. *Бинарные деревья используют для отображения структуры арифметических выражений. Узлам ветвления ставят в соответствие знаки бинарных операций, а концевым узлам — операнды. Трем способам обхода дерева соответствуют три формы записи арифметических выражений (термин «польская запись» предложен польским математиком Я. Лукасевичем):*

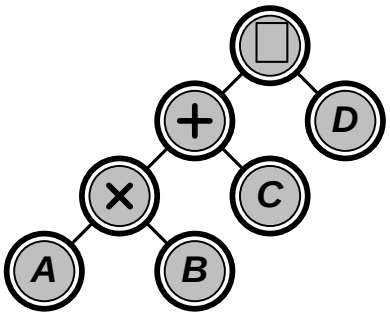


- *префиксная (прямая польская):* $\times\, A\, B;$
- *инфиксная:* $A \times B;$
- *постфиксная (обратная польская):* $A\, B \times .$

Например, выражение

$$(A \times B + C)/D$$

с учетом приоритета операций должно представляться в виде следующего бинарного дерева



Можно выделить три формы записи этого выражения (так как представление бинарных арифметических операций не может быть интерпретировано двояко, нет необходимости использовать скобки):

- *префиксная*

$$\div(+(\times A\, B)C)\, D \text{ или } \div + \times A\, B\, C\, D;$$

- *инфиксная*

$$((A \times B) + C) \div D \text{ или } A \times B + C \div D;$$

- *постфиксная*

$$((A\,B\times)C+)D\div \text{ или } A\,B\times C+D\div .$$

Приведенная ниже рекурсивная процедура ***InorderTreeWalk*** выполняет *обход* бинарного дерева во внутреннем порядке. Ее первый вызов должен производиться с параметром ***Root***.

```
InorderTreeWalk(Current)
  IF Current ≠ 0
    THEN InorderTreeWalk(Left(Current))
         Выполнение действия над Current.
         InorderTreeWalk(Right(Current))
```

Примечание. *Алгоритмы обхода бинарного дерева в прямом (**PreorderTreeWalk**) и обратном порядках (**PostorderTreeWalk**) рекомендуется реализовать самостоятельно.*

Рекурсия устраняется с помощью стека ***STACK***, в котором запоминается последовательность пройденных вершин (символ «⇐» обозначает операции включения и исключения). Ниже представлен нерекурсивный вариант обхода дерева во внутреннем порядке.

```
InorderTreeWalk(Root)
  Current ← Root
  STACK ← ∅
  WHILE Current ≠ 0 OR STACK ≠ ∅
    DO WHILE Current ≠ 0
         DO STACK ⇐ Current
            Current ← Left(Current)
       Current ⇐ STACK
       Выполнение действия над Current.
       Current ← Right(Current)
```

Каждая вершина бинарного дерева может быть дополнена полем ссылки ***Parent*** на вершину верхнего уровня ($Parent(Root) = 0$). Такая структура называется **деревом с обратными связями** (см. рисунок 3.5).

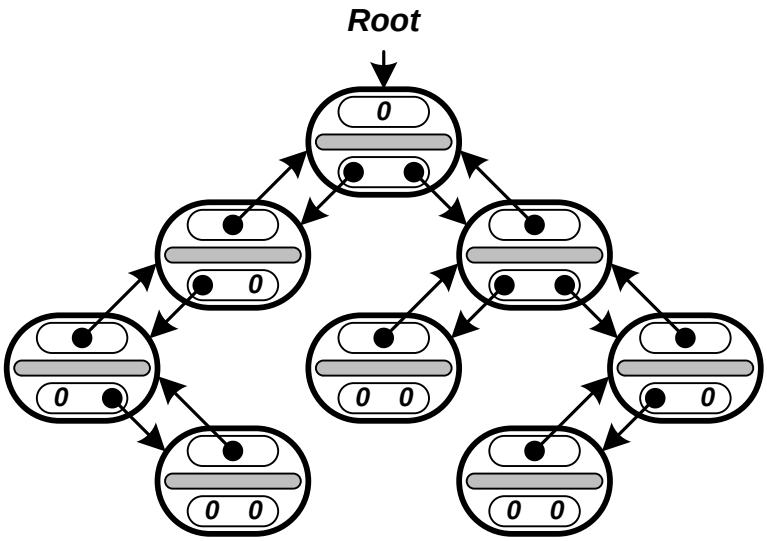


Рисунок 3.5 — Бинарное дерево с обратными связями

Бинарное дерево может быть дополнено ограничителем ***Nil***. Ссылками на корень дерева будут $Left(Nil)$ или $Right(Nil)$ (см. рисунок 3.6).

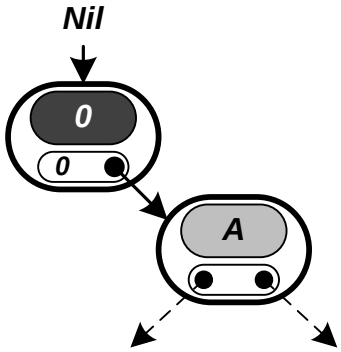


Рисунок 3.6 — Фрагмент бинарного дерева с ограничителем

Такое дерево не может быть пустым, поэтому отпадает необходимость вводить соответствующий контроль при его обработке.

Любое дерево с произвольным ветвлением, если число потомков каждой его вершины ограничено, может быть представлено аналогом бинарного дерева (см. рисунки 3.7 и 3.8).

По терминологии теории деревьев для вершины ***Current*** все вершины нижнего уровня считаются «детьми» (*англ.* ***child*** *— ребенок*), а вершины равного уровня — «родственниками» (*англ.* ***sibling*** *— кровный родственник*). На самого левого потомка вершины ***Current*** будет указывать $\boldsymbol{Left(Current)}$ (если ***Current*** не имеет потомков, то $\boldsymbol{Left(Current)=0}$). На ближайшего по старшинству «родственника» вершины ***Current*** будет указывать $\boldsymbol{Right(Current)}$ (если ***Current*** — последний потомок родительской вершины, то $\boldsymbol{Right(Current)=0}$). Подобная схема называется «левый потомок — правый родственник» (*англ.* ***left child — right sibling***).

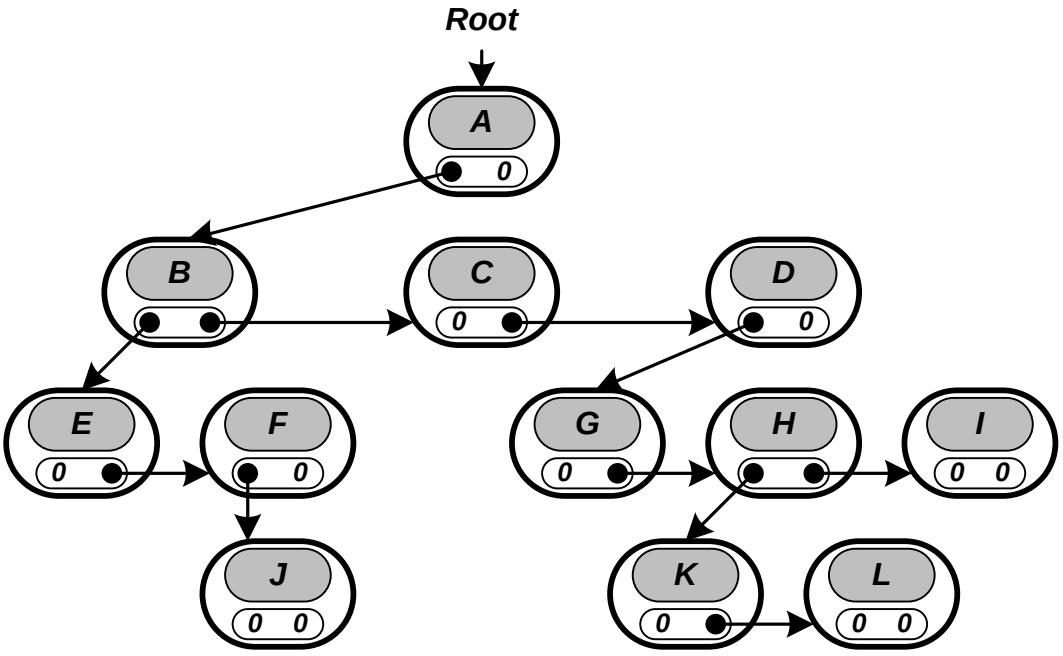


Рисунок 3.7 — Представление дерева с рисунка 3.1 в виде бинарного дерева

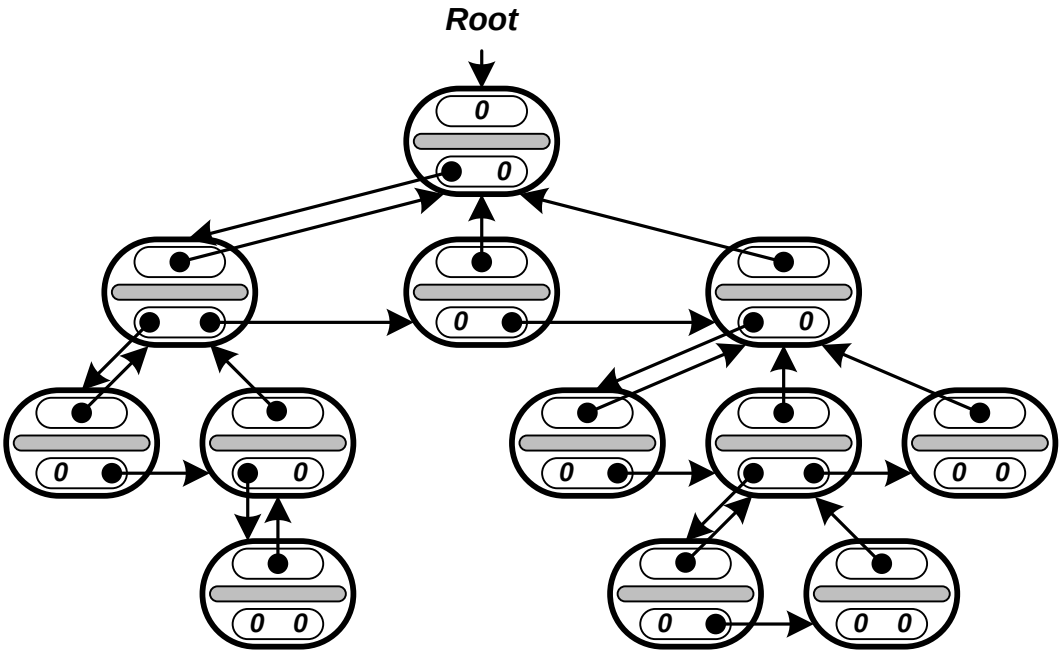


Рисунок 3.8 — Представление дерева с рисунка 3.1 в виде бинарного дерева с обратными связями

3.3 Бинарные деревья поиска

Бинарные деревья поиска или ***BST*** (*англ.* ***binary search trees***) позволяют с высокой производительностью обнаруживать элементы в больших структурах данных. Каждая вершина дерева имеет ключевое поле ***Key***, содержимое которого служит для ее идентификации (обычно в качестве ключей используют натуральные числа). Само дерево обладает свойством упорядоченности.

> *Пусть* $\boldsymbol{u}$ *— произвольная вершина бинарного дерева поиска. Если вершина* $\boldsymbol{v}$ *находится в левой ее ветви, то* $\boldsymbol{Key(v) < Key(u)}$. *В противном случае* $\boldsymbol{Key(v) \geq Key(u)}$.

Дерево с рисунка 3.4., представленное как бинарное дерево поиска, изображено на рисунке 3.9.

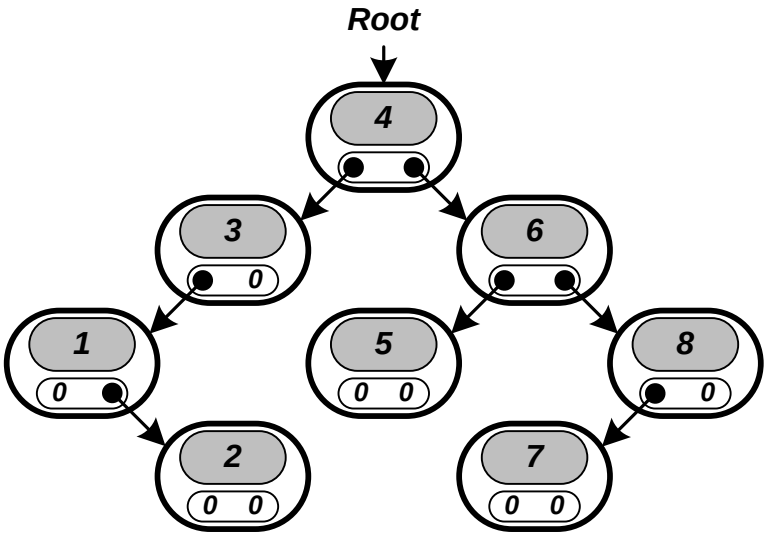


Рисунок 3.9 — Бинарное дерево поиска

Создание вершины бинарного дерева ***New*** с ключом ***key***, содержащей в поле ***Info*** значение ***value***, выполняет процедура ***CreateNode***. Она также возвращает ссылку на эту вершину.

```
CreateNode(key, value)
  Create(New)
  Key(New) ← key
  Info(New) ← value
  Left(New) ← Right(New) ← 0
  RETURN New
```

Вставка в дерево новой вершины ***New*** не должна приводить к потере им упорядоченности. Место вставки находится в процессе поиска. Положение ***New*** относительно текущей вершины ***Current*** определяется после сравнения ключей $Key(New)$ и $Key(Current)$. После завершения операции вершина ***New*** будет концевой. Описанные действия выполнит процедура ***TreeInsert***.

```
TreeInsert(New)
  IF Root = 0
    THEN Root ← New
```

```
  ELSE Current ← Root
       WHILE Current ≠ 0
         DO Prev ← Current
            IF Key(New) < Key(Current)
              THEN Current ← Left(Current)
              ELSE Current ← Right(Current)
       IF Key(New) < Key(Prev)
         THEN Left(Prev) ← New
         ELSE Right(Prev) ← New
```

Рекурсивная процедура ***RecursiveTreeInsert***, вызываемая с параметром ***Root***, решит эту же задачу.

```
RecursiveTreeInsert(Current, New)
  IF Current = 0
    THEN Current ← New
    ELSE IF Key(New) < Key(Current)
           THEN TreeInsert(Left(Current), New)
           ELSE TreeInsert(Right(Current), New)
```

Поиск в дереве ключа ***key*** выполнит процедура ***TreeSearch***. Она вернет ссылку на вершину ***Current***, у которой $Key(Current) = key$, или пустую ссылку, если значение ***key*** не найдено.

```
TreeSearch(key)
  Current ← Root
  WHILE Current ≠ 0 AND Key(Current) ≠ key
    DO IF key < Key(Current)
         THEN Current ← Left(Current)
         ELSE Current ← Right(Current)
  RETURN Current
```

Рекурсивная процедура ***RecursiveTreeSearch***, вызываемая с параметрами ***Root*** и ***key***, решит эту же задачу.

```
RecursiveTreeSearch(Current, key)
  IF Current = 0 OR Key(Current) = key
    THEN RETURN Current
  IF key < Key(Current)
    THEN RETURN TreeSearch(Left(Current), key)
    ELSE RETURN TreeSearch(Right(Current), key)
```

Определение в поддереве с корнем ***Current*** вершины с минимальным ключом (самой «левой») выполнит процедура ***TreeMinimum***. Процедура ***TreeMaxixum*** нахождения вершины с максимальным ключом (самой «правой») симметрична.

```
TreeMinimum(Current)
  WHILE Left(Current) ≠ 0
    DO Current ← Left(Current)
  RETURN Current
```

Рекурсивная процедура ***TreeClear***, вызываемая с параметром ***Root***, *удалит* все вершины дерева из памяти.

```
TreeClear(Current)
  IF Current ≠ 0
    THEN TreeClear(LeftLink(Current))
         TreeClear(RightLink(Current))
         Delete(Current)
```

Дерево типа ***BST*** с ***N*** вершинами содержит ***N + 1*** пустых концевых ссылок. Их можно использовать, как «нити» (*англ.* ***threads***). Каждая левая нить ведет к вершине с предшествующим, а каждая правая — к вершине с последующим значением ключа. Самая левая и самая правая нити ведут к корневой вершине. Такое дерево называется «прошитым» (*англ.* ***threaded tree***; см. рисунок 3.10).

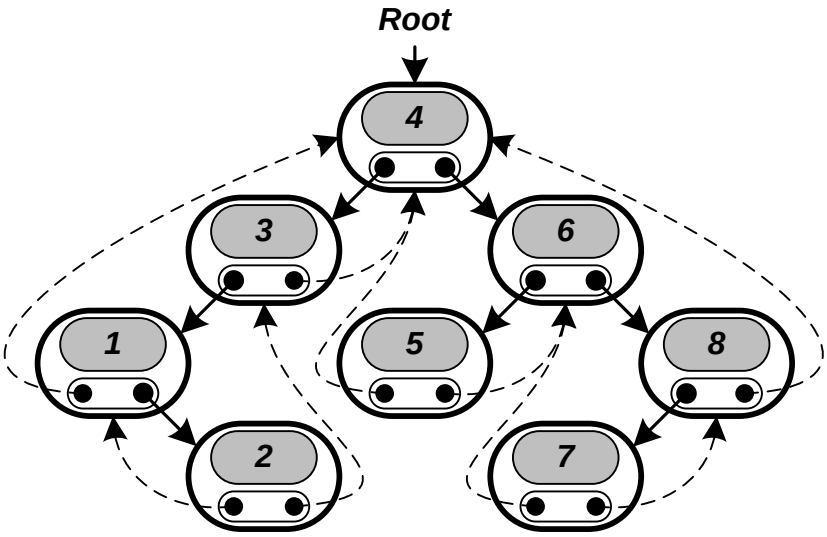


Рисунок 3.10 — «Прошитый» аналог бинарного дерева поиска с рисунка 3.9

Каждая вершина прошитого дерева должна содержать два дополнительных логических поля, определяющих порядок использования ссылок ***Left*** и ***Right*** (например, ***LeftThread*** и ***RightThread***). Если содержимое этих полей — «истина», то соответствующая ссылка является «нитью». В противном случае она указывает на вершину нижнего уровня.

Бинарное дерево поиска может быть дополнено обратными связями (см. рисунок 3.5). *Вставку* вершины ***New*** в такое дерево выполняет следующая процедура.

```
TreeInsert(New)
  IF Root = 0
    THEN Root ← New
    ELSE Current ← Root
         WHILE Current ≠ 0
           DO Parent(New) ← Current
              IF Key(New) < Key(Current)
                THEN Current ← Left(Current)
                ELSE Current ← Right(Current)
         IF Key(New) < Key(Parent(New))
```

```
        THEN Left(Parent(New)) ← New
        ELSE Right(Parent(New)) ← New
```

Примечание. *Для любой новой вершины* **New** *до ее вставки в дерево принимается* $Parent(New) = 0$.

Пусть имеется вершина ***Current***. Определение в дереве вершины со следующим за значением $Key(Current)$ ключа (*англ.* ***successor***) выполнит процедура ***TreeSuccessor***. Она вернет ссылку на нее или пустую ссылку, если нужное значение не найдено. Процедура ***TreePredecessor*** нахождения в дереве предшествующего значению $Key(Current)$ ключа (*англ.* ***predecessor***) является симметричной.

```
TreeSuccessor(Current)
  IF Right(Current) ≠ 0
    THEN RETURN TreeMinimum(Right(Current))
  Top ← Parent(Current)
  WHILE Top ≠ 0 AND Current = Right(Top)
    DO Current ← Top
       Top ← Parent(Top)
  RETURN Top
```

Можно удалить из дерева вершину ***Current*** без нарушения упорядоченности по ключам одним из способов:

- если ***Current*** — концевая вершина, она удаляется из дерева обычным способом (см. рисунок 3.11-а);
- если ***Current*** — неполная вершина, она удаляется из дерева путем реорганизации связей (см. рисунок 3.11-б);
- если ***Current*** — полная вершина, она заменяется вершиной с последующим значением ключа, у которой *левая ветвь будет пустой*, после чего вершина-«последователь» удаляется обычным образом (см. рисунок 3.11-в).

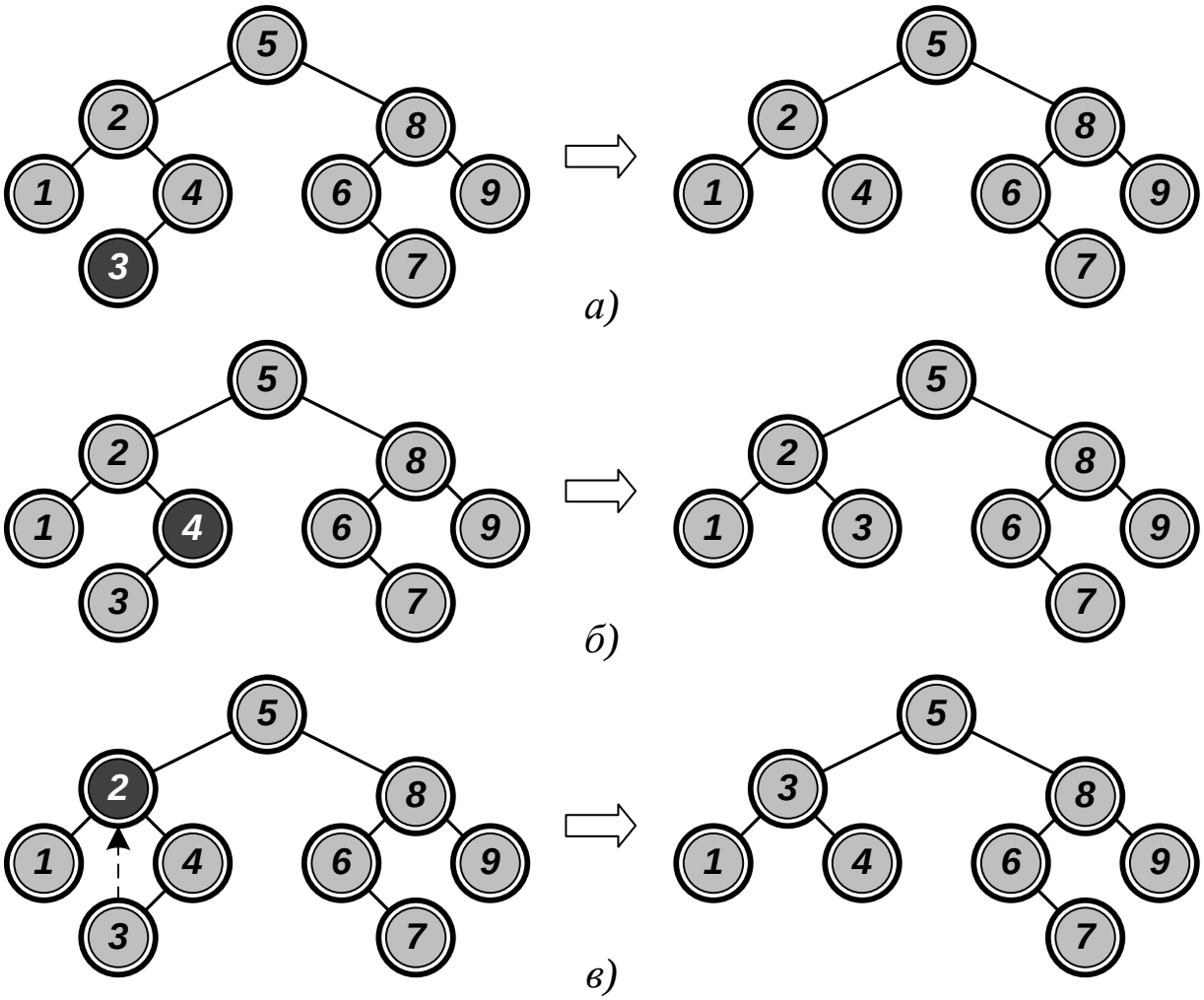


Рисунок 3.11 — Удаление из бинарного дерева поиска: а) концевой вершины; б) неполной вершины; в) полной вершины

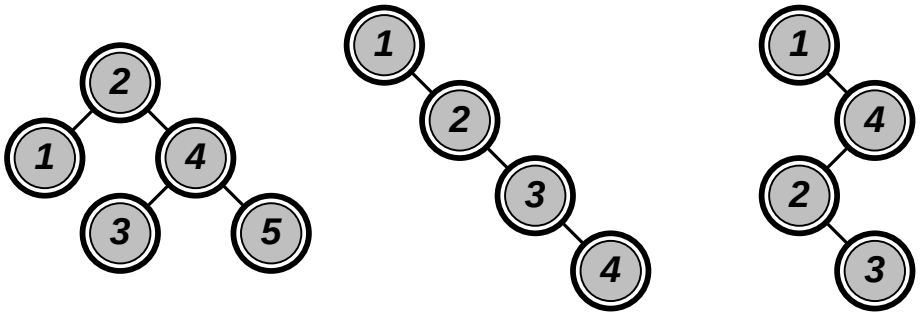
Процедура ***TreeDelete*** *удалит* из дерева вершину ***Current*** (указатели ***Cut*** и ***Child*** — временные).

```
TreeDelete(Current)
  IF Left(Current) = 0 OR Right(Current) = 0
    THEN Cut ← Current
    ELSE Cut ← TreeSuccessor(Current)
  IF Left(Cut) ≠ 0
    THEN Child ← Left(Cut)
    ELSE Child ← Right(Cut)
  IF Child ≠ 0
    THEN Parent(Child) ← Parent(Cut)
```

```
IF Parent(Cut) = 0
  THEN Root ← Child
  ELSE IF Cut = Left(Parent(Cut))
         THEN Left(Parent(Cut)) ← Child
         ELSE Right(Parent(Cut)) ← Child
IF Cut ≠ Current
  THEN Key(Current) ← Key(Cut)
       Info(Current) ← Info(Cut)
Delete(Cut)
```

3.4 Сбалансированные бинарные деревья

Деревья типа ***BST*** удобны для поиска их вершин по ключевым значениям. Оценка эффективности поиска в структуре размерностью N имеет порядок $\log_2 N$ (так, для обнаружения нужного значения среди $10\,000$ элементов понадобится не более 14 сравнений; при аналогичном поиске в линейной структуре число сравнений в среднем составит $5\,000$). Однако, если ключи генерируются случайным образом, дерево может выродиться (см. рисунок 3.12).



а) *б)* *в)*

Рисунок 3.12 — Формирование бинарного дерева поиска при включении элементов: а) случайным образом; б) в порядке возрастания; в) сериями монотонности

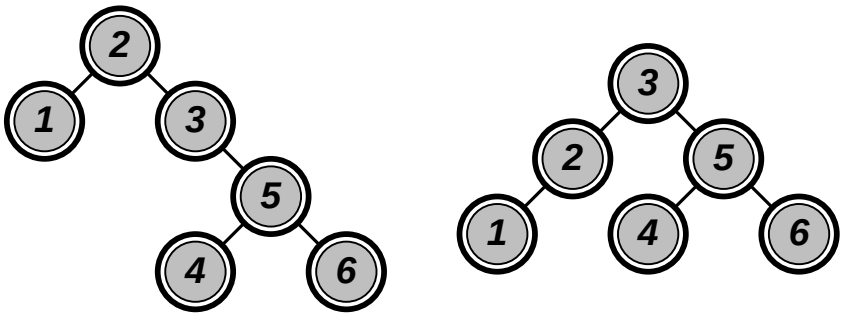
Можно предположить, что поиск будет более быстрым, если элементы, которые ищутся чаще, находятся ближе к корню дерева.

Бинарное дерево, построенное с учетом вероятности поиска, называется **оптимальным**.

Примечание. *Создание оптимальных деревьев требует, как правило, больших затрат времени, что нивелирует выигрыш в скорости поиска. Более перспективным считается использование сбалансированных деревьев.*

Бинарное дерево называется **идеально сбалансированным**, если для любой его вершины справедливо, что число вершин в ее левой и правой ветвях различается не более, чем на ***1***,

Примечание. *Поддержание идеального баланса требует значительной перестройки дерева, затраты времени на которую могут существенно превышать время поиска. На практике используются т. н.* ***AVL****-деревья (по фамилиям московских математиков Г.М. Адельсон-Вельского и Е.М. Ландиса), требования к сбалансированности которых менее строгие.*



а) *б)*

Рисунок 3.13 — Представление массива: а) в виде бинарного дерева поиска; б) в виде ***AVL***-дерева

Бинарное дерево называется **сбалансированным по *AVL***, если для любой его вершины справедливо, что высота ее левой и правой ветвей различается не более чем на ***1*** (см. рисунок 3.13).

В состав каждой вершины ***AVL***-дерева включается оценка ее *высоты* ***Height***. *Наибольшую высоту имеет корень дерева. Высота*

концевых вершин равна нулю. Для любой вершины ***Current*** условие баланса записывается в виде

$$\left|Height\left(Left\left(Current\right)\right) - Height\left(Right\left(Current\right)\right)\right| < 2.$$

Примечание. *Для любой новой вершины* ***New*** *до ее вставки в дерево принимается* $Height\left(New\right) = 0$.

Операции над деревьями ***AVL*** и ***BST*** аналогичны. Исключение составляют вставка и удаление вершин, при которых сбалансированность ***AVL***-дерева может нарушиться.

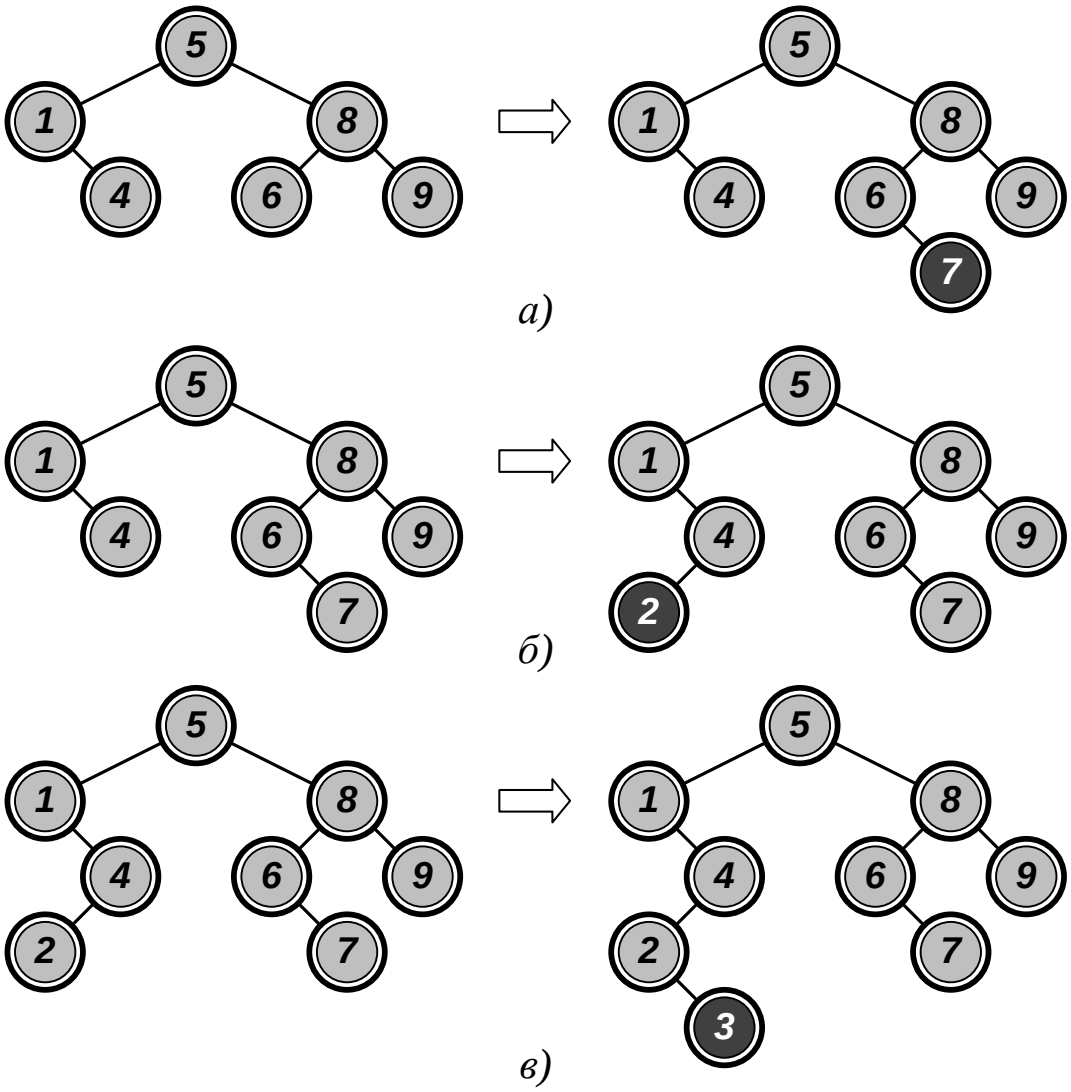


Рисунок 3.14 — Вставка элемента в ***AVL***-дерево

При вставке в ***AVL***-дерево новой вершины возможны следующие ситуации:

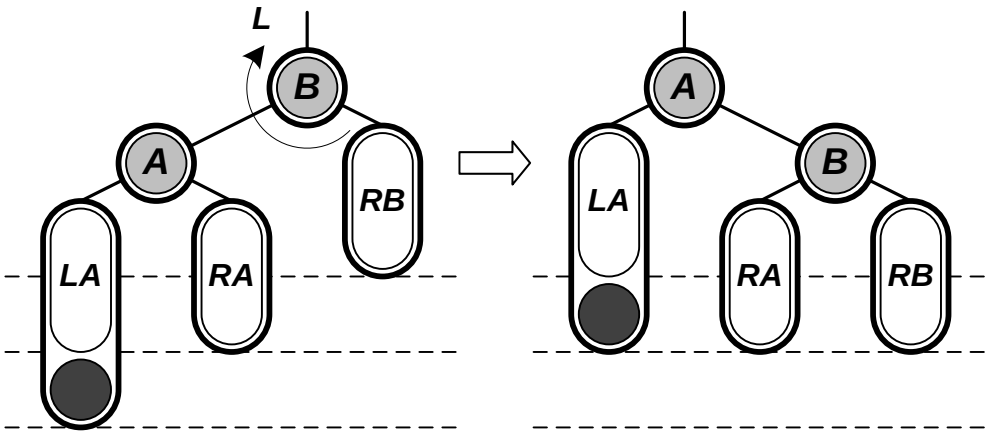
- дерево изначально сбалансировано. При вставке удлиняется одна из его ветвей, но в целом сбалансированность по ***AVL*** не нарушается (см. рисунок 3.14-а);
- дерево изначально сбалансировано по ***AVL***. Вставка вершины производится в более короткую ветвь (см. рисунок 3.14-б), что только улучшает показатель сбалансированности дерева;
- дерево изначально сбалансировано по ***AVL***. Вставка вершины производится в более длинную ветвь (см. рисунок 3.14-в), после чего баланс дерева необходимо восстановить.

Если после вставки у какой-либо вершины левого поддерева недопустимо вырастает левая ветвь (см. рисунок 3.15-а), то баланс восстанавливается *однократным левым вращением* ***LL***. Если при выполнении этой операции у какой-либо вершины правого поддерева недопустимо вырастает правая ветвь (см. рисунок 3.15-б), то баланс восстанавливается *однократным правым вращением* ***RR***.

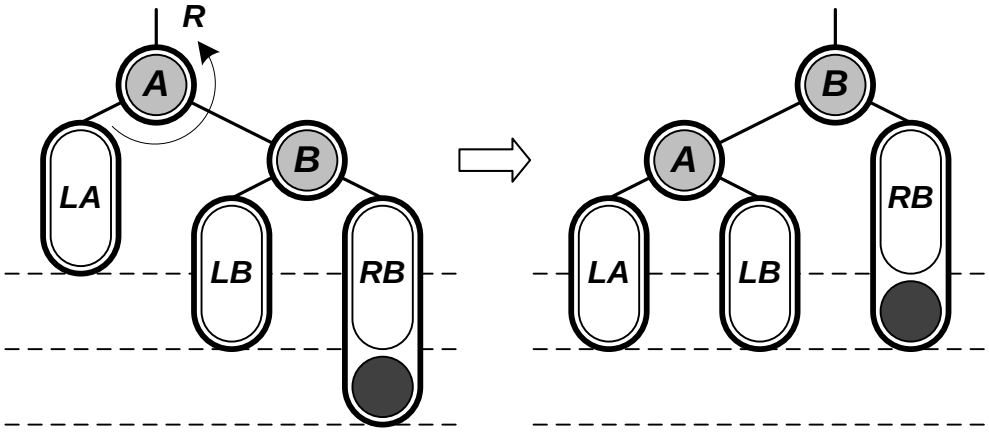
Для поддерева с корнем ***Current*** однократное левое вращение выполнит процедура ***RotationLL***.

```
RotationLL(Current)
  L ← Left(Current)
  Left(Current) ← Right(L)
  Right(L) ← Current
  Height(Current) ← 1 +
    Max(H(Left(Current)), H(Right(Current))
  Height(L) ← 1 +
    Max(H(Left(L)), H(Current))
  Current ← L
```

Процедура однократного правого вращения ***RotationRR*** является симметричной.



а)



б)

Рисунок 3.15 — Однократные вращения: (а) левое типа ***LL***; (б) правое типа ***RR*** (закрашенная область ветви показывает увеличение ее длины)

Примечание. *Функция* ***H*** *определения высоты вершины исключает возможность разыменования пустых ссылок.*

```
H(Current)
  IF Current ≠ 0
    THEN RETURN Height(Current)
    ELSE RETURN -1
```

Если после вставки у какой-либо вершины левого поддерева недопустимо вырастает правая ветвь (см. рисунок 3.16-а), то баланс восстанавливается *двукратным вращением* ***RL***. Если при выполне-

нии этой операции у какой-либо вершины правого поддерева недопустимо вырастает левая ветвь (см. рисунок 3.16-б), то баланс восстанавливается *двукратным вращением* ***LR***. (Знак «**?**» показывает, что вершина добавляется *либо в ветвь* ***LC****, либо в ветвь* ***RC****, но не в обе одновременно*).

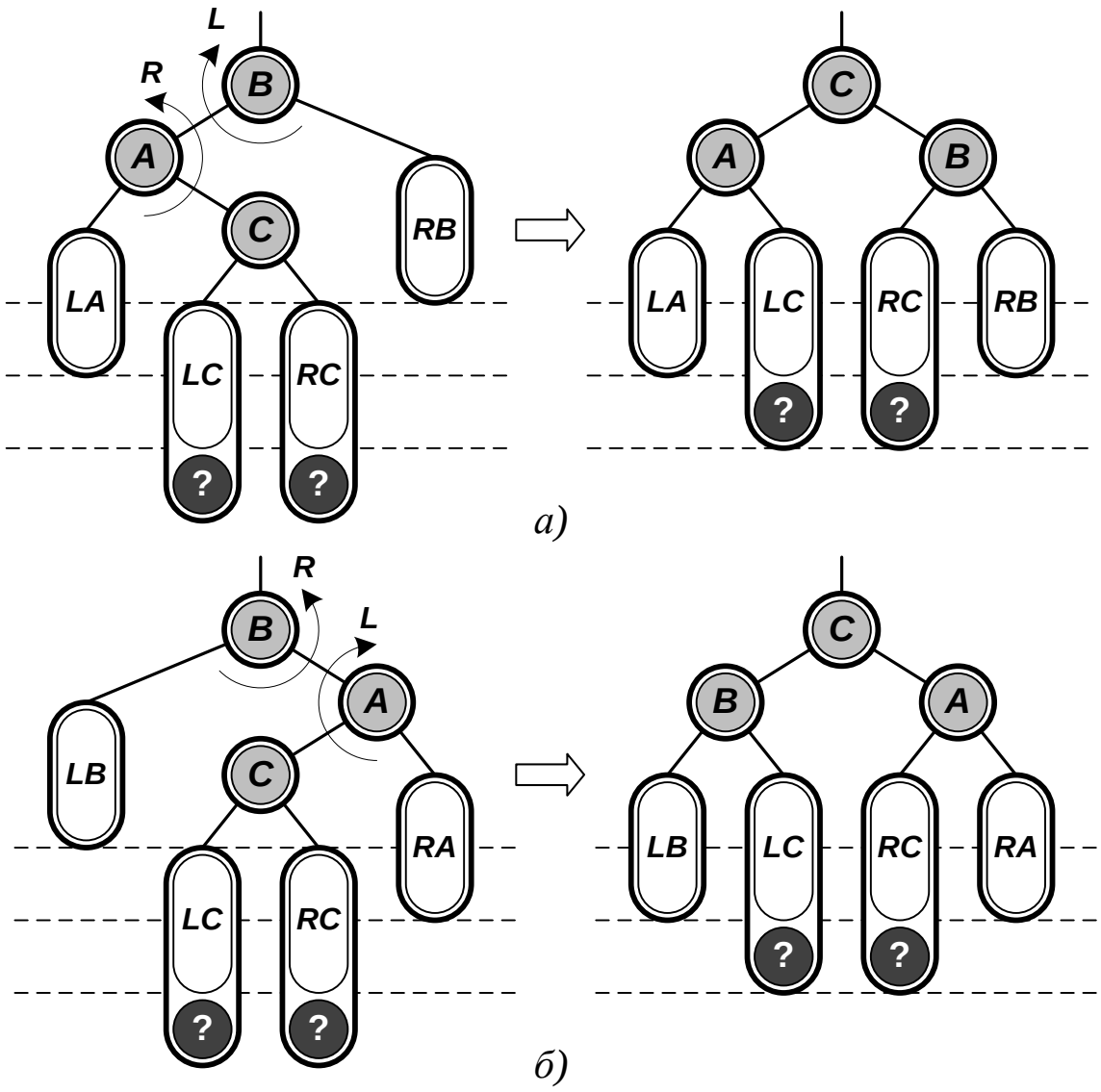


Рисунок 3.16 — Двукратные вращения: а) типа ***RL***; б) типа ***LR***

Для поддерева с корнем ***Current*** *двукратное вращение* ***RL*** выполнит процедура ***RotationRL***. Процедура *двукратного вращения* ***LR*** (см. рисунок 3.16-б) симметрична.

```
RotationRL(Current)
  RotationRR(Left(Current))
  RotationLL(Current)
```

Вставка в ***AVL***-дерево новой вершины ***New*** включает в себя следующие этапы:

- поиск по дереву места для вставки вершины;
- добавление вершины в найденную для нее позицию;
- восстановление при необходимости баланса для всех вершин по пути вверх от добавленной вершины к корню дерева.

Процедура ***AVLTreeInsert***, вызываемая с параметрами ***Root*** и ***New***, решает эту задачу рекурсивно. *Балансировку* дерева выполняет процедура ***Balance***.

```
AVLTreeInsert(Current, New)
  IF Current = 0
    THEN Current ← New
    ELSE IF Key(New) < Key(Current)
           THEN AVLTreeInsert(Left(Current), New)
           ELSE AVLTreeInsert(Right(Current), New)
         Balance(Current, New)
```

```
Balance(Current, New)
  IF Key(New) < Key(Current)
    THEN IF H(Left(Current) – H(Right(Current)) > 1
           THEN IF Key(New) < Key(Left(Current))
                  THEN RotationLL(Current)
                  ELSE RotationRL(Current)
    ELSE IF H(Right(Current) – H(Left(Current)) > 1
           THEN IF Key(New) > Key(Right(Current))
                  THEN RotationRR(Current)
                  ELSE RotationLR(Current)
  Height(Current) ← 1 +
    Max(H(Left(Current)), H(Right(Current))
```

Удаление из ***AVL***-дерева включает в себя три этапа:

- поиск вершины, которую необходимо удалить;
- удаление вершины с сохранением упорядоченности дерева;
- восстановление при необходимости баланса для всех вершин по пути вверх к корню дерева от места, из которого была удалена вершина (см. рисунок 3.11-а и 3.11-б) или из которого была взята вершина для замены удаляемой (см. рисунок 3.11-в).

Примеры балансировки ***AVL***-дерева после удаления из него вершин показаны на рисунке 3.17.

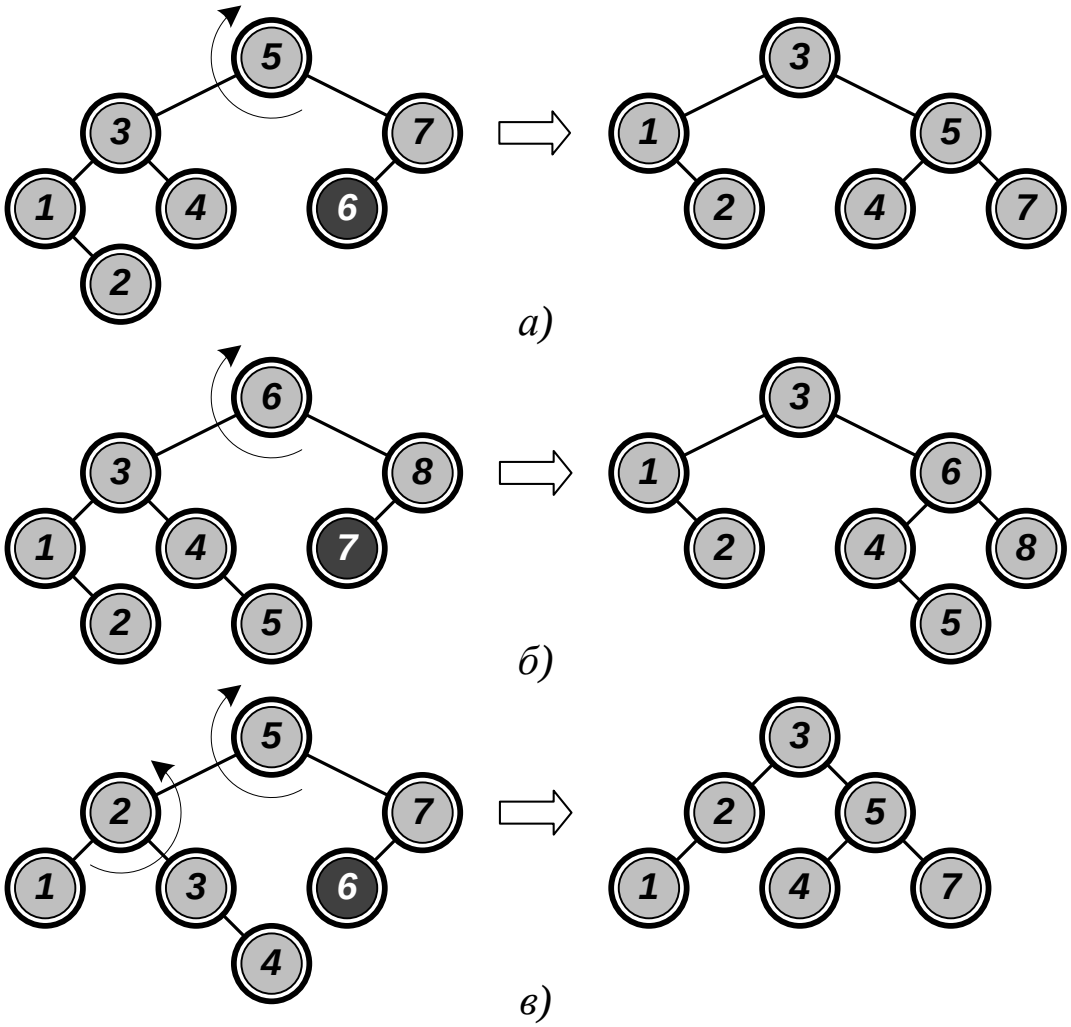


Рисунок 3.17 — Восстановление сбалансированности ***AVL***-дерева: а) и б) поворотом ***LL***; в) поворотом ***RL***

Достоинством ***AVL***-деревьев является повышение эффективности поиска их элементов по ключевым значениям. Однако сложность балансировки приводит к тому, что эти деревья используются лишь в тех случаях, когда поиск выполняется значительно чаще, чем включение или исключение элементов.

Примечание. *Экспериментально установлено, что в среднем балансировка необходима один раз на каждые два включения и один раз на каждые пять удалений (однократное и двукратное вращения равновероятны). При этом добавление вершины в дерево может вызвать в худшем случае одно вращение двух или трех узлов. Удаление вершины из дерева может потребовать вращений в каждой вершине по пути поиска.*

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие структуры данных считаются иерархическими? Какие основные операции выполняются над ними?
2. По какому принципу организуются структуры данных типа «дерево»? Какие основные понятия теории деревьев?
3. Что понимается под поиском по дереву? Какие существуют виды поиска по дереву с произвольным ветвлением?
4. Как рекурсивно определяется бинарное дерево? Какие бинарные деревья считаются подобными, какие — эквивалентными? Какие бинарные деревья считаются полными, какие — завершенными, какие — вырожденными?
5. Какие существуют порядки обхода бинарного дерева?
6. Как строится бинарное дерево с использованием технологии связного распределения памяти?
7. Как строится бинарное дерево с обратными связями с использованием технологии связного распределения памяти?
8. Как в виде бинарного дерева представляются деревья с произвольной структурой?

9. Для чего бинарное дерево дополняется ограничителем?

10. Какое бинарное дерево называется бинарным деревом поиска? Что понимается под его упорядоченностью?

11. По каким правилам выполняются основные операции над бинарными деревьями поиска?

12. Какие бинарные деревья поиска называются «прошитыми»? В чем назначение «прошивки» деревьев?

13. Для чего необходимо балансировать бинарное дерево поиска? Какое бинарное дерево поиска называется оптимальным, какое — идеально сбалансированным, какое — сбалансированным по ***AVL***?

14. Когда нарушается сбалансированность по ***AVL*** при вставке вершин в дерево?

15. Когда нарушается сбалансированность по ***AVL*** при удалении вершин из дерева?

16. Как восстанавливается сбалансированность ***AVL***-дерева? Какие виды вращений существуют и когда они применяются?

17. В чем заключаются достоинства и недостатки ***AVL***-деревьев?

4 МНОГОСВЯЗНЫЕ СТРУКТУРЫ ДАННЫХ

4.1 Общие сведения

Многосвязной структурой или **графом** (*англ.* ***graph***) называется структура данных, которая определяется как $G = \langle V, E \rangle$, где V — конечное непустое множество **вершин** (*англ.* ***vertex set***), а $E \subseteq V \times V$ — бинарное отношение на V или множество **ребер** (*англ.* ***edge set***). Число вершин графа $|V| = N$, число ребер $|E| = M$.

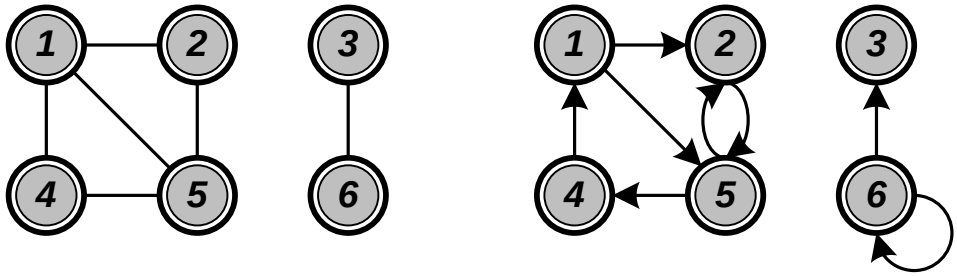
Если множество ребер состоит из *неупорядоченных пар* вершин

$$E \subseteq \{\{u,v\} \mid u \in V, v \in V, u \neq v\},$$

граф считается неориентированным (см. рисунок 4.1-а). Если множество ребер (или **дуг**) состоит из *упорядоченных пар* вершин:

$$E \subseteq \{(u,v) \mid u \in V, v \in V\},$$

граф является ориентированным (см. рисунок 4.1-б). Ориентированный граф может содержать параллельные дуги и петли. В неориентированном графе каждое ребро должно ограничиваться двумя различными вершинами.



а) *б)*

Рисунок 4.1 — Графы: а) неориентированный; б) ориентированный

Можно получить *ориентированный вариант неориентированного графа*, заменив каждое ребро $\{u,v\}$ парой дуг (u,v) и (v,u).

Неориентированный вариант ориентированного графа получается путем отказа от направленности дуг, удаления петель и объединения встречных дуг (u,v) и (v,u) в ребра $\{u,v\}$.

Если в ориентированном графе G имеется дуга (u,v), то говорят, что она *выходит из вершины* u *и входит в вершину* v, а вершина v *смежна с вершиной* u (обозначается $u \to v$). (На рисунке 4.1-б вершина 2 смежна с вершиной 1, но вершина 1 не смежна с вершиной 2.) Для неориентированного графа отношение смежности является симметричным. Про его ребро $\{u,v\}$ говорят, что оно *инцидентно* вершинам u и v.

Если $u = v$, дуга называется *петлей* ($(6,6)$ на рисунке 4.1-б). Дуги с общими граничными вершинами являются *параллельными* (кратными). Если направления дуг относительно общих вершин совпадают, они *строго параллельны*, если нет — *нестрого параллельны*. Две нестрого параллельные дуги могут заменяться ребром. Ориентированный граф, не содержащий петель, называется *простым*. Граф, содержащий параллельные дуги и петли, называется *мультиграфом*. Граф, содержащий и ребра, и дуги, называется *смешанным*.

Степень (валентность) вершины v графа $0 \le d(v) \le N-1$ — число инцидентных ей ребер. Степень *изолированной* вершины равна 0. Степень *висячей (конечной)* вершины равна 1. Если степени всех вершин равны, граф *регулярный (однородный)*. (На рисунке 4.1-а вершины 1 и 5 имеют степень 3, вершины 2 и 4 — степень 2, вершины 3 и 6 — степень 1.)

Степень вершины ориентированного графа определяется как сумма ее *исходящей и входящей степеней* — числа выходящих и входящих дуг $d^-(v)$ и $d^+(v)$. (На рисунке 4.1-б вершина 5 имеет входящую степень 2 и исходящую степень 2.)

Путь p из вершины u в вершину v — последовательность вершин $\langle v_0, v_1, v_2, \ldots, v_k \rangle$, где $v_0 = u$ — *начало* пути, $v_k = v$ — его *конец*. Путь считается *простым*, если все вершины в нем различны.

(На рисунке 4.1-б путь $\langle 1,2,5,4\rangle$ — простой, путь $\langle 5,2,5,4\rangle$ простым не является.) Если существует путь p из u в v, то говорят, что вершина v *достижима из* u *по пути* p.

Длина пути k — количество входящих в него (возможно повторяющихся) ребер $\{v_{i-1},v_i\}\in E, i=1,2,...,k$. *Расстояние* между вершинами u и v — длина кратчайшего соединяющего их простого пути $d(u,v)$. *Диаметр* графа G — самое большое расстояние между парами его вершин $D(G)$. Множество вершин, находящихся на одинаковом расстоянии L от вершины v, называется *ярусом*

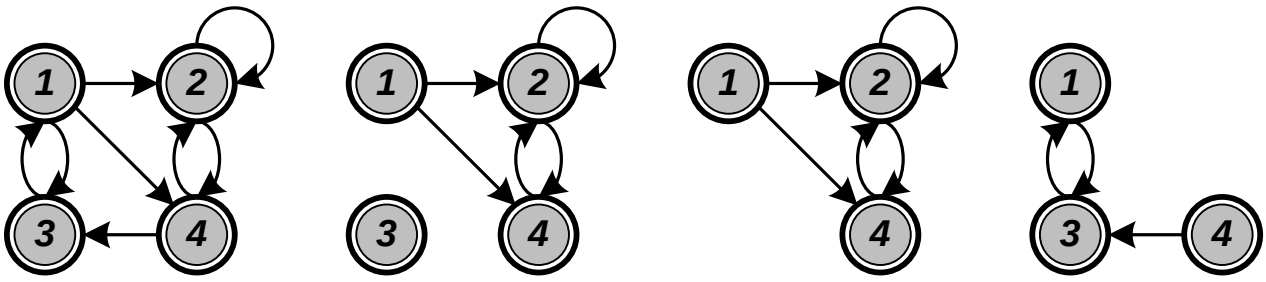
$$D(v,L)=\{u\in V \mid d(v,u)=L\}.$$

Цикл в ориентированном графе — путь ненулевой длины, в котором начальная и конечная вершины совпадают. Один и тот же цикл длины k может быть представлен k различными путями. Цикл считается *простым*, если в нем нет одинаковых вершин, кроме первой и последней. (На рисунке 4.1-б пути $\langle 1,5,4,1\rangle$, $\langle 5,4,1,5\rangle$ и $\langle 4,1,5,4\rangle$ представляют один и тот же простой цикл; цикл $\langle 1,5,2,5,4,1\rangle$ простым не является.) Петля является циклом единичной длины ($\langle 6,6\rangle$ на рисунке 4.1-б). Граф, в котором нет циклов, называется *ациклическим*.

В неориентированном графе путь считается (простым) циклом, если его длина $k\geq 3$, и все вершины в нем различны. (На рисунке 4.1-а показаны циклы $\langle 1,2,5,1\rangle$, $\langle 1,5,4,1\rangle$ и $\langle 1,2,5,4,1\rangle$.)

Граф $G'=\langle V',E'\rangle$ является частью графа $G=\langle V,E\rangle$, если $V'\subseteq V$ и $E'\subseteq E$. Часть, которая содержит некоторое подмножество ребер графа и все его вершины, называется **суграфом**. Часть, которая содержит некоторое подмножество ребер графа и все инцидентные им вершины, называется **подграфом** (исходный граф по отношению к его подграфу считается **надграфом**, а по отношению к суграфу — **сверхграфом**). Совокупность всех ребер графа, не

принадлежащих подграфу, вместе с инцидентными им вершинами образует **дополнение** подграфа (см. рисунок 4.2).



а) *б)* *в)* *г)*

Рисунок 4.2 — Граф и его части: а) исходный граф; б) суграф; в) подграф; г) дополнение подграфа

Говорят, что подграфы $G' = \langle V', E' \rangle$ и $G'' = \langle V'', E'' \rangle$ *разделены ребрами*, если они не имеют общих ребер ($E' \cap E'' = \varnothing$), и *разделены вершинами*, если у них нет общих вершин ($V' \cap V'' = \varnothing$).

Два графа $G = \langle V, E \rangle$ и $G' = \langle V', E' \rangle$ **изоморфны**, если существует взаимно однозначное соответствие между множествами их вершин $f : V \to V'$, сохраняющее смежность: $\{u, v\} \in E$ тогда и только тогда, когда $\{f(u), f(v)\} \in E'$. (На рисунке 4.3 приведен пример изоморфных графов с множествами вершин $V = \{1, 2, 3, 4\}$ и $V' = \{\alpha, \beta, \gamma, \delta\}$; изоморфизм определяет функция f, для которой $f(1) = \alpha$, $f(2) = \beta$, $f(3) = \gamma$, $f(4) = \delta$.) Графы всегда рассматриваются *с точностью до изоморфизма*.

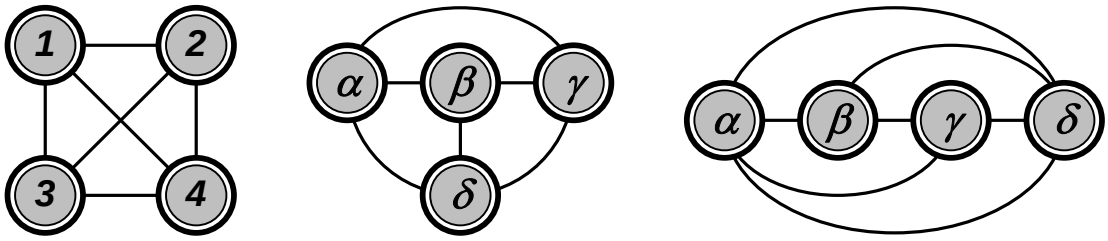
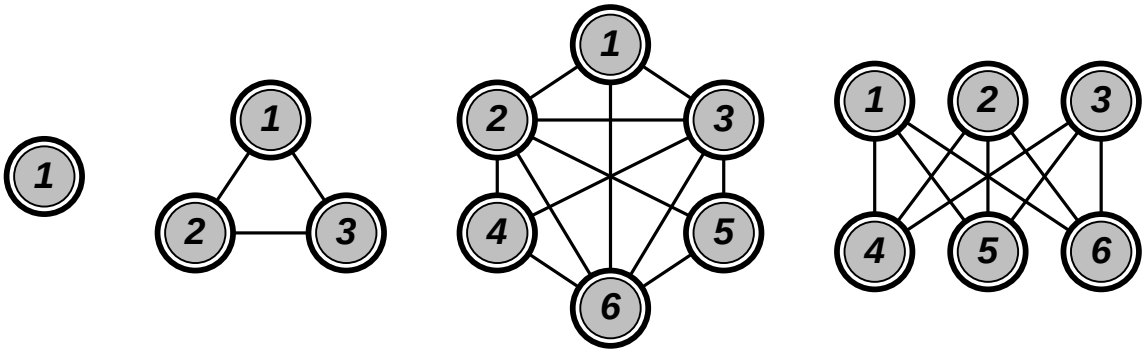


Рисунок 4.3 — Изоморфные графы

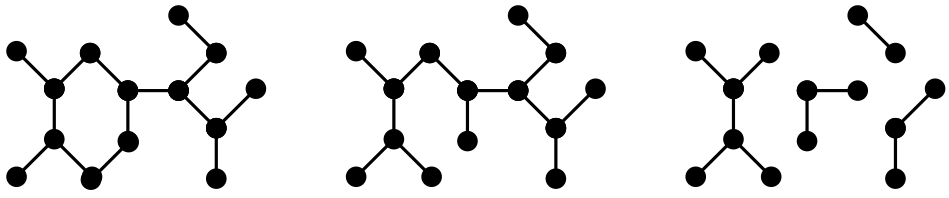
Граф считается **плоским (планарным)**, если изоморфный ему граф можно изобразить на плоскости без пересечения ребер (см. рисунок 4.3). Граф называется **тривиальным**, если он состоит из одной вершины. Граф, состоящий из простого цикла с k вершинами, обозначается C_k. Граф, в котором любая пара вершин смежна, называется **полным**. Неориентированный граф, множество вершин которого можно разбить на два непересекающихся подмножества так, чтобы любое ребро соединяло вершины разных подмножеств, называется **двудольным** (биграфом).



а) *б)* *в)* *г)*

Рисунок 4.4 — Специальные графы: а) тривиальный граф; б) граф C_3; в) полный граф; г) двудольный граф

Несвязный ациклический неориентированный граф называют **лесом**, а связный ациклический неориентированный граф называют **деревом без выделенного корня** (см. рисунок 4.5).

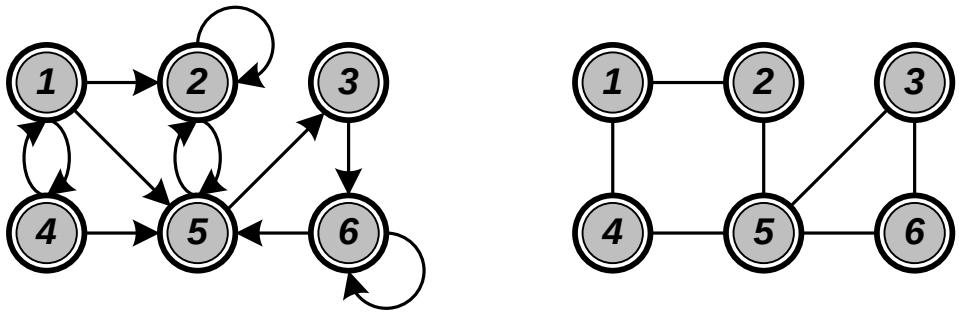


а) *б)* *в)*

Рисунок 4.5 — Виды графов: а) связный циклический граф; б) дерево без выделенного корня; в) лес

4.2 Алгоритмическое представление графа

Способы представления структуры графов в программах различаются требуемым для этого объемом памяти и скоростью решения задач. Выбор любого из них оказывает принципиальное влияние на эффективность алгоритмов.



а) *б)*

Рисунок 4.6 — Примеры графов: а) ориентированного; б) неориентированного

1) Представление с помощью матрицы инцидентности. Матрица имеет размерность $N \times M$. Для ориентированного графа ее столбец, соответствующий дуге (u,v), содержит ***–1*** в строке ***u***, ***1*** в строке ***v*** и нули во всех остальных строках (петлю, т. е. дугу вида (u,u) представляют иным значением в строке ***u***, например ***2***). Для неориентированного графа столбец, соответствующий ребру $\{u,v\}$, содержит ***1*** в строках ***u*** и ***v*** и нули в остальных строках.

С алгоритмической точки зрения это один из худших способов представления графа. Для хранения сильно разреженной структуры используется $N \times M$ элементов. Неудобен также доступ к информации (ответ на элементарные вопросы «существует ли данное ребро», «к каким вершинам ведут ребра из данной вершины» требует перебора всех столбцов матрицы).

	(1, 2)	(1, 4)	(1, 5)	(2, 2)	(2, 5)	(3, 6)	(4, 1)	(4, 5)	(5, 2)	(5, 3)	(6, 5)	(6, 6)
1	–1	–1	–1	0	0	0	1	0	0	0	0	0
2	1	0	0	2	–1	0	0	0	1	0	0	0
3	0	0	0	0	0	–1	0	0	0	1	0	0
4	0	1	0	0	0	0	–1	–1	0	0	0	0
5	0	0	1	0	1	0	0	1	–1	–1	1	0
6	0	0	0	0	0	1	0	0	0	0	–1	2

а)

	{1, 2}	{1, 4}	{2, 5}	{3, 5}	{3, 6}	{4, 5}	{5, 6}
1	1	1	0	0	0	0	0
2	1	0	1	0	0	0	0
3	0	0	0	1	1	0	0
4	0	1	0	0	0	1	0
5	0	0	1	1	0	1	1
6	0	0	0	0	1	0	1

б)

Рисунок 4.7 — Матрицы инцидентности: а) для графа с рисунка 4.6-а; б) для графа с рисунка 4.6-б

	1	2	3	4	5	6
1	0	1	0	1	1	0
2	0	1	0	0	1	0
3	0	0	0	0	0	1
4	1	0	0	0	1	0
5	0	1	1	0	0	0
6	0	0	0	0	1	1

а)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	0	0	0	0	1	1
4	1	0	0	0	1	0
5	0	1	1	1	0	1
6	0	0	1	0	1	0

б)

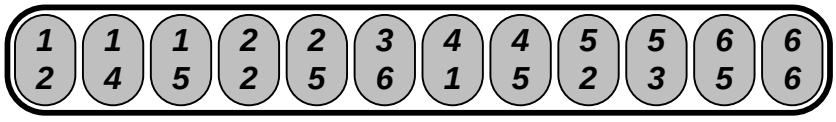
Рисунок 4.8 — Матрицы смежности: а) для графа с рисунка 4.6-а; б) для графа с рисунка 4.6-б

2. Представление с помощью матрицы смежности. Матрица имеет размерность $N \times N$. Элемент ее u-й строки и v-го столбца равен 1, если вершина v смежна с вершиной u. В противном случае

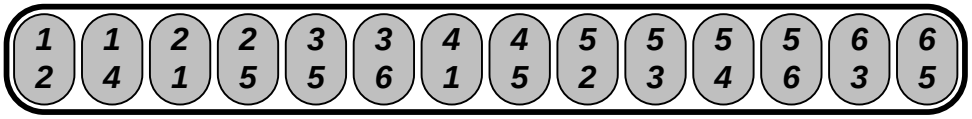
он равен 0. (Очевидно, что для неориентированного графа матрица смежности симметрична.)

Основное преимущество матрицы смежности — возможность сразу получить ответ на вопрос, существует ли ребро $\{u,v\}$. Основной недостаток — независимость ее размера от числа ребер (необходимо всегда размещать $N \times N$ элементов).

3) Представление с помощью списка инцидентности. Каждый элемент списка содержит номера смежных вершин.



а)



б)

Рисунок 4.9 — Списки инцидентности: а) для графа с рисунка 4.6-а; б) для графа с рисунка 4.6-б

Этот метод более экономный (особенно в случае неплотных графов, когда $M << N \times N$), так как расход памяти пропорционален $2 \times M$. Однако, доступ к информации неудобен (ответ на вопрос «к каким вершинам ведут ребра из данной вершины» требует перебора элементов списка).

Примечание. *Для более эффективного решения задач элементы списка необходимо упорядочить лексикографически.*

4) Представление графа с помощью списков смежности. Эта структура, в которой каждой вершине $v \in V$ ставится в соответствие список $LIST[v]$. В него включаются (в произвольном порядке) ссылки на все смежные с v вершины.

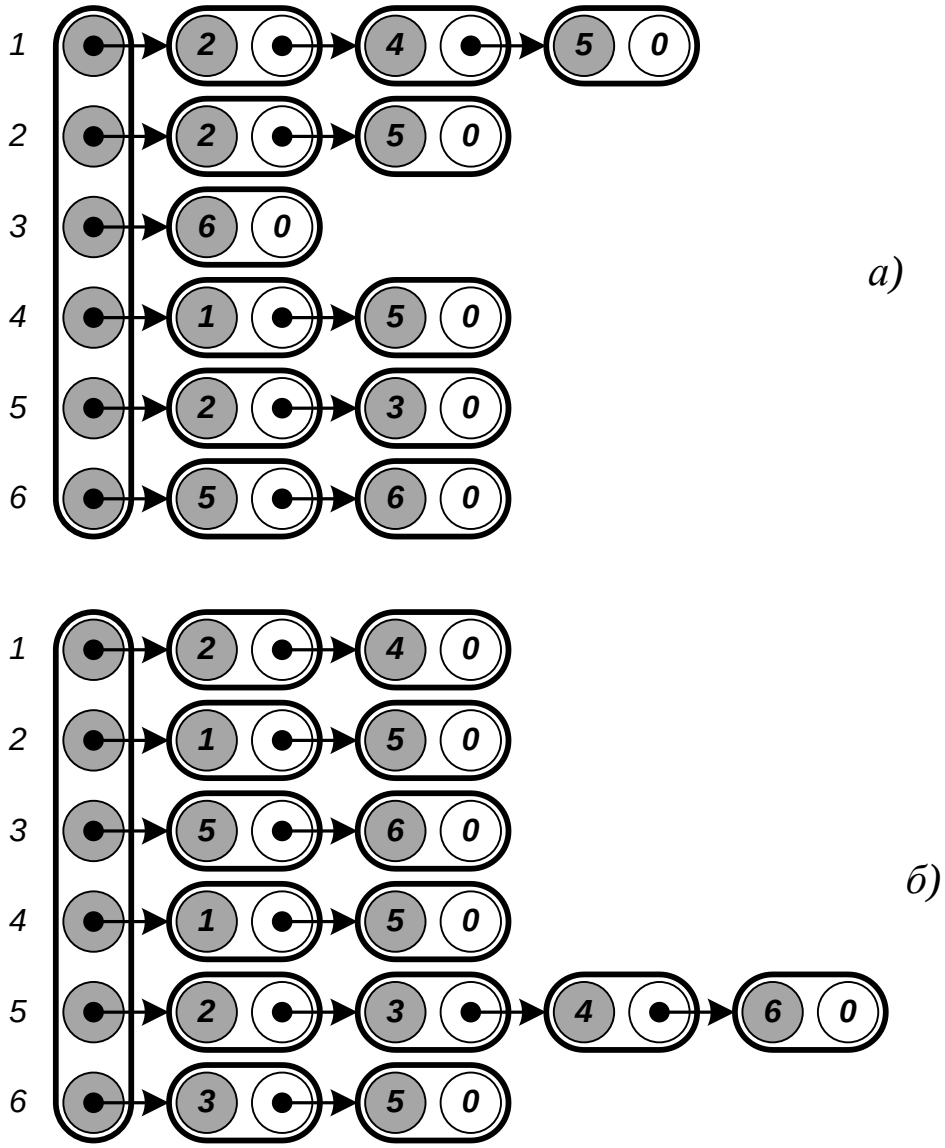


Рисунок 4.10 — Списки смежности: а) для графа с рисунка 4.6-а; б) для графа с рисунка 4.6-б

Основное достоинство списка смежности — простота модификации при изменении структуры графа. Объем памяти, необходимой для представления графа, будет иметь порядок $\boldsymbol{N + M}$.

4.3 Поиск в графах

Поиском в графах (по аналогии с деревьями) считается систематический перебор их вершин. Основные требования к нему следующие:

- алгоритм решения конкретной задачи должен *легко интегрироваться* с процедурой поиска;
- каждое ребро должно анализироваться *конечное число раз*;
- обработка информации для каждой вершины графа должна выполняться *один раз*.

Если структура графа не меняется, то наиболее полезными оказываются следующие виды поиска.

Поиск в глубину (*англ.* ***DFS — depth first search***) производится в следующем порядке:

- посещается в общем случае произвольная *новая* (еще не обработанная) вершина v. Она перестает быть *новой*, что отмечается ее свойством $done[v]$;
- выполняется *рекурсивное обращение* к процедуре поиска для каждой смежной с v *новой* вершины u_j, причем u_j также становится исходной точкой поиска. Обход всех ребер (v, u_j) производится с учетом установленного для них порядка;
- если новых смежных с v вершин не существует, то производится возврат в вершину, предшествующую v.

Рекурсивная процедура ***DFS*** выполнит *поиск в глубину* в произвольном графе, начиная от вершины v.

```
DFS(v)
  Выполнение действия над v.
  done[v] ← True
  FOR u ∈ LIST[v]
    DO IF NOT done[u]
         THEN DFS(u)
```

Устранить рекурсию можно с помощью стека, в котором запоминаются последовательность просмотренных вершин. Нерекурсивный алгоритм поиска в глубину представлен ниже (символ «⇐» обозначает операции включения и исключения).

```
DFS(v)
  STACK ← ∅
  Выполнение действия над v.
  STACK ⇐ v
  done[v] ← True
  WHILE STACK ≠ ∅
    DO v ⇐ STACK
       STACK ⇐ v
       FOR u ∈ LIST[v]
         DO IF NOT done[u]
              THEN break
       IF u = 0
         THEN v ⇐ STACK
         ELSE Выполнение действия над u.
              STACK ⇐ u
              done[u] ← True
```

На каждом шаге условного цикла в стеке ***STACK*** запоминается последняя обработанная вершина. Ее список инцидентности просматривается до обнаружения необработанной вершины (над ней выполняется необходимые действия) или до конца. После этого вершина удаляется из стека. *Чем раньше посещается вершина, тем позже она используется*.

Примечание. *Действия* $v \Leftarrow STACK$ *и* $STACK \Leftarrow v$ *определяют последнюю вершину в стеке без удаления из него.*

Поиск в ширину (*англ.* ***BFS — breadth first search***) начинается с произвольной новой вершины. После этого просматриваются все ее соседи, затем соседи соседей и т. д. Процедура ***BFS*** выполнит *поиск в ширину* в произвольном графе, начиная от вершины ***v***.

```
BFS(v)
  QUEUE ← ∅
  QUEUE ⇐ v
```

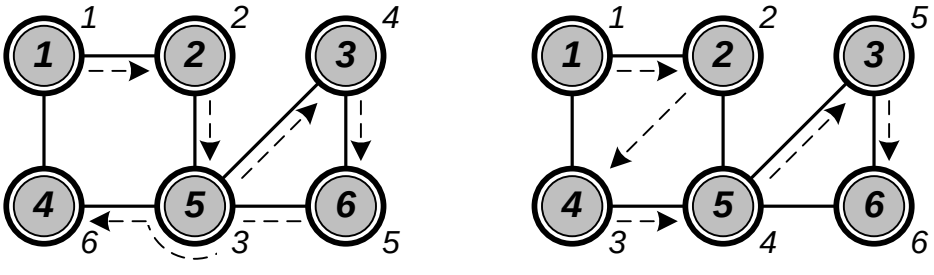
```
done[v] ← True
WHILE QUEUE ≠ ∅
  DO v ⇐ QUEUE
     Выполнение действия над v.
     FOR u ∈ LIST[v]
       DO IF NOT done[u]
            THEN QUEUE ⇐ u
                 done[u] ← True
```

Поиск основывается на замене стека очередью ***QUEUE***. *Чем раньше посещается вершина, тем раньше она используется*.

Фрагмент алгоритма, инициирующего поиск в глубину от всех необработанных вершин (возможно несвязного) графа, приведен ниже (для поиска в ширину достаточно заменить ***DFS*** на ***BFS***).

```
FOR v ∈ V
  DO done[v] ← False
FOR v ∈ V
  DO IF NOT done[v]
       THEN DFS(v)
```

Порядок обхода графа при поиске в глубину и в ширину показан на рисунке 4.11.



а) *б)*

Рисунок 4.11 — Очередность просмотра вершин графа: а) при поиске в глубину; б) при поиске в ширину

Оба вида поиска могут использоваться для *нахождения путей в графе* (для построения пути $u \to v$ достаточно начать обход из u и вести его до посещения v). Поиск в глубину является также универсальным средством решения большого числа практических задач.

4.4 Компоненты связности

Связность является отношением эквивалентности на множестве вершин вида: «u достижимо из v и v достижимо из u». Классы эквивалентности называются **компонентами связности**, их число равно k. (Граф, показанный на рисунке 4.1-а, имеет две компоненты $\{1,2,4,5\}$ и $\{3,6\}$.) Граф *связный*, если $k = 1$. Граф *несвязный*, если $k > 1$. Граф *вполне несвязный*, если $k = N, M = 0$ (все вершины изолированы). В связном неориентированном графе для любой пары вершин существует соединяющий их (простой) путь.

Между числом вершин, числом ребер и числом компонент связности существует взаимосвязь

$$N - k \le M \le \frac{1}{2} \cdot (N - k) \cdot (N - k + 1).$$

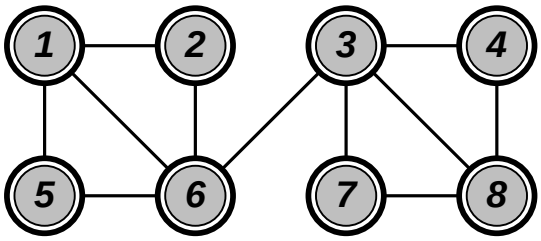
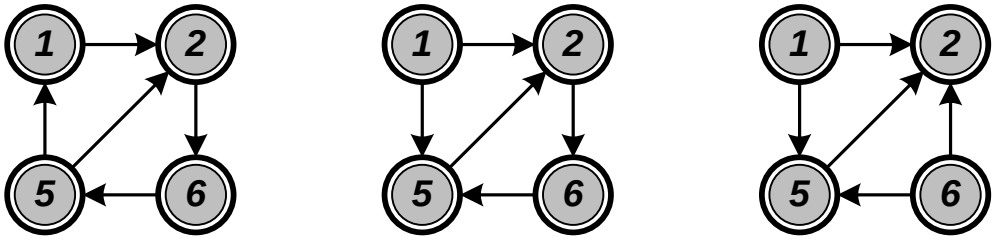


Рисунок 4.12 — Граф, имеющий точки сочленения и мост

Ребро, удаление которого из графа увеличивает число компонент связности, называется **мостом**. Вершина, удаление которой из графа вместе с инцидентными ребрами увеличивает число компонент связности графа, называется **точкой сочленения**. Связный

граф, не имеющий точек сочленения, называется **блоком**. (Граф, показанный на рисунке 4.12, имеет две точки сочленения ***3*** и ***6*** и один мост $\{3,6\}$.)

В ориентированном графе ***G*** отношение связности вершин несимметрично. Вершины ***u*** и ***v*** считаются *сильно связными*, если существуют пути из ***u*** в ***v*** и из ***v*** в ***u***. Вершины ***u*** и ***v*** считаются *односторонне связными*, если существуют пути либо из ***u*** в ***v***, либо из ***v*** в ***u***. Вершины ***u*** и ***v*** считаются *слабо связными*, если они связаны только в графе ***G'***, полученном из ***G*** после отказа от направленности дуг. Если все вершины в ориентированном графе сильно (односторонне) связаны, то он называется *сильно (односторонне) связным* (см. рисунок 4.13).



а) *б)* *в)*

Рисунок 4.13 — Ориентированные графы с а) сильной, б) односторонней и в) слабой связностью

Для определения компонент связности графа используется поиск в глубину. Принадлежность вершины ***v*** компоненте распознается по ее «цвету» ***color***. С самого начала граф «не раскрашен» ($color[v] = 0, v \in V$). В процессе поиска все вершины, достижимые от начальной, раскрашиваются в один цвет. Число компонент связности графа определится числом использованных цветов.

Примечание. *Начальной может стать любая вершина, еще не отнесенная ни к одной компоненте связности (т. е. еще не «раскрашенная»).*

Модифицированная процедура ***DFS*** определит все вершины, принадлежащие одной компоненте связности (переменная ***k*** фиксирует число компонент связности).

```
DFS(v)
  color[v] ← k
  FOR u ∈ LIST[v]
    DO IF color[u] = 0
         THEN DFS(u)
```

Фрагмент алгоритма, инициирующего определение числа компонент связности графа, представлен ниже.

```
FOR v ∈ V
  DO color[v] ← 0
k ← 0
FOR v ∈ V
  DO IF color[v] = 0
       THEN k ← k + 1
            DFS(v)
```

4.5 Стягивающие деревья

Стягивающим деревом (или **каркасом**) графа $G = \langle V, E \rangle$ называется его произвольный связный подграф без циклов $T = \langle V', E' \rangle$, где $V' = V$ и $E' \subseteq E$. Ребра такого дерева называются *ветвями*, остальные ребра графа — *хордами*. *Стягивающее дерево всегда будет содержать* $N - 1$ *ветвь.*

Простой способ построения каркаса — поиск в графе. Достижение из вершины v новой вершины u вызывает включение в T ветви $\{v, u\}$. Построенное дерево будет:

- *связным*, так как всегда существует путь $v \to u$;
- *ациклическим*, так как одна из вершин $\{v, u\}$ всегда новая;

- *стягивающим*, так как просматриваются все вершины.

Модифицированная процедура ***DFS*** включит все вершины в стягивающее дерево графа.

```
DFS(v)
  done[v] ← True
  FOR u ∈ LIST[v]
    DO IF NOT done[u]
         THEN T ⇐ {v, u}
              DFS(u)
```

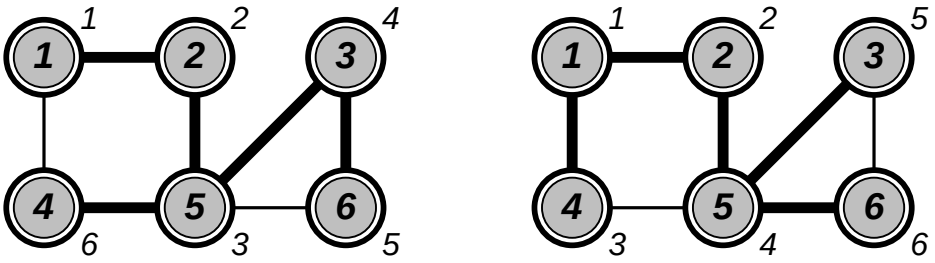
Эту же задачу можно решить с помощью поиска в ширину.

```
BFS(v)
  QUEUE ← ∅
  QUEUE ⇐ v
  done[v] ← True
  WHILE QUEUE ≠ ∅
    DO v ⇐ QUEUE
       FOR u ∈ LIST[v]
         DO IF NOT done[u]
              THEN T ⇐ {v, u}
                   QUEUE ⇐ u
                   done[u] ← True
```

Фрагмент алгоритма, строящего каркас графа от его произвольной вершины поиском в глубину, приведен ниже (для решения задачи поиском в ширину достаточно заменить ***DFS*** на ***BFS***).

```
FOR v ∈ V
  DO done[v] ← False
T ← ∅
DFS(v)
```

Стягивающие деревья, построенные различными способами, показаны на рисунке 4.14.



а) *б)*

Рисунок 4.14 — Построение стягивающего дерева графа от вершины ***1***: а) поиском в глубину; б) поиском в ширину

Каждому ребру $\{u,v\}$ графа может быть приписан вес $w(u,v)$, количественно отражающий некоторую его характеристику (например, длину). Тогда можно поставить задачу построения **оптимального каркаса**, для которого выполнялось бы условие

$$\sum_{\{u,v\}\in T} w(u,v) \to min\,.$$

Задача построения оптимального каркаса решается пошагово. На каждом шаге выбирается наиболее предпочтительное ребро. Если оно не образует циклов с уже сформированными ветвями, его включают в каркас.

Алгоритм Крускала (*Kruskal*) строит минимальный каркас T на основе леса. Сначала множество T пусто, и каждая вершина v образует собственную компоненту связности («раскрашивается» в собственный цвет $color[v]$). Ребра включаются в каркас в порядке возрастания их весов. Чтобы добавляемое ребро $\{u,v\}$ не образовывало цикла, вершины u и v должны принадлежать разным компонентам связности. После включения в каркас «безопасного» ребра связанные им компоненты объединяются («раскрашиваются» в один цвет). Работа алгоритма показана на рисунке 4.15.

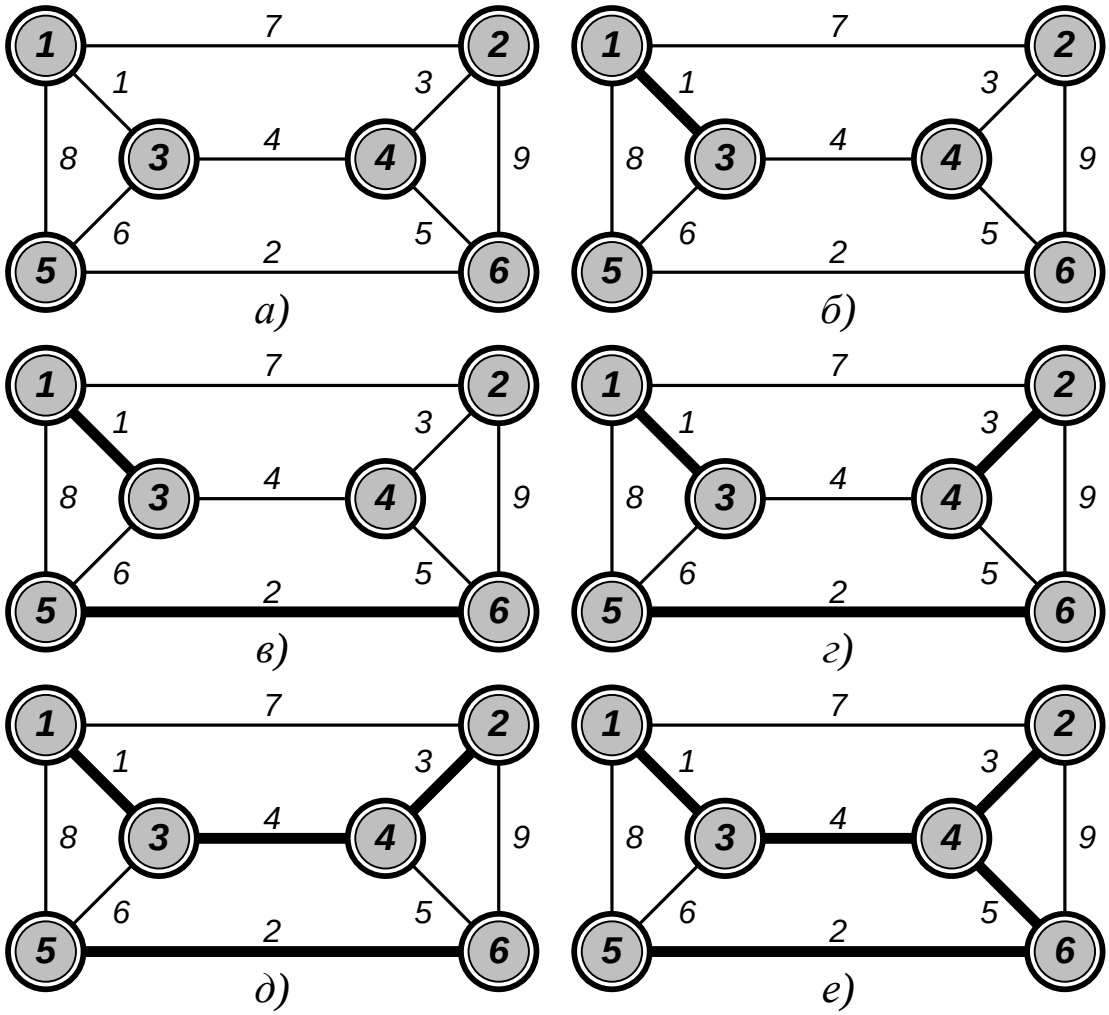


Рисунок 4.15 — Построение минимального каркаса алгоритмом Крускала

Алгоритм Крускала предусматривает выполнение следующих действий.

```
FOR v ∈ V
  DO color[v] ← v
T ← ∅
FOR {u, v} ∈ E
  Отбор ребра {u, v} по приоритету.
  IF color[u] ≠ color[v]
    THEN T ⇐ {u, v}
         FOR p ∈ V
```

```
DO IF color[p] = color[v]
     THEN color[p] ← color[u]
```

Алгоритм Прима (*Prim*) строит минимальный каркас ***T*** от произвольного корня ***r***. Его работа завершается за $N-1$ шаг. На каждом шаге в ***T*** включается ближайшая к нему *новая* вершина (это исключает появление циклов). Работа алгоритма показана на рисунке 4.16.

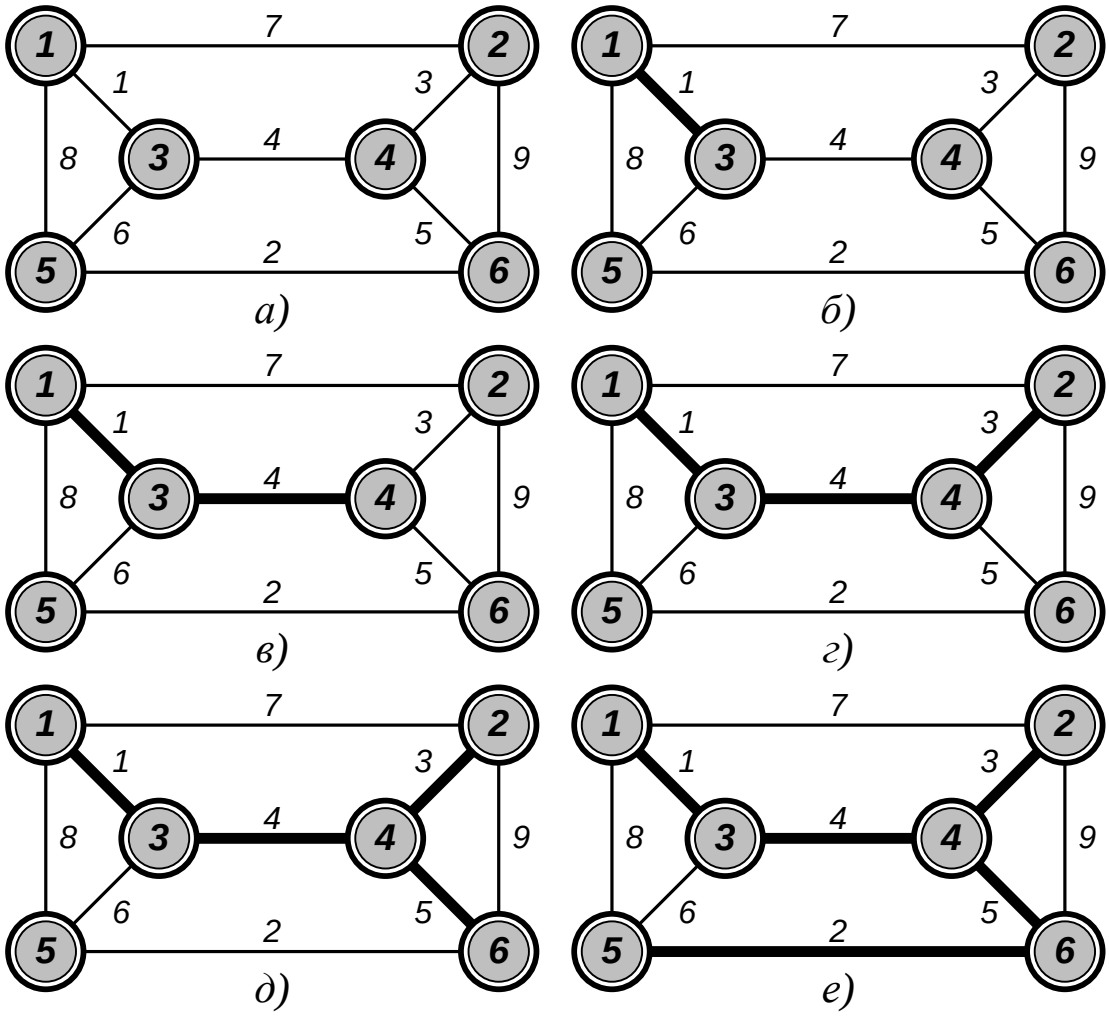


Рисунок 4.16 — Построение минимального стягивающего дерева от вершины ***1*** алгоритмом Прима

Алгоритм предусматривает выполнение следующих действий.

```
FOR v ∈ V
  DO done[v] ← False
done[r] ← True
FOR k ∈ [1, N − 1]
  DO min ← ∞
     FOR v ∈ V
       DO IF NOT done[v]
            THEN FOR u ∈ V
                   DO IF done[u] AND w(u, v) < min
                        THEN min ← w(u, v)
                             i ← u
                             j ← v
  T ⇐ {i, j}
  done[j] ← True
```

4.6 Циклы

Для любого графа можно определить минимальное **множество фундаментальных циклов**, ни один из которых не может быть получен линейной комбинацией остальных. Любой цикл графа может быть представлен симметрический разностью♣ (сложением по модулю ***2***) некоторого числа элементов этого множества. Количество фундаментальных циклов называется *циклическим рангом* или *цикломатическим числом* графа. Оно равно $M - N + 1$.

Фундаментальные циклы графа $G = \langle V, E \rangle$ строятся относительно его каркаса T. Добавление в T любого ребра из множества $E \setminus T$ порождает ровно один цикл (см. рисунок 4.17).

♣ Симметрической разностью двух множеств A и B называется операция $A \oplus B = (A \cup B) \setminus (A \cap B)$.

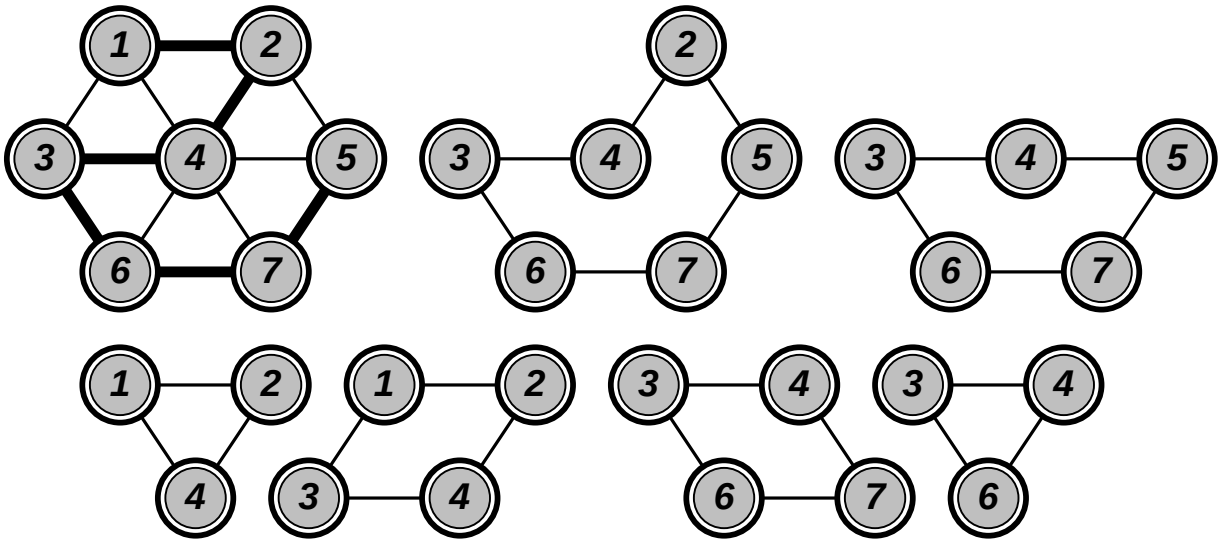


Рисунок 4.17 — Каркас графа, построенный поиском в глубину от вершины ***1***, и множество фундаментальных циклов графа

Модифицированный алгоритм ***DFS*** находит множество фундаментальных циклов графа (переменная ***k*** подсчитывает вершины графа, переменная ***s*** фиксирует число элементов стека).

```
DFS(v)
  k ← k + 1
  ord[v] ← k
  STACK ⇐ v
  s ← s + 1
  FOR u ∈ LIST[v]
    DO IF ord[u] = 0
         THEN DFS(u)
         ELSE IF u ≠ STACK[s − 1] AND ord[u] < ord[v]
                THEN Отображение STACK[j]
                     от j = s до STACK[j] = u
  v ⇐ STACK
  s ← s − 1
```

В процессе поиска вершины нумеруются в том порядке, в котором они включаются в каркас (что отмечается свойством ***ord***). Каж-

дая новая вершина ***v*** помещается в стек ***STACK***, из которого она удаляется после использования. Стек всегда содержит последовательность вершин от ***v*** до корня стягивающего дерева. Анализируемое ребро $\{v,u\}$ замыкает цикл, если вершина ***u*** рассматривалась ранее ($ord[u] \neq 0$), и она не предшествует ***v*** ($ord[u] < ord[v]$). Цикл, замыкаемый ребром $\{v,u\}$, представляется верхней группой элементов стека.

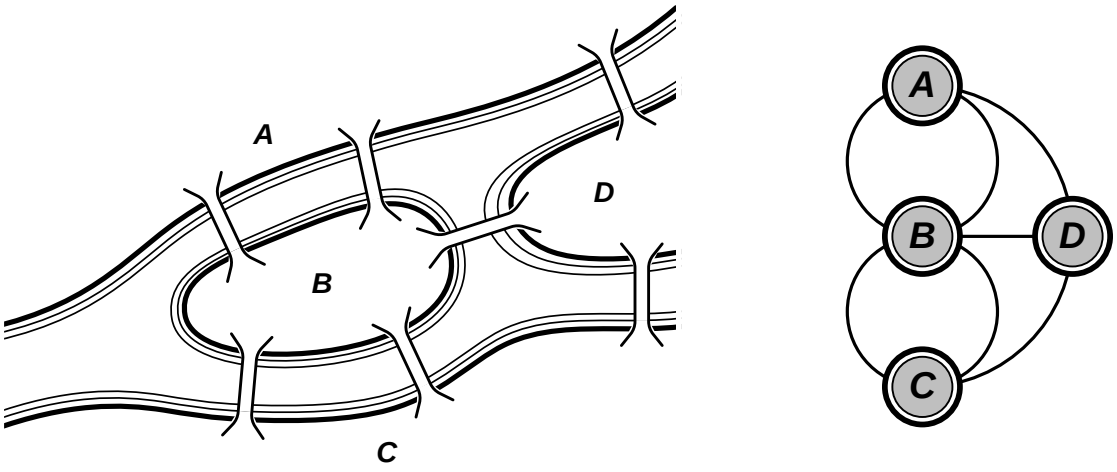
Фрагмент алгоритма, инициирующего построение каркаса графа поиском в глубину от произвольной вершины и нахождение его фундаментальных циклов, приведен ниже.

```
FOR v ∈ V
  DO ord[v] ← 0
k ← 0
s ← 0
STACK ← ∅
DFS(v)
```

Цикл, проходящий через каждое ребро связного графа один раз, называется **эйлеровым**. Чтобы такой цикл существовал, в графе *не должно быть вершин нечетной степени*.

Примечание. *Происхождение названия цикла восходит к старинной математической задаче о кёнигсбергских мостах, в которой требовалось узнать, как можно пройти по всем мостам Кёнигсберга, не проходя ни по одному из них дважды (см. рисунок 4.18). Леонардом Эйлером (Euler) было доказано, что эта задача не имеет решения*

Для построения эйлерового цикла используется модифицированный алгоритм ***DFS*** (последовательность вершин сохраняется в стеке ***STACK***).



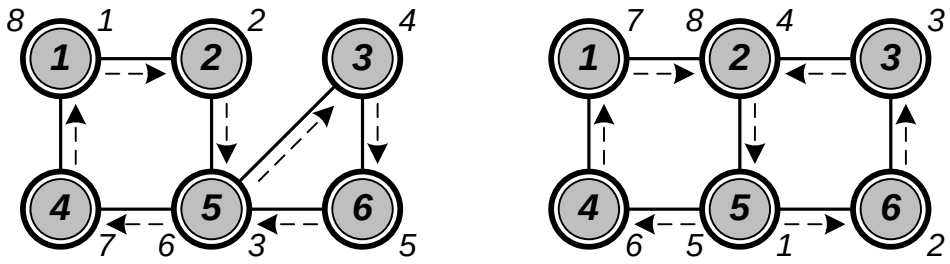
а) *б)*

Рисунок 4.18 — Иллюстрация к задаче о кенигсбергских мостах: а) план города; б) граф

```
DFS(v)
  FOR u ∈ LIST[v]
    DO LIST[v] ← LIST[v] \ {u}
       LIST[u] ← LIST[u] \ {v}
       DFS(u)
  STACK ⇐ v
```

Поиск начинается от произвольной вершины ***v***. Образующие путь ребра удаляются из графа. Если дальнейшее удлинение пути невозможно (что распознается по условию $LIST[v] = \varnothing$), последняя пройденная вершина переносится в ***STACK***, и поиск продолжается от предыдущей вершины. Пример эйлерового цикла показан на рисунке 4.19-а.

Если связный граф содержит не более двух вершин нечетной степени, то в нем существует *эйлеров путь*, проходящий через каждое ребро по одному разу. Его началом и концом всегда являются вершины нечетной степени (пример эйлерового пути показан на рисунке 1.19-б).



а) *б)*

Рисунок 4.19 — Специальные случаи обхода графа: а) эйлеров цикл; б) эйлеров путь

Простой цикл, проходящий через каждую вершину (но не обязательно через каждое ребро) связного графа один раз, называется **гамильтоновым** (по названию известной задачи о кругосветном путешествии, придуманной Уильямом Гамильтоном (*Hamilton*)). *Гамильтонов путь* определяется очевидным образом.

Простые условия существования гамильтоновых циклов неизвестны. Универсальный алгоритм их построения в произвольном графе за разумное время отсутствует. Задача решается перебором возможных вариантов. Для сокращения числа проб используют технику *«программирования с возвратом»*. Ее идея следующая.

Пусть искомое решение имеет вид последовательности вершин $\langle x_1,...,x_n \rangle$, которая вначале является пустой. Если уже имеется частное решение $\langle x_1,...,x_{k-1} \rangle$, и обнаруживается новая допустимая вершина x_k, то последовательность расширяется до $\langle x_1,...,x_k \rangle$. В противном случае производится возврат (*англ.* ***backtracking***) к найденному ранее частному решению и продолжается поиск приемлемой, но еще не рассмотренной вершины x_k.

Рекурсивная процедура ***HAMILTON*** служит для расширения последовательности решений.

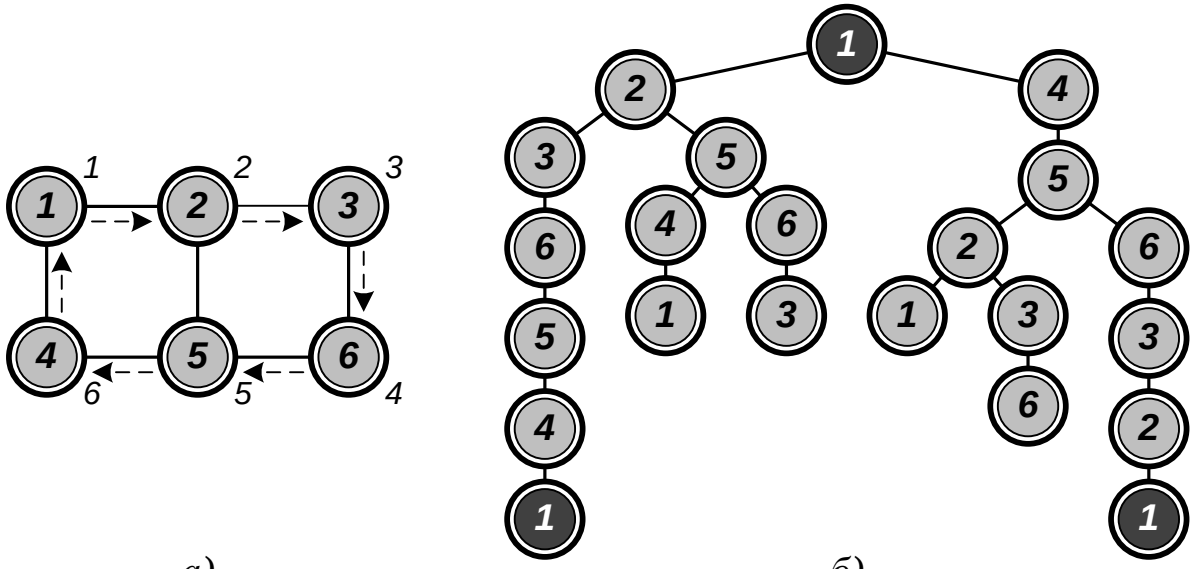
```
HAMILTON(k)
  FOR y ∈ LIST[x[k − 1]]
    DO IF k = N + 1 AND y = v0
```

```
        THEN Использовать <x[1],... x[n], v0>
        ELSE IF NOT done[y]
               THEN x[k] ← y
                    done[y] ← True
                    HAMILTON(k + 1)
                    done[y] ← False
```

Фрагмент алгоритма, инициирующего построение всех гамильтоновых циклов в графе от вершины v_0, приведен ниже.

```
FOR v ∈ V
  DO done[v] ← False
x[1] ← v0
done[v0] ← True
HAMILTON(2)
```

Пример гамильтонова цикла показан на рисунке 4.20-а. Технология его построения путем поиска в дереве возможных решений отражена на рисунке 4.20-б.



а) *б)*

Рисунок 4.20 — Гамильтонов цикл в графе: а) порядок обхода вершин; б) дерево поиска всех гамильтоновых циклов графа

4.7 Кратчайшие пути

Пусть каждой дуге (u,v) ориентированного графа приписан вес $a(u,v)$, определяющий ее длину (если отсутствует путь из u в v, то $a(u,v)=\infty$; если отсутствует петля (u,u), то $a(u,u)=\infty$). Тогда можно поставить задачу поиска *кратчайшего пути между фиксированными вершинами* s (*англ.* ***source***) и t (*англ.* ***target***). Если $s \in V$ — начало последовательности вершин $\langle v_0, v_1, v_2 ..., v_k \rangle$, а $t \in V$ — ее конец, то длина такого пути (или *расстояние от* s *до* t) определится как сумма

$$d(s,t) = \sum_{i=1}^{k} a(v_{i-1}, v_i).$$

Если не существует ни одного пути из s в t, то $d(s,t)=\infty$.

Если каждый контур графа имеет положительную длину, то кратчайший путь всегда будет простым. Если в графе существует контур отрицательной длины, расстояние между некоторыми вершинами становится неопределенным.

Кратчайшие пути находятся следующим образом. Для произвольных вершин s и t (где $s \neq t$) существует такая вершина v, что $d(s,t) = d(s,v) + a(v,t)$ (т. е. v — предпоследняя вершина произвольного кратчайшего пути из s в t). Далее можно найти вершину u, для которой $d(s,v) = d(s,u) + a(u,v)$, и т. д. При положительной длине всех контуров графа созданная таким образом последовательность $t, v, u, ...$ не содержит повторений и оканчивается вершиной s. При обращении очередности она определяет кратчайший путь из s в t.

Большинство алгоритмов нахождения кратчайших путей между двумя фиксированными вершинами используют матрицу весов дуг $A[u,v]$ ($u \in V, v \in V$). С ее помощью вычисляются некоторые верхние ограничения $D[v]$ на расстояния от s до всех вершин $v \in V$.

Каждый раз, когда устанавливается, что $D[v] > D[u] + A[u,v]$, оценка $D[v]$ улучшается: $D[v] \leftarrow D[u] + A[u,v]$. Процесс прерывается, когда дальнейшее улучшение ни одного из ограничений невозможно. Очевидно, что значение каждой из переменных $D[v]$ равно расстоянию $d(s,v)$.

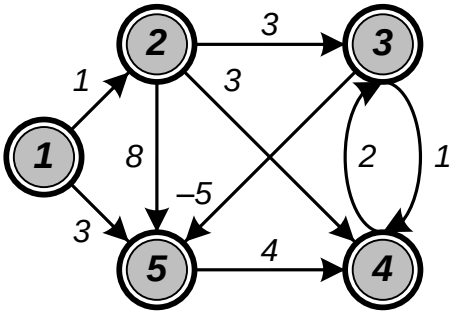
Ниже приведен алгоритм Форда (*Ford*) и Беллмана (*Bellman*) вычисления кратчайших расстояний $D[v]$ от источника s до всех вершин графа $v \in V$. *Допустимы дуги с отрицательным весом, но не контура отрицательной длины*. Для каждой вершины v определяется номер $P[v]$ ее предшественницы в кратчайшем пути, который используется при построении этого пути.

```
FOR v ∈ V
  DO D[v] ← A[s, v]
     P[v] ← s
D[s] ← 0
P[s] ← 0
FOR k ∈ [1, N − 2]
  DO FOR v ∈ V / {s}
       DO FOR u ∈ V
            DO IF D[v] > D[u] + A[u, v]
                 THEN D[v] ← D[u] + A[u, v]
                      P[v] ← u
```

Отобразить найденный кратчайший путь из t в s можно с помощью следующего алгоритма.

```
v ← t
WHILE P[v] ≠ 0
  DO Отобразить v
     v ← P[v]
Отобразить s
```

Работа алгоритма Форда-Беллмана проиллюстрирована на рисунке 4.21 (исходной является вершина ***1***).



$$A = \begin{pmatrix} \infty & 1 & \infty & \infty & 3 \\ \infty & \infty & 3 & 3 & 8 \\ \infty & \infty & \infty & 3 & -5 \\ \infty & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & 4 & \infty \end{pmatrix}$$

k	*D*[1]	*D*[2]	*D*[3]	*D*[4]	*D*[5]
0	***0***	***1***	∞	∞	***3***
1	***0***	***1***	***4***	***4***	***–1***
2	***0***	***1***	***4***	***3***	***–1***
3	***0***	***1***	***4***	***3***	***–1***

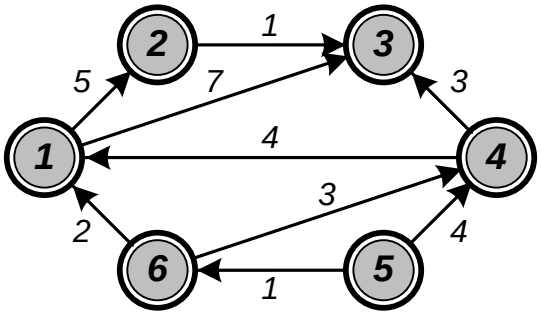
Рисунок 4.21 — Работа алгоритма Форда-Беллмана

Если *веса всех дуг неотрицательны*, то можно использовать эффективный алгоритм Дейкстры (*Dijkstra*).

```
FOR v ∈ V
  DO D[v] ← A[s, v]
     P[v] ← s
D[s] ← 0
P[s] ← 0
Q ← V \ {s}
WHILE Q ≠ ∅
  DO u ← Произвольная вершина, определяемая из условия
          D[u] = min{D[v], v ∈ Q}
     Q ← Q \ {u}
     FOR v ∈ Q
       DO IF D[v] > D[u] + A[u, v]
```

```
            THEN D[v] ← D[u] + A[u, v]
                 P[v] ← u
```

Работа алгоритма проиллюстрирована на рисунке 4.22 (исходной является вершина ***5***)



$$A = \begin{pmatrix} \infty & 5 & 7 & \infty & \infty & \infty \\ \infty & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ 4 & \infty & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & 4 & \infty & 1 \\ 2 & \infty & \infty & 3 & \infty & \infty \end{pmatrix}$$

k	$D[1]$	$D[2]$	$D[3]$	$D[4]$	$D[5]$	$D[6]$
0	∞	∞	∞	***4***	***0***	***1***
1	∞	∞	∞	***4***	***0***	***1***
2	***3***	∞	∞	***4***	***0***	***1***
3	***3***	***8***	***10***	***4***	***0***	***1***
4	***3***	***8***	***7***	***4***	***0***	***1***
5	***3***	***8***	***7***	***4***	***0***	***1***

Рисунок 4.22 — Работа алгоритма Дейкстры

Эффективный алгоритм Флойда (*Floyd*) и Уоршалла (*Warshall*), использующий технику динамического программирования, вычисляет **кратчайшие расстояния между всеми парами вершин** ориентированного графа (*допустимы дуги с отрицательным весом, но не контура отрицательной длины*). Матрица $D[u,v]$ хранит ограничения на расстояния между вершинами. Матрица предшествования $P[u,v]$ используется для построения кратчайших путей.

```
FOR i ∈ [1, N]
  DO FOR j ∈ [1, N]
       DO D[i, j] ← A[i, j]
          IF A[i, j] ≠ ∞ AND i ≠ j
            THEN P[i, j] ← i
            ELSE P[i, j] ← 0
     D[i, i] ← 0
     P[i, i] ← 0
FOR k ∈ [1, N]
  DO FOR i ∈ [1, N]
       DO FOR j ∈ [1, N]
            DO IF D[i, j] > D[i, k] + D[k, j]
                 THEN D[i, j] ← D[i, k] + D[k, j]
                      P[i, j] ← P[k, j]
```

Примечание. *Отобразить найденный кратчайший путь из* t *в* s *можно с помощью приведенного выше алгоритма, заменив элемент вектора* $P[v]$ *элементом матрицы* $P[s,v]$.

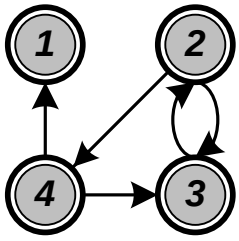
Алгоритм Флойда-Уоршалла можно использовать для решения вопроса о существовании пути между любой парой вершин ориентированного графа. Отношение достижимости вершин называется **транзитивным замыканием** графа. При его вычислении считают, что все дуги имеют единичный вес. Если существует путь из u в v, то $D[u,v] < N$, иначе $D[u,v] = \infty$.

Примечание. *На практике вместо арифметических действий используют логические операции. Если существует путь из* u *в* v, *то* $D[u,v] = 1$, *иначе* $D[u,v] = 0$.

```
FOR i ∈ [1, N]
  DO FOR j ∈ [1, N]
       DO IF A[i, j] ≠ ∞ OR (i, j) ∈ E
            THEN D[i, j] ← 1
```

```
            ELSE D[i, j] ← 0
FOR k ∈ [1, N]
  DO FOR i ∈ [1, N]
       DO FOR j ∈ [1, N]
            DO D[i, j] ← D[i, j] ∨ (D[i, k] ∧ D[k, j])
```

Пример графа и матрицы, вычисленные с помощью этого алгоритма, показаны на рисунке 4.23.



$$D^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}, \quad D^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}, \quad D^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix},$$

$$D^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}, \quad D^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}.$$

Рисунок 4.23 — Ориентированный граф и матрицы, вычисленные алгоритмом транзитивного замыкания

При положительных весах ребер задача поиска кратчайших путей в неориентированном графе легко сводится к аналогичной задаче для ориентированного графа. Достаточно заменить каждое ребро $\{u,v\}$ двумя дугами (u,v) и (v,u) с таким же весом. Однако при

наличии ребер с неположительным весом это может приводить к возникновению контуров с неположительной длиной.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие структуры данных считаются многосвязными? Как определяется граф? В чем отличие ориентированного графа от неориентированного? Как можно выполнить их взаимное преобразование?
2. Что считается путем в графе? Как определяется длина пути? Какой путь является простым?
3. Что считается циклом в графе? Какой цикл является простым?
4. Что называется подграфом, суграфом, надграфом, сверхграфом, дополнением?
5. Какие графы считаются изоморфными?
6. Какие графы считаются плоскими?
7. Какие графы считаются тривиальными?
8. Какие графы считаются полными?
9. Какие графы считаются двудольными?
10. Какие графы обозначаются C_k?
11. Как представить граф с помощью матриц инцидентности и смежности?
12. Как представить граф с помощью списков инцидентности и смежности?
13. Как осуществляется поиск по графу в глубину?
14. Как осуществляется поиск по графу в ширину?
15. Что понимается под связностью ориентированных и неориентированных графов? В чем отличие компонент сильной, односторонней и слабой связности?
16. Как определить число компонент связности графа?
17. Что называется стягивающими деревьями графа? Как они строятся?

18. Что называется экстремальными стягивающими деревьями графа? Как они строятся?

19. Что считается фундаментальными циклами графа? Как находится их множество? Что такое цикломатическое число графа?

20. Какой цикл в графе считается эйлеровым? Каковы условия его существования? Как он строится? Что считается эйлеровым путем в графе?

21. Какой цикл в графе считается гамильтоновым? Как он строится? Что считается гамильтоновым путем в графе?

22. По какому принципу отыскиваются кратчайшие пути между фиксированными вершинами графа?

23. Как работает алгоритм Форда-Беллмана поиска кратчайшего пути между фиксированными вершинами графа?

24. Как работает алгоритм Дейкстры поиска кратчайшего пути между фиксированными вершинами графа?

25. Как работает алгоритм Флойда-Уоршалла поиска кратчайших путей между всеми парами вершин графа?

26. Что считается транзитивным замыканием графа? Как оно строится?

5 РЕАЛИЗАЦИЯ СТРУКТУР ДАННЫХ СРЕДСТВАМИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

5.1 Структуры данных в языке Pascal

5.1.1 Массивы и строки

В языке ***Pascal*** массив определяется следующим образом

```
Имя-массива: array [Тип-индекса] of Тип-элемента;
```

Примечание. *Точка с запятой — элемент описания.*

Имя массива является идентификатором. В качестве типа индекса могут использоваться упорядоченные типы (кроме ***longint***), ограниченные типы (диапазоны) и типы-перечисления. Для многомерных массивов допускается указывать типы индексов через запятую. Типом элементов может быть любой тип языка (в том числе структура данных). Количество элементов массива определяется числом значений индекса. *Типы индексов и элементов должны выбираться так, чтобы размер массива не превышал* ***64 К.***

В блоке констант изображение массива строится из списка изображений его элементов, которые должны соответствовать фактическому описанию структуры *по количеству и типу*. Многомерные массивы считаются образованными структурами меньшей размерности. Их изображения формируются с учетом порядка размещения массивов в машинной памяти при их векторизации.

Пример*: объявление и инициализация массивов (матрица представляется в виде массива строк).*

```
type
  TLogical = array [boolean] of boolean;
  TVector = array [1 .. 5] of integer;
  TMatrix = array [1 .. 2, 1 .. 3] of double;
```

```
const
  confirm: TLogical = (False, True);
  inverse: TLogical = (True, False);
  x: TVector = (9, 7, 5, 3, 1);
  a: TMatrix = ((1.0, 2.0, 3.0), (2.0, 3.0, 4.0));
```

• • •

Элемент массива считается переменной с индексами

***Имя-массива*[*Список-индексных-выражений*].**

Типы индексных выражений должны *строго соответствовать описанию индексов* (в том числе и по диапазонам значений).

Все действия над массивом выполняются по отдельности над каждым его элементом. Набор допустимых операций *полностью определяется типом элемента*.

Пример: *вычисление произведения матриц* $\boldsymbol{C = A \cdot B}$.

```
var
  i, j, ij: integer;
  a: array [1 .. k, 1 .. m] of double;
  b: array [1 .. m, 1 .. n] of double;
  c: array [1 .. k, 1 .. n] of double;
...
  for i := 1 to k do
    for j := 1 to n do begin
      c[i, j] := 0.0;
      for ij := 1 to m do
        c[i, j] := c[i, j] + a[i, ij] * b[ij, j];
    end;
{ Вывод матрицы в общепринятом виде. }
  for i := 1 to k do begin
    for j := 1 to n do Write(c[i, j]: 4);
    WriteLn;
  end;
...
```

• • •

Пример: *подсчет числа вхождений символов в строку.*

```
var
  i: char; j: byte;
  workString: array [byte] of char;
  count: array [char] of byte;
...
{ Сброс начальных значений счетчиков. }
for i := #0 to #255 do
  count[i] := 0;
{ Подсчет числа вхождений символов. }
for j := 0 to 255 do
  Inc(count[workString[j]]);
{ Вывод результатов. }
for i := #0 to #255 do
  WriteLn(count[i]);
...
```

• • •

Над массивами в целом можно выполнять только операции присваивания, в которых производится их *поэлементное копирование*. Типы массивов, участвующих в присваивании, должны быть *эквивалентны*.

Пример: *копирование строк матриц.*

```
type
  TVector = array [1 .. 5] of double;
var
  a, b: array [1 .. 3] of TVector;
  c: array [1 .. 3] of array [1 .. 5] of double;
...
b[2] := a[1]; { Допустимо. Типы a и b эквивалентны.}
c[3] := a[1]; { Недопустимо! Тип c не эквивалентен a и b. }
...
```

• • •

В процедуры и функции можно передать как копию массива, так и ссылку на него. Правила языка требуют именной эквивалентности типов формальных и фактических параметров, поэтому в заголовках подпрограмм необходимо использовать синонимы структурных типов. Функция не может возвращать массив в качестве своего результата, однако получение ссылки на него возможно.

Пример: *вычисление произведения квадратных матриц* $\boldsymbol{C = A \cdot B}$ *с помощью функции.*

```
const
  n = 2;

type
  TMatrix = array [1 .. n, 1 .. n] of integer;
  PMatrix = ^TMatrix;

const
  a: TMatrix = ((1, 2), (3, 4));
  b: TMatrix = ((1, 3), (2, 4));

var
  i, j: integer;
  c: PMatrix;

{ Функция умножения квадратных матриц. }
function Mul(a, b: PMatrix): PMatrix;
var
  i, j, ij: integer;
  c: PMatrix;
begin
  New(c);
  for i := 1 to n do
    for j := 1 to n do begin
      c^[i, j] := 0;
      for ij := 1 to n do
        c^[i, j] := c^[i, j] + a^[i, ij] * b^[ij, j];
```

```
    end;
  Mul := c;
end;

begin
  c := Mul(@a, @b);
{ Вывод матрицы в общепринятом виде. }
  for i := 1 to n do begin
    for j := 1 to n do Write(c^[i, j]: 4);
    WriteLn;
  end;
  Dispose(c);
end.
```

• • •

Для работы с одномерными массивами символов, индексы которых заданы в виде диапазона с начальным единичным значением, язык ***Pascal*** располагает дополнительными средствами:

- их разрешается изображать в виде строки, длина которой должна *точно соответствовать* объявленному размеру;
- их разрешается сравнивать по правилам, принятым для строк (элементы сравниваются последовательно);
- их имена разрешается использовать в операторах вывода; при выполнении этих действий возможно объединение массивов с помощью операции «+».

Пример: *использование массивов символов.*

```
const
  s1: array [1 .. 15] of char = 'Длинная строка.';
var
  s2: array [1 .. 16] of char;
...
s2 := 'Короткая строка.';
if s1 > s2
  then WriteLn(s1)
  else WriteLn(s2);
```

```
WriteLn(s1 + ' ' + s2 + ' Они объединены.');
...
```

• • •

Разновидностью массивов являются строки заранее определенной длины (см. п. 2.2), описание которых имеет вид

***Имя-строки*: string [*Константное-выражение*];**

Константное выражение должно принимать значение в диапазоне ***1 .. 255***. Если этот параметр не задан, длина строки считается равной ***255*** символов.

Строке можно присвоить значение до начала работы программы. Ее фактическая длина определяется по изображению.

***Пример**: задание строк через блок констант.*

```
const
  anyString: string = 'All right!'#10#13'Isn''t it?';
  anotherString: string[80] = 'All right now.'^G^G^G;
```

• • •

Строки могут участвовать в ограниченном числе операций.

Операция	Особенность выполнения
:=	Посимвольное копирование. Длина строки-приемника устанавливается равной длине строки-источника. Если она превышает объявленную длину, происходит усечение до нужного размера.
+	Конкатенация (объединение). Если образуется строка длиннее ***255*** символов, выполняется ее усечение до нужного размера.
> < = <>	Посимвольное сравнение. Выполняется слева направо с учетом упорядоченности кодов симво-

Операция	Особенность выполнения
<= >=	лов. Более короткая строка «меньше» более длинной строки с такими же начальными символами

Если в операции одновременно участвуют несколько строк, то *эквивалентности их типов не требуется*.

***Пример**: операции над строками.*

```
var
  longString: string[20];
  shortString: string[5];
  tinyString: string[2];
...
shortString := 'ABCDE';
WriteLn(shortString); { Выводится ABCDE. }
tinyString := shortString;
WriteLn(tinyString); { Выводится AB. }
if shortString > tinyString
  then WriteLn('Более длинная строка ', shortString);
longString := shortString + tinyString;
WriteLn(longString); { Выводится ABCDEAB. }
...
```

• • •

Возможен доступ к отдельному символу строки как к элементу одномерного массива. С нулевым байтом строки можно работать по общим правилам. При обращениях к нему *фактическая длина строки не изменяется.*

***Пример**: операции над элементами строки.*

```
var
  workString: string; i: integer;
...
```

```
workString := 'A'; { Фактическая длина строки 1. }
for i := 2 to 5 do
  workString[i] := 'A'; { Длина строки не меняется. }
WriteLn(workString); { Выводится A. }
workString[0] := #5; { Фактическая длина строки 5. }
WriteLn(workString); { Выводится AAAAA. }
workString := 'A'; { Фактическая длина строки 1. }
{ Длина строки модифицируется при объединении. }
for i := 2 to 5 do
  workString := workString + 'A';
WriteLn(workString); { Выводится AAAAA. }
...
```

• • •

Ввод и вывод строк реализуется с помощью процедур ***Read***, ***ReadLn***, ***Write*** и ***WriteLn***. При работе со строками могут также использоваться стандартные процедуры и функции языка.

Функции и процедуры	Результат
Length(s)	Значение фактической длины строки **s**.
Pos(s1,s2)	Номер символа строки ***s2***, с которого начинается последовательность ***s1***, или нуль, если искомый фрагмент в строке отсутствует.
Copy(s,k,n)	Подстрока длиной ***n*** символов, выделенная из строки ***s***, начиная с позиции ***k***. Если ***k*** больше размера ***s***, возвращается пустая строка. Если необходимых символов в строке нет, возвращается остаток строки.
Delete(s,k,n)	Удаление из строки ***s***, начиная с позиции ***k***, подстроки длиной ***n*** символов. Если ***k*** больше размера ***s***, удаления не происходит. Если символов в строке недостаточно, удаляется остаток

Функции и процедуры	Результат
	строки.
Insert(s1,s2,k)	Вставка подстроки ***s1*** в строку ***s2***, начиная с позиции ***k***. Строка слишком большого размера усекается до ***255*** символов.

Пример: *поиск заданного контекста в строке и замена его новым.*

```
const
  oldContext = 'Old';
  NewContext = 'New';
var
  position: integer;
  workString: string;
...
position := Pos(oldContext, workString);
while position > 0 do begin
  Delete(workString, position, Length(oldContext));
  Insert(NewContext, workString, position);
  position := Pos(oldContext, workString);
end;
...
```

• • •

Строки допускается передавать подпрограммам в качестве параметров. В отличие от массивов, они могут возвращаться функциями как результаты их работы.

Пример: *функция обращения строки.*

```
function Reverse(s: string): string;
var
  i: integer; tmp: char;
begin
  for i := 1 to Length(s) div 2 do begin
```

```
    tmp := s[i];
    s[i] := s[Length(str) + 1 - i];
    s[Length(str) + 1 - i] := tmp;
  end;
  Reverse := s;
end;
```

• • •

5.1.2 Записи

Запись считается комбинированным типом данным. Она определяется следующим образом

```
Имя-типа = record
  Имя-поля: Тип-поля;
  Имя-поля: Тип-поля;
  ...
end;
```

Примечание. *Точка с запятой — элемент описания.*

Типом поля может быть *любой тип языка* ***Pascal***. Количество полей не ограничивается. Сами записи могут использоваться для построения более сложных структур данных (например, массивов и других записей).

В блоке констант изображение записи строится из списка изображений ее полей. Элементы списка должны соответствовать фактическому описанию *по количеству, порядку следования и типу*.

Пример*: объявление записей и задание их значений через блок констант.*

```
type
{ Вспомогательный тип «месяц». }
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
    Sep, Oct, Nov, Dec);
{ Вспомогательный тип «образование». }
```

```
  TEducation = (None, Primary, Secondary, Vocational,
    Trade, Classical, Commercial, Art, Higher);
{ Тип данных «дата». }
  TDate = record
    Day: 1 .. 31; Month: TMonth; Year: integer;
  end;
{ Тип данных «персональные сведения». }
  TPerson = record
    Name, SecondName, SurName: string;
    Birthday: TDate;
    Education: TEducation;
  end;
{ Тип данных «персонал». }
  TPersonnel = array [1 .. 100] of TPerson;

const
  newYear: TDate = (Day: 1; Month: Jan; Year: 2015);
  minister: TPerson = (
    Name : 'John';
    SecondName: 'Wesley';
    SurName : 'Harding';
    Birthday : (Day: 15; Month: Jun; Year: 1956);
    Education : Commercial);
```

• • •

Действия над записями выполняются по отдельности над каждым полем, доступ к которому производится с помощью селектора

Имя-записи.Имя-поля.

Набор допустимых операций *определяется типом поля*.

Пример: *присваивание значений полям записи.*

```
var
  secretary: TPerson;
...
secretary.Name := 'Bernard';
```

```
secretary.SecondName := 'Spencer';
secretary.SurName := 'Wooly';
secretary.Birthday.Day := 27;
secretary.Birthday.Month := Mar;
secretary.Birthday.Year := 1969;
secretary.Education := Art;
...
```

• • •

Для более эффективной обработки полей записи используется оператор присоединения общего имени ***with***.

Пример: *присоединение общего имени полям записи.*

```
var
  worker: TPersonnel;
  i, aMonth, anEducation: integer;
...
for i := 1 to 100 do
  with worker[i] do begin
    ReadLn(Name); ReadLn(SurName); ReadLn(SecondName);
    with Birthday do begin
      ReadLn(Day);
      ReadLn(aMonth); { Число в диапазоне 1 .. 12. }
      Month := TMonth(Pred(aMonth));
      ReadLn(Year);
    end;
    ReadLn(anEducation); { Число в диапазоне 1 .. 9. }
    Education := TEducation(Pred(anEducation));
  end;
...
```

• • •

Над записями в целом можно выполнять только операции присваивания, при которых производится *копирование соответствующих полей*. Типы записей, участвующих в таких операциях, должны быть *эквивалентны*.

Пример: *определение типа «точка на плоскости».*

```
type
  TPoint = record X, Y: double; end;
const
  initialPoint: TPoint = (X: 10.0; Y: 20.0);
var
{ Типы point и initialPoint эквивалентны. }
  point: TPoint;
{ Тип anotherPoint не эквивалентен типу point. }
  anotherPoint: record X, Y: Double; end;
...
point := initialPoint; { Допустимо. }
anotherPoint := point; { Недопустимо! }
...
```

• • •

В процедуры и функции записи могут передаваться по значению или по ссылке. Правила языка требуют именной эквивалентности типов формальных и фактических параметров, поэтому в заголовках подпрограмм необходимо использовать синонимы структурных типов. Функция не может возвращать запись в качестве своего результата (возможен возврат ссылки на структуру).

Пример: *использование записей для работы с комплексными числами.*

```
type TComplex = record Re, Im, Abs, Arg: double; end;

{ Вспомогательная функция знака. }
function Sign(x: double): double;
begin
  if x < 0 then Sign := -1.0 else
    if x > 0 then Sign := 1.0 else Sign := 0.0;
end;

{ Процедура формирования комплексного числа. }
procedure Create(var x: TComplex; r, i: double);
```

```
begin
  x.Re := r; x.Im := i;
  x.Abs := SqRt(Sqr(x.Re) + Sqr(x.Im));
  if x.Re <> 0.0
    then x.Arg := ArcTan(x.Im / x.Re)
    else x.Arg := Sign(x.Im) * Pi / 2.0;
end;

{ Процедура сложения комплексных чисел. }
procedure Add(x1, x2: TComplex; var x3: TComplex);
begin
  x3.Re := x1.Re + x2.Re; x3.Im := x1.Im + x2.Im;
  x3.Abs := SqRt(Sqr(x3.Re) + Sqr(x3.Im));
  if x3.Re <> 0.0
    then x3.Arg := ArcTan(x3.Im / x3.Re)
    else x3.Arg := Sign(x3.Im) * Pi / 2.0;
end;

{ Процедура вычитания комплексных чисел. }
procedure Sub(x1, x2: TComplex; var x3: TComplex);
begin
  x3.Re := x1.Re - x2.Re; x3.Im := x1.Im - x2.Im;
  x3.Abs := SqRt(Sqr(x3.Re) + Sqr(x3.Im));
  if x3.Re <> 0.0
    then x3.Arg := ArcTan(x3.Im / x3.Re)
    else x3.Arg := Sign(x3.Im) * Pi / 2.0;
end;

{ Процедура умножения комплексных чисел. }
procedure Mul(x1, x2: TComplex; var x3: TComplex);
begin
  x3.Abs := x1.Abs * x2.Abs; x3.Arg := x1.Arg + x2.Arg;
  x3.Re := x3.Abs * Cos(x3.Arg);
  x3.Im := x3.Abs * Sin(x3.Arg);
end;

{ Процедура деления комплексных чисел. }
procedure Dvd(x1, x2: TComplex; var x3: TComplex);
```

```
begin
  x3.Abs := x1.Abs / x2.Abs; x3.Arg := x1.Arg - x2.Arg;
  x3.Re := x3.Abs * Cos(x3.Arg);
  x3.Im := x3.Abs * Sin(x3.Arg);
end;

{ Процедура вывода комплексного числа. }
procedure Print(x: TComplex);
begin
  WriteLn(x.Re: 7: 4, ' + ', x.Im: 7: 4, 'i');
end;

var
  a, b, c: TComplex;

begin
  Create(a, 1.0, 2.0); Print(a); { Первый операнд. }
  Create(b, 2.0, 4.0); Print(b); { Второй операнд. }
  Add(a, b, c); Print(c);        { Сумма.          }
  Sub(a, b, c); Print(c);        { Разность.       }
  Mul(a, b, c); Print(c);        { Произведение.   }
  Dvd(a, b, c); Print(c);        { Частное.        }
end.
```

• • •

Если, в зависимости от значения некоторого поля, в пределах одной структуры необходимо хранить различную информацию, то используют **записи с вариантами**. Поле, значение которого будет критерием выбора, называется *полем варианта*. Его тип должен быть *упорядоченным*. Имена полей внутри вариантов должны различаться. Они не могут совпадать с именами полей безвариантной части записи.

В объявлении записи может быть только одна вариантная часть, которая *должна завершать* список ее полей. Любой вариант может иметь свою вариантную часть, которая также должна завершать его описание.

***Пример**: использование записей с вариантами для объявления типов геометрических фигур.*

```
type
  TFigure = record
    X, Y: double; { Базовые координаты фигуры. }
    case FigureKind: (Circle, Square, Rect) of
      Circle: (Radius: double); { Окружность. }
      Square: (Side: double); { Квадрат. }
      Rect:   (Height, Width: double); { Прямоугольник. }
  end;
var
 figure1, figure2: TFigure;
...
figure1.FigureKind := Circle;
figure1.Radius := 25.7;
figure2.FigureKind := Rect;
figure2.Height := 17.8;
figure2.Width := 7.2;
...
```

• • •

Допускается в качестве критерия выбора варианта использовать значения любого упорядоченного типа данных (потребность в поле варианта при этом отпадает, и размер записи несколько сокращается).

***Пример**: упрощенное объявление записи с вариантами.*

```
type
  TFigure = record
    X, Y: double;
    case byte of
      1: (Radius: double);         { Окружность. }
      2: (Side: double);           { Квадрат. }
      3: (Height, Width: double); { Прямоугольник. }
  end;
```

• • •

Все варианты занимают одно и то же пространство памяти. Объем памяти, отводимый на хранение записи, определяется по самому большому варианту. Контроль логики работы с вариантами отсутствует.

***Пример**: нарушение логики работы с вариантами.*

```
...
figure1.FigureKind := Circle;
figure1.Height := 17.8;
figure1.Width := 7.2;
...
```

• • •

Так как поля записи с вариантами «совмещаются» в пространстве памяти, ее содержимое можно интерпретировать по-разному. Эта особенность структуры часто используется для выполнения специфических преобразований типов.

***Пример**: вывод побайтового представления числа типа **double**.*

```
type
  TData = record
    case byte of
      1: (Data: double);
      2: (Byte: array [1 .. 8] of byte);
  end;

var
  x: TData; i: integer;
...
x.Data = 3.7;
for i := 1 to SizeOf(x.Data) do
  WriteLn(x.Byte[i]);
...
```

• • •

5.1.3 Множества

В язык ***Pascal*** включен стандартный тип, позволяющий работать со счетными множествами. Он определяется следующим образом

```
Имя-типа = set of Упорядоченный-Базовый-тип;
```

Примечание. *Точка с запятой — элемент описания.*

Размер множества не может превышать ***256*** элементов, поэтому в качестве базового допускается использовать:

- один из стандартных типов ***boolean***, ***char***, ***shortint*** и ***byte***;
- тип-диапазон, не выходящий за пределы ***0 .. 255***;
- тип-перечисление, значения элементов которого отображается на диапазон ***0 .. 255***.

При объявлении множественных типов определяется только принципиально возможный набор значений элементов, а не их действительные значения. Проинициализировать множество можно в программе или через блок констант. Значения константных выражений, указываемых в изображениях элементов, должны соответствовать базовому типу множества.

Пример: *объявление и инициализация множеств.*

```
type
  TSymbols = set of char;
  TDigits = set of 0 .. 9;
  TDays = set of (Sun, Mon, Tue, Wed, Thu, Fri, Sat);

const
  Alphabet: TSymbols = ['A' .. 'Z'];
  OddDigit: TDigits = [1, 3, 5, 7, 9];
  WorkDays: TDays = [Mon .. Fri];
```

• • •

Множества могут участвовать в ограниченном числе операций, которые выполняются максимально эффективно.

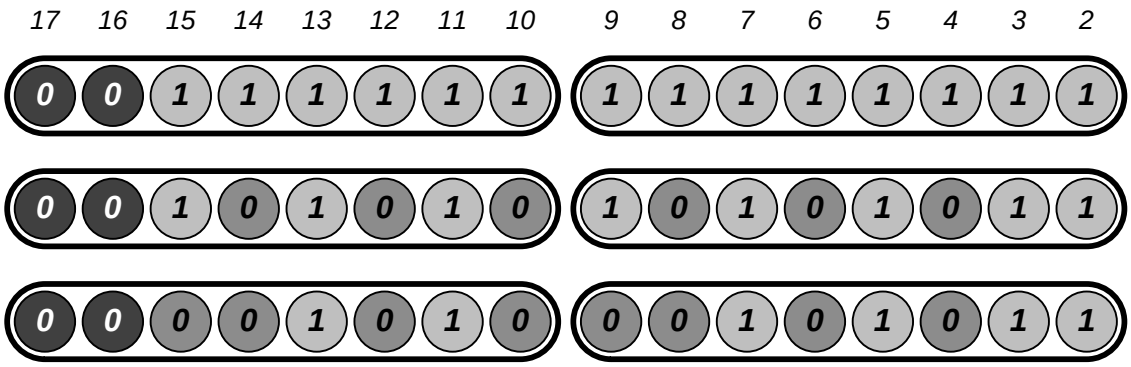
Операция	Особенность выполнения
`:=`	**Присваивание.** В операции могут принимать участие только значения, которые входят в объявленный диапазон значений множества.
`+ * -`	**Объединение, пересечение и вычитание.** Операции аналогичны традиционным математическим действиям.
`= <>` `<= >=`	**Проверки** на равенство, неравенство и включение множеств.
`in`	**Проверка принадлежности** указанного значения множеству.

Операция обращения к отдельному элементу множества в языке ***Pascal*** отсутствует. Если необходимо выполнить подобное действие, организуется циклический перебор всех возможных значений базового типа. На каждом шаге до выполнения нужной операции проверяется принадлежность множеству очередного значения параметра цикла.

Пример: *вычисление простых чисел в заданном диапазоне с помощью «решета Эратосфена» с использованием множеств.*

Алгоритм греческого математика и философа Эратосфена, использующий множества для нахождения простых чисел, меньших или равных значению **n**, *заключается в следующем. В исходное множество включаются все числа в диапазоне от* ***2*** *до* **n**. *Его «просеивание», которое начинается с наименьшего ключевого значения* ***2***, *заключается в удалении всех элементов, кратных ключу. Следующее по значению число в множестве будет простым. Оно стано-*

*вится новым ключом, и процесс «просеивания» повторяется. Работа алгоритма продолжается до тех пор, пока не будет просканировано все множество для всех возможных ключей. Работа «решета» показана на рисунке (шаг первый — исходное множество чисел от **2** до **15**; шаг второй — множество после удаления чисел, кратных **2**; шаг третий (и последний) — множество после удаления чисел, кратных **3**).*



Данный алгоритм реализуется с помощью программы.

```
const
  n = 20;
  sieve: set of 2 .. n = [2 .. n];
  primes: set of 2 .. n = [];

var
  current, next: 2 .. n;

begin
  current := 2;
  repeat
{ Поиск в sieve наименьшего простого числа. }
    while not (current in sieve) do
      Inc(current);
{ Размещение этого числа в primes. }
    primes := primes + [current];
{ Удаление из sieve всех чисел, кратных next. }
    next := current;
    while next <= n do begin
```

```
      sieve := sieve - [next]; Inc(next, current);
    end;
  until sieve = [];
{ Вывод простых чисел из заданного диапазона. }
  for current := 2 to n do
    if current in primes then WriteLn(current: 4);
end.
```

• • •

В процедуры и функции множества можно передать как по значению, так и по ссылке. Чтобы типы формальных и фактических параметров были эквивалентны, в заголовках подпрограмм необходимо использовать синонимы множественных типов. Функция не может возвращать множество в качестве своего результата. Возврат ссылки на структуру возможен, однако, учитывая специфику применения множеств, потребность в этом возникает редко.

5.1.4 Списки

Списочный тип не входит в систему типов языка ***Pascal***, поэтому его необходимо описать явно.

Примечание. *Программа не должна принципиально меняться при изменении типа обрабатываемых данных, поэтому базовый тип списка рекомендуется определять через псевдоним.*

```
type
  TData = integer; { Псевдоним базового типа. }
  PNode = ^TNode; { Указатель на списочный тип. }
  TNode = record { Списочный тип. }
    Info: TData;
    Next: PNode;
  end;
```

До начала работы необходимо определить указатель на голов-

ной элемент списка ***Head*** и присвоить ему пустую ссылку.

```
var
  Head: PNode;
...
{ В теле программы. }
  Head := Nil;
...
```

Функция, которая создает новый элемент списка, имеет следующий вид

```
function CreateNode(value: TData; NextNode: PNode): PNode;
var NewNode: PNode;
begin
  New(NewNode);
  NewNode^.Info := value; NewNode^.Next := NextNode;
  CreateNode := NewNode;
end;
```

Стандартная процедура

```
Dispose(Current);
```

удалит элемент списка ***Current*** из динамической памяти.

В качестве примера приведены вставка нового элемента в упорядоченный по возрастанию список (упорядоченность структуры при этом не нарушается) и удаление из списка элемента с заданным значением (предполагается, что процедуры ***InsertAt*** и ***RemoveAt*** реализованы так, как это рассмотрено в п. 2.5).

Пример: *вставка элемента в упорядоченный по возрастанию список.*

```
procedure Insert(value: TData; Head: PNode);
var Prev, Current: PNode;
begin
  Prev := Nil; Current := Head;
  while (Current <> Nil) and
```

```
        (Current^.Info < value) do begin
    Prev := Current; Current := Current^.Next;
  end;
  InsertAt(value, Prev);
end;
```

• • •

Пример: *удаление из списка элемента с заданным значением.*

```
procedure Delete(value: TData; Head: PNode);
var Prev, Current: PNode;
begin
  Prev := Nil; Current := Head;
  while (Current <> Nil) and
        (Current^.Info <> value) do begin
    Prev := Current; Current := Current^.Next;
  end;
  if Current <> Nil
    then RemoveAt(Prev, Current)
    else WriteLn('Значение не найдено!');
end;
```

• • •

5.1.5 Деревья

Будут рассматриваться бинарные деревья поиска. Чтобы иметь возможность производить их обработку, в программе необходимо определить тип вершины и ввести указатель на корень дерева.

```
type
  TData = integer; { Псевдоним базового типа. }
  PNode = ^TNode; { Указатель на вершину. }
  TNode = record { Тип вершины. }
    Key: word;
    Info: TData;
    Left, Right: PNode;
  end;
```

```
var
  Root: PNode;
...
{ В теле программы. }
  Root := Nil;
...
```

Ниже приведен пример функции, которая создает новую вершину дерева со значением ключевого поля ***keyValue*** и со значением поля ***Info***, равным ***value***.

```
function CreateNode(keyValue: word; value: TData): PNode;
var NewNode: PNode;
begin
  New(NewNode);
  NewNode^.Key := keyValue; NewNode^.Info := value;
  NewNode^.Left := Nil; NewNode^.Right := Nil;
  CreateNode := NewNode;
end;
```

В качестве примера показана процедура нерекурсивного обхода дерева во внутреннем порядке с выводом содержимого поля ***Info*** для каждой вершины. Для временного хранения ссылок на вершины используется стек.

Примечание. *В примере используется статический стек на основе массива. Его размер должен превышать число вершин дерева. Исходя из специфики задачи, исключен контроль переполнения и опустошения стека.*

Пример*: обход бинарного дерева во внутреннем порядке.*

```
procedure InorderTreeWalk(Root: PNode);

  const
    StackSize = 20;
    top: word = 0;
```

```
  var
    Current: PNode;
    stack: array [1 .. StackSize] of PNode;

{ Процедура записи информации в стек. }
  procedure Push(Current: PNode);
  begin
    Inc(top); stack[top] := Current;
  end;

{ Функция чтения информации из стека }
  function Pop: PNode;
  begin
    Pop := stack[top]; Dec(top);
  end;

begin
  Current := Root;
  while (Current <> Nil) or (top <> 0) do begin
{ Спуск по левой ветви }
    while Current <> Nil do begin
      Push(Current);
      Current := Current^.Left;
    end;
    Current := Pop;
    WriteLn(Current^.Info);
{ Переход в правую ветвь }
    Current := Current^.Right;
  end;
end.
```

• • •

5.1.6 Графы

В программу рекомендуется включить определение типа вершины графа, который кроме необходимой информации, может содержать признак обработки вершины и ее цвет.

```
type
  TData = char; { Псевдоним базового типа. }
  TVertex = record
    Info: TData;    { Информация. }
    Done: boolean;  { Признак обработки. }
    Color: integer; { Цвет. }
  end;
```

Необходимая информация сохраняется в массиве элементов типа ***TVertex*** нужной размерности. Для каждой вершины поля ***Info*** и ***Color*** инициализируются, исходя из условий задачи. У всех вершин содержимое поля ***Done*** до начала работы с графом должно быть равно ***False***.

Ниже представлена программа нерекурсивного поиска в глубину по графу, изображенному на рисунке 4.6-б, с выводом содержимого поля ***Info*** для каждой вершины. Данные хранятся в массиве вершин ***vertex***. Структура графа отображается матрицей смежности ***adjacent***. Для временного хранения номеров вершин используется стек. Защиту от записи в заполненный стек и от чтения из пустой структуры необходимо реализовать самостоятельно.

Пример: *программа нерекурсивного поиска в глубину по неориентированному графу.*

```
const
{ Число вершин графа. }
  n = 6;
type
  TData = integer;
  TVertex = record
    Info: TData; Done: boolean;
  end;

const
{ Информация о вершинах графа. }
  vertex: array [1 .. n] of TVertex =
    ((Info: 1; Done: False),
```

```
      (Info: 2; Done: False),
      (Info: 3; Done: False),
      (Info: 4; Done: False),
      (Info: 5; Done: False),
      (Info: 6; Done: False));
{ Матрица смежности. }
  adjacent: array [1 .. n, 1 .. n] of integer =
    ((0, 1, 0, 1, 0, 0),
     (1, 0, 0, 0, 1, 0),
     (0, 0, 0, 0, 1, 1),
     (1, 0, 0, 0, 1, 0),
     (0, 1, 1, 1, 0, 1),
     (0, 0, 1, 0, 1, 0));

var
  v, top: integer;
  stack: array [1 .. n] of integer;

{ Процедура записи данных в стек. }
procedure Push(data: integer);
  begin
    Inc(top); stack[top] := data;
  end;

{ Функция извлечения данных из стека. }
function Pop: integer;
  begin
    Pop := stack[top]; Dec(top);
  end;

{ Процедура поиска в глубину. }
procedure DFS(v: integer);
  var
    u: integer;
  begin
    top := 0;
    Write(vertex[v].Info: 4);
    Push(v); vertex[v].Done := True;
```

```
    while top > 0 do begin
{ Определение последней обработанной вершины. }
      v := Pop; Push(v);
{ Поиск по списку необработанных вершин. }
      for u := 1 to n do
        if adjacent[v][u] <> 0 then
          if not vertex[u].Done then break;
      if (u = n) and vertex[u].Done
{ Список просмотрен до конца. }
        then v := Pop
        else begin
{ Обработка новой вершины. }
          Write(vertex[u].Info: 4);
          Push(u); vertex[u].Done := True;
        end;
    end;
    WriteLn;
  end;

begin
  for v := 1 to n do
    vertex[v].Done := False;
  for v := 1 to n do
    if not vertex[v].Done
      then DFS(v);
end.
```

• • •

Можно определить тип ребра (дуги) графа, представляющий как информацию о смежности вершин, так и дополнительные характеристики (например, вес ребра или дуги).

```
TEdge = record
  U, V: integer;    { Номера смежных вершины }
  Weight: integer; { Вес ребра }
end;
```

В качестве примера приведена программа, строящая каркас графа, изображенного на рисунке 4.15, с помощью алгоритма Крускала. Информация о графе отображается массивами вершин ***vertex*** и ребер ***edge***. Для хранения ветвей каркаса используется вспомогательный массив.

Примечание. *Процедуру предварительного упорядочивания ребер по приоритету необходимо разработать самостоятельно.*

Пример*: программа построения каркаса неориентированного графа методом Крускала.*

```
const
{ Число вершин графа }
  n = 6;
{ Число ребер графа }
  m = 9;

type
  TData = integer;
  TVertex = record
    Info: TData; Done: boolean; Color: integer;
  end;
  TEdge = record u, v, Weight: integer; end;

const
{ Информация о вершинах графа }
  vertex: array [1 .. n] of TVertex =
    ((Info: 1; Done: False; Color: 0),
     (Info: 2; Done: False; Color: 0),
     (Info: 3; Done: False; Color: 0),
     (Info: 4; Done: False; Color: 0),
     (Info: 5; Done: False; Color: 0),
     (Info: 6; Done: False; Color: 0));
{ Информация о ребрах графа }
  edge: array [1 .. m] of TEdge =
    ((U: 1; V: 5; Weight: 1),
     (U: 3; V: 4; Weight: 2),
```

```
      (U: 2; V: 6; Weight: 3),
      (U: 5; V: 6; Weight: 4),
      (U: 3; V: 6; Weight: 5),
      (U: 4; V: 5; Weight: 6),
      (U: 1; V: 2; Weight: 7),
      (U: 1; V: 4; Weight: 8),
      (U: 2; V: 3; Weight: 9));
{ Счетчик ветвей каркаса графа }
  k: integer = 0;

var
{ Каркас графа }
  tree: array [1 .. n - 1] of TEdge;
  i, j: integer;

begin
{ Раскраска вершин. }
  for i := 1 to n do
    vertex[i].Color := i;
{ Перебор ребер графа. }
  for i := 1 to n - 1 do
    if vertex[edge[i].U].Color <>
       vertex[edge[i].V].Color then begin
{ Включение нового безопасного ребра в каркас. }
         Inc(k); tree[k] := edge[i];
{ Перекраска графа. }
         for j := 1 to n do
           if vertex[j].Color = vertex[tree[k].V].Color
             then vertex[j].Color := vertex[tree[k].U].Color;
       end;
{ Отображение результатов }
  for i := 1 to n - 1 do
    WriteLn(tree[i].U: 4, tree[i].V: 4);
end.
```

• • •

5.2 Структуры данных в языке C++

5.2.1 Массивы и строки

В языке ***C++*** объявление массива имеет вид

```
Тип Имя-массива[Размерность-1]...[Размерность-N] =
  { Список-инициализации };
```

Примечание. *Точка с запятой — элемент описания.*

Имя массива является идентификатором. Размерность массива является константным выражением. Она определяется на этапе компиляции программы и не может изменяться впоследствии. Под массив отводится область памяти размером не более ***64 К***.

При объявлении массива его элементам можно присвоить начальные значения из списка инициализации. Они должны задаваться в виде константных выражений, соответствующих фактическому описанию массива *по количеству и типу*. Если размерность массива не указана, она будет вычислена компилятором по списку инициализации. Если этот список опущен, элементы массива считаются *неопределенными*.

Многомерные массивы образуются из структур меньшей размерности. Их списки инициализации формируются с учетом порядка размещения элементов в машинной памяти при векторизации структуры. Размер самого левого измерения вычисляется компилятором, поэтому он *может не указываться*. Для наглядности значения по каждому измерению могут группироваться.

Пример*: объявление и инициализация массивов.*

```
int count[100];
int sqr[][2] = {{1, 1}, (2, 4}, {3, 9}, {4, 16}};
double data[7] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0};
char anyWord[] = {'C', '+', '+', '\0'};
```

• • •

Элемент массива считается переменной с индексами

***Имя-массива*[*Выражение-1*]...[*Выражение-N*].**

При выполнении программы значения индексных выражений *преобразуются к необходимому целочисленному типу*.

Все действия над массивом выполняются по отдельности над каждым элементом, доступ к которому производится по индексу. Набор допустимых операций *полностью определяется типом элемента*. Так как индекс определяет смещение элемента относительно начала структуры, выраженное в единицах типа элемента, то *индексация по любому измерению начинается с нуля*.

***Пример**: вычисление произведения матриц* $\boldsymbol{C = A \cdot B}$.

```
double a[k][m], b[m][n], c[k][n];
for (int i = 0; i < k; i++)
  for (int j = 0; j < n; j++)
  {
    c[i][j] = 0.0;
    for (int ij = 0; ij < m; ij++)
      c[i][j] += a[i][ij] * b[ij][j];
  }
// Вывод матрицы в общепринятом виде.
for (i = 0; i < k; i++)
{
  for (int j = 0; j < n; j++)
    printf("%4d", c[i][j]);
  printf("\n");
}
```

• • •

***Пример**: подсчет числа вхождений символов в строку.*

```
char workString[100];
int i, count[256];
for (i = 0; i < 256; i++)
  count[i] = 0; // Сброс начальных значений счетчиков.
```

```
for (i = 0; i < 100; i++)
  count[workString[i]]++; // Подсчет вхождений символов.
for (i = 0; i < 256; i++)
  printf("%4d\n", count[i]); // Вывод результатов.
```

• • •

Во время работы программы принадлежность индексов допустимым диапазонам *не проверяется*, и содержимое самих массивов не контролируется. Следующий фрагмент программы компилируется без ошибок, хотя при его выполнении нарушается граница массива, и теряется содержимое смежных участков памяти.

Пример: *некорректные действия над массивом.*

```
double x[10];
for (int i = 0; i < 100; i++)
  x[i] = i;
```

• • •

Если массив используется в качестве аргумента функции, то в нее передается ссылка на первый элемент массива. В прототипе функции ее параметр нужно объявить как переменную с индексами. Для многомерных массивов все размерности, кроме первой, должны быть указаны явно.

Пример: *массивы как аргументы функций.*

```
const n = 2;
const m = 3;
void main()
{
  extern void f1(double []);
  extern void f2(double [][n]);
  double x[n], y[m][n];
  f1(x); f2(y);
}
void f1(double p[]) { ... }
```

```
void f2(double q[][n]) { ... }
```

• • •

Массивы тесно связаны с указателями (*имя массива без индекса — константная ссылка на его начальный элемент*). Указатели могут индексироваться и модифицироваться. Их использование позволяет обращаться к многомерным массивам как к линейным структурам, что сокращает затраты времени на их индексацию.

Примечание. *Часто адресная арифметика оказывается эффективнее индексации.*

Пример*: индексация и модификация указателей.*

```
int x[n], y[n][m], i, j;
// Использование индексации.
for (i = 0; i < n; i++)
{
  x[i] = i;
  for (j = 0; i < m; j++) y[i][j] = i * j;
}
// Использование адресной арифметики.
for (i = 0; i < n; i++)
{
  *(x + i) = i;
  for (j = 0; i < m; j++) *(*(y + i) + j) = i * j;
}
```

• • •

Пример*: векторизация массивов с помощью указателей.*

```
double a[n][m], *p;
p = &a[0][0]; // Получение ссылки на начало структуры.
for (int i = 0; i < n; i++)
  for (int j = 0; j < m; j++)
    p[i * m + j] = i * j; // Эквивалентно a[i][j] = i * j.
```

• • •

С помощью указателей можно возвращать массивы как результаты работы функций.

Пример: *вычисление произведения квадратных матриц* $\boldsymbol{C = A \cdot B}$ *с помощью функции (массивы векторизованы).*

```
const int n = 2;
void main()
{
  extern int * Mul(int *, int *);
  int a[] = {1, 2, 3, 4}, b[] = {1, 3, 2, 4};
  int * c = Mul(a, b);
// Вывод матрицы в общепринятом виде.
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < n; j++)
      printf("%4d", c[i * n + j]);
    printf("\n");
  }
  delete [] c;
}

// Функция умножения векторизованных квадратных матриц.
int * Mul(int * a, int * b)
{
  int * c = new int[n * n];
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
    {
      c[i * n + j] = 0;
      for (int ij = 0; ij < n; ij++)
        c[i * n + j] += a[i * n + ij] * b[ij * n + j];
    }
  return c;
}
```

• • •

Указатели можно применять для создания «свободных» многомерных массивов, элементы которых имеют разную длину. Память для их размещения *выделяется и освобождается явно*.

***Пример**: использование «свободных» массивов.*

```
int * x[3];
x[0] = new int[3]; x[1] = new int[2]; x[2] = new int[4];
```

• • •

При работе с указателями необходимо следить, чтобы не произошла непреднамеренная ошибочная адресация структуры.

***Пример**: ошибочное использование указателей.*

```
int x[] = { 1, 2, 3, 4, 5 }, y[] = { 9, 8, 7, 6, 5 },
    *p = x, // Указатель p – синоним x.
    *q = y; // Указатель q – синоним y.
p[2] = 0; // Новое содержимое x – 1, 2, 0, 4, 5.
q[2] = 0; // Новое содержимое y – 9, 8, 0, 6, 5.
p = q;    // Потеря ссылки на массив x.
p[0] = 0; // Содержимое x не изменяется.
q[0] = 0; // Новое содержимое y – 0, 8, 0, 6, 5.
```

• • •

Одномерный символьный массив, заканчивающийся нулевым элементом, в **C++** считается строкой. В отличие от обычных массивов, строки могут инициализироваться литералами, в конец которых компилятор автоматически добавляет нулевой символ. *Если финализирующий символ отсутствует, действия над строкой имеют непредсказуемые последствия.*

***Пример**: объявление и инициализация строк.*

```
char anyString[20];
char helloString[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

```
char anotherString[] = "See you later...";
```

• • •

Действия над отдельными символами строки определяются общими правилами, принятыми для массивов.

Для ввода и вывода строк рационально использовать функции ***gets*** и ***puts***. Прототипы других полезных функций манипуляции со строками определены в файле ***string.h***.

Функция	Результат
strlen(s)	Значение фактической длины строки **s**.
strcmp(s1,s2)	Значение ***0***, если ***s1 = s2***. Отрицательное значение, если ***s1 < s2***. Положительное значение, если ***s1 > s2***. Сравнение выполняется слева направо с учетом упорядоченности кодов символов. Более короткая строка считается «меньше» более длинной строки с такими же начальными символами.
strcat(s1,s2)	Конкатенация (присоединение) ***s2*** к ***s1***.
strcpy(s1,s2)	Копирование ***s2*** в ***s1***.
strchr(s1,ch)	Указатель на первое вхождение символа ***ch*** в строку ***s1***.
strstr(s1,s2)	Указатель на первое вхождение подстроки ***s2*** в строку ***s1***.

Пример: *операции над строками.*

```
char s1[80]; gets(s1);
char s2[80]; gets(s2);
printf("%s %d\n", s1, strlen(s1));
puts(strcat(s1, s2));
```

```
if (!strcmp(s1, s2))
  printf("Строки равны.\n");
if (strchr("Hello!", 'l'))
  printf("\"Hello!\" содержит символ \'l\'.\n");
if (strstr("Hello!", "ll"))
  printf("\"Hello!\" содержит подстроку \"ll\".\n");
```

• • •

Для работы со строками можно использовать указатели на стандартный тип ***char***. Выделение и освобождение памяти для размещения структур в этом случае *должны производиться явно*.

Пример: *копирование строк (временные указатели* ***s1*** *и* ***s2*** *перемещаются по строкам и после завершения операции — обнаружения нулевого символа — выходят за их пределы).*

```
char *str1 = "Temporary string.";
char *str2 = new char[20];
char *s1 = str1, *s2 = str2;
while(*s2++ = *s1++);
puts(str1); puts(str2);
delete[] s2;
```

• • •

Строки могут передаваться функциям в качестве параметров и (так как они напрямую связаны с указателями) возвращаться функциями как результаты их работы.

Пример: *функция обращения строки.*

```
char * Reverse(char * s)
{
  for (int i = 0; i < strlen(s) / 2; i++)
  {
    char tmp = s[i];
    s[i] = s[strlen(s) - i - 1];
    s[strlen(s) - i - 1] = tmp;
```

```
  }
  return s;
}
```

• • •

5.2.2 Структуры и объединения

Структура считается комбинированным типом данным. Ее объявление имеет следующий вид

```
struct Имя-Типа
{
  Тип-поля Имя-поля;
  Тип-поля Имя-поля;
  ...
};
```

Примечание. *Точка с запятой — элемент описания.*

Типом поля может быть *любой тип языка*. Количество полей не ограничивается. Сами структуры могут использоваться для построения более сложных типов данных (например, массивов и других структур).

При объявлении структурной переменной ее полям можно присвоить начальные значения из списка инициализации. Они должны задаваться в виде константных выражений, соответствующих фактическому описанию *по количеству, порядку следования и типу*. Если этот список опущен, поля считаются *неопределенными*.

Пример*: объявление и инициализация структур.*

```
// Вспомогательный тип данных «месяц».
enum TMonth {Jan, Feb, Mar, Apr, May, Jun, Jul,
  Aug, Sep, Oct, Nov, Dec};
// Вспомогательный тип данных «образование».
enum TEducation {None, Primary, Secondary, Vocational,
  Trade, Classical, Commercial, Art, Higher};
```

```
// Тип данных «дата».
struct TDate
{
  int Day; TMonth Month; int Year;
};
// Тип данных «персональные сведения».
struct TPerson
{
  char Name[20], Surname[20], SecondName[20];
  TDate BirthDay;
  TEducation Education;
};
TDate newYear = {1, Jan, 2008};
TPerson minister = { "John", "Wesley", "Harding",
  {15, Jun, 1956}, Commercial};
```

• • •

Если структура определена, как переменная (по имени), то доступ к ее отдельному полю производится с помощью селектора

Имя-структуры.Имя-поля.

Пример: *доступ к структуре по имени.*

```
TPerson worker[100];
for (int i = 0; i < 100; i++)
{
  gets(worker[i].Name);
  gets(worker[i].Surname);
  gets(worker[i].SecondName);
  scanf("%u", &worker[i].BirthDay.Day);
  scanf("%u", &worker[i].BirthDay.Month);
  scanf("%u", &worker[i].BirthDay.Year);
  scanf("%u", &worker[i].Education);
}
```

• • •

Если структура определена с помощью указателя, то доступ к ее полям производится с помощью ссылочного селектора

Указатель-на-структуру->Имя-поля.

Пример: *доступ к структуре по указателю.*

```
TPerson * chief = new TPerson;
...
puts((*chief).Name); // Стандартное обращение.
puts(chief->Name);   // Эквивалентное обращение.
switch(chief->Education)
{
  case None       : printf("None.\n"); break;
  case Primary    : printf("Primary.\n"); break;
  case Secondary  : printf("Secondary.\n"); break;
  case Vocational: printf("Vocational.\n"); break;
  case Trade      : printf("Trade.\n"); break;
  case Classical  : printf("Classical.\n"); break;
  case Commercial: printf("Commercial.\n"); break;
  case Art        : printf("Art.\n"); break;
  case Higher     : printf("Higher.\n");
}
```

• • •

Все действия над структурами выполняются по отдельности над каждым полем. Набор допустимых операций *определяется типом поля*. Над структурами в целом можно выполнять только операции присваивания, при которых производится *копирование соответствующих полей*. Типы структур, участвующих в таких операциях, должны быть *эквивалентны*.

Пример: *агрегатное присваивание значения структуре.*

```
// Типы a и b эквивалентны.
struct { int Field1, Field2; } a, b;
// Тип c определен в собственном описании.
struct { int Field1, Field2; } c;
```

```
a = b; // Допустимо.
b = c; // Недопустимо!
```

• • •

В подпрограммы структуры могут передаваться как по значению, так и по ссылке (правила языка требуют именной эквивалентности типов формальных и фактических параметров). Структуры могут возвращаться функциями в качестве результата своего выполнения.

Пример: *использование структур для работы с комплексными числами.*

```
struct TComplex { double Re, Im, Abs, Arg; };

// Вспомогательная функция знака.
double Sign(double x)
{
  return (x < 0.0) ? -1 : (x > 0.0) ? 1 : 0;
}

// Функция формирования комплексного числа.
TComplex Create(double r, double i)
{
  TComplex x;
  x.Re = r; x.Im = i;
  x.Abs = sqrt(x.Re * x.Re + x.Im * x.Im);
  if (x.Re) x.Arg = atan(x.Im / x.Re);
  else x.Arg = Sign(x.Im) * M_PI_2;
  return x;
}

// Функция сложения комплексных чисел.
TComplex Add(TComplex x1, TComplex x2)
{
  TComplex x;
  x.Re = x1.Re + x2.Re; x.Im = x1.Im + x2.Im;
```

```
  x.Abs = sqrt(x.Re * x.Re + x.Im * x.Im);
  if (x.Re) x.Arg = atan(x.Im / x.Re);
  else x.Arg = Sign(x.Im) * M_PI_2;
  return x;
}

// Функция вычитания комплексных чисел.
TComplex Sub(TComplex x1, TComplex x2)
{
  TComplex x;
  x.Re = x1.Re - x2.Re; x.Im = x1.Im - x2.Im;
  x.Abs = sqrt(x.Re * x.Re + x.Im * x.Im);
  if (x.Re) x.Arg = atan(x.Im / x.Re);
  else x.Arg = Sign(x.Im) * M_PI_2;
  return x;
}

// Функция умножения комплексных чисел.
TComplex Mul(TComplex x1, TComplex x2)
{
  TComplex x;
  x.Abs = x1.Abs * x2.Abs; x.Arg = x1.Arg + x2.Arg;
  x.Re = x.Abs * cos(x.Arg); x.Im = x.Abs * sin(x.Arg);
  return x;
}

// Функция деления комплексных чисел.
TComplex Div(TComplex x1, TComplex x2)
{
  TComplex x;
  x.Abs = x1.Abs / x2.Abs; x.Arg = x1.Arg - x2.Arg;
  x.Re = x.Abs * cos(x.Arg); x.Im = x.Abs * sin(x.Arg);
  return x;
}

// Функция вывода комплексного числа.
void Print(TComplex x)
{
```

```
  printf("%7.4f + %7.4fi\n", x.Re, x.Im);
}

void main()
{
  TComplex a = Create(1.0, 2.0); Print(a);
  TComplex b = Create(2.0, 4.0); Print(b);
  Print(Add(a, b)); Print(Sub(a, b));
  Print(Mul(a, b)); Print(Div(a, b));
}
```

• • •

Язык ***C++*** поддерживает работу с т. н. «битовыми» полями структур, полезными при доступе к отдельным битам внутри байта или при дефиците машинной памяти. Общий вид определения битового поля следующий

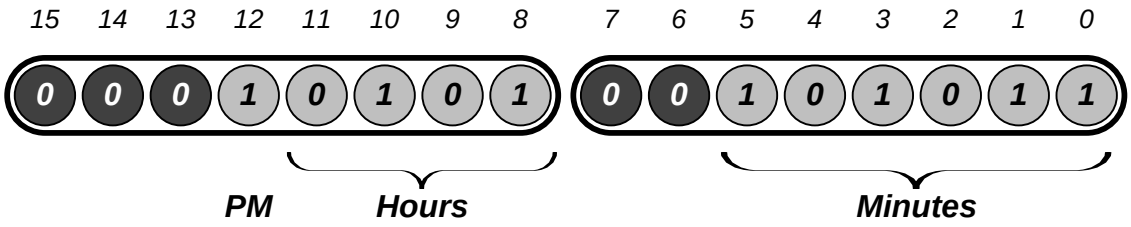
***Тип-поля Имя-поля*: *Размер-в-битах*;**

Тип битового поля может быть ***int***, ***signed*** или ***unsigned***. При распределении памяти битовые поля необходимо выравнивать по границе байта.

Пример: *компактное определение типа «текущее время» (под каждое поле отводится минимально необходимое количество бит).*

```
struct TTime
{
  unsigned Minutes: 6; // Минуты в диапазоне 0 .. 59.
  unsigned        : 2; // Выравнивание по границе байта.
  unsigned Hours  : 4; // Часы в диапазоне 0 .. 11.
  unsigned PM     : 1; // Признак «пополудни».
  unsigned        : 3; // Выравнивание по границе байта.
};
```

*Размещение структуры в памяти (занято **2** байта, представлено время **17** часов **43** минуты):*



• • •

Частным случаем структур можно считать объединения, все поля которых размещаются в одной и той же области машинной памяти. Их объявления строятся по тем же правилам, что и объявления структур (ключевое слово ***struct*** заменяется на ***union***). Имена полей объединения должны различаться. Размер памяти, отводимой для хранения объединения, определяется размером его самого большого компонента.

Объединения могут использоваться для создания эквивалентов записей с вариантами языка ***Pascal***. Имена полей такого комбинированного типа не могут повторяться в пределах структуры, в которой они объявлены.

Пример: *использование структур и объединений для объявления типов геометрических фигур.*

```
struct TFigure
{
  double X, Y; // Базовые координаты фигуры.
  union
  {
    struct { double Radius; } Circle; // Окружность.
    struct { double Side; } Square; // Квадрат.
    struct { double Height, Width; } Rect; // Прямоугольник.
  };
};
...
TFigure figure;
figure.Circle.Radius = 1.7;
figure.Rect.Height = 4.3;
```

```
figure.Rect.Width = 7.9;
...
```

• • •

Так как поля объединения «совмещаются» в пространстве памяти, его содержимое можно интерпретировать различными способами. По этой причине подобные структуры данных часто используют для выполнения специфических преобразований типов.

***Пример**: вывод побайтового представления числа типа **double**.*

```
union TUnion
{
  unsigned short Byte[8]; double Data;
};
...
TUnion x; x.Data = 3.7;
for (int i = 0; i < sizeof(x.Data); i++)
  printf("%04X ", x.Byte[i]);
...
```

• • •

5.2.3 Множества

Язык ***C++*** не располагает встроенными средствами работы с множествами. Использование для этой цели массивов оправдано, если это позволяет компактно или эффективно представить решение задачи.

***Пример**: вычисление простых чисел в заданном диапазоне с помощью «решета Эратосфена» (см. п. 5.1.3).*

```
void main()
{
  const int n = 20;
  int s[n + 1], p[n], i, j;
// Исходное множество чисел.
```

```
  for (i = 2; i <= n; i++) s[i] = 1;
// «Просеивание».
  for (i = 2; i * i <= n; i++)
    for (j = 2; i * j <= n; j++)
// Числа, кратные i, удаляются из множества.
      s[i * j] = 0;
// Формирование массива простых чисел.
  for (i = 2, j = 0; i <= n; i++)
    if (s[i]) p[j++] = i;
// Вывод результата.
  for (i = 0; i < j; i++)
    printf("%d\n", p[i]);
}
```

• • •

В ***C***++ множества имитируются посредством массивов целых чисел, с элементами которых оперируют на уровне битов (как показано в п. 2.4). В качестве примера в приложении А приведен шаблон множественного типа, реализованный с использованием технологии объектно-ориентированного программирования.

5.2.4 Списки

В языке ***C***++ нет встроенных средств для работы со списками. В программе необходимо явно определить списочный тип.

Примечание. *Программа на языке* ***C***++ *не должна принципиально меняться при изменении типа обрабатываемых данных, поэтому рекомендуется использовать шаблоны (****templates****), которым базовый тип списка передается, как параметр.*

```
template <class T>
struct TNode
{
  T Info;
  TNode<T> * Next;
};
```

В программу необходимо ввести указатель на головной элемент списка ***Head*** и явно присвоить ему значение пустой ссылки (в примере список строится на базе типа ***int***).

```
...
// В теле программы.
  TNode<int> * Head = 0;
...
```

Функция создания элемента списка имеет вид

```
template <class T>
TNode<T> * CreateNode(T value, TNode<T> * NextNode)
{
  TNode<T> * NewNode = new TNode<T>;
  NewNode->Info := value; NewNode->Next := NextNode;
  return NewNode;
}
```

Удаление элемента списка ***Current*** из динамической памяти выполняется оператором

```
delete Current;
```

Ниже представлены функции вставки элемента в упорядоченный по возрастанию список (упорядоченность структуры сохраняется) и удаления из списка элемента с заданным значением (функции ***InsertAt*** и ***RemoveAt*** описаны в п. 2.5).

Пример: *вставка элемента в упорядоченный по возрастанию список.*

```
template <class T>
void Insert(T value, TNode<T> * Head)
{
  TNode<T> * Prev = 0;
  TNode<T> * Current = Head;
  while (Current && Current->Info < value)
  {
```

```
    Prev = Current; Current = Current->Next;
  }
  InsertAt(value, Prev);
}
```

• • •

***Пример**: удаление из списка элемента с заданным значением.*

```
template <class T>
void Delete(T value, TNode<T> * Head)
{
  TNode<T> * Prev = 0;
  TNode<T> * Current = Head;
  while (Current && Current->Info != value)
  {
    Prev = Current; Current = Current->Next;
  }
  if (Current)
    RemoveAt(Prev, Current);
  else
    printf("Значение не найдено!\n");
}
```

• • •

5.2.5 Деревья

Чтобы иметь возможность работать с бинарными деревьями поиска, в программе необходимо дать описание типа вершины и ссылки на нее. До начала работы необходимо определить указатель на корень дерева и присвоить ему пустую ссылку.

```
template <class T>
struct TNode
{
  unsigned Key;
  T Info;
  TNode<T> * Left, * Right;
```

```
};
TNode<int> * Root = 0;
```

Приведенная ниже функция создает новую вершину дерева со значением ключевого поля ***keyValue*** и со значением поля ***Info***, равным ***value***.

```
template <class T>
TNode<T> * CreateNode(unsigned keyValue, T value)
{
  TNode<T> * NewNode = new TNode<T>;
  NewNode->Key = keyValue; new->Info = value;
  NewNode->Left = NewNode->Right = 0;
  return NewNode;
}
```

В качестве примера продемонстрирована технология нерекурсивного обхода дерева во внутреннем порядке с выводом содержимого поля ***Info*** для каждой вершины. Для временного хранения ссылок на вершины используется стек.

Примечание. *В примере используется статический стек на основе массива, размер которого должен превышать число вершин дерева. Исходя из специфики задачи, исключен контроль переполнения и опустошения стека.*

Пример*: обход бинарного дерева во внутреннем порядке.*

```
TNode<T> * stack[20];
int top = -1;

// Функция для записи информации в стек.
template <class T>
void Push(TNode<T> * data)
{
  stack[++top] = data;
}
```

```
// Функция для чтения информации из стека.
template <class T>
TNode<T> * Pop()
{
  return stack[top--];
}

template <class T>
void InorderTreeWalk(TNode<T> * Root)
{
  TNode<T> * Current = Root;
  while (Current || top != -1)
  {
// Спуск по левой ветви.
    while (Current)
    {
      Push(Current);
      Current = Current->Left;
    }
    Current = Pop();
    printf("%d\n", Current->Info);
// Переход в правую ветвь
    Current = Current->Right;
  }
}
```

• • •

5.2.6 Графы в языке C++

В программе должен быть определен тип вершины графа как структуры. Для удобства работы в его состав рекомендуется включить дополнительные поля — признак обработки вершины и (при необходимости) ее цвет.

```
template <class T>
struct TVertex
{
```

```
  T Info;    // Информация
  int Done;  // Признак обработки
  int Color; // Цвет
};
```

Сам граф представляется массивом элементов типа ***TVertex*** нужной размерности. Поля ***Info*** и ***Color*** инициализируются, исходя из условий задачи. До начала работы с графом содержимое поля ***Done*** для каждой вершины должно быть равно нулю.

В качестве примера ниже представлена программа нерекурсивного поиска в глубину по графу, изображенному на рисунке 4.6-б, с выводом содержимого поля ***Info*** для каждой вершины. Необходимая информация хранится в массиве вершин ***vertex***. Структура графа отображается матрицей смежности ***adjacent***. Иллюстрируется использование стека для хранения номеров вершин (контроль его переполнения и опустошения необходимо реализовать самостоятельно).

Пример: *программа нерекурсивного поиска в глубину по неориентированному графу.*

```
// Число вершин графа.
const int n = 6;
template <class T>
struct TVertex { T Info; int Done; };
// Информация о вершинах графа.
TVertex<int> vertex[n] = {{1, 0, 0}, {2, 0, 0},
                          {3, 0, 0}, {4, 0, 0},
                          {5, 0, 0}, {6, 0, 0}};
// Матрица смежности.
int adjacent[n][n] = {{0, 1, 0, 1, 0, 0},
                      {1, 0, 0, 0, 1, 0},
                      {0, 0, 0, 0, 1, 1},
                      {1, 0, 0, 0, 1, 0},
                      {0, 1, 1, 1, 0, 1},
                      {0, 0, 1, 0, 1, 0}};
```

```
// Стек для временного хранения данных.
int stack[n];
// Начальное значение указателя верхушки стека.
int top = -1;

// Процедура записи данных в стек.
void Push(int data)
{
  stack[++top] = data;
}
// Функция извлечения данных из стека.
int Pop()
{
  return stack[top--];
}

// Функция поиска в глубину.
void DFS(int v)
{
  top = -1;
  printf("%d\n", vertex[v].Info);
  Push(v); vertex[v].Done = 1;
  while (top > -1)
  {
// Определение последней обработанной вершины.
    v = Pop(); Push(v);
// Поиск по списку необработанных вершин.
    for (int u = 0; u < n; u++)
      if (adjacent[v][u])
        if (!vertex[u].Done) break;
    if (u == n)
// Список просмотрен до конца.
      v = Pop();
    else
    {
// Обработка новой вершины.
      printf("%d\n", vertex[u].Info);
      Push(u); vertex[u].Done = 1;
```

```
    }
  }
}

void main()
{
  for (int v = 0; v < n; v++)
    vertex[v].Done = 0;
  for (int v = 0; v < n; v++)
    if (!vertex[v].Done)
      DFS(v);
}
```

• • •

Можно определить тип ребра или дуги графа, представляющий (как минимум) информацию о смежности вершин, а также о дополнительных характеристиках ребер.

```
struct TEdge
{
  int U, V;   // Номера смежных вершин.
  int Weight; // Вес ребра.
};
```

Ниже приведена программа построения каркаса графа, изображенного на рисунке 4.15, с помощью алгоритма Крускала. Информация о графе отображается массивами вершин ***vertex*** и ребер ***edge***. Иллюстрируется использование вспомогательного массива для хранения ветвей каркаса.

Примечание. *Процедуру предварительного упорядочивания ребер по приоритету следует разработать самостоятельно.*

Пример: *программа построения каркаса неориентированного графа методом Крускала.*

```
// Число вершин графа.
const int n = 6;
```

```
// Число ребер графа.
const int m = 9;

template <class T>
struct TVertex { T Info; int Done, Color; };
struct TEdge { int U, V, Weight; };

// Информация о вершинах графа.
TVertex<int> vertex[n] = {{1, 0, 0}, {2, 0, 0},
                          {3, 0, 0}, {4, 0, 0},
                          {5, 0, 0}, {6, 0, 0}};
// Информация о ребрах графа.
TEdge edge[m] = {{1, 5, 1}, {3, 4, 2}, {2, 6, 3},
                 {5, 6, 4}, {3, 6, 5}, {4, 5, 6},
                 {1, 2, 7}, {1, 4, 8}, {2, 3, 9}};
// Каркас графа.
TEdge tree[n - 1];
// Счетчик ветвей каркаса графа.
int k = -1;

void main()
{
// Раскраска вершин.
  for (int i = 0; i < n; i++)
    vertex[i].Color = i;
// Перебор ребер графа.
  for (i = 0; i < n - 1; i++)
    if (vertex[edge[i].U - 1].Color !=
        vertex[edge[i].V - 1].Color)
    {
// Включение нового ребра в каркас.
      tree[++k] = edge[i];
// Перекраска графа.
      for (int j = 0; j < n; j++)
        if (vertex[j].Color == vertex[tree[k].V - 1].Color)
          vertex[j].Color = vertex[tree[k].U - 1].Color;
    }
```

```
// Отображение результатов.
  for (i = 0; i < n - 1; i++)
    printf("%4d %4d\n", tree[i].U, tree[i].V);
}
```

• • •

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как объявляется массив в языке ***Pascal***? Каковы ограничения на размер массива? Как организуются многомерные массивы? Как задать значения элементов массива через блок констант?

2. Как осуществляется доступ к элементу массива? Какие действия можно производить над отдельным элементом массива, а какие — над массивом в целом? Контролируется ли принадлежность индексов допустимым диапазонам?

3. Как осуществляется передача массивов в процедуры и функции языка ***Pascal***? Можно ли возвращать массивы как результаты работы функций?

4. Как организованы строки в языке ***Pascal***? Как они объявляются? В каких операциях строки могут принимать участие? Какие функции могут использоваться при работе со строками?

5. Как объявляется запись в языке ***Pascal***? Как задать значения полям записи через блок констант?

6. Как осуществляется доступ к полю записи? Что дает присоединение общего имени к полям записи?

7. Какие действия можно производить над отдельным полем записи, а какие — над записью в целом?

8. Как осуществляется передача записей в процедуры и функции языка ***Pascal***? Можно ли возвращать записи как результаты работы функций?

9. Для чего используются записи с вариантами? Как они объявляются? Как осуществляется контроль работы с вариантами?

10. Как объявляется множество в языке ***Pascal***? Каковы огра-

ничения на размер множества? Как задать значения элементов множества через блок констант?

11. Какие операции можно производить над множествами? Как осуществляется доступ к отдельному элементу множества?

12. Как осуществляется передача множеств в процедуры и функции языка ***Pascal***? Можно ли возвращать множества как результаты работы функций?

13. Как организуются и обрабатываются линейные связные списки в языке ***Pascal***?

14. Как организуются и обрабатываются деревья в языке ***Pascal***?

15. Как организуются и обрабатываются графы в языке ***Pascal***?

16. Как объявляется массив в языке ***C***++? Каковы ограничения на размер массива? Как организуются многомерные массивы? Как задать значения элементов массива через список инициализации?

17. Как осуществляется доступ к элементу массива? Какие действия можно производить над отдельным элементом массива? Контролируется ли принадлежность индексов допустимым диапазонам?

18. Как осуществляется передача массивов в функции языка ***C***++? Можно ли возвращать массивы как результаты работы функций?

19. Как работать с массивами с помощью указателей?

20. Как организованы строки в языке ***C***++? Как они объявляются? В каких операциях строки могут принимать участие? Какие функции могут использоваться при работе со строками?

21. Как объявляется структура в языке ***C***++? Как задать значения полям структуры через список инициализации?

22. Как осуществляется доступ к полю структуры? Как получить доступ к полю структуры с помощью указателя?

22. Какие действия можно производить над отдельным полем структуры, а какие — над структурой в целом?

24. Как осуществляется передача структур в функции языка

C++? Можно ли возвращать структуры как результаты работы функций?

25. Для чего используются «битовые» поля структур? Как с ними работать?

26. Для чего используются объединения? Как они объявляются?

27. Как решаются задачи теории множеств в языке ***C++***?

28. Как организуются и обрабатываются линейные связные списки в языке ***C++***?

29. Как организуются и обрабатываются деревья в языке ***C++***?

30. Как организуются и обрабатываются графы в языке ***C++***?

6 ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ

6.1 Организация практикума

Практикум предполагает выполнение семи работ, тематика которых в целом соответствует структуре настоящего пособия. В каждой работе решается одна конкретная задача.

Обязательным условием допуска студента к выполнению работы является усвоение соответствующего теоретического материала. Работа считается выполненной и защищенной, если студент ответил на все вопросы по методике ее проведения и теории, получил необходимые результаты и дал им правильную оценку, продемонстрировал уверенное владение соответствующими инструментальными средствами.

Примечание. *Для проверки теоретических знаний могут использоваться контрольные вопросы, приведенные в соответствующих разделах пособия.*

Практикум не претендует на полноту охвата всех проблем, связанных с проектированием программного обеспечения. Не ставилась также целью изучение приемов решения прикладных задач в конкретной программной среде. Тем не менее, автор надеется, что реализация предлагаемой программы практической подготовки позволит активизировать самостоятельную работу студентов, упрочить их теоретические знания и усовершенствовать навыки решения задач в области программной инженерии.

6.2 Работа №1. Обработка массивов

Цель работы — изучение принципов организации регулярных структур данных и усвоение правил выполнения над ними типовых операций.

Содержание работы — создание программного приложения для обработки одномерных и многомерных массивов данных.

Предварительно необходимо самостоятельно изучить:

- основы линейной алгебры, необходимые для выполнения векторно-матричных операций;
- правила объявления одномерных и многомерных массивов и выполнения над ними основных операций путем перебора элементов или обращения к стандартным функциям;
- приемы программирования операций над массивами на языках высокого уровня.

При выполнении работы необходимо:

- сформировать алгоритм решения задачи, выбранной в соответствии с вариантом;
- составить программу решения задачи на языке высокого уровня;
- подготовить тестовый набор исходных данных, достаточный для проверки корректности программы;
- проверить с помощью набора исходных данных правильность работы программы.

Содержание отчета о работе:

- описание задачи;
- алгоритм решения задачи;
- текст программы на языке высокого уровня, выбранном для решения задачи;
- тестовые данные;
- результаты работы программы, подтверждающие ее правильность.

Все необходимые теоретические сведения приведены в данном пособии. Варианты заданий для выполнения работы приведены в приложении Б.

Большинство задач на обработку массивов решаются методом систематического перебора их элементов. Так как размерность мас-

сива определена до начала выполнения программы, для этой цели традиционно используют циклы с параметром.

При возникновении необходимости для одномерных массивов рекомендуется использовать следующие приемы решения типовых задач ($Length(x)$ — длина или число элементов массива x).

```
// Вычисление суммы элементов вектора.
Sum(x)
  s ← 0
  FOR i ∈ [1, Length(x)]
    DO s ← s + x[i]
  RETURN s

// Вычисление произведения элементов вектора.
Prod(x)
  p ← 1
  FOR i ∈ [1, Length(x)]
    DO p ← p × x[i]
  RETURN p

// Определение минимального элемента вектора.
Min(x)
  min ← x[1]
  FOR i ∈ [2, Length(x)]
    DO IF x[i] < min
         THEN min ← x[i]
  RETURN min

// Определение номера минимального элемента вектора.
NMin(x)
  nmin ← 1
  FOR i ∈ [2, Length(x)]
    DO IF x[i] < x[nmin]
         THEN nmin ← i
  RETURN nmin
```

Для массивов большей размерности рекомендуется самостоятельно модифицировать эти схемы.

Примеры выполнения работы на языках ***Pascal*** и ***C++*** приведены ниже.

***Пример**: Задана целочисленная матрица **A** размерностью **n×m**. Сформировать матрицу **B** размерностью **n×m** путем переноса в нее положительных элементов соответствующих строк **A**.*

*Алгоритм решения задачи следующий. Строки матрицы **A** перебираются циклически. В процессе последовательного просмотра каждой строки **A** ее положительные элементы копируются в матрицу **B** на вакантное место. Если строка **A** обработана полностью, и в строке **B** есть вакантные места, туда записывается минимальное целое число. Для фиксации числа положительных элементов в строке матрицы **B** используется счетчик.*

*Вариант программы на языке **Pascal**:*

```
const
  n = 4; m = 3;
  a: array [1 .. n, 1 .. m] of integer =
    (( 1,  2,  3), (-1,  2,  3),
     (-1, -2, -3), ( 1, -2, -3));

var
  b: array [1 .. n, 1 .. m] of integer;
  i, j, k: integer;

begin
  for i := 1 to n do begin
    k := 0;
    for j := 1 to m do
      if a[i, j] >= 0 then begin
        Inc(k); b[i, k] := a[i, j];
      end;
    for j := k + 1 to m do
```

```
      b[i, j] := -MaxInt;
  end;
  for i := 1 to n do begin
    if b[i, 1] = -MaxInt then begin
      WriteLn('Строка матрицы пустая!'); Continue;
    end;
    j := 1;
    while (j <= m) and (b[i][j] <> -MaxInt) do begin
      Write(b[i][j]: 4); Inc(j);
    end;
    WriteLn;
  end;
end.
```

• • •

Вариант программы на языке ***C++****:*

```
# include <stdio.h>
# include <stdlib.h>

void main()
{
  const int n = 4, m = 3;
  int a[n][m] = {{ 1,  2,  3}, {-1,  2,  3},
                 {-1, -2, -3}, { 1, -2, -3}},
      b[n][m], i, j, k;
  for (i = 0; i < n; i++)
  {
    for (j = k = 0; j < m; j++)
      if (a[i][j] >= 0) b[i][k++] = a[i][j];
    for ( ; k < m; b[i][k++] = -INT_MAX);
  }
  for (i = 0; i < n; i++)
  {
    if (b[i][0] == -INT_MAX)
    {
      printf("Строка матрицы пустая!\n"); continue;
    }
```

```
    for (j = 0; j < m && b[i][j] != -INT_MAX; j++)
      printf("%4d", b[i][j]);
    printf("\n");
}
```

• • •

6.3 Работа №2. Обработка строк

Цель работы — изучение принципов организации текстовых данных как совокупности строк и усвоение правил выполнения над ними типовых операций.

Содержание работы — создание программного приложения для обработки текстов.

Предварительно необходимо самостоятельно изучить:

- основы морфологии, орфографии и синтаксиса языка, на основе которого будут подготовлены данные для решения задачи;
- правила объявления строк и выполнения над ними основных операций путем перебора элементов или обращения к стандартным функциям;
- приемы программирования операций над строками на языках высокого уровня.

При выполнении работы необходимо:

- сформировать алгоритм решения задачи, выбранной в соответствии с вариантом;
- составить программу решения задачи на языке высокого уровня;
- подготовить тестовый набор исходных данных, достаточный для проверки корректности программы;
- проверить с помощью набора исходных данных правильность работы программы.

Содержание отчета о работе:

- описание задачи;
- алгоритм решения задачи;

- текст программы на языке высокого уровня, выбранном для решения задачи;
- тестовые данные;
- результаты работы программы, подтверждающие ее правильность.

Все необходимые теоретические сведения приведены в данном пособии. Варианты заданий для выполнения работы приведены в приложении В.

Приемы, используемые для массивов, естественным образом распространяются на строки. Особые случаи работы связаны с поиском в тексте слов и фраз.

Обычно признаком окончания слова можно считать группу пробелов. Если после слова располагаются знаки препинания, то при решении конкретных задач (например, определении длины слова) их необходимо учесть.

Признаками окончания фразы можно считать знаки препинания. Троеточие должно учитываться только в том случае, если следующее за ним слово начинается с заглавной буквы.

Пробелы и знаки препинания не могут использоваться в качестве начальных символов текста. Завершаться текст должен одним из знаков препинания (но не пробелом).

При выполнении работы, если специально не оговорено иное, фрагмент текста целесообразно представлять одной строкой, составленной из слов русского языка. При решении задачи рекомендуется использовать стандартные строковые функции языка программирования.

Примеры выполнения работы на языках ***Pascal*** и ***C++*** приведены ниже.

***Пример**: Задан фрагмент текста, образованный словами английского языка. Определить длину каждого его слова.*

Используется следующий алгоритм. Фиксируется начальный символ первого слова, после чего начинается последовательный просмотр текста до обнаружения пробела. Так как после слова могут идти знаки препинания, организуется обратное движение по тексту до обнаружения последнего символа слова. Разность позиций последнего и первого символа определит длину слова. Группа повторяющихся пробелов пропускается, фиксируется начальный символ следующего слова, и вышеописанная процедура повторяется, пока не будет просмотрен весь текст.

Вариант программы на языке ***Pascal****:*

```
const
  n = 100;

var
  l: array [1 .. n] of integer; { Массив длин слов. }
  s: string; { Рабочая строка. }
  i, j, jj, k: integer;

begin
  ReadLn(s);
  jj := 1; k := 0;
  for i := 1 to Length(s) do
    if (s[i] = ' ') or (i = Length(s)) then begin
{ Закончились слово или текст. Фильтрация символов. }
      j := i;
      while not(s[j] in ['a' .. 'z', 'A' .. 'Z',
        '0' .. '9']) do Dec(j); { Возврат по строке. }
      Inc(k);
      l[k] := j - jj + 1; { Длина текущего слова. }
{ Пропуск повторяющихся пробелов. }
      while (i < Length(s)) and (s[i + 1] = ' ') do
        Inc(i);
      jj := i + 1; { Начало следующего слова. }
    end;
  for i := 1 to k do
```

```
    WriteLn(l[i]);
end.
```

• • •

Вариант программы на языке ***C++****:*

```
# include <stdio.h>
# include <string.h>

void main()
{
  const int n = 100;
  int length[n]; // Массив длин слов.
  char s[n + n]; // Рабочая строка.
  int i, j, jj, k;
  gets(s);
  for (i = jj = k = 0; i < strlen(s); i++)
    if (s[i] == ' ' || i == strlen(s) - 1)
    {
// Закончились слово или текст. Фильтрация символов.
      for (j = i - 1;
        !(s[j] >= 'a' && s[j] <= 'z') && // Не буква.
        !(s[j] >= 'A' && s[j] <= 'Z') && // Не буква.
        !(s[j] >= '0' && s[j] <= '9');   // Не цифра.
        j--); // Возврат по строке.
      length[k++] = ++j - jj; // Длина текущего слова.
// Пропуск повторяющихся пробелов.
      for ( ; i < strlen(s) && s[i + 1] == ' '; i++);
      jj = i + 1; // Начало следующего слова.
    }
  for (i = 0; i < k; i++)
    printf("%d\n", length[i]);
}
```

• • •

6.4 Работа №3. Обработка данных комбинированного типа

Цель работы — изучение принципов организации комбинированных структур данных и усвоение правил выполнения над ними типовых операций.

Содержание работы — создание программного приложения для обработки комбинированных структур данных.

Предварительно необходимо самостоятельно изучить:

- основы аналитической геометрии, необходимые для выполнения работы;
- правила объявления комбинированных типов данных и выполнения над ними основных операций;
- приемы программирования операций над данными комбинированных типов на языках высокого уровня.

При выполнении работы необходимо:

- сформировать алгоритм решения задачи, выбранной в соответствии с вариантом;
- составить программу решения задачи на языке высокого уровня;
- подготовить тестовый набор исходных данных, достаточный для проверки корректности программы;
- проверить с помощью набора исходных данных правильность работы программы.

Содержание отчета о работе:

- описание задачи;
- алгоритм решения задачи;
- текст программы на языке высокого уровня, выбранном для решения задачи;
- тестовые данные;
- результаты работы программы, подтверждающие ее правильность.

Все необходимые теоретические сведения приведены в данном пособии. Варианты заданий для выполнения работы приведены в приложении Г.

В работе комбинированные типы данных используются для представления характеристик геометрических объектов:

- точка на плоскости или в пространстве (в структуру объединяются ее координаты);
- прямая на плоскости (в структуру объединяются коэффициенты ее уравнения $A \cdot x + B \cdot y + C = 0$);
- плоскость в пространстве (в структуру объединяются коэффициенты ее уравнения $A \cdot x + B \cdot y + C \cdot z + D = 0$);
- окружность на плоскости или сфера в пространстве (в структуру объединяются координаты ее центра и радиус).

Подобная группировка позволяет создавать массивы геометрических объектов, все элементы которых обрабатываются единообразно с помощью циклов.

Большинство задач решаются методом систематического перебора возможных комбинаций объектов, при котором повторяющиеся группировки не рассматриваются. Для этого необходима грамотная организация управляющих циклов.

```
// Отбор неповторяющихся пар из n объектов.
FOR i ∈ [1, n - 1]
  DO FOR j ∈ [i + 1, n]
       DO Отобрано сочетание (i, j).

// Отбор неповторяющихся троек из n объектов.
FOR i ∈ [1, n - 2]
  DO FOR j ∈ [i + 1, n - 1]
       DO FOR k ∈ [j + 1, n]
            DO Отобрано сочетание (i, j, k).

// Отбор неповторяющихся четверок из n объектов.
FOR i ∈ [1, n - 3]
```

```
DO FOR j ∈ [i + 1, n – 2]
     DO FOR k ∈ [j + 1, n – 1]
          DO FOR l ∈ [k + 1, n]
               DO Отобрано сочетание (i, j, k, l).
```

Все вопросы из области геометрии должны решаться с помощью соответствующей справочной литературы.

Примеры выполнения работы на языках ***Pascal*** и ***C++*** приведены ниже.

Пример. *В трехмерном пространстве задано множество точек. Определить среднее расстояние между ними.*

Алгоритм решения задачи следующий. Расстояние между ***i****-й и* ***j****-й точкой рассчитывается по формуле*

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}\,.$$

В процессе перебора несовпадающих пар точек (i, j) *вычисляется сумма расстояний меду ними. Нужный результат получается делением полученного значения на число точек.*

Вариант программы на языке ***Pascal****:*

```
const
  n = 10;

{ Определение типа «точка в пространстве». }
type
  TPoint = record
    X, Y, Z: double;
  end;

const
  p: array [1 .. n] of TPoint =
    ((X: 1.3; Y: 2.7; Z: 9.2),
     (X: 7.1; Y: 1.2; Z: 1.3),
```

```
      (X: 0.8; Y: 0.3; Z: 3.1),
      (X: 2.3; Y: 7.4; Z: 2.8),
      (X: 3.2; Y: 4.7; Z: 7.3),
      (X: 8.9; Y: 9.5; Z: 4.3),
      (X: 7.5; Y: 5.0; Z: 8.6),
      (X: 5.4; Y: 6.8; Z: 7.5),
      (X: 2.1; Y: 9.2; Z: 2.9),
      (X: 4.7; Y: 2.0; Z: 9.7));

{ Определение расстояния между двумя точками. }
function Dist(p1, p2: TPoint): double;
begin
  Dist := SqRt(Sqr(p1.X - p2.X) +
               Sqr(p1.Y - p2.Y) +
               Sqr(p1.Z - p2.Z));
end;

var
  i, j: 1 .. n;
  sumDist: double;

begin
  sumDist := 0.0;
  for i := 1 to n - 1 do
    for j := i + 1 to n do
      sumDist := sumDist + Dist(p[i], p[j]);
  WriteLn((sumDist / n): 10: 4);
end.
```

• • •

Вариант программы на языке ***C++****:*

```
# include <stdio.h>
# include <math.h>

// Определение типа «точка в пространстве».
struct TPoint
{
```

```
  double X, Y, Z;
};

void main()
{
  extern double Dist(TPoint , TPoint);
  const int n = 10;
  TPoint point[] =
    {{1.3, 2.7, 9.2}, {7.1, 1.2, 1.3},
     {0.8, 0.3, 3.1}, {2.3, 7.4, 2.8},
     {3.2, 4.7, 7.3}, {8.9, 9.5, 4.3},
     {7.5, 5.0, 8.6}, {5.4, 6.8, 7.5},
     {2.1, 9.2, 2.9}, {4.7, 2.0, 9.7}};
  double sumDist = 0.0;
  for (int i = 0; i < n - 1; i++)
    for (int j = i + 1; j < n; j++)
      sumDist += Dist(point[i], point[j]);
  printf("%8.4f\n", sumDist / n);
}

// Определение расстояния между двумя точками.
double Dist(TPoint point1, TPoint point2)
{
  return sqrt(pow(point1.X - point2.X, 2.0) +
              pow(point1.Y - point2.Y, 2.0) +
              pow(point1.Z - point2.Z, 2.0));
}
```

• • •

6.5 Работа №4. Обработка множеств

Цель работы — изучение принципов организации данных множественного типа и усвоение правил выполнения над ними типовых операций.

Содержание работы — создание программного приложения для обработки множеств.

Предварительно необходимо самостоятельно изучить:

- основы теории множеств, необходимые для выполнения работы;
- правила объявления множественных типов данных и выполнения над ними основных операций;
- приемы программирования операций над данными множественных типов на языках высокого уровня.

При выполнении работы необходимо:

- сформировать алгоритм решения задачи, выбранной в соответствии с вариантом;
- составить программу решения задачи на языке высокого уровня;
- подготовить тестовый набор исходных данных, достаточный для проверки корректности программы;
- проверить с помощью набора исходных данных правильность работы программы.

Содержание отчета о работе:

- описание задачи;
- алгоритм решения задачи;
- текст программы на языке высокого уровня, выбранном для решения задачи;
- тестовые данные;
- результаты работы программы, подтверждающие ее правильность.

Все необходимые теоретические сведения приведены в данном пособии. Варианты заданий для выполнения работы приведены в приложении Д.

До начала проектирования программы необходимо оценить возможность использования множественных типов. К типовым случаям их применения относятся:

- осмысленная *группировка и перегруппировка* данных в соответствии с поставленным условием;

- *проверка* принадлежности конечному набору значений;
- *обнаружение* необходимой комбинации значений;
- *получение* необходимой комбинации значений в результате выполнения множественных операций.

В программе множество задается одним из способов, описанных в пп. 5.1.3 и 5.2.3.

Большинство задач решаются методом систематического перебора элементов множества, при котором их повторяющиеся группировки не рассматриваются (правила отбора неповторяющихся группировок рассмотрены в предыдущей работе).

Для выполнения действий над множествами необходимо разработать соответствующие программные модули или использовать средства языка программирования (при решении задачи на языке ***Pascal***). При необходимости возможно заимствование фрагментов программ из приложения А.

Все вопросы из области теории множеств должны решаться с помощью соответствующей литературы.

Примеры выполнения работы на языках ***Pascal*** и ***C++*** приведены ниже.

Пример. *Для произвольного символьного множества из* **n** *элементов сгенерировать все подмножества.*

Алгоритм решения задачи следующий. Задается нижняя и верхняя границы диапазона значений элементов, по которым находится мощность исходного множества N *и число его подмножеств* 2^N *(изначально они пустые). Генерируются* 2^N *различных целых чисел в диапазоне от* 0 *до* 2^{N-1}*, двоичный код которых ставится в соответствие каждому подмножеству. В процессе перебора возможных значений элементов производится (или не производится) их включение в соответствующее подмножество согласно его кодовой комбинации.*

*В приведенных примерах считается, что мощность исходного множества не превышает **8** элементов.*

*Вариант программы на языке **Pascal**:*

```
const
{ Нижняя граница множества. }
  Lo = 's';
{ Верхняя граница множества. }
  Hi = 'w';
{ Число возможных подмножеств без учета пустого. }
  TopIdx = (1 shl (ord(hi) - ord(lo) + 1)) - 1;

var
  subSet: array [0 .. TopIdx] of set of Lo .. Hi;
  i: 0 .. TopIdx; j: Lo .. Hi;

begin
  for i := 0 to TopIdx do begin
{ Получение битовой комбинации подмножества i. }
    subSet[i] := [];
    for j := Lo to Hi do
{ Проверка наличия бита j в комбинации i. }
      if (i and (1 shl (ord(j) - ord(Lo)))) <> 0 then
{ Включение элемента j в подмножество. }
        subSet[i] := subSet[i] + [j];
  end;
{ Вывод результатов. }
  for i := 0 to TopIdx do begin
    for j := Hi downto Lo do if j in subSet[i] then Write(j);
    WriteLn;
  end;
end.
```

• • •

*Вариант программы на языке **C++**:*

```
# include <stdio.h>
# include <stdlib.h>
```

```
void main()
{
// Нижняя граница множества.
  const int lo = 's';
// Верхняя граница множества.
  const int hi = 'w';
// Число возможных подмножеств (с учетом пустого).
  const int numSet = 1 << (hi - lo + 1);
  unsigned short i, subSet[numSet];
  for (i = 0; i < numSet; i++)
// Генерация битовой комбинации подмножества i.
    subSet[i] = i;
// Вывод результатов.
  for (i = 0; i < numSet; i++)
  {
    for (char j = hi; j >= lo; j--)
      if (subSet[i] & 1 << (j - lo)) printf("%c", j);
    printf("\n");
  }
}
```

• • •

6.6 Работа №5. Обработка списков

Цель работы — изучение принципов организации списочных структур данных и усвоение правил выполнения над ними типовых операций.

Содержание работы — создание программного приложения для обработки линейных связных списков.

Предварительно необходимо самостоятельно изучить:

- правила описания списочных типов данных и выполнения над ними основных операций;
- приемы программирования операций над данными списочных типов на языках высокого уровня.

При выполнении работы необходимо:

- сформировать алгоритм решения задачи, выбранной в соответствии с вариантом;
- составить программу решения задачи на языке высокого уровня;
- подготовить тестовый набор исходных данных, достаточный для проверки корректности программы;
- проверить с помощью набора исходных данных правильность работы программы.

Содержание отчета о работе:

- описание задачи;
- алгоритм решения задачи;
- текст программы на языке высокого уровня, выбранном для решения задачи;
- тестовые данные;
- результаты работы программы, подтверждающие ее правильность.

Все необходимые теоретические сведения приведены в данном пособии. Варианты заданий приведены в приложении Е.

6.7 Работа №6. Обработка деревьев

Цель работы — изучение принципов организации деревьев как иерархических структур данных и усвоение правил выполнения над ними типовых операций.

Содержание работы — создание программного приложения для обработки деревьев.

Предварительно необходимо самостоятельно изучить:

- правила описания деревьев как структур данных и выполнения над ними основных операций;
- приемы программирования операций над бинарными деревьями на языках высокого уровня.

При выполнении работы необходимо:

- сформировать алгоритм решения задачи, выбранной в соответствии с вариантом;
- составить программу решения задачи на языке высокого уровня;
- подготовить тестовый набор исходных данных, достаточный для проверки корректности программы;
- проверить с помощью набора исходных данных правильность работы программы.

Содержание отчета о работе:

- описание задачи;
- алгоритм решения задачи;
- текст программы на языке высокого уровня, выбранном для решения задачи;
- тестовые данные;
- результаты работы программы, подтверждающие ее правильность.

Все необходимые теоретические сведения приведены в данном пособии. Варианты заданий для выполнения работы приведены в приложении Ж.

Основой для решения большинства задач служат поисковые схемы. Для временного хранения информации может потребоваться вспомогательная структура данных — стек. Ее рекомендуется организовывать на базе статического массива.

6.8 Работа №7. Обработка графов

Цель работы — изучение принципов организации графов как многосвязных структур данных и усвоение правил выполнения над ними типовых операций.

Содержание работы — создание программного приложения для обработки графов.

Предварительно необходимо самостоятельно изучить:

- правила описания графов как структур данных и выполнения над ними основных операций;
- приемы программирования операций над графами на языках высокого уровня.

При выполнении работы необходимо:

- сформировать алгоритм решения задачи, выбранной в соответствии с вариантом;
- составить программу решения задачи на языке высокого уровня;
- подготовить тестовый набор исходных данных, достаточный для проверки корректности программы;
- проверить с помощью набора исходных данных правильность работы программы.

Содержание отчета о работе:

- описание задачи;
- алгоритм решения задачи;
- текст программы на языке высокого уровня, выбранном для решения задачи;
- тестовые данные;
- результаты работы программы, подтверждающие ее правильность.

Все необходимые теоретические сведения приведены в данном пособии. Варианты заданий для выполнения работы приведены в приложении И.

Структуру графа необходимо выбрать самостоятельно. Граф должен содержать не менее 6 вершин и не менее 9 ребер (дуг).

В программе структуру графа целесообразно представлять с помощью списков смежности или матрицы смежности. При небольшом числе вершин эти способы одинаково эффективны. Следующие алгоритмические схемы эквивалентны ($LIST[v]$ — список

смежности вершины ***v***, ***A*** — матрица смежности графа, ***n*** — число вершин графа):

```
// Использование списка смежности.
FOR v ∈ V
  DO FOR u ∈ LIST[v]
       DO Выполнение действия.
  IF u = 0
    THEN Список просмотрен.

// Использование матрицы смежности.
FOR v ∈ [1, n]
  DO FOR u ∈ [1, n]
       DO IF A[v, u] ≠ 0
             THEN Выполнение действия.
  IF u > n
    THEN Список просмотрен.
```

Основой для решения большинства задач служат поисковые схемы. Все вопросы из области теории графов должны решаться с помощью соответствующей справочной литературы.

Для временного хранения информации могут потребоваться вспомогательные структуры данных — стек, очередь и множество. Их рекомендуется организовывать на базе статических массивов.

СПИСОК ЛИТЕРАТУРЫ

1. Кнут, Д. Искусство программирования [Текст] : в 3 т. / Дональд Э. Кнут. — М. : Вильямс, 2010-2015. — (Искусство программирования).

Т. 1 : Основные алгоритмы. — 2010. — 720 с. — ISBN 978-5-8459-0080-7 — ISBN 0-201-89683-4.

Т. 2 : Получисленные алгоритмы. — 2011. — 832 с. — ISBN 978-5-8459-0081-4 — ISBN 5-8459-0081-6, 0-201-89684-2.

Т. 3 : Сортировка и поиск. — 2015. — 824 с. — ISBN 978-5-8459-0082-1 — ISBN 0-201-89685-0

2. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы. Построение и анализ [Текст] / Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн. — М. : Вильямс, 2015. — 1296 с. — (Классические учебники: Computer science). — ISBN 978-5-8459-0857-5 — ISBN 0-07-013151-1.

3. Вирт, Н. Алгоритмы и структуры данных [Текст] / Никлаус Вирт. — М. : ДМК Пресс, 2011. — 272 с. — (Классика программирования). — ISBN 978-5-94074-584-6 — ISBN 0-13-022005-9.

4. Фаронов, В. В. Turbo Pascal [Текст] : Учебное пособие / В.В Фаронов. — СПб. : Питер, 2010. — 368 с. — ISBN 978-5-469-01295-5.

5. Фаронов, В. В. Turbo Pascal 7.0. Практика программирования [Текст] : Учебное пособие / В.В Фаронов. — М. : КноРус, 2015. — 240 с. — ISBN 978-5-406-01925-2.

6. Немнюгин, С.А. Turbo Pascal. Программирование на языке высокого уровня [Текст] / С.А. Немнюгин. — СПб. : Питер, 2008. — 544 с. — (Учебник для вузов). — ISBN 978-5-94723-509-8.

7. Шилдт, Г. C++ для начинающих. Шаг за шагом [Текст] / Герберт Шилдт. — М. : ЭКОМ Паблишерз, 2010. — 640 с. —

(Самоучитель). — ISBN 978-5-9790-0127-2 — ISBN 0-07-219467-7.

8. Шилдт, Г. Полный справочник по C++ [Текст] / Герберт Шилдт. — М. : Вильямс, 2007. — (Полный справочник). — 800 с. — ISBN 978-5-8459-0489-8 — ISBN 5-8459-0489-7 — ISBN 0-07-222680-3.

9. Страуструп, Б. Язык программирования C++ [Текст] / Бьерн Страуструп. — СПб. : Бином, Невский Диалект, 2008. — 1136 с. — ISBN 5-7989-0226-2 — ISBN 5-7940-0064-3 — ISBN 0-201-70073-5.

10. Седжвик, Р. Алгоритмы на C++ [Текст] / Роберт Седжвик. — М. : Вильямс, 2011. — 1156 с. — ISBN 978-5-8459-1650-1 — ISBN 978-0-321-60633-4.

11. Хэзфилд, Р., Кирби, Л. Искусство программирования на C Фундаментальные алгоритмы, структуры данных и примеры приложений [Текст] / Р. Хэзфилд, Л. Кирби. — М. : ДиаСофт, 2001. — 736 с. — ISBN 966-7393-82-8.

12. Топп, У., Форд, У. Структуры данных в C++ [Текст] / Уильям Топп, Уильям Форд. — СПб. : Бином-Пресс, 2006. — 814 с. — ISBN 5-7989-0017-7 — ISBN 0-13-320938-5.

13. Подбельский, В.В. Язык Си++ [Текст] / В. В. Подбельский. — М. : Финансы и статистика, 2007. — 560 с. — ISBN 978-5-279-02204-5.

14. Подбельский, В.В. Стандартный Си++ [Текст] / В. В. Подбельский. — М. : Финансы и статистика, 2008. — 688 с. — ISBN 978-5-279-03243-3.

15. ИНТУИТ. Национальный открытый университет [Электронный ресурс] / Интернет-университет информационных технологий — дистанционное образование. — Электрон. дан. — [М. : Открытые системы, 2003-]. — Режим доступа: http://www/intuit.ru, свободный. — Загл. с экрана. — Яз. рус., англ.

ПРИЛОЖЕНИЕ А

Реализация множественного типа на языке C++

```
// Шаблон класса. * * * * * * * * * * * * * * * * * * * *
template <class T>
class TSet
{
  private:
    long loRange, hiRange;
    unsigned size;
    unsigned short * data;
    unsigned ArrayIndex(T);
    unsigned short BitMask(T);
  public:
    TSet(long, long);
    TSet(TSet<T> &);
   ~TSet(void);
    void Clear(void);
    void Display(char *);
    int Contain(T);
    int Empty(void);
    int Capacity(void);
    TSet<T> & operator = (TSet<T> &);
    TSet<T> & operator + (TSet<T> &);
    TSet<T> & operator * (TSet<T> &);
    TSet<T> & operator - (TSet<T> &);
    TSet<T> & operator += (T);
    TSet<T> & operator -= (T);
    int operator == (TSet<T> &);
    int operator != (TSet<T> &);
}
// Конструктор. * * * * * * * * * * * * * * * * * * * * *
template <class T>
TSet<T>::TSet(long lo, long hi): loRange(lo),
  hiRange(hi), size((hi - lo + 8) >> 3)
{
```

```
  data = new unsigned short [size];
  for (unsigned i = 0; i < size; i++)
    data[i] = 0;
}
// Конструктор копии. * * * * * * * * * * * * * * * * * *
template <class T>
TSet<T>::TSet(TSet<T> & x):
  loRange(x.loRange), hiRange(x.hiRange),
  size(x.size)
{
  data = new unsigned short [size];
  for (unsigned i = 0; i < size; i++)
    data[i] = x.data[i];
}
// Деструктор.  * * * * * * * * * * * * * * * * * * * * *
template <class T>
TSet<T>::~TSet(void)
{
  delete [] data;
}
// Вычисление номера байта. * * * * * * * * * * * * * * *
template <class T>
unsigned TSet<T>::ArrayIndex(T x)
{
  if (x < loRange || x > hiRange) exit(1);
  return (x - loRange) >> 3;
}
// Формирование маски.  * * * * * * * * * * * * * * * * *
template <class T>
unsigned short TSet<T>::BitMask(T x)
{
  if (x < loRange || x > hiRange) exit(1);
  return 1 << ((x - loRange) & 7);
}
// Очистка множества. * * * * * * * * * * * * * * * * * *
template <class T>
void TSet<T>::Clear(void)
{
```

```
  for (unsigned i = 0; i < size; i++)
    data[i] = 0;
}
// Отображение множества. * * * * * * * * * * * * * * *
template <class T>
void TSet<T>::Display(char * format)
{
  if (Empty())
    { printf("Set is empty!\n"); return; }
  for (long i = loRange; i <= hiRange; i++)
    if (Contain(i)) printf(format, i);
  printf("\n");
}
// Проверка принадлежности множеству. * * * * * * * * *
template <class T>
int TSet<T>::Contain(T x)
{
  return (int) data[ArrayIndex(x)] & BitMask(x);
}
// Проверка пустоты множества.  * * * * * * * * * * * *
template <class T>
int TSet<T>::Empty(void)
{
  for (unsigned i = 0, res = 1; i < size; i++)
    res *= data[i] == 0;
  return res;
}
// Определение мощности множества.  * * * * * * * * * *
template <class T>
int TSet<T>::Capacity(void)
{
  for (long i = loRange, res = 0; i <= hiRange; i++)
    if (Contain(i)) res++;
  return res;
}
// Присваивание множеству значения. * * * * * * * * * *
template <class T>
TSet<T> & TSet<T>::operator = (TSet<T> & x)
```

```
{
  if (loRange != x.loRange ||
      hiRange != x.hiRange) exit(1);
  for (unsigned i = 0; i < size; i++)
    data[i] = x.data[i];
  return * this;
}
// Объединение множеств.  * * * * * * * * * * * * * * * *
template <class T>
TSet<T> & TSet<T>::operator + (TSet<T> & x)
{
  if (loRange != x.loRange ||
      hiRange != x.hiRange) exit(1);
  for (unsigned i = 0; i < size; i++)
    data[i] |= x.data[i];
  return * this;
}
// Пересечение множеств.  * * * * * * * * * * * * * * * *
template <class T>
TSet<T> & TSet<T>::operator * (TSet<T> & x)
{
  if (loRange != x.loRange ||
      hiRange != x.hiRange) exit(1);
  for (unsigned i = 0; i < size; i++)
    data[i] &= x.data[i];
  return * this;
}
// Разность множеств. * * * * * * * * * * * * * * * * * *
template <class T>
TSet<T> & TSet<T>::operator - (TSet<T> & x)
{
  if (loRange != x.loRange ||
      hiRange != x.hiRange) exit(1);
  for (unsigned i = 0; i < size; i++)
    data[i] &= ~x.data[i];
  return * this;
}
// Включение в множество. * * * * * * * * * * * * * * * *
```

```
template <class T>
TSet<T> & TSet<T>::operator += (T x)
{
  data[ArrayIndex(x)] |= BitMask(x);
  return * this;
}
// Исключение из множества. * * * * * * * * * * * * * * *
template <class T>
TSet<T> & TSet<T>::operator -= (T x)
{
  data[ArrayIndex(x)] &= ~BitMask(x);
  return * this;
}
// Проверка эквивалентности множеств. * * * * * * * * * *
template <class T>
int TSet<T>::operator == (TSet<T> & x)
{
  for (unsigned i = 0, res = 1; i < size; i++)
    res *= data[i] == x.data[i];
  return res;
}
// Проверка неэквивалентности множеств. * * * * * * * * *
template <class T>
int TSet<T>::operator != (TSet<T> & x)
{
  for (unsigned i = 0, res = 1; i < size; i++)
    res *= data[i] != x.data[i];
  return res;
}

// Пример основной программы. * * * * * * * * * * * * * *
void main()
{
  const int n = 10;
  int i, j;
// Формирование множества натуральных чисел.
  TSet<int> a(2, n);
  for (i = 2; i <= n; a << i++);
```

```
  TSet<int> b(a);
// Формирование множества простых чисел
// с помощью решета Эратосфена.
  for (i = 2; i <= n; i++) if (a.Contain(i))
    for (j = 2 * i; j <= n; j += i) a >> j;
// Множество простых чисел: 2, 3, 5, 7.
  a.Display("%4d");
// Множество оставшихся чисел: 1, 4, 6, 8, 9, 10.
  (b - a).Display("%4d");
}
```

ПРИЛОЖЕНИЕ Б

МАССИВЫ

Варианты заданий для самостоятельного решения

1. ↓ Задан вектор X размерностью n, представляющий последовательность целых чисел. Определить число инверсий в этой последовательности.

   Рекомендации: *инверсией считать пару элементов, в которой большее число находится перед меньшим (* $x_i > x_j$*, если* $i < j$*).*

2. ↓ Задан вектор X размерностью n, представляющий последовательность целых чисел, отличных от 1. Найти максимальное значение k, при котором $x_i^k = i$.

   Рекомендации: *при подборе значения* k *учитывать, что оно не может превышать* n.

3. ↓ Задан вектор X размерностью n, представляющий последовательность целых чисел, отличных от 1. Найти максимальное значение k, при котором значение $\left|x_i^k - i\right|$ минимально (см. задачу 2).

4. Задан вектор X размерностью n. Найти самую длинную последовательность его элементов, которая является арифметической или геометрической прогрессией.

5. ↓ Задан вектор X размерностью n. Построить такой вектор Y, каждый элемент которого y_i равен числу элементов из X в диапазоне индексов от 1 до $i-1$, превосходящих по модулю x_i.

6. ↓ Задан вектор X размерностью n. Найти максимальную по длине монотонную (т. е. либо неубывающую, либо невозрастающую) последовательность его элементов.

7. ↓ Задан вектор ***X*** размерностью ***n***. Найти такие ***i*** и ***j***, при которых значение $\sum_{k=i}^{j} a_k$ максимально.

8. ↓ Задан целочисленный вектор ***X*** размерностью ***n***. Определить наличие среди его элементов чисел Фибоначчи.

Рекомендации: *числа Фибоначчи определяются рекуррентными соотношениями* $f_0 = f_1 = 1,\ f_k = f_{k-1} + f_{k-2},\ k = 2,3,...$

9. Задан целочисленный вектор ***X*** размерностью ***n***. Определить длину его начального участка, для которого произведение числа вхождений степеней двойки на число вхождений чисел Фибоначчи максимально (см. задачу 8).

10. Задан целочисленный вектор ***X*** размерностью ***n***. Проверить, образуют ли его элементы перестановку начального отрезка последовательности натуральных чисел.

Рекомендации: *в перестановку натуральных чисел от* ***1*** *до* ***n*** *все значения из этого диапазона входят по одному разу.*

11. ↓ Задан целочисленный вектор ***X*** размерностью ***n***. Найти наибольший общий делитель его элементов.

Рекомендации: *для поиска наибольшего общего делителя двух целых чисел* ***m*** *и* ***n*** *использовать алгоритм Евклида*[♣]*:*

```
GCD(m, n)
  IF n = 0
    THEN вернуть результат m
    ELSE вызвать GCD(n, m mod n)
```

(где операция ***mod*** *вычисляет остаток от деления). Числа* ***m*** *и* ***n*** *предварительно упорядочить по убыванию.*

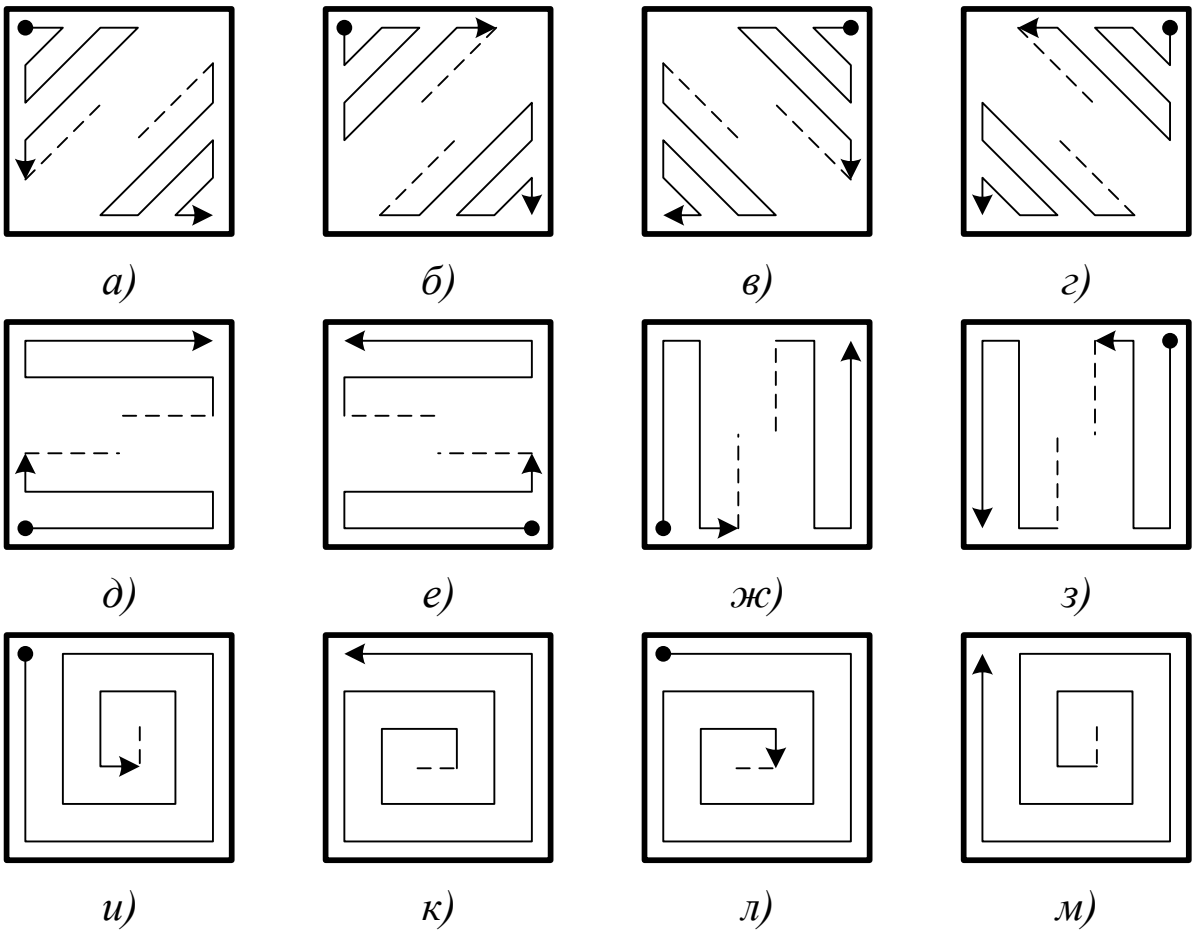
♣ ***GCD*** (*англ.* ***Greatest Common Divisor***) — наибольший общий делитель.

12. ↓ Задан целочисленный вектор ***X*** размерностью ***n***. Найти наименьшее общее кратное его элементов.

Рекомендации: *для поиска наименьшего общего кратного двух целых чисел* ***m*** *и* ***n*** *использовать их наибольший общий делитель (см. задачу 11).*

13. Задан целочисленный вектор ***X*** размерностью ***n***. Присвоить его элементам изображения в двоичной системе счисления натуральных чисел, начиная с заданного значения ***M***.

14. Задана матрица ***A*** размерностью $n \times n$. Вывести ее элементы в указанном порядке.



а) *б)* *в)* *г)*

д) *е)* *ж)* *з)*

и) *к)* *л)* *м)*

15. Задана матрица ***A*** размерностью $n \times m$. Найти строки матрицы, максимально удаленные друг от друга.

Рекомендации: *за расстояние между* ***i****-й и* ***k****-й строками матрицы принять значение*

$$\sqrt{\sum_{j=1}^{m}\left(a_{ij}-a_{kj}\right)^2}\ .$$

16. Задана матрица A размерностью $n\times m$. Найти максимальное из чисел, встречающихся в ней более одного раза.

17. ↓ Задана матрица A размерностью $n\times m$. Определить ее нормы.

Рекомендации: *нормы матрицы считать по формулам:*

$$\|A\|_k=\sqrt{\sum_i\sum_j a_{ij}^2},\quad \|A\|_l=\max_j\sum_i\left|a_{ij}\right|,\quad \|A\|_m=\max_i\sum_j\left|a_{ij}\right|.$$

18. Задана матрица A размерностью $n\times n$. Построить вектор длиной $2\cdot n-1$ из наибольших элементов диагоналей, параллельных побочной диагонали A.

19. Задана целочисленная матрица A размерностью $n\times m$. Найти количество попарно сходных строк матрицы.

Рекомендации: *две строки матрицы считать сходными, если множества значений их элементов эквивалентны.*

20. Задана целочисленная матрица A размерностью $n\times m$. Переставить строки матрицы в порядке возрастания их характеристик.

Рекомендации: *характеристикой строки матрицы считать сумму ее положительных четных элементов.*

21. Построить целочисленную матрицу A размерностью $n\times m$ по правилам

$$a_{ij}=\begin{cases}C_j^i, & i<j,\\ C_i^j, & i\ge j,\end{cases}$$

где C_i^j — число сочетаний из i элементов по j.

22. Задана матрица A размерностью $n\times m$. Определить, является ли она ортонормированной.

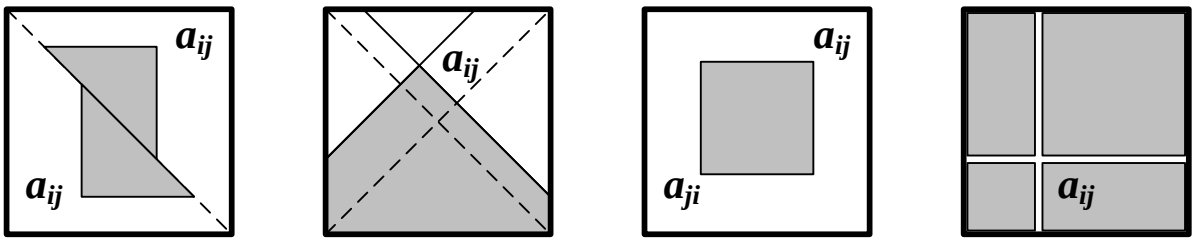
Рекомендации: *матрицу считать ортонормированной, если скалярное произведение любой пары ее различных строк равно* ***0****, а скалярное произведение каждой ее строки на себя равно* ***1****.*

23. Задана матрица A размерностью $n \times n$. Найти максимум среди сумм элементов диагоналей, параллельных ее главной диагонали.

24. Задана матрица A размерностью $n \times n$. Найти минимум среди сумм элементов диагоналей, параллельных ее побочной диагонали.

25. Заданы матрицы A и B размерностью $n \times n$. Построить матрицу C того же размера путем последовательного выполнения операций умножения каждой строки A на сумму элементов соответствующего столбца B и сложения каждого столбца полученной матрицы с произведением элементов соответствующей строки B.

26. Задана матрица A размерностью $n \times n$. Построить матрицу B того же размера, каждый элемент которой равен сумме квадратов элементов заштрихованной области в A.



а) *б)* *в)* *г)*

27. Задана матрица A размерностью $(2 \cdot n) \times (2 \cdot n)$, составленная из квадратов размером 2×2. Преобразовать A, упорядочив эти квадраты по значениям сумм их элементов.

Рекомендации: *если сумма элементов квадрата* A_1 *меньше, чем у* A_2*, то* A_1 *следует располагать выше или левее* A_2*.*

28. Задана матрица A размерностью $n \times m$. Найти ее седловые точки.

Рекомендации: *седловой точкой матрицы считать элемент* a_{ij}*, если он является минимальным в* i*-й строке и максимальным в* j*-м столбце или является максимальным в* i*-й строке и минимальным в* j*-м столбце.*

29. Задана матрица A размерностью $n \times m$. Найти максимальный из элементов ее строк, упорядоченных по возрастанию или по убыванию.

30. Задана матрица A размерностью $n \times m$. Переставить столбцы матрицы по возрастанию их характеристик.

Рекомендации: *характеристикой столбца матрицы считать сумму модулей его отрицательных нечетных элементов.*

31. ↑ Задана матрица A размерностью $n \times n$. Применяя операции перестановки строк и столбцов, преобразовать ее так, чтобы диагональные элементы новой матрицы были упорядочены по убыванию.

32. Задана матрица A размерностью $n \times m$. Подсчитать количество ее столбцов, которые составлены из различных чисел.

33. ↓ Задана матрица A размерностью $n \times m$. Преобразовать ее, последовательно переставив первую строку с последней, вторую — с предпоследней и т. д., затем переставить первый столбец — с последним, второй — с предпоследним и т. д.

34. Задана целочисленная матрица A размерностью $n \times m$. Среди ее строк, содержащих только нечетные элементы, найти ту, для которой $\sum_j \left|a_{ij}\right|$ максимально.

35. Задана целочисленная матрица A размерностью $n \times m$. Среди ее столбцов, содержащих только четные элементы, по модулю не превосходящие 10, найти тот, для которого $\prod_i \left|a_{ij}\right|$ минимально.

36. Задана целочисленная матрица A размерностью $n \times n$. Определить, является ли она «магическим квадратом».

Рекомендации: *матрицу считать «магическим квадратом», если суммы элементов всех ее строк и столбцов одинаковы.*

37. Задана матрица A размерностью $n \times m$. Пусть $M(A,i)$ — номер столбца A, в котором находится минимум i-й строки. Проверить выполнение условия $M(A,1) \le M(A,2) \le \ldots \le M(A,n)$.

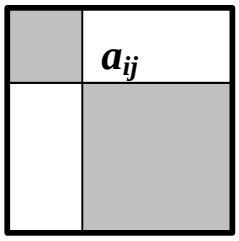
38. Задана матрица A размерностью $n \times m$. Построить матрицу B той же размерности путем сглаживания A.

Рекомендации: *элемент* b_{ij} *сглаженной матрицы находить как среднее арифметическое элемента* a_{ij} *и его соседей.*

39. ↓ Задана матрица A размерностью $n \times m$, каждая строка которой представляет координаты одного вектора. Определить, являются ли эти векторы попарно линейно независимыми.

Рекомендации: *два линейно зависимых вектора параллельны.*

40. Задана матрица A размерностью $n \times m$. Любой ее элемент a_{ij} определяет разбиение A на четыре клетки, хотя бы одна из которых не пуста.



Построить матрицу B такого же размера, в которой b_{ij} равен наименьшему из максимальных элементов непустых клеток, определяемых в A элементом a_{ij}.

41. Задана матрица A размерностью $n \times m$. Проверить, выполняется ли условие $a_{ij} \ge a_{i-1,j} + a_{i,j-1}$ для всех $i > 1$ и для всех $j > 1$.

42. ↑ Заданы матрицы A и B размерностью $n \times n$. Проверить, можно ли получить матрицу B из A применением конечного числа

раз операций транспонирования относительно главной и побочной диагоналей.

43. Задана матрица A размерностью $n \times m$. Подсчитать количество ее локальных минимумов.

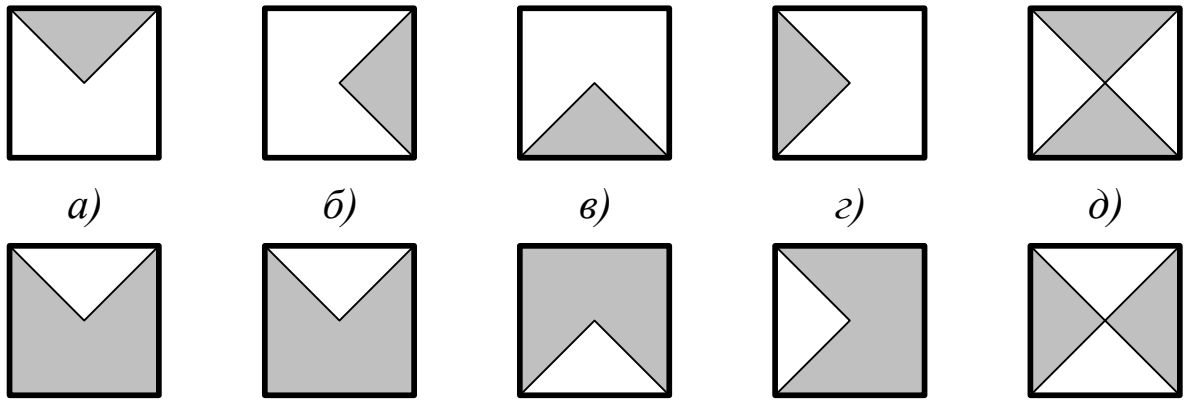
Рекомендации: *элемент матрицы считается локальным минимумом, если он строго меньше всех его соседей.*

44. Задана матрица A размерностью $n \times m$. Найти наибольший среди всех ее локальных минимумов (см. задачу 43).

45. Задана матрица A размерностью $n \times n$. Определить, становится ли она симметричной относительно главной диагонали после обнуления ее локальных минимумов (см. задачу 43).

46. Задана матрица A размерностью $n \times n$. Вычислить скалярное произведение строки, содержащей наибольший элемент A, на столбец, в котором находится наименьший элемент матрицы.

47. Задана матрица A размерностью $n \times n$. Найти наибольшее из значений, расположенных в ее заштрихованной части.



а) *б)* *в)* *г)* *д)*

е) *ж)* *з)* *и)* *к)*

48. Взаимно однозначное отображение элемента a_{ij} матрицы на себя можно задать с помощью двух целочисленных матриц. В позиции (i, j) в первой матрице указывается номер строки, куда перемещается данный элемент, а во второй матрице — номер столбца. Построить две матрицы, задающие транспонирование матрицы A размерностью $n \times n$.

49. Задан вектор X размерностью n. Построить следующую матрицу A размерностью $(m+1)\times(m+1)$

$$\begin{pmatrix} n & \sum x_i & \sum x_i^2 & \dots & \sum x_i^m \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \dots & \sum x_i^{m+1} \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \dots & \sum x_i^{m+2} \\ \dots & \dots & \dots & \dots & \dots \\ \sum x_i^m & \sum x_i^{m+1} & \sum x_i^{m+2} & \dots & \sum x_i^{2\cdot m} \end{pmatrix}$$

50. Задана матрица A размерностью $n\times m$, вектор X размерностью m и вектор Y размерностью $n+1$. Сформировать матрицу B размерностью $(n+1)\times(m+1)$ путем вставки в A после строки p вектора X и вставки после столбца q вектора Y.

51. Задана матрица A размерностью $n\times m$ и вектор X размерностью q. Заменить нулями элементы A с четной суммой индексов, если они встречаются в X.

52. Задана матрица A размерностью $(2\cdot n)\times(2\cdot n)$. Преобразовать ее путем перестановки блоков размером $n\times n$ указанным ниже образом.

а) *б)* *в)* *г)*

53. Построить целочисленную матрицу A размерностью $n\times m$, где $a_{ij}=(i!)^j$.

54. ↑ Заданы матрицы A и B размерностью $n\times n$. Получить матрицу $A^T\cdot(B-I)+C$, где I — единичная матрица, а элементы матрицы C вычисляются по формуле $c_{ij}=1/(i+j)$.

55. ↑ Заданы матрицы A и B размерностью $n\times n$. Получить матрицу $A^T\cdot(B^2-I)$, где I — единичная матрица.

56. ↑ Задана матрица A размерностью $n \times n$. Вычислить значение выражения $\sum_{i=0}^{m} A^i$.

Рекомендации: *возможно непосредственное применение формулы* $A^l = A \cdot A \cdot ... \cdot A$ (l *сомножителей). Однако, такой алгоритм неэффективен. Если* $l = 2 \cdot k$ *— четное число, то* $A^l = A^k \cdot A^k$. *Если* $l = 2 \cdot k + 1$ *— нечетное число, то* $A^l = A \cdot A^k \cdot A^k$. *Значение* A^k *вычисляется аналогично. Например, для получения* A^9 *достаточно* **4** *операции умножения: нахождение* A^2, A^4, A^8 *и* $A \cdot A^8$.

57. ↑ Задана матрица A размерностью $n \times n$. Привести ее к верхнему треугольному виду и вычислить ее определитель.

Рекомендации: *для приведения к треугольному виду использовать схему исключения Гаусса*

$$a_{ij} \leftarrow a_{ij} - \frac{a_{ik} \cdot a_{kj}}{a_{kk}}, \quad k = 1,...n-1, \quad i = k+1,...n, \quad j = k,...n.$$

Определитель треугольной матрицы находить как произведение элементов главной диагонали.

58. ↑ Задана матрица A размерностью $n \times n$ и вектор B размерностью n. Определить вектор X размерностью n, являющийся решением системы линейных уравнений $A \cdot X = B$.

Рекомендации: *для решения системы линейных уравнений матрицу* A *привести к треугольному виду (см. задачу 57). При обработке* i*-й строки* A *вектор* B *пересчитывать по схеме*

$$b_i \leftarrow b_i - \frac{a_{ik} \cdot b_k}{a_{kk}}, \quad k = 1,...n-1, \quad i = k+1,...n.$$

Элементы вектора решения находить по формулам

$$x_i \leftarrow \frac{1}{a_{ii}} \cdot \left[b_i - \sum_{j=i+1}^{n} a_{ij} \cdot x_j \right], \quad i = n, n-1, \ldots 1 .$$

Сумму считать равной нулю, если верхний предел суммирования меньше нижнего.

59. ↑ Задана матрица A размерностью $n \times n$. Вычислить обратную матрицу A^{-1}.

Рекомендации: *для вычисления обратной матрицы* $A^{-1} = Z$ *необходимо* n *раз решить систему линейных уравнений (см. задачу 58)* $A \cdot Z^{\langle i \rangle} = I^{\langle i \rangle}$, $i = 1, \ldots, n$, *где* $Z^{\langle i \rangle}$ *и* $I^{\langle i \rangle}$ *— i-е столбцы искомой обратной матрицы Z и единичной матрицы I. Приведение матрицы A к треугольному виду выполнить один раз.*

60. ↑ Задана матрица A размерностью $n \times n$. Вычислить ее определитель, используя схему разложения по строке (столбцу).

Рекомендации: *если используется разложение по первой строке, то* $\det A = \sum_{k} (-1)^{k+1} \cdot a_{1k} \cdot \det B_k$, *где матрица* B_k *получается вычеркиванием из A первой строки и k-го столбца. Решить задачу рекурсивно.*

ПРИЛОЖЕНИЕ В

ТЕКСТЫ

Варианты заданий для самостоятельного решения

1. ↓ Задан фрагмент текста. Найти в нем самое длинное симметричное слово.

2. ↓ Задан фрагмент текста. Найти в нем пары слов, из которых одно является обращением другого.

3. ↓ Задан фрагмент текста. Для каждого слова указать, сколько раз оно встречается в тексте.

4. Задан фрагмент текста. Для каждого предложения перечислить все слова, которые состоят из тех же букв, что и его первое слово.

5. Задан фрагмент текста. Найти в нем пару наиболее далеко удаленных слов заданной длины.

Рекомендации: *расстояние между двумя словами равно числу позиций между первыми символами слов.*

6. ↓ Задан фрагмент текста. Отредактировать его, удаляя слова, которые уже встречались в тексте раньше.

7. ↓ Задан фрагмент текста. Выбрать из него слова, встречающиеся ровно один раз, и вывести их в том порядке, как они появляются в тексте.

8. Задан фрагмент текста, все слова которого начинаются с различных букв. Если возможно, напечатать их так, чтобы последняя буква каждого слова совпадала с первой буквой следующего слова.

9. Заданы два фрагмента текста. Найти множество слов, которые встречаются в каждом из них.

10. Заданы два фрагмента текста, составленные из попарно различных слов. Определить, можно ли получить второй текст из первого удалением некоторых его слов.

11. Заданы два фрагмента текста. Найти их самое длинное общее слово.

12. Заданы два фрагмента текста. Найти самое короткое из слов первого фрагмента, которого нет во втором фрагменте.

13. ↓ Задан фрагмент текста. Среди его слов найти такое, которое имеет наибольшее число вхождений в текст.

14. ↓ Задан фрагмент текста. Проверить, действительно ли всякое симметричное его слово имеет четную длину.

15. ↓ Задан фрагмент текста. Отредактировать его, удаляя слова, которые встречаются заданное число раз.

16. Задан фрагмент текста. Найти в нем слово, в котором доля гласных максимальна.

17. Задан фрагмент текста. Переставить его слова в соответствии с ростом доли в них согласных.

18. ↓ Задан фрагмент текста. Заменить окончание ***ing*** каждого слова, встречающегося в тексте, на ***ed***.

19. ↓ Задан фрагмент текста. Отредактировать его, удаляя слова, целиком составленные из вхождений не более чем двух букв.

20. ↓ Задан фрагмент произвольного текста. Отредактировать его так, чтобы между словами был ровно один пробел, а между предложениями — два пробела.

21. Задан фрагмент произвольного текста. Отредактировать его, заменяя изображения чисел последовательностью числовых триад и пробелов так, чтобы в дробной части числа располагались полные триады (например, ***22452,5276*** должно представиться как ***22 452,527 600***).

22. ↓ Задан фрагмент произвольного текста. Изменить порядок следования в нем слов на противоположный.

23. ↓ Задан фрагмент произвольного текста. В каждом слове заменить порядок следования в нем букв на противоположный.

24. Задан фрагмент текста, не содержащий скобок. Выполнить его сжатие, т е. заменить всякую последовательность, составленную более чем из трех вхождений одного и того же символа ***s***, на ***(k)s***, где $k > 3$ — количество повторений ***s***.

25. Задан фрагмент текста. Отредактировать его, заменяя всякое вхождение слов вида $\alpha\beta\tilde{\alpha}$ на β, где α, β — подслова, $\tilde{\alpha}$ — обращение слова α.

26. Задан фрагмент текста, в котором есть вхождения каждой из букв латинского алфавита. Пусть C_α —слово, начинающееся с α. Проверить, действительно ли длины слов упорядочены в соответствии с порядком букв алфавита (т. е. C_A не длиннее C_B и т. д.).

27. Задан фрагмент текста. Напечатать его так, чтобы каждое слово целиком находилось в одной и той же строке (т. е. избавиться от переносов).

28. ↑ Задан фрагмент текста, образованный словами русского языка. Отредактировать его, представляя слова, оканчивающиеся на ***-онок*** или ***-енок***, во множественном числе.

Рекомендации: *для большинства существительных, оканчивающихся на* ***-онок*** *и* ***-енок****, множественное число образуется от другой основы. Как правило, в ней перед последней буквой* ***т*** *пишется* ***а****, если предыдущая буква шипящая, в противном случае пишется* ***я*** *(например,* ***цыпленок — цыплята, мышонок — мышата*** *и т. д.). Имеются слова — исключения, основные из которых:* ***ребенок — дети, бесенок — бесенята, опенок — опята, звонок — звонки, позвонок — позвонки, подонок — подонки, колонóк — колонки, жаворонок — жаво-***

ронки, бочонок — бочонки *(малоупотребительные исключения не рассматриваются).*

29. ↑ Задан фрагмент текста, среди символов которого имеются пробелы и отсутствуют скобки, образованный словами русского языка. За существительными мужского рода, оканчивающимися на ***-ок***, следуют конструкции ***(и)***, ***(р)***, ***(д)***, ***(в)***, ***(т)***, ***(п)***, определяющие нужный падеж (именительный, родительный, дательный, винительный, творительный, предложный). Отредактировать текст, представляя все существительные в указанном падеже.

Рекомендации: *при склонении существительных мужского рода, оканчивающихся на* ***-ок****, буква* ***о*** *может выступать как беглая гласная (****кружок — кружка****,* ***белок — белком*** *и т. д.). При склонении некоторых слов гласная* ***о****, тем не менее, сохраняется. Во-первых, это слова из трех букв (****ток****,* ***сок*** *и т. д.). Во-вторых, это слова* ***скок, блок, волок, восток, шток*** *и слова, основа которых оканчивается на подобные сочетания букв (т. е.* ***перескок****,* ***пищеблок****,* ***юго-восток*** *и т. д.). В-третьих, это слова-исключения, среди которых* ***брелок, щелок, войлок, челнок, зарок, срок, урок, знаток, поток, сток, артишок****.*

30. ↑ Задан фрагмент текста, образованный словами русского языка. Выполнить расстановку переносов в тексте.

Рекомендации: *как показывают многочисленные эксперименты, разбиение русского слова на части для переноса с одной строки на другую с большой вероятностью выполняется правильно, если пользоваться следующими приемами:*

- *две идущие подряд гласные можно разделить, если первой из них предшествует согласная, а за второй идет хотя бы одна буква (буква* ***й*** *вместе с предшествующей гласной рассматривается как единое целое);*
- *две идущие подряд согласные можно разделить, если первой из них предшествует гласная, а в той части слова,*

которая идет за второй согласной, имеется хотя бы одна гласная (буквы ***ь*** *и* ***ъ*** *вместе с предшествующей согласной рассматриваются как единое целое);*

- *если не удается применить перечисленные приемы, то следует попытаться разбить слово так, чтобы первая часть содержала более чем одну букву и оканчивалась на гласную, а вторая содержала хотя бы одну гласную.*

Вероятность правильного разбиения увеличивается, если предварительно воспользоваться хотя бы неполным списком приставок, содержащих гласные, и попытаться, прежде всего, выделить из слова такую приставку.

31. Задан фрагмент текста, образованный словами русского языка. Вывести его с выравниванием правой границы строк.

Рекомендации: *необходимо решить задачу выбора подходящих промежутков между словами. За счет изменения групп пробелов следует добиться, чтобы в одной строке размеры этих групп различались не более чем на единицу, а в конце строки пробелы отсутствовали.*

32. ↑ Задан фрагмент текста, образованный словами русского языка. Выполнить расстановку в нем переносов и перегруппировку слов по строкам так, чтобы вывод осуществлялся строками заданной длины ***m***, выровненных по правой границе (см. задачи 30 и 31).

33. Задано натуральное число. Записать его словами (например, ***22452*** должно представиться как «двадцать две тысячи четыреста пятьдесят два»).

34. ↑ Задано натуральное число и символ ***и***, ***р***, ***д***, ***в***, ***т***, или ***п***, указывающий падеж (именительный, родительный, дательный, винительный, творительный, предложный). Записать это число словами в соответствующем падеже (например, ***22452д*** должно предста-

виться как «двадцати двум тысячам четыремстам пятидесяти двум»).

35. Задано натуральное число с начальным значением ***0***, которое играет роль счетчика. Записать словами его последовательные значения с указанием объекта счета в соответствующем числе и падеже (например, ***«один человек»***, ***«одна тысяча сто двадцать три человека»*** и т. п.).

36. Задано натуральное число. Записать его изображение в римской системе счисления (например, ***1998*** должно представиться как ***MCMXCVIII***).

Рекомендации: *в римской системе счисления в качестве цифр используются* ***I(1)***, ***V(5)***, ***X(10)***, ***L(50)***, ***C(100)***, ***D(500)***, ***M(1000)***. *Вес цифры не зависит от ее положения в числе. Значение числа определяется как сумма или разность цифр. Если меньшая цифра стоит слева от большей, она вычитается, если справа — прибавляется.*

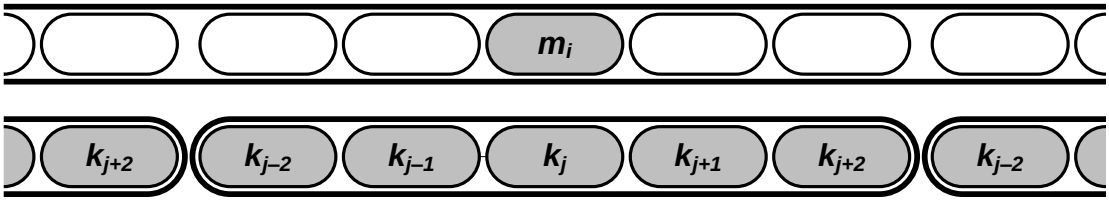
37. ↑ Задано изображение числа в римской системе счисления и символ ***и***, ***р***, ***д***, ***в***, ***т***, или ***п***, указывающий падеж (именительный, родительный, дательный, винительный, творительный, предложный). Записать это число словами в соответствующем падеже (например, ***MCMXCVIIIд*** должно представиться как «одной тысяче девятистам девяносто восьми»).

38. Задан фрагмент текста, образованный словами русского языка. Зашифровать его с помощью кода Цезаря. Выполнить расшифровку текста.

Рекомендации: *код Цезаря заключается в следующем. При шифровке каждая буква заменяется другой, отстоящей от нее в алфавите на* **n** *позиций. Организация алфавита считается циклической. Если* ***n = 3****, то вместо* ***А*** *используется* ***Г****, вместо* ***Я*** *—* ***В****. Расшифровка выполняется в обратном порядке.*

39. ↑ Задан фрагмент текста, образованный словами русского языка. Зашифровать его с помощью кода Виженера. Выполнить расшифровку текста.

Рекомендации: *код Виженера использует кодовое слово* **k** *заданной длины, которое последовательно сопоставляется с кодируемым сообщением* **m**, *начиная с первого символа.*



При шифровке каждая буква $\mathbf{m_i}$ *заменяется другой, отстоящей от нее в алфавите на* $\mathbf{k_j}$ *позиций (*$\mathbf{k_j}$ *— номер* **j***-й буквы кодового слова в алфавите, организация алфавита циклическая). Расшифровка выполняется в обратном порядке.*

40. ↑ Задан фрагмент текста длиной **n** символов, образованный словами русского языка. Зашифровать его с помощью сгенерированного случайным образом числового ключа. Выполнить расшифровку текста.

Рекомендации: *ключ* **k** *является последовательностью из* **n** *чисел, выбираемых случайным образом в диапазоне от* **1** *до* **q**, *(***q** *— число символов алфавита). Каждая буква кодируемого сообщения* $\mathbf{m_i}$ *заменяется другой, отстоящей от нее в алфавите на* $\mathbf{k_j}$ *позиций (*$\mathbf{k_j}$ *— значение* **j***-й цифры ключа, организация алфавита циклическая). Расшифровка выполняется в обратном порядке.*

41. ↑ Задан фрагмент текста. Зашифровать его с помощью числового ключа. Выполнить расшифровку текста.

Рекомендации: *ключ является перестановкой чисел от* **1** *до* **k**, *которая применяется при шифровке исходного текста к каждой из последовательно идущих групп по* **k** *символов. Например, при* $\mathbf{k = 4}$ *и перестановке* **3, 1, 4, 2** *группа символов* s_1, s_2, s_3, s_4

заменяется на s_3, s_1, s_4, s_2. *Если в последней группе меньше* k *символов, ее дополняют пробелами.*

42. ↑ Задан фрагмент текста. Переписать его в матрицу размерностью $n \times m$. Зашифровать текст путем чтения элементов матрицы при обходе ее по спирали против часовой стрелки, начиная с нижнего правого угла. Выполнить расшифровку текста.

Рекомендации: *параметры* n *и* m *выбрать самостоятельно из условия экономичного размещения текста.*

43. ↑ Задан фрагмент текста. Составить матричный ключ и зашифровать текст с его помощью. Выполнить расшифровку текста.

Рекомендации: *ключ* k *представляет собой матрицу размерностью* $n \times n$, *число элементов которой должно превышать число символов в тексте. Ключ состоит из* $n^2/4$ *нулей и* $3 \cdot n^2/4$ *единиц. Он совмещается с матрицей размерностью* $n \times n$, *в которую при шифровании на позиции нулей ключа последовательно переносятся* $n^2/4$ *символов исходного текста. После этого ключ поворачивается на* $90°$, *и шифруется группа из следующих* $n^2/4$ *символов. Так повторяется еще два раза. Расшифровка выполняется в обратном порядке. Параметр* n *выбрать самостоятельно из условия экономичного размещения текста. При составлении ключа следить, чтобы из элементов* $k_{ij}, k_{n-i+1,j}, k_{i,n-j+1}, k_{n-i+1,n-j+1}$ *только один равнялся нулю.*

44. ↑ Задан фрагмент текста, состоящий только из десятичных цифр. Зашифровать текст, заменяя каждую четную цифру заключенной в круглые скобки последовательностью символов «+», а каждую нечетную цифру — последовательностью символов «–». Длина последовательности должна быть равна числу, изображаемому цифрой. Выполнить расшифровку текста.

45. Задан фрагмент текста длиной n символов, составленного из 0 и 1. Зашифровать его, используя правила

$$B_i = \begin{cases} 1, & A_i = A_{i-1}, \\ 0, & A_i \neq A_{i-1}, \end{cases}$$

где $i = 2,\ldots,n$. Выполнить расшифровку текста.

46. ↑ Задан фрагмент текста, составленного из **0** и **1**, предназначенный для передачи по каналам связи в условиях действия помех. Выполнить помехоустойчивое шифрование и последующее дешифрование текста. При возникновении ошибки выдать соответствующее сообщение.

Рекомендации: *для снижения вероятности ошибки при передаче каждый символ утраивается. Каждая триада символов дополняется* **0** *или* **1** *так, чтобы число единиц в сообщении было четным. Например, последовательность символов* **101** *будет зашифрована как* **1110001110**, *а* **100** *— как* **1110000001**. *При расшифровке три последовательные цифры заменяются той, которая встречается среди них не менее двух раз. С помощью последнего символа оценивается правильность передачи сообщения.*

47. ↑ Заданы две строки, состоящие только из десятичных цифр и представляющие очень длинные целые чисел. Сформировать строки, хранящие сумму и разность этих чисел.

48. ↑ Задана строка, изображающая константное выражение в постфиксной форме (см. п. 3.2). Вычислить значение этого выражения.

Рекомендации: *для решения задачи необходим дополнительный массив, организованный по принципу стека. Выражение просматривается слева направо. Если встречается операнд (число), то его значение запоминается в стеке. Если встречается знак операции, то из стека извлекаются два последних элемента (операнды), над ними выполняется операция, ее ре-*

зультат записывается в стек. После просмотра строки в стеке будет находиться значение выражения.

49. ↑ Задана строка, изображающая выражение в инфиксной форме (см. п. 3.2). Перевести это выражение в постфиксную форму.

Рекомендации: *для решения задачи необходим дополнительный массив, организованный по принципу стека. Для хранения выражений используются строки* **infix** *и* **postfix.** *Алгоритм решения задачи следующий. В стек записывается открывающая скобка, и строка* **infix** *просматривается слева направо.*

- *Если встречается операнд (число или переменная), то его изображение переносится в строку* **postfix.**
- *Если встречается открывающая скобка, то она запоминается в стеке.*
- *Если встречается закрывающая скобка, то из стека извлекаются находящиеся там знаки операций до ближайшей открывающей скобки (она также удаляется из стека), которые в порядке извлечения записываются в строку* **postfix.**
- *Если встречается знак операции, то из стека извлекаются (до ближайшей скобки) знаки операций, приоритеты которых не меньше приоритета текущей операции, которые записываются в строку* **postfix***, после чего знак текущей операции записывается в стек.*

В заключение из стека извлекаются находящиеся там знаки операций до ближайшей открывающей скобки (она также удаляется из стека), которые в порядке извлечения записываются в строку **postfix.**

50. ↑ Задана строка, изображающая выражение в постфиксной форме. Перевести это выражение в инфиксную форму.

Рекомендации: *алгоритм решения задачи аналогичен тому, который рассмотрен в задаче 49. Лишние скобки не печатать.*

ПРИЛОЖЕНИЕ Г

ГЕОМЕТРИЯ

Варианты заданий для самостоятельного решения

1. На плоскости задано множество точек. Выбрать в нем две различные точки так, чтобы проходящая через них прямая делила оставшееся множество на группы, различающиеся по числу точек минимально.

2. На плоскости задано множество точек. Выбрать три различные точки так, чтобы проходящая через них окружность делила это множество на группы, различающиеся по числу точек минимально. Определить радиус и центр этой окружности.

Рекомендации: *центр окружности, проходящей через три различные точки, находится на пересечении перпендикуляров, восстановленных к сторонам треугольника, образованного этими точками, в их серединах. Радиусом окружности является расстояние от ее центра до любой из точек.*

3. ↓ На плоскости задано множество точек. Определить, можно ли через любые две из них провести прямую, оставляющую все остальные точки с одной своей стороны.

4. На плоскости задано множество точек. Определить радиус и центр окружности, на которой лежит наибольшее число точек (см. задачу 2).

5. ↓ На плоскости задано множество точек. Выбрать пару точек с максимальным расстоянием между ними.

6. На плоскости заданы два множества точек. Найти расстояние между ними.

Рекомендации: *расстоянием между двумя множествами точек считать расстояние между наиболее близко расположенными точками этих множеств.*

7. На плоскости задано множество точек, являющимися вершинами многоугольника (не обязательно выпуклого), заданными в порядке обхода его границы. Определить площадь многоугольника.

Рекомендации: *площадь многоугольника находится как сумма площадей составляющих его треугольников.*

8. ↓ В трехмерном пространстве задано множество точек. Выбрать одну из них так, чтобы шар заданного радиуса с центром в этой точке содержал максимальное число точек множества.

9. На плоскости задано множество прямых линий. Определить, находятся ли они в общем положении.

Рекомендации: *Прямые находятся в общем положении, если все они различны, никакие две из них не параллельны и никакие три не пересекаются в одной точке.*

10. ↓ В трехмерном пространстве задано множество материальных точек. Найти ту из них, которая наиболее близко расположена к центру тяжести этого множества.

Рекомендации: *материальная точка, кроме координат, характеризуется массой* $\boldsymbol{m}$*. Координаты центра тяжести множества этих точек рассчитываются по формулам:*

$$x_C = \frac{\sum x_i \cdot m_i}{\sum m_i}, \quad y_C = \frac{\sum y_i \cdot m_i}{\sum m_i}, \quad z_C = \frac{\sum z_i \cdot m_i}{\sum m_i}.$$

11. В трехмерном пространстве задано множество материальных точек (см. задачу 10). Каждая из точек с максимальной массой исчезает, теряя десятую часть своей массы и раздавая оставшуюся массу поровну всем остальным точкам. Определить суммарную массу множества в тот момент, когда все оставшиеся в нем материальные точки имеют одинаковую массу.

12. На плоскости задано множество точек. Перечислить точки в соответствии с их порядком.

Рекомендации: *точки* **u** *и* **v** *упорядочены, если* **v** *располагается ниже или правее* **u**.

13. На плоскости заданы два множества точек. Построить пересечение и разность этих множеств.

14. На плоскости задано множество точек. Определить его регулярность.

Рекомендации: *множество точек считать регулярным, если для каждой пары точек можно указать третью точку, образующую вместе с ними правильный треугольник.*

15. На плоскости задано n множеств точек. Выбрать множество, содержащее точку, которая принадлежит наибольшему числу множеств.

16. На плоскости заданы множества точек P и окружностей C. Выбрать две точки из P так, чтобы проходящая через них прямая пересекала максимальное количество окружностей из C.

17. На плоскости заданы множества точек P и прямых L. Выбрать две точки из P так, чтобы проходящая через них прямая была параллельна наибольшему количеству прямых из L.

18. На плоскости заданы множество точек P и точка $d \in P$. Подсчитать количество различных (не обязательно упорядоченных) троек точек из P, образующих совместно с d параллелограмм.

19. ↓ На плоскости задано множество точек. Выбрать три различные точки так, чтобы проходящая через них окружность содержала внутри себя наибольшее количество точек. Определить радиус и центр этой окружности (см. задачу 2).

20. На плоскости задано множество точек. Выбрать три различные точки так, чтобы образованный ими треугольник делил это

множество на две группы, различающиеся по числу точек минимально. Определить длины сторон треугольника.

21. ↓ В трехмерном пространстве задано множество попарно различных плоскостей. Выбрать максимальное подмножество попарно непараллельных плоскостей.

22. В трехмерном пространстве задано множество точек. Найти минимальный радиус шара с центром в любой из этих точек, содержащего ровно ***n*** точек множества.

23. ↓ На плоскости задано множество точек. Выбрать три различные не лежащие на одной прямой точки, которые образуют треугольник наибольшего периметра.

24. ↓ На плоскости задано множество точек. Выбрать три различные не лежащие на одной прямой точки, которые образуют треугольник наименьшей площади.

25. ↑ На плоскости задано множество точек, причем никакие три из них не лежат на одной прямой. Определить число таких треугольников с вершинами в этих точках, что никакие два из них не пересекаются и не содержат друг друга.

26. На плоскости задано множество точек. Выбрать три различные точки так, чтобы внутри образованного ими треугольника содержалось максимальное количество точек множества.

27. На плоскости задано множество точек. Выбрать две различные точки так, чтобы окружности заданного радиуса с центрами в этих точках содержали внутри себя одинаковое количество точек.

28. На плоскости заданы множество точек ***P*** и окружность. Выбрать из ***P*** две различные точки так, чтобы проходящая через них прямая пересекала окружность и делила подмножество ее внутренних точек на группы, различающиеся по числу точек минимально.

29. ↓ На плоскости задано два непересекающихся множества точек. Определить, существует ли окружность, проходящая через $k \geq 3$ точек каждого из множеств (см. задачу 2).

30. На плоскости задано два непересекающихся множества точек. Выбрать из первого из них три различные точки, образующие треугольник, который содержит строго внутри себя равное количество точек первого и второго множеств.

31. ↓ На плоскости задано два непересекающихся множества точек. Определить, существует ли окружность, проходящая через $k \geq 3$ точек первого множества и содержащая строго внутри себя m точек второго множества (см. задачу 2).

32. ↓ На плоскости задано множество попарно различных прямых линий. Выбрать из них прямую, имеющую максимальное число пересечений с остальными прямыми.

33. На плоскости задано два непересекающихся множества точек. Определить, существует ли окружность, проходящая через $k \geq 3$ точек первого множества и содержащая строго внутри равное число точек обоих множеств (см. задачу 2).

34. На плоскости задано два непересекающихся множества точек. Выбрать из первого из них три различные точки, образующие треугольник, который содержит строго внутри себя все точки второго множества и имеет минимальную площадь.

35. На плоскости задано два непересекающихся множества точек. Выбрать из первого из них четыре различные точки, образующие прямоугольник, который содержит строго внутри себя все точки второго множества и имеет минимальную площадь.

36. На плоскости задано два непересекающихся множества точек. Выбрать из первого из них три различные точки так, чтобы проходящая через них окружность содержала строго внутри себя все точки второго множества и ограничивала круг минимальной площади (см. задачу 2).

37. На плоскости задано множество точек. Найти ромб наибольшей площади с вершинами — точками этого множества.

38. На плоскости задано множество точек. Подсчитать количество равносторонних треугольников с вершинами — точками этого множества и различными длинами сторон.

39. На плоскости задано множество точек. Выделить в нем максимальное подмножество точек, лежащих на одной прямой.

40. ↑ На плоскости задано множество точек. Найти множество всех различных выпуклых четырехугольников с вершинами — точками этого множества.

41. ↑ На плоскости задано множество точек. Найти множество всех различных остроугольных треугольников с вершинами — точками этого множества.

42. ↑ На плоскости заданы множество точек и окружность радиусом $\boldsymbol{R}$ с центром в начале координат. Найти множество всех треугольников с вершинами в заданных точках, имеющих непустое пересечение с окружностью.

43. На плоскости задано множество точек. Выбрать три различные точки так, чтобы разность между площадью треугольника с вершинами в этих точках и площадью круга, ограниченного проходящей через эти точки окружностью, была минимальной (см. задачу 2).

44. На плоскости задано множество точек. Выбрать три различные точки так, чтобы сторонам образованного ими треугольника принадлежало максимальное число точек заданного множества.

45. ↑ На плоскости задано множество точек. Построить два таких треугольника с вершинами — точками этого множества, чтобы один треугольник лежал строго внутри другого.

46. ↑ На плоскости задано множество окружностей. Выбрать максимальное подмножество попарно не связанных друг с другом окружностей.

Рекомендации: *две окружности считать связанными, если они пересекаются, либо существует третья окружность, связанная с ними.*

47. На плоскости задано множество точек. Найти все различные содержащие более двух точек подмножества точек, лежащих на одной прямой.

48. ↓ На плоскости задано множество точек. Определить радиус и центр окружности минимального радиуса, проходящей хотя бы через три различные точки (см. задачу 2).

49. ↓ На плоскости задано множество точек. Найти точку, сумма расстояний от которой до остальных точек минимальна.

50. На плоскости заданы множества точек ***P*** и окружностей ***C***. Определить, есть ли на плоскости точка, принадлежащая всем кругам, ограниченным окружностями из ***C***.

51. На плоскости задано множество окружностей. Определить, есть ли среди них три попарно пересекающиеся.

52. ↑ На плоскости задано множество окружностей. Найти все уединенные окружности.

Рекомендации: *две окружности считать уединенными, если они не имеют общих точек с остальными окружностями, не лежат целиком внутри и не заключают внутри себя какой-либо из остальных окружностей.*

53. На плоскости задано множество точек, из которых никакие три точки не лежат на одной прямой. Найти число медиан этого множества.

Рекомендации: *медианой множества считать прямую, соединяющую две точки множества, с обеих сторон от которой равное число точек.*

54. ↑ На плоскости задано множество попарно различных точек. Найти выпуклый многоугольник с вершинами в некоторых из точек, который содержит все точки множества.

Рекомендации: *многоугольник представить последовательностью вершин.*

55. На плоскости задано множество попарно различных точек, последовательность которых определяет ломаную линию. Определить, имеет ли эта линия самопересечения.

56. На плоскости задано множество точек, причем никакие три из них не лежат на одной прямой. Определить, имеются ли среди них три вершины прямоугольника, и вычислить координаты четвертой вершины.

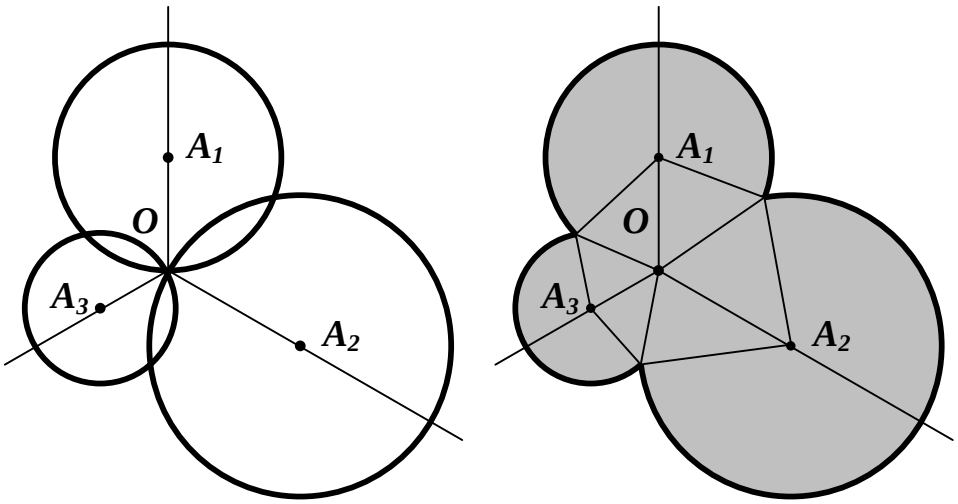
57. ↑ На плоскости задано множество (возможно пересекающихся) прямоугольников со сторонами, параллельными координатным осям. Найти площадь образованной ими фигуры.

58. ↑ На плоскости заданы три точки, являющиеся вершинами треугольника, с координатами $(0,0)$, $(0,1)$ и (x, y). Найти точку, сумма расстояний от которой до вершин треугольника минимальна (разрешается использовать методы приближенных вычислений).

Рекомендации: *по теореме Штейнера для треугольника с углами, не превосходящими* $2 \cdot \pi/3$*, эта точка совпадает с точкой Торричелли, т. е. точкой, из которой все стороны треугольника видны под углом* $2 \cdot \pi/3$*. Если в треугольнике имеется угол, больший* $2 \cdot \pi/3$*, то решением задачи будет вершина этого угла.*

59. ↑ На плоскости из общей точки O проведены три луча. Углы между соседними лучами равны $2 \cdot \pi/3$. На лучах выбраны точ-

ки A_1, A_2 и A_3, из которых как из центров проведены окружности, проходящие через точку O. Считая расстояния OA_1, OA_2 и OA_3 известными, вычислить площадь заштрихованной фигуры (ее разбиение на элементарные составляющие показано на рисунке).



60. ↑ Обобщение задачи 58. Рассматривается $n \geq 2$ лучей, проведенных из общей точки O, на которых выбраны точки $A_1, A_2, ..., A_n$. Углы между соседними лучами равны $2 \cdot \pi / n$. Считая расстояния $OA_1, OA_2, ..., OA_n$ известными, вычислить площадь фигуры, аналогичной описанной в предыдущей задаче.

ПРИЛОЖЕНИЕ Д

МНОЖЕСТВА

Варианты заданий для самостоятельного решения

1. ↓ Задать множества перечислением их элементов. Доказать справедливость следующих высказываний

1) $A \setminus (B \cup C) = (A \setminus B) \cap (A \setminus C)$,

2) $A \setminus (B \cap C) = (A \setminus B) \cup (A \setminus C)$,

3) $A \setminus (A \setminus B) = A \cap B$,

4) $A \setminus B = A \setminus (A \cap B)$,

5) $A \cap (B \setminus C) = (A \cap B) \setminus (A \cap C) = (A \cap B) \setminus C$,

6) $(A \setminus B) \setminus C = (A \setminus C) \setminus (B \setminus C)$,

7) $A \cup B = A \cup (B \setminus A)$,

8) $(A \cap B) \cup (A \cap \overline{B}) = A$,

9) $(A \cup B) \cap (A \cup \overline{B}) = A$,

10) $(\overline{A} \cup B) \cap A = A \cap B$,

11) $(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C)$,

12) $A \setminus (B \setminus C) = (A \setminus B) \cup (A \cap C)$,

13) $A \setminus (B \cup C) = (A \setminus B) \setminus C$,

14) $A \cup B \subset C \Leftrightarrow A \subset C \;\&\; B \subset C$,

15) $A \subset B \cap C \Leftrightarrow A \subset B \;\&\; A \subset C$,

16) $A \subset B \cup C \Leftrightarrow A \cap \overline{B} \subset C$,

17) $A \subset B \Rightarrow C \setminus B \subset C \setminus A$,

18) $A \subset B \Rightarrow \overline{B} \subset \overline{A}$,

19) $A \cap B = A \cup B \Rightarrow A = B$,

20) $A = \overline{B} \Leftrightarrow A \cap B = \varnothing \;\&\; A \cup B = U$,

21) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$,

22) $(A \cup B) \cap A = (A \cap B) \cup A = A$,

23) $A \cap (B \setminus A) = \varnothing$,

24) $(A \cap B) \cup (C \cap D) = (A \cup C) \cap (B \cup C) \cap (A \cup D) \cap (B \cup D)$,

25) $A \subset B \Leftrightarrow A \cup B = B \Leftrightarrow A \cap B = A \Leftrightarrow A \setminus B = \varnothing \Leftrightarrow \overline{A} \cup B = U$.

2. ↓ Задать множества перечислением их элементов. Доказать справедливость следующих высказываний:

1) $\bigcup\limits_{i \in I} \bigcup\limits_{j \in J} A_{ij} = \bigcup\limits_{j \in J} \bigcup\limits_{i \in I} A_{ij}$,

2) $\bigcap\limits_{i \in I} \bigcap\limits_{j \in J} A_{ij} = \bigcap\limits_{j \in J} \bigcap\limits_{i \in I} A_{ij}$,

3) $\bigcup\limits_{i \in I} \bigcap\limits_{j \in J} A_{ij} \subset \bigcap\limits_{i \in I} \bigcup\limits_{j \in J} A_{ij}$,

4) $\overline{\bigcup\limits_{i \in I} A_i} = \bigcap\limits_{i \in I} \overline{A_i}$,

5) $\overline{\bigcap\limits_{i \in I} A_i} = \bigcup\limits_{i \in I} \overline{A_i}$,

6) $\bigcup\limits_{i \in I} A_i \cup \bigcup\limits_{i \in I} B_i = \bigcup\limits_{i \in I} (A_i \cup B_i)$,

7) $\bigcup\limits_{i\in I}\left(B\cap A_i\right)=B\cap\left(\bigcup\limits_{i\in I}A_i\right)$,

8) $\bigcap\limits_{i\in I}\left(B\cup A_i\right)=B\cup\left(\bigcap\limits_{i\in I}A_i\right)$.

3. ↓ Задать множества перечислением их элементов. Доказать, что $(A\times B)\cup(C\times D)\subset(A\cup C)\times(B\cup D)$. При каких A, B, C и D включение можно заменить равенством?

4. ↓ Задать множества перечислением их элементов. Доказать справедливость следующих высказываний:

1) $(A\cup B)\times C=(A\times C)\cup(B\times C)$,

2) $(A\cap B)\times(C\cap D)=(A\times C)\cap(B\times D)$,

3) $A\times(B\setminus C)=(A\times B)\setminus(A\times C)$,

4) $(A\setminus B)\times C=(A\times C)\setminus(B\times C)$.

5. Для натуральных чисел A и B определим операцию

$$A\oplus B=A-B+A \mod B,$$

где ***mod*** — вычисление остатка от деления. Сформировать множество пар натуральных чисел, не превосходящих N, для которых $A\oplus B=B\oplus A$.

6. Сформировать множество простых чисел, не превосходящих N, двоичная запись которых представляет собой симметричную последовательность нулей и единиц, начинающуюся единицей.

7. Сформировать множество натуральных чисел, не превосходящих N, делителями которых являются только числа 2, 3 и 5.

8. Сформировать множество натуральных чисел, не превосходящих ***N***, в двоичном представлении которых номера ненулевых разрядов образуют арифметическую прогрессию.

9. Сформировать множество натуральных чисел, не превосходящих ***N***, которые равны сумме кубов своих цифр.

10. Сформировать множество таких пар двузначных натуральных чисел ***M*** и ***N***, что значение произведения $M \times N$ не изменится, если поменять местами цифры каждого из сомножителей (такой парой будет, например, ***38*** и ***83***).

11. Сформировать множество натуральных чисел, не превосходящих ***N***, которые нацело делятся на каждую из своих цифр.

12. Заданы три натуральных числа ***A***, ***B*** и ***N***. Сформировать множество натуральных чисел, не превосходящих ***N***, которые можно представить в виде суммы (произвольного числа) слагаемых, каждое из которых — ***A*** или ***B***.

13. Сформировать множество натуральных чисел, не превосходящих ***N***, десятичная запись которых есть строго возрастающая или строго убывающая последовательность цифр.

14. Сформировать множество натуральных чисел, не превосходящих ***N***, которые могут быть представлены в виде суммы квадратов двух различных натуральных чисел.

15. Сформировать множество натуральных чисел, не превосходящих ***N***. Заменить каждое из них числом, которое получается записью десятичных цифр исходного числа в обратном порядке.

16. Подсчитать количество различных значащих цифр в десятичной записи натурального числа ***N*** и напечатать в порядке возрастания все цифры, не входящие в эту запись.

17. Сколько существует целых чисел от ***1*** до ***16500***, которые а) не делятся на ***5***; б) не делятся ни на ***5***, ни на ***3***; в) не делятся ни на ***5***, ни на ***3***, ни на ***11***?

18. Сколько существует целых чисел от 1 до 33000, которые не делятся ни на 3, ни на 5, но делятся на 11?

19. Сколько существует натуральных чисел, меньших 1000, которые делятся на 3? На 5? На 15? Не делятся ни на 3, ни на 5?

20. Сколько существует натуральных чисел, меньших 1000, которые не делятся ни на 5, ни на 7?

21. Сколько существует шестизначных чисел, в записи которых есть хотя бы одна четная цифра?

22. Имеется множество C, состоящее из N элементов. Сколькими способами можно выбрать в C два подмножества A и B так, чтобы а) множества A и B не пересекались; б) множество A содержалось бы в множестве B?

23. Можно ли разбить множество целых чисел на три подмножества так, чтобы для любого целого значения N числа N, $N-50$, $N+1987$ принадлежали трём разным подмножествам?

24. Доказать, что можно разбить все множество натуральных чисел на 100 непустых подмножеств так, чтобы в любой тройке a, b, c такой, что $a+99\cdot b=c$, нашлись два числа из одного подмножества.

25. В множестве, состоящем из N элементов, выбрано 2^{N-1} подмножеств, каждые три из которых имеют общий элемент. Доказать, что все эти подмножества имеют общий элемент.

26. Часть подмножеств некоторого конечного множества выделена. Каждое выделенное подмножество состоит в точности из $2\cdot k$ элементов (k — фиксированное натуральное число). Известно, что в каждом подмножестве, состоящем не более чем из $(k+1)^2$ элементов, либо не содержится ни одного выделенного подмножества, либо все в нем содержащиеся выделенные подмножества имеют общий элемент. Доказать, что все выделенные подмножества имеют общий элемент.

27. Сколькими способами можно выбрать из полной колоды (***52*** карты) ***10*** карт так, чтобы а) среди них был ровно один туз? б) среди них был хотя бы один туз?

28. Задан фрагмент текста. Перечислить все его слова, которые состоят из тех же букв, что и первое слово.

29. Задан фрагмент текста. Найти:

а) все гласные буквы, которые входят в каждое слово;

б) все согласные буквы, которые не входят ни в одно слово;

в) все согласные буквы, которые входят только в одно слово;

г) все глухие согласные буквы, которые не входят ни в одно слово.

30. Задан фрагмент текста. Найти:

а) все звонкие согласные буквы, которые входят более чем в одно слово;

б) все гласные буквы, которые не входят более чем в одно слово;

в) все звонкие согласные буквы, которые входят в каждое нечетное слово и не входят ни в одно четное слово;

г) все глухие согласные буквы, которые входят в каждое нечетное слово и не входят хотя бы в одно четное слово.

ПРИЛОЖЕНИЕ Е

СПИСКИ

Варианты заданий для самостоятельного решения

1. ↓ Сформировать линейный список ***L***, хранящий данные числового типа. Разработать процедуры и функции для выполнения следующих действий:

а) подсчет числа элементов в списке;

б) определение наибольшего значения в списке;

в) определение ссылки на первое отрицательное число в списке (если таких чисел нет, возвращается пустая ссылка);

г) проверка на наличие в списке совпадающих значений;

д) удаление из списка совпадающих значений.

2. ↓ Сформировать линейный список ***L***, хранящий данные произвольного типа. Разработать процедуры и функции для выполнения следующих действий:

а) определение числа элементов в списке;

б) поиск в списке элемента с заданным значением (если таких элементов нет, возвращается пустая ссылка);

в) замена ***i***-го элемента копией ***j***-го элемента;

г) вставка после ***i***-го элемента копии ***j***-го элемента;

д) удаление из списка ***i***-го элемента.

3. ↓ Сформировать линейный список ***L***, хранящий данные числового типа. Разработать процедуры и функции для выполнения следующих действий:

а) определение наличия элементов в списке;

б) определение среднего арифметического значения элементов непустого списка;

в) замена всех вхождений ***i*** на ***j***;

г) перестановка первого и последнего элемента списка;

д) определение упорядоченности элементов списка по возрастанию.

4. ↓ Сформировать линейный список ***L***, хранящий данные строкового типа. Разработать процедуры и функции, определяющие число элементов списка, которые:

а) начинаются и оканчиваются одним и тем же символом;

б) начинаются с того же символа, что и следующие строки;

в) совпадают с содержимым последнего элемента.

5. ↓ Сформировать линейный список ***L***, хранящий данные числового типа. Построить список L_1 из положительных элементов ***L*** и L_2 — из остальных его элементов.

6. ↓ Сформировать односвязный линейный список ***L***, хранящий данные целого типа. Разработать процедуры и функции для выполнения следующих действий:

а) вставка нового элемента в конец списка;

б) вставка нового элемента после каждого вхождения ***i***;

в) вставка нового элемента перед каждым вхождением ***i***;

г) вставка значения ***i*** после каждого четного элемента;

д) удаление из списка всех отрицательных элементов.

7. ↑ Сформировать линейный список ***L***, хранящий данные числового типа. Разработать процедуры упорядочивания списка по возрастанию и по убыванию путем реорганизации связей.

8. Сформировать линейные списки L_1 и L_2, хранящие данные произвольного типа. Разработать процедуры и функции для выполнения следующих действий:

а) проверка списков на равенство;

б) проверка на вхождение второго списка в первый;

в) проверка списков на наличие хотя бы двух одинаковых элементов;

г) исключение из первого списка всех элементов, входящих во второй список;

д) слияние списков.

9. Сформировать односвязный линейный список ***L***, хранящий данные числового типа. Разработать процедуры и функции для выполнения следующих действий:

а) замена группы идущих подряд одинаковых элементов одним значением;

б) оставление в списке только первых элементов с одинаковыми значениями;

в) изменение знака всех элементов на противоположный;

г) обнуление элементов, значения которых лежат вне заданного диапазона;

д) удваивание каждого вхождения ***i*** в список.

10. ↑ Сформировать линейный список ***L***, хранящий данные произвольного типа. Разработать процедуру, изменяющую порядок следования элементов на противоположный путем реорганизации связей.

11. Сформировать линейные списки L_1 и L_2, хранящие данные произвольного типа. Разработать процедуры и функции, формирующие новый список ***L*** из элементов, которые:

а) входят хотя бы в один из списков L_1 и L_2;

б) входят одновременно в оба списка L_1 и L_2;

в) входят в один из списков L_1 или L_2, но не входят в другой.

12. ↑ Сформировать упорядоченные по возрастанию линейные списки L_1 и L_2, хранящие данные числового типа. Разработать процедуры их объединения (с сохранением упорядоченности):

а) путем построения нового списка ***L***;

б) путем реорганизации связей в L_1 и L_2 и их объединения.

13. Сформировать линейный односвязный список ***L***, хранящий данные числового типа. Разработать рекурсивные процедуры и функции для выполнения следующих действий:

а) определение вхождения элемента с заданным значением в список;

б) подсчет числа вхождений элемента с заданным значением в список;

в) определение максимального элемента списка;

г) замена вхождения ***i*** на ***j***;

д) вывод списка в обратном порядке.

Рекомендации: *для вывода списка в обратном порядке возможно использование рекурсивного прохода по списку.*

14. Сформировать линейный список ***L***, хранящий данные числового типа. Разработать рекурсивные процедуры и функции для выполнения следующих действий:

а) удаление из списка первого вхождения ***i***;

б) удаление из списка всех вхождений ***i***;

в) построение копии списка;

г) удваивание вхождения всех элементов в список;

д) подсчет среднего арифметического значения всех элементов списка.

15. Сформировать линейный список ***L***, хранящий данные строкового типа. Разработать процедуры и функции для выполнения следующих действий:

а) перестановка первого и последнего из непустых слов (если такие есть в списке);

б) вывод первых букв всех слов списка;

в) удаление из непустых слов списка первых букв;

г) вывод непустых слов списка;

д) определение количества слов, отличных от последнего.

16. Представить полином

$$P(x) = \sum_{i=0}^{n} a_i \cdot x^i$$

с числовыми коэффициентами в виде списка (если $a_i = 0$, соответствующий элемент в список не включается). Разработать процедуры и функции для выполнения следующих действий:

а) вычисление значения полинома для аргумента x;

б) сложение двух полиномов P_1 и P_2;

в) интегрирование полинома по аргументу x (строится новый полином со свободным нулевым коэффициентом);

Рекомендации: *информационная часть каждого элемента списка должна содержать поля* **Coef** *(значение коэффициента* a_i*) и* **Power** *(степень* **i***).*

17. Представить полином

$$P(x) = \sum_{i=0}^{n} a_i \cdot x^i$$

с числовыми коэффициентами в виде списка (см. задачу 16). Разработать процедуры и функции для выполнения следующих действий:

а) дифференцирование полинома по аргументу x (строится новый полином);

б) вывод полинома в общепринятом виде. Например, для полинома $10{,}7 \cdot x^{24} + 3{,}2 \cdot x^8 + x$ необходимо вывести

$$P(x) = 10{,}7 * x\text{^}24 + 3{,}2 * x\text{^}8 + x;$$

в) проверка на равенство двух полиномов P_1 и P_2.

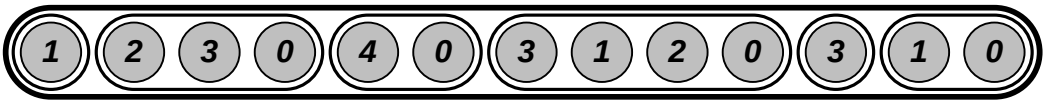
18. ↑ Сформировать иерархический линейный список L, хранящий данные произвольного типа. Разработать рекурсивные процедуры и функции для выполнения следующих действий:

а) формирование списка с заданным числом уровней иерархии;

б) проверка на вхождение элемента заданного уровня;

в) вывод всех элементов списка.

Рекомендации: *иерархическим считается список, содержимым которого могут быть как отдельные элементы, так и списки (см. рисунок).*



Число уровней вложенности элементов может быть произвольным.

19. Сформировать циклический двухсвязный список ***L*** с ограничителем, хранящий данные числового типа. Разработать процедуры и функции для выполнения следующих действий:

а) определение наличия элементов в списке;

б) подсчет числа элементов с одинаковыми «соседями»;

в) удаление из списка первого отрицательного элемента;

г) удваивание каждого вхождения элемента ***i***;

д) изменение порядка следования элементов между первым и последним вхождением ***i*** на противоположный.

20. Сформировать циклический двухсвязный список ***L*** с ограничителем, хранящий данные произвольного типа. Разработать процедуры и функции для выполнения следующих действий:

а) печать непустого списка в обратном порядке;

б) подсчет числа элементов, одинаковых с их правыми (по кругу) «соседями»;

в) удаление из списка всех элементов с одинаковыми «соседями»;

г) построение списка по однонаправленному списку;

д) добавление в конец списка всех его элементов в обратном порядке (например, по списку ***1,2,3*** строится список ***1,2,3,3,2,1***).

21. ↓ («Детская считалка».) Сформировать циклический список из ***n*** элементов. Начиная с первого, последовательно удалять каждый ***k***-й элемент, не теряя связности списка. Определить порядок удаления элементов.

22. («Игра в домино») Используя двухсвязный список, разработать процедуры и функции, имитирующие следующие действия игроков в домино:

а) определение возможности сделать ход;

б) выполнение хода;

в) определение, закончена ли игра.

23. ↑ Решить задачу 48 приложения В, заменив строку связным списком и статический стек — связным стеком.

24. ↑ Решить задачу 49 приложения В, заменив строку связным списком и статический стек — связным стеком.

25. ↑ Решить задачу 50 приложения В, заменив строку связным списком и статический стек — связным стеком.

ПРИЛОЖЕНИЕ Ж

ДЕРЕВЬЯ

Варианты заданий для самостоятельного решения

1. Сформировать бинарное дерево ***T***, хранящее данные числового типа. Разработать процедуры и функции для выполнения следующих действий:

а) определение числа вхождений заданного значения поля ***Info*** в дерево;

б) определение среднего арифметического значения содержимого всех полей ***Info***;

в) замена отрицательных значение полей ***Info*** их абсолютными значениями;

г) обмен максимального и минимального значений полей ***Info***;

д) вывод содержимого всех полей ***Info*** в процессе обхода дерева в прямом порядке.

2. Сформировать бинарное дерево ***T***, хранящее данные числового типа. Разработать рекурсивные процедуры и функции для выполнения следующих действий:

а) определение вхождения заданного значения поля ***Info*** в дерево;

б) определение числа вхождений заданного значения поля ***Info*** в дерево;

в) определение суммы всех полей ***Info***;

г) нахождение максимального и минимального значений полей ***Info***.

3. ↓ Сформировать бинарные деревья T_1 и T_2, хранящие данные произвольного типа. Разработать рекурсивные и нерекурсивные функции для проверки T_1 и T_2 на эквивалентность.

4. ↓ Сформировать бинарное дерево ***T***, хранящее данные произвольного типа. Разработать процедуру для построения копии этого дерева.

5. Задать изображение арифметического выражения. Разработать процедуры и функции для выполнения следующих действий:

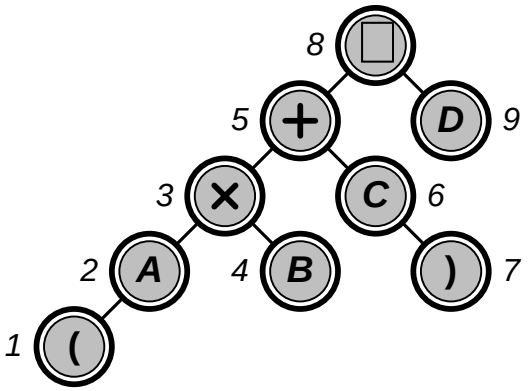
а) представление арифметического выражения в виде бинарного дерева поиска;

б) вывод изображения арифметического выражения, представленного бинарным деревом поиска, в общепринятом виде.

Рекомендации: *правила представления выражений в виде бинарного дерева приведены в п. 3.2. Необходимо сформировать массив вершин, которые будут включаться в дерево последовательно по уровням или по ветвям. У каждой вершины поле* ***Info*** *должно содержать изображение операнда, знака операции или скобки. Значение поля* ***Key*** *должно быть таким, чтобы после включения вершины в дерево упорядоченность последнего сохранялась. Так, выражение*

$$(A \times B + C)/D$$

с учетом приоритета операций и ключевых значений должно представляться следующим бинарным деревом поиска



6. Задать изображение арифметического выражения в префиксной форме без скобок (см. п. 3.2). Разработать процедуры для выполнения следующих действий:

а) представление арифметического выражения в виде бинарного дерева;

б) вывод изображения формулы, представленной бинарным деревом, в общепринятом виде;

в) проверка выражения на правильность.

Рекомендации: *правила проверки на правильность следующие: если элемент выражения — операнд, соответствующая ему вершина является концевой; если элемент выражения — оператор, соответствующая ему вершина является полной; если элемент выражения — скобка, соответствующая ему вершина является неполной.*

7. Задать изображение арифметического выражения в префиксной форме без скобок (см. п. 3.2). Разработать процедуры для выполнения следующих действий:

а) представление арифметического выражения в виде бинарного дерева;

б) вывод изображения формулы, представленной бинарным деревом, в общепринятом виде;

в) вычисление значения выражения.

Рекомендации: *каждая вершина дерева должна содержать символьное поле* **Info** *для хранения изображения операнда или знака операции и числовое поле* **Value** *для хранения соответствующего значения (только для операндов). Бинарное дерево обходится в обратном порядке. Если встречается вершина-операнд, то значение ее поля* **Value** *запоминается в стеке. Если встречается вершина-знак операции, то из стека извлекаются два последних элемента, над ними выполняется операция, ее результат записывается в стек. После просмотра строки в стеке будет находиться значение выражения.*

8↑. Задать изображение арифметического выражения в префиксной форме без скобок (см. п. 3.2), операндами которого могут

быть переменные и числовые константы. Разработать процедуры для выполнения следующих действий:

а) представление арифметического выражения в виде бинарного дерева;

б) упрощение полученного бинарного дерева путем замены поддеревьев, соответствующих выражениям $x+0$, $0+x$, $x\times 1$ и $1\times x$, на x.

9↑. Задать изображение арифметического выражения в префиксной форме без скобок (см. п. 3.2), операндами которого могут быть переменные и числовые константы. Разработать процедуры для выполнения следующих действий:

а) представление арифметического выражения в виде бинарного дерева;

б) упрощение полученного бинарного дерева путем замены поддеревьев, соответствующих выражениям $(x_1 \pm x_2)\times x_3$ и $x_1 \times (x_2 \pm x_3)$, на поддеревья, соответствующие выражениям $x_1 \times x_3 \pm x_2 \times x_3$ и $x_1 \times x_2 \pm x_1 \times x_3$.

10↑. Задать изображение арифметического выражения в префиксной форме без скобок (см. п. 3.2), операндами которого являются переменные с длиной имени 1 символ. Разработать процедуру для построения дерева — частной производной по данному выражению по выбранной переменной.

11. Разработать процедуру «горизонтального вывода» бинарного дерева поиска.

Рекомендации: *под «горизонтальным выводом» понимается печать вершин дерева по уровням без изображения ветвей с поворотом на* $90°$ *против часовой стрелки. Например, дерево из задачи 5 будет представлено в виде*

Ключ	Уровни				
	0	1	2	3	4
9		D			
8	□				
7				)	
6			C		
5		+			
4				B	
3			×		
2				A	
1					(

Необходимо проанализировать порядок вывода изображения вершин в зависимости от их ключевых значений и разработать технологию формирования соответствующего отступа для каждой строки.

12. ↓ Разработать процедуру рекурсивного построения и нерекурсивного обхода бинарного дерева поиска в прямом порядке.

13. ↑ Разработать процедуры, выполняющие основные операции (создания, уничтожения, вставки и удаления элементов) над деревом с произвольным ветвлением.

14. ↑ Разработать процедуры, выполняющие основные операции (создания, уничтожения, вставки и удаления элементов) над «прошитым» деревом поиска.

15. ↑ Разработать процедуры, выполняющие основные операции (создания, уничтожения, вставки и удаления элементов) над деревом, сбалансированным по ***AVL***.

ПРИЛОЖЕНИЕ И

ГРАФЫ

Варианты заданий для самостоятельного решения

1. ↓ Задан ориентированный граф. Проверить правильность утверждения, что для любой пары его вершин одна из этих вершин достижима от другой.

2. ↓ Задан ориентированный граф. Найти все его вершины, недостижимые от заданной вершины.

3. ↓ Задан неориентированный граф. Для двух его выделенных вершин построить соединяющий их простой путь.

4. ↓ Задан ориентированный граф. Найти такую нумерацию его вершин, при которой любая дуга ведет от вершины с меньшим номером к вершине с большим номером.

5. Задан ориентированный граф. Найти все его истоки и стоки.
Рекомендации: *истоком ориентированного графа называют вершину, от которой достижимы все другие вершины, стоком — вершину, достижимую от всех других вершин.*

6. Задан неориентированный граф. Найти такую вершину заданного графа, которая принадлежит каждому пути между двумя различными вершинами и отлична от каждой из них.

7. Задан ориентированный граф. Удалить из него все вершины вместе с инцидентными им ребрами, от которых недостижима заданная вершина.

8. ↓ Задан ориентированный граф. Найти минимальное подмножество его вершин, от которых достижимы все остальные вершины.

9. Задан ориентированный граф. Найти такое множество $\boldsymbol{P}$ путей между его выделенными вершинами $\boldsymbol{u}$ и $\boldsymbol{v}$, что ни одна другая вершина, кроме $\boldsymbol{u}$ и $\boldsymbol{v}$, не лежит на двух путях из $\boldsymbol{P}$ и любое расширение $\boldsymbol{P}$ приводит к нарушению этого правила.

10. ↓ Задан ориентированный граф. Найти максимальное по мощности подмножество попарно несмежных его вершин.

11. ↓ Задан неориентированный граф. Найти максимальное по мощности подмножество попарно несмежных его ребер.

12. Задан неориентированный граф. Определить, является ли он гамильтоновым, и построить все гамильтоновы циклы.

13. Задан неориентированный граф. Найти максимальный по количеству вершин его полный подграф.

14. Задан неориентированный граф. Определить, совпадают ли степени всех его ребер и, если нет, можно ли удалить из него одну вершину вместе с инцидентными ребрами так, чтобы полученный граф обладал этим свойством.

Рекомендации: *степенью ребра графа* $\{\boldsymbol{u},\boldsymbol{v}\}$ *будет считаться неупорядоченная пара* $\{\boldsymbol{d}(\boldsymbol{u}),\boldsymbol{d}(\boldsymbol{v})\}$*, где* $\boldsymbol{d}(\boldsymbol{u})$ *и* $\boldsymbol{d}(\boldsymbol{v})$ *— степени вершин* $\boldsymbol{u}$ *и* $\boldsymbol{v}$*.*

15. Задан ориентированный граф. Проверить, является ли он транзитивным.

Рекомендации: *граф будет считаться транзитивным, если для любых трех вершин* $\boldsymbol{u}$*,* $\boldsymbol{v}$*, и* $\boldsymbol{w}$ *выполняется условие: если смежны как вершины* $\boldsymbol{u}$ *и* $\boldsymbol{w}$*, так и* $\boldsymbol{v}$ *и* $\boldsymbol{w}$*, то также смежны вершины* $\boldsymbol{u}$ *и* $\boldsymbol{v}$*.*

16. Задан неориентированный граф. Указать все его полные подграфы, состоящие из $\boldsymbol{k}$ вершин.

17. Задан неориентированный граф. Определить, является ли он каркасом.

18. Задан неориентированный граф. Найти все его подграфы, которые являются каркасами.

19. Задан неориентированный граф с циклами. Проверить, можно ли удалить одну вершину вместе с инцидентными ей ребрами так, чтобы новый граф становился ациклическим.

20. Задан ориентированный граф. Подсчитать количество компонент связности в его дополнении.

21. Задан неориентированный граф. Найти все его точки сочленения.

22. Задан неориентированный граф. Найти все его мосты.

23. Задан ориентированный граф. Определить его компоненты сильной связности.

24. Задан неориентированный граф. Найти минимальное по мощности подмножество его ребер, удаление которых превращает связный граф в несвязный.

25. Задан неориентированный граф. Найти длину кратчайшего его цикла.

26. Задан ориентированный граф. Найти в нем самый длинный простой путь.

27. Задан ориентированный граф. Найти все его вершины, к которым существует путь заданной длины от выделенной вершины.

28. Задан ориентированный граф. Найти все его вершины, от которых существует путь заданной длины к выделенной вершине.

29. ↓ Задан ориентированный граф. Найти его диаметр и медиану.

Рекомендации: *медианой графа будет считаться вершина, сумма расстояний от которой до остальных вершин минимальна.*

30. Задан ориентированный граф. Пронумеровать его вершины так, чтобы число обратных дуг было минимальным.

Рекомендации: *дуга будет считаться обратной, если она ведет от вершины с большим номером к вершине с меньшим номером.*

31. Задан неориентированный граф ***G*** с ***N*** вершинами. Раскрасить его вершины цветами в диапазоне от ***1*** до $k < N$ так, чтобы любые две смежные вершины были разного цвета.

32. Задан ориентированный граф. Найти минимальное доминирующее подмножество его вершин.

Рекомендации: *подмножество вершин будет считаться доминирующим, если с ним смежна хотя бы одна вершина не из этого подмножества.*

33. Задан неориентированный граф. Построить минимальное по числу ребер покрытие заданного графа.

Рекомендации: *подмножество ребер, покрывающее вершины графа, образуют ребра, инцидентные его вершинам.*

34. ↓ Заданы два неориентированных графа. Определить, изоморфны ли они.

35. Задан неориентированный граф. Определить, изоморфен ли он своему дополнению.

36. Подсчитать количество попарно не изоморфных графов с ***N*** вершинами и четырьмя ребрами.

37. Подсчитать количество попарно не изоморфных графов, содержащих не более четырех вершин.

38. Задан неориентированный граф. Определить, является ли он двудольным.

39. Для заданного целого ***N*** построить неориентированный граф, в котором степень каждой вершины равна ***4***.

40. Для заданного целого N построить неориентированный граф, не содержащий циклов длиной 3, в котором степени всех вершин равны 3.

41. Задан неориентированный граф G. Построить граф G' с тем же множеством вершин, которые смежны тогда, когда расстояние между ними в G не превышает 2. Проверить, совпадают ли степени всех вершин в G', и если нет, то нельзя ли удалить из него одну вершину так, чтобы полученный граф удовлетворял этому требованию.

42. Задан неориентированный граф G. Построить граф G' по правилам: число вершин G' равно числу ребер G; две вершины в G' смежны тогда, когда смежны соответствующие ребра в G. В G' найти все вершины, расстояние от которых до некоторой выделенной вершины превышает 2.

43. Задан неориентированный граф G. Построить последовательность графов $G_1,...,G_k$ с тем же множеством вершин, что и у G. Вершины в G_i смежны, если расстояние между соответствующими вершинами в G не превышает i.

44. ↑ Задан ориентированный граф G. Построить граф G' последовательным применением (пока это возможно) следующей операции: если v — вершина G с единственным предшественником u ($u \neq v$) и единственным преемником w ($w \neq v$), то она удаляется из G вместе с дугами (u,v) и (v,w), и в G добавляется новая дуга (u, w); если v — вершина без предшественников или без преемников, то она просто удаляется из G.

45. ↑ Задан неориентированный граф. Последовательным применением операции склеивания исключить из него все треугольники вершин.

Рекомендации: *треугольником будет считаться любая тройка различных и попарно смежных вершин. Склеиванием называется операция удаления из графа вершин треугольника вме-*

сте с инцидентными им ребрами. В граф добавляется новая вершина **v**. *Новое ребро* $\{w,v\}$ *добавляется только тогда, когда вершина* **w** *была смежна хотя бы с одной вершиной удаленного треугольника.*

46. Задана система односторонних дорог. Найти путь, соединяющей города ***A*** и ***B***, который не проходит через заданное множество городов.

47. Задана система двусторонних дорог. Определить, является ли она трехсвязной.

Рекомендации: *система двусторонних дорог будет считаться трехсвязной, если для любой четверки разных городов* ***A***, ***B***, ***C*** *и* ***D*** *существует два различных пути из* ***A*** *в* ***D***, *причем один из них проходит через* ***B***, *а другой — через* ***C***.

48. ↓ Задана система двусторонних дорог. Для каждой пары городов определить длину кратчайшего пути между ними.

49. ↓ Задана система двусторонних дорог. Найти два города и соединяющий их путь, который проходит через каждую из дорог ровно один раз.

50. ↓ Задана система двусторонних дорог. Найти замкнутый путь длиной не более ***100*** км, проходящий через каждую дорогу ровно один раз.

51. Задана система двусторонних дорог, причем для любой пары городов существует соединяющий их путь. Найти город, для которого сумма расстояний до остальных городов минимальна.

52. ↓ Задана система односторонних дорог. Определить, есть ли в ней город, из которого можно добраться до каждого из остальных городов, проезжая не более ***100*** км.

53. ↓ Задана система односторонних дорог. Определить, есть ли в ней город, куда можно попасть из любого другого города, проезжая не более ***100*** км.

54. Задана система двусторонних дорог. Определить, можно ли, построив какие-нибудь новые три дороги заданной длины, из заданного города добраться до каждого из остальных городов, проезжая не более ***100*** км.

55. Задана система двусторонних дорог. Определить, можно ли, закрыв какие-нибудь три дороги, добиться того, чтобы из города ***A*** нельзя было попасть в город ***B***.

56. Задана система двусторонних дорог, связывающих фиксированное число городов. Для заданного расстояния ***L*** определить периферию столицы.

Рекомендации: *периферией считается множество городов, расстояние от которых до выделенного города больше* ***L.***

57. Задана система односторонних дорог. Определить, можно ли проехать из города ***A*** в город ***B*** таким образом, чтобы посетить город ***C*** и не проезжать никакой дороги более одного раза.

58. Задана система двусторонних дорог. За проезд по каждой из них взимается пошлина. Найти путь из города ***A*** в город ***B*** с минимальной суммой длин дорог ***S*** и взимаемых пошлин ***P***.

59. Заданы две системы двусторонних железных и шоссейных дорог с одним и тем же множеством городов. Найти минимальный по длине путь из города ***A*** в город ***B***, который может проходить как по железным, так и по шоссейным дорогам, и места пересадок с одного вида транспорта на другой на этом пути.

60. Задана система односторонних дорог. Найти длину самого длинного простого пути от города ***A*** до города ***B***.