БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Г. А. Расолько Е. В. Кремень Ю. А. Кремень

ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ

В трех частях

Часть 3
СРЕДСТВА ОБРАБОТКИ ДАННЫХ

Рекомендовано
Учебно-методическим объединением
по естественно-научному образованию
в качестве учебно-методического пособия
для студентов учреждений высшего образования,
обучающихся по специальности «математика»

Учебное электронное издание

Минск, БГУ, 2025

ISBN 978-985-881-748-0 (ч. 3)
ISBN 978-985-881-745-9

УДК 004.42(075.8)(076.5)
ББК 32.973-018я73-1

Р е ц е н з е н т ы:
кафедра инженерной психологии и эргономики
Белорусского государственного университета
информатики и радиоэлектроники
(заведующий кафедрой доктор психологических наук,
член-корреспондент Международной академии психологических наук,
профессор *Т. В. Казак*);
доктор физико-математических наук, доцент *А. С. Кравчук*

Расолько, Г. А. Практикум по программированию [Электронный ресурс] : учеб.-метод. пособие. В 3 ч. Ч. 3. Средства обработки данных / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2025. – 1 электрон. опт. диск (CD-ROM). – ISBN 978-985-881-748-0.

Представлен учебный материал для проведения занятий по дисциплине «Практикум по программированию». Рассмотрены задачи, не ориентированные на конкретный алгоритмический язык, по темам: стиль программирования, динамические структуры данных, методы разработки алгоритмов, современные языки программирования.

Предназначено для студентов учреждений высшего образования, обучающихся по специальности «математика».

Минимальные системные требования:
PC, Pentium 4 или выше;
RAM 1 Гб; Windows XP/7/10; Adobe Acrobat.

Оригинал-макет подготовлен в программе MicroSoft Word.

Ответственный за выпуск *Т. М. Турчиняк*.
Дизайн обложки *В. П. Явуз.* Технический редактор *В. П. Явуз*.
Компьютерная верстка *В. П. Явуз*.
Корректор *Е. О. Алёшина.*

Подписано к использованию 30.04.2025. Объем 2,60 МБ.

Белорусский государственный университет.
Управление редакционно-издательской работы.
Пр. Независимости, 4, 220030, Минск.
e-mail: urir@bsu.by
http://elib.bsu.by

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Настоящий практикум направлен на формирование у студентов навыков решения различных типов задач с использованием современных информационных технологий. Основная цель издания – развитие алгоритмического мышления, изучение современных методов программирования и приобретение навыков работы с современными вычислительными средствами.

В процессе занятий студенты должны:

- развивать математическое, логико-алгоритмическое и программистское мышление;
- применять практические знания и умения, современные методы и системы программирования;
- осваивать приемы и основы методологии структурного и модульного программирования;
- формировать творческий подход к конструированию алгоритмов, что способствует развитию аналитических способностей.

Основное внимание уделяется не только кодированию программ, но и их проектированию, где особый акцент делается на современных технологиях: проектирование сверху вниз, модульное программирование с использованием подпрограмм и модулей, анализ эффективности и оптимизация программ, широкое использование рекурсии. Все это направлено на привитие определенного стиля программирования и дальнейшее развитие навыков посредством обучения объектно ориентированному программированию.

Третья часть учебно-методического пособия посвящена более сложным темам, включая стиль программирования, динамические структуры данных и методы разработки алгоритмов, что необходимо для создания сложных и оптимизированных программных решений.

В результате изучения третьей части практикума студенты должны знать:

- жизненный цикл программного продукта, документирование и стандартизацию;
- работу с динамическими структурами данных, включая списки, стеки, очереди и деревья;
- методы разработки алгоритмов, такие как «разделяй и властвуй», «жадные» алгоритмы, полный перебор, поиск с возвратом и фракталы;
- современные языки и стили программирования.

Перечисленные знания помогут студентам не только разрабатывать эффективные алгоритмы и программы, но и использовать современные подходы и технологии программирования, что является важным шагом на пути к профессиональному росту в области информационных технологий.

Тема 11
СТИЛЬ ПРОГРАММИРОВАНИЯ
(4 часа)

Под технологией программирования будем понимать совокупность производственных процессов, приводящих к созданию требуемой программной системы, а также описание данной совокупности, начиная с момента зарождения идеи этого средства до создания необходимой программной документации.

Каждый процесс данной совокупности базируется на использовании каких-либо *методов* и *средств*, например компьютера. Тогда говорится о компьютерной технологии программирования.

В историческом аспекте в развитии технологии программирования выделяют следующие этапы.

1. *Стихийное программирование* – отсутствие сформулированной технологии, когда программирование было, по сути, искусством. Стихийно использовалась разработка сверху вниз – подход, при котором вначале проектировали и реализовывали сравнительно простые подпрограммы, из которых потом пытались построить сложную программу. Развитие программирования шло по пути замены машинных языков ассемблерами, а затем алгоритмическими языками (Fortran, Algol).

2. *Структурный подход к программированию* – в основе его лежит декомпозиция сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. Программирование осуществлялось сверху вниз и подразумевало представление задачи в виде иерархии подзадач простейшей структуры со своим тщательно проработанным интерфейсом (заголовками подпрограмм). Поддержка принципов структурного программирования была заложена в основу процедурных языков программирования (PL/1, Algol-68, Pascal, C). Появилась и начала развиваться технология модульного программирования. Модули раздельно разрабатывались и компилировались, могли возникать неточности на этапе эксплуатации таких программных систем по причине ошибок в состыковке модулей. Модули взаимодействуют между собой, обращаясь к ресурсам друг друга.

3. *Объектный подход к программированию* (с середины 80-х до конца 90-х годов XX века) – использование способов организации программ, основанных на механизмах инкапсуляции, наследования, полиморфизма. В дальнейшем развитие объектного подхода в технологии программирования привело к созданию сред визуального программирования. Появились языки визуального объектно ориентированного программирования (Delphi, C++ Builder, Visual C++, C#).

4. *Компонентный подход и Case-технологии* (с середины 90-х годов XX века и до наших дней) – построение программного обеспечения из отдельных компонентов – физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы.

Важнейшая особенность современного этапа технологии программирования – широкое использование компьютерных технологий создания и сопровождения программных систем на всех этапах их жизненного цикла.

Технологии программирования развивались вместе с ЭВМ и языками программирования.

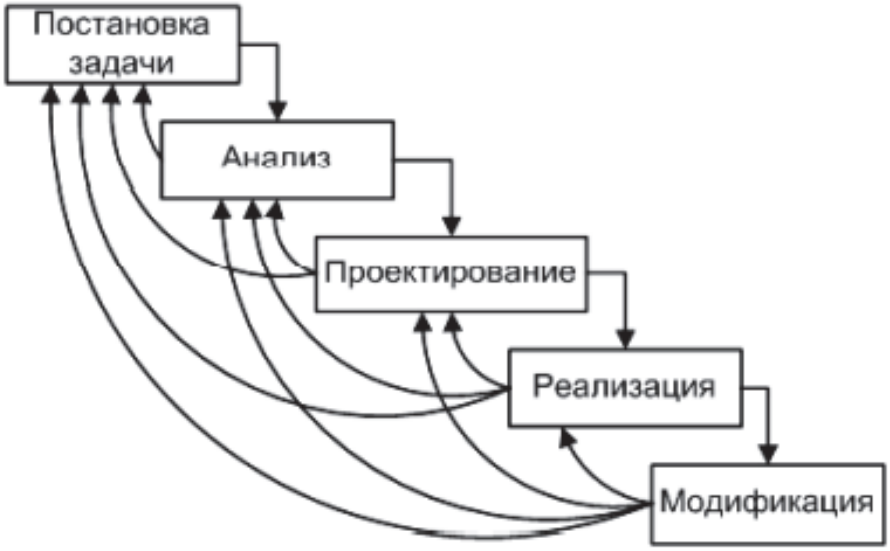
11.1. Жизненный цикл программного продукта

Под жизненным циклом программного продукта (*software life cycle*) понимают весь период его разработки и эксплуатации, начиная от момента возникновения замысла программного продукта, определения его целевого назначения и заканчивая выводом его из эксплуатации. Понятие «жизненный цикл» охватывает весь процесс создания и использования программного продукта. Этот процесс может быть организован по-разному в зависимости от особенностей коллектива разработчиков и разрабатываемого программного продукта.

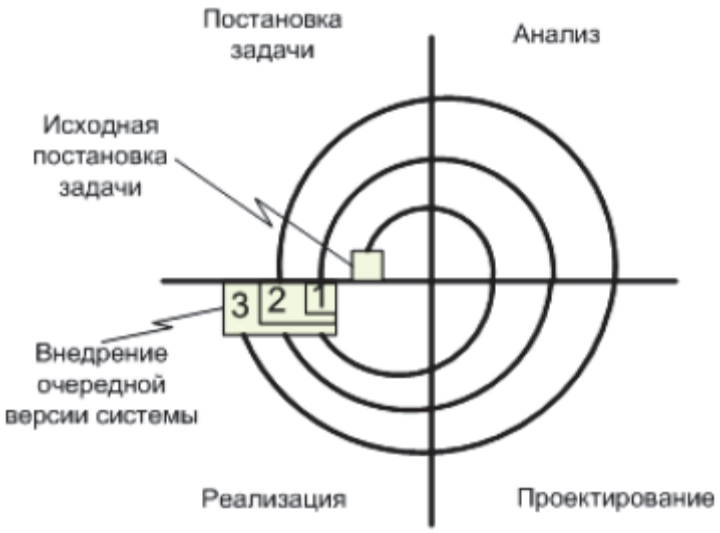
В настоящее время наиболее широко используют три базовые стратегии разработки программного продукта: каскадную, инкрементную и эволюционную.

Каскадная стратегия представляет собой однократный проход этапов разработки. Каскадная (или водопадная) модель реализует каскадную стратегию. Процесс разработки состоит из цепочки этапов, выполняемых последовательно друг за другом. На каждом этапе создается документация, используемая на последующем этапе. Это классический подход к созданию программного продукта.

Итеративная (инкрементная) модель является классическим примером *инкрементной стратегии* конструирования. Она объединяет элементы последовательной водопадной модели с итерационной философией макетирования. Процесс разработки заключается в разбиении создаваемой программной системы на набор частей. Каждая часть реализуется путем нескольких логических проходов последовательности всех работ. На первой итерации разрабатывается независимая часть системы, при этом чаще всего проводится полный цикл работ. Затем оцениваются полученные результаты. Следующая итерация заключается либо в исправлении недочетов первой части, либо в проектировании следующего фрагмента, зависящего от первого, либо добавлением новых функций. В результате на каждой итерации можно анализировать промежуточные результаты работ, учитывать мнение конечных пользователей и вносить изменения на следующих итерациях. Каждая итерация в процессе выполнения может содержать полный цикл видов деятельности начиная с анализа требований.



Продолжением идеи итераций является спиральная модель жизненного цикла программного обеспечения – классический пример применения *эволюционной стратегии* конструирования. В данной модели разработки каждая итерация начинается с анализа целей, разработки основных альтернатив, определения необходимых ограничений, оценки рисков и т. д. Результатом каждой итерации будет оценка проведенных в ее рамках работ. Следует отметить общую структуру действий на каждой итерации – планирование, определение задач, ограничений и вариантов решений, оценка предложенных решений и рисков, выполнение основных работ итерации и оценка их результатов. Название «спиральная» эта модель получила из-за изображения хода работ в полярных координатах, где угол соответствует выполняемому этапу в рамках общей структуры итераций, а удаление от начала координат – затраченным ресурсам.



Исследовательское программирование предполагает быструю (насколько это возможно) реализацию рабочих версий программ, выполняющих основные функции. После опытного применения созданных программ переходят к их модификации с целью устранения недочетов и приближения их к реальным требованиям пользователей. Такой подход соответствовал ранним этапам развития программирования. В настоящее время этот подход применяется для разработки систем искусственного интеллекта.

Прототипирование моделирует начальную фазу работы вплоть до создания рабочих версий программ с целью установить (уточнить) требования заказчика к программному продукту.

Основная цель создания прототипа – снять неопределенности в требованиях заказчика. В дальнейшем обычно следует разработка программы по установленным требованиям в рамках какого-либо другого подхода (например, классического).

Достоинством данного подхода является определение полных требований к программному продукту.

Недостатки: когда заказчик видит уже работающую версию программного продукта, возникает желание принять ее за готовую версию и оставить нерешенными вопросы качества и удобства сопровождения. При этом заказчик не может адекватно оценить объем будущих работ и требует получить рабочий продукт в нереально короткие сроки.

Следует также отметить, что для быстрого получения работающего макета программист может идти на определенные компромиссы, например использовать не самые подходящие языки программирования и операционные системы или для демонстрации функциональных возможностей применить неэффективный алгоритм.

Сборочное программирование заключается в конструировании программ из стандартных компонент, которые уже существуют в разработанных библиотеках.

Эти библиотеки компонент достаточно хорошо разработаны и многократно используются при создании сложных программных систем, содержащих типовые фрагменты обработки данных.

Такой процесс разработки состоит скорее из сборки программ из компонент, чем из их программирования.

Весь жизненный цикл программного продукта можно разбить на три крупные фазы:

1) разработка;

2) эксплуатация;

3) сопровождение и разработка новых версий.

В фазе разработки программный продукт разрабатывается, документируется, тестируется и выпускается.

В фазе эксплуатации разработанный программный продукт используется на практике конкретными потребителями, при этом могут выявляться ошибки, недоработки, а также изменяться требования к пользовательскому интерфейсу.

В фазе сопровождения программный продукт изменяется в соответствии с поступившими требованиями, и появляются его принципиально новые версии.

Фазу разработки обычно разделяют на следующие логические этапы:

1) анализ требований;

2) проектирование;

3) программирование (кодирование);

4) отладка и тестирование;

5) документирование;

6) выпуск.

На *этапе анализа требований* для заказчика обосновывается необходимость нового программного продукта, выявляются наиболее общие требования к нему. Результатом системного анализа является выработка спецификации требований на программный продукт, содержащая указанные требования в формальном виде.

Этот документ является описанием поведения программы с точки зрения ее будущего пользователя и с фиксацией требований относительно его качества. По единой системе программной документации такая спецификация называется техническим заданием.

На *этапе проектирования* сформулированные общие требования к будущему программному продукту последовательно реализуются в рабочий проект. Он в деталях описывает структуру программной системы, применяемые форматы данных и алгоритмы, внешний вид интерфейса пользователя. Результатом проектирования является технический проект.

Этап кодирования при наличии детального технического проекта является достаточно рутинным. По сути, кодирование представляет собой процесс автоматической реализации предложенных алгоритмов обработки на конкретном языке программирования с использованием определенной методологии и инструментария.

Результатом данного этапа являются программы в исходном тексте и выполняемые файлы.

Этап кодирования сопровождается процессом документирования.

Этапы проектирования и кодирования часто перекрываются, иногда довольно сильно.

Этап отладки и тестирования предназначен для выявления и устранения ошибок и недочетов в программах. Результатом данного этапа являются отлаженные программы, для которых тестированием установлено (насколько это возможно) соответствие запрашиваемым требованиям.

На *этапе документирования* к созданной программной системе подготавливается пакет документации, описывающей создаваемый программный продукт. Каждый документ ориентирован на конкретный тип пользователей: конечных пользователей, системных администраторов, прикладных программистов и т. д.

На *этапе выпуска* программный продукт подвергается итоговым испытаниям по утвержденной методике. После этого программный комплекс продается или внедряется в работу фирмы, заказавшей его.

Задачи

Репдиджит – это число, которое состоит из повторяющихся цифр (например, 5555).

В разных системах счисления репдиджитами оказываются разные десятичные числа. Например, число $(13)_{10}$ будет репдиджитом в троичной системе счисления, потому что там оно выглядит как $(111)_3$.

1. Для некоторого случайного целого числа из промежутка [*A*, *B*] необходимо определить, существуют ли системы счисления, в которых данное число является репдиджитом.

2. Сформируйте все репдиджиты в заданной *p*-ричной системе счисления из промежутка [*A*, *B*]. Разработайте алгоритм решения задачи, по которому формируются только репдиджиты. Оформите отчет по данной задаче, отражая жизненный цикл разработки данного программного продукта в соответствии с требованиями, изложенными выше.

3. Для всех систем счисления от двоичной системы счисления до $p_к < 17$ сформируйте все репдиджиты, которые в десятичной системе счисления представляют собой четырехзначные коды. Разработайте алгоритм решения задачи, по которому формируются только репдиджиты. Оформите отчет по данной задаче, отражая жизненный цикл разработки данного программного продукта в соответствии с требованиями, изложенными выше.

4. В типизированном файле хранятся целые числа, в количестве, кратном трем. Пока не закончится информация из файла нужно инициализировать коэффициенты полного квадратного уравнения и решать его на множестве целых чисел. Если существуют целые корни текущего уравнения, тогда результат, содержащий три коэффициента и два корня, вывести в результирующий типизированный файл.

11.2. Документирование и стандартизация

При разработке программного продукта создается большой объем документации. Она используется как средство передачи информации между разработчиками и как средство описания информации, необходимой пользователям для применения программы. Также программная документация может быть использована для тестирования программы.

Документирование должно начинаться одновременно с разработкой продукта.

Существуют следующие основные программные документы:

• *текст программы* – запись программы с необходимыми комментариями;

• *описание программы* – сведения о логической структуре и функционировании программы;

• *программа* и *методика испытаний* – требования, подлежащие проверке при испытании программы, а также порядок и методы их контроля;

• *техническое задание* – описание поведения программы с точки зрения ее будущего пользователя и фиксация требований относительно ее качества;

• *пояснительная записка* – содержит общее описание алгоритма, часто в виде граф-схем, схему функционирования и взаимосвязи программных модулей, а также обоснование принятых технических и экономических решений.

Раздел эксплуатационных документов включает в себя:

• *руководство пользователя* – сведения об области назначения программы, области ее применения, используемых при реализации методах, ограничениях алгоритма, конфигурации требуемых технических средств; описание интерфейса пользователя и допустимых к использованию функций;

• *руководство системного администратора* – сведения для обеспечения установки, функционирования и настройки программ в условиях конкретного применения.

Интерфейсом пользователя называют совокупность способов и правил взаимодействия программы с пользователем.

Задачи

1. Для некоторого случайного целого числа из промежутка [A, B] необходимо определить, существуют ли системы счисления, в которых данное число является палиндромом.

2. Сформируйте все палиндромы в заданной p-ричной системе счисления из промежутка [A, B]. Разработайте алгоритм решения задачи, по которому формируются только палиндромы. Оформите отчет по данной задаче, отражая жизненный цикл разработки программного продукта в соответствии с требованиями, изложенными выше.

3. Для всех систем счисления от двоичной системы счисления до $p_к < 17$ сформируйте все репдиджиты, которые в десятичной системе счисления представляют собой четырехзначные коды. Разработайте алгоритм решения задачи, по которому формируются только репдиджиты. Оформите отчет по данной задаче, отражая жизненный цикл разработки программного продукта в соответствии с требованиями, изложенными выше.

4. В типизированном файле хранятся целые числа в количестве, кратном трем. Пока не закончится информация, из файла нужно инициализировать коэффициенты полного квадратного уравнения и решать его на множестве целых чисел. Если существуют целые корни текущего уравнения, то результат, содержащий три коэффициента и два корня, выведите в результирующий типизированный файл.

Тема 12
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ
(14 часов)

Абстрактный тип данных (АТД) – это математическая модель с совокупностью данных и операторов, определенных в рамках этой модели.

Между объектами материального мира, поведение которых моделируют программы, существуют разнообразные, постоянно изменяющиеся связи. Одни объекты могут исчезать, другие – появляться. Если в программе требуется моделировать группы с переменным числом объектов, между которыми связи могут быть изменчивыми, то нужны языковые средства для установления, изменения и расторжения связей между объектами, которые моделируются, а также для *рождения* и *уничтожения* объектов. Этой цели в языке Pascal служат динамические переменные и указатели.

Самый простой способ связать множество элементов – соединить их линейно в список, очередь или стек. Если элементы связаны рекурсивно, то их представляют деревом (примером является генеалогическое древо человека). Более сложные связи между элементами дают графы.

Динамические структуры данных появились, когда использование регулярных структур (массивов, строк) начало замедлять написание программ несовершенным аппаратом.

12.1. Списки и их классификация

Основные положения

Список – это структура данных, каждый элемент которой содержит ссылку, связывающую его с последующим элементом – *узлом* (*звеном*) списка.

Различают списки:

- линейные:
  - с одной связью;
  - несколькими связями;
- кольцевые (циклические):
  - с одной связью;
  - несколькими связями;
- ассоциативные;
- иерархические.

Существуют два метода хранения списков – последовательный и связный.

При *последовательном* хранении элементы списка (или узлы списка) располагаются в массиве зафиксированного размера. Этот прием используется, когда нельзя работать с динамической памятью.

При *связном* хранении узлы списка располагаются динамически в `Heap` при помощи указателей. Для организации связного списка, как упоминалось выше при решении задачи на считалочку, используются узлы списка – записи, состоящие из двух частей. Одна часть – тело узла списка – имеет информацию, подлежащую обработке, вторая часть – ссылочная – содержит указатель на следующий узел списка.

Связные списки

Связный список, как и массив, является универсальной структурой данных, которая широко используется программистами. Однако, в отличие от массива, связный список не входит в состав стандартного языка Pascal.

Тем не менее в Pascal создать связный список довольно просто. Все, что для этого нужно, – иметь в составе языка указатель.

По своей сути связный список – это цепочка узлов или звеньев, содержащих описания, образующие информационную часть узла списка. При этом каждый узел имеет указатель, который указывает на следующий узел в списке или равен `nil`, если он последний. В случае кольцевого списка последний узел должен ссылаться на первый.

Список начинается с первого узла, от которого, путем последовательных переходов по ссылке, можно обойти все остальные узлы. В связном списке узлы могут быть разбросаны по разным местам памяти, а их чередование определяется ссылками. Память под каждый узел в `Heap` выделяется отдельно.

Тип узла – это запись, в котором хранятся данные и указатель на следующий узел списка.

Действия со списками

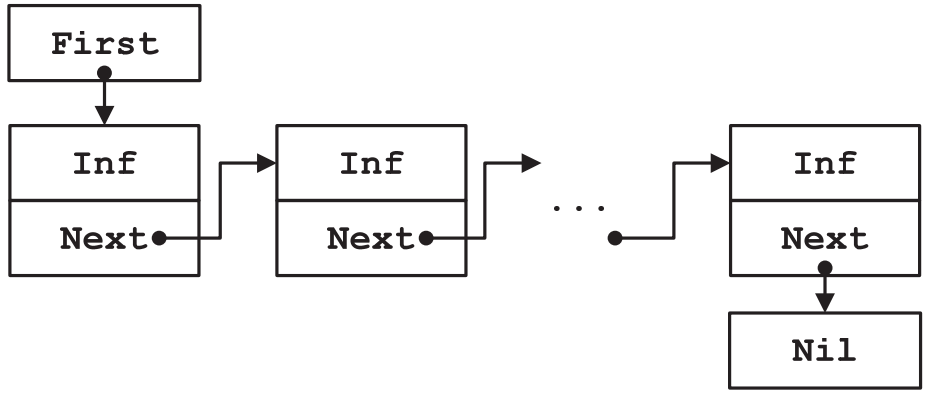
При работе со списками часто выполняются следующие действия:

1) найти узел с заданным свойством;
2) удалить узел;
3) определить по номеру нужный узел;
4) добавить узел;
5) распечатать узлы списка;
6) упорядочить узлы списка по некоторому ключу и прочее.

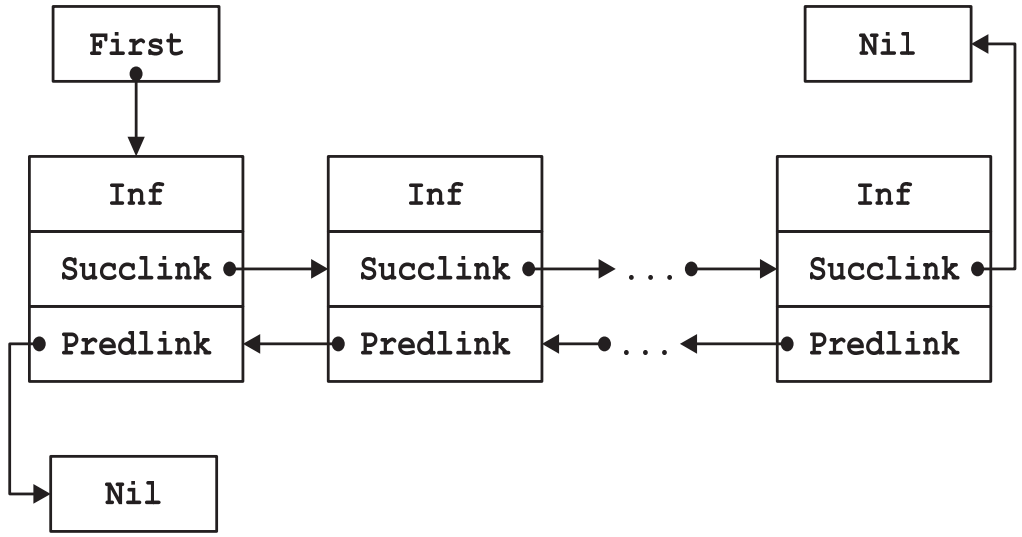
Типы списков по методам доступа к узлам

Приведем графическую интерпретацию методов связного хранения некоторых простых типов списков.

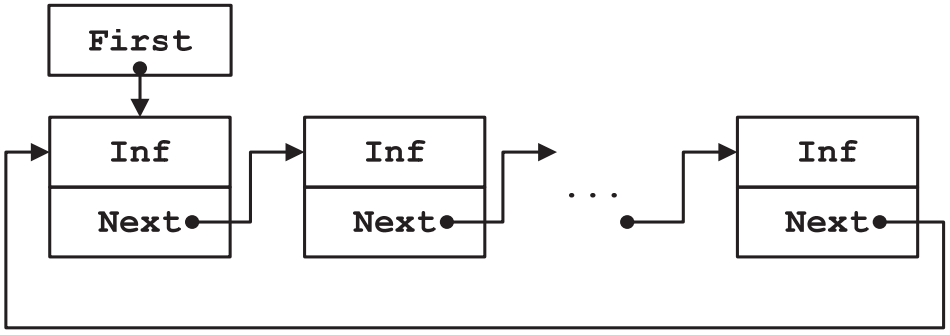
1. Обычный линейный список с одной связью.



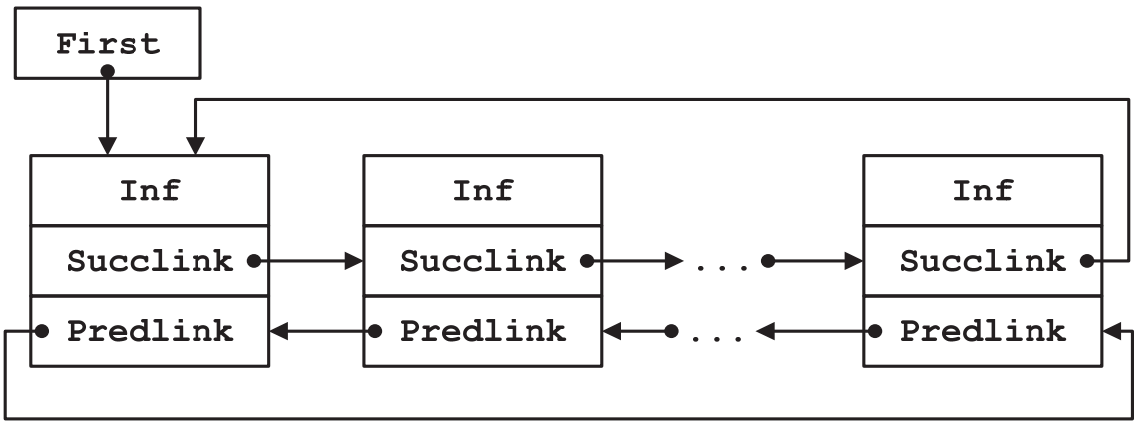
2. Обычный линейный список с двумя связями.



3. Обычный линейный циклический список с одной связью.



4. Обычный линейный циклический список с двумя связями.



Во всех перечисленных списках доступ возможен к любому узлу списка. Если в линейном списке добавление узла происходит в конце списка, а уничтожение – с его начала, то такой список называется *очередью*. Если же в линейном списке добавление узла происходит в начало списка, а уничтожение идет с его начала, то такой список называется *стеком*.

Ресурсы по работе со списками лучше объединить в отдельном модуле, образуя таким образом АТД «линейный список» и т. п.

Для работы со списками желательно предусмотреть следующие действия:

1) размещение узла списка в динамической памяти;

2) инициализация информационной части узла;

3) включение узла в нужное место списка;

4) поиск определенного узла списка;

5) распечатка информационной части узла списка;

6) распечатка информационных частей всех узлов списка (распечатка списка);

7) нахождение и удаление конечного числа узлов списка;

8) сортировка узлов списка по некоторому ключу из информационной части.

Однонаправленные связные списки

Предположим, требуется работать со списком, который содержит сведения о каком-либо автомобиле. Информационная часть узла такого списка будет содержать сведения о марке автомобиля, номере двигателя, шасси, государственном номере регистрации, владельце и т. д. Но если речь идет о непосредственной работе с узлами списка, то эта информация может быть несущественна. Поэтому сделаем в модуле только такие описания:

```
unit  Spis_New;
interface
```

```
type
      TInf   = record ... end;
      TLink  = ^TZveno;
      TZveno = record
                 Inf  : TInf;
                 Next : TLink
               end;
procedure  Print_inf (a : TZveno);
procedure  Init_inf  (a : TZveno);
procedure  Sozd_spis (var   First : TLink);
procedure  Show_spis (const First : TLink);
           .....
implementation
.....
end.
```

Здесь одно из полей записи предназначено для связывания узлов списка в единое целое – однонаправленный связный список. Это поле – `Next : TLink`. Структура связного списка предполагает, что очередной узел в списке через этот указатель связывается с последующим узлом. Последний узел списка имеет в поле `Next` указатель на `nil`.

Указатель на первый узел списка (и, соответственно, на весь список целиком) содержится в переменной `First`.

Алгоритм создания списка

Это тривиальная задача. Добавлять узлы в создаваемый список можно перед первым или после последнего узла. Рассмотрим алгоритм, основанный на добавлении нового узла первым.

1. Изначально, когда список пуст, указатель `First` устанавливается в `nil`.
2. Затем в динамической области под указатель `A` выделяется память.

```
A := New(TLink);
```

3. Информационная часть `A^`, например, при помощи процедуры `Init_inf`, наполняется информацией.
4. Для выполнения операции вставки узла в список требуется ссылочному полю `Next` присвоить ссылку на `First` (первоначально фактически на `nil`).
5. Меняя значение указателя `First`, узел вставляется в список.

```
First := A;
```

Далее указатель `A` можно использовать для других целей или освободить память, выделенную под него.

Пункты 2–5 повторяются.

Рассмотрите самостоятельно алгоритм, основанный на добавлении узла последним.

Вставка узла в список

При вставке очередного узла в список могут возникнуть следующие ситуации:

- включить первым;
- включить в середину;
- поместить в конец списка последним, но не единственным.

Включение узла в список первым

Напишем процедуру `AddFirst` для включения очередного узла `A` в начало списка.

```
procedure AddFirst (var A, First : TLink);
begin
     A^.Next := First;          {(1)}
     First   := A;                   {(2)}
end;
```

Включение узла в середину

Пусть `Old` – указатель, который указывает на место вставки. Рассмотрим две процедуры включения, так как здесь могут возникнуть такие ситуации, когда нужно включить:

- после узла, на который указывает указатель `Old` (процедура `AddAfter`);
- перед узлом, на который указывает указатель `Old` (процедура `AddBefore`).

Для включения после узла, на который указывает указатель `Old`, получим процедуру `AddAfter`.

```
procedure  AddAfter(var  A, Old : TLink);
begin
   A^.Next   := Old^.Next;       {(1)}
   Old^.Next := A;                    {(2)}
end;
```

Если нужно включить узел перед узлом, на который указывает `Old`, то однонаправленная цепочка связей создает трудность, поскольку нет доступа к узлу, который предшествует данному (`Old`). Тогда можно предложить такой прием:

1) содержимое `A^.Inf` переслать во вспомогательный элемент `Q`;
2) на место содержимого `A^` послать содержимое `Old^`;
3) на место содержимого `Old^.Inf` послать вспомогательный элемент `Q`;
4) установить ссылку `Old^.Next = A`.

В результате выполнения операции включения очередного узла A в середину списка перед узлом, на который указывает Old, получим следующую процедуру:

```
procedure AddBefore(var  A, Old : TLink);
var   Q : TInf;
begin
   Q         := A^.Inf;              {(1)}
   A^        := Old^;                {(2)}
   Old^.Inf  := Q;                   {(3)}
   Old^.Next := A;                   {(4)}
end;
```

Помещение узла в конец списка последним, но не единственным

Рассмотрим включение очередного узла в конец списка последним, но не единственным. Ссылка на последний узел находится в указателе Last.

```
procedure AddLast (var  A, Last : TLink);
begin
     Last^.Next := A;                {(1)}
     A^.Next    := nil;              {(2)}
     Last       := A;                {(3)}
end;
```

Удаление узла списка

При удалении узла списка также могут возникнуть различные ситуации:
- удаление первого узла списка (DelFirst);
- удаление узла в середине списка:
  - узел, который удаляется, стоит после узла Old (DelAfter);
  - удаление самого Old (DelCurrent);
- удаление последнего узла (DelLast).

Удаление первого узла списка

Рассмотрим удаление первого узла списка.

```
procedure  DelFirst(var First, A : TLink);
begin
      A     := First;                  {(1)}
      First := First^.Next;            {(2)}
end;
```

Далее в вызывающей программе можно уничтожить (Dispose(A)) объект, на который ссылается указатель A или использовать его для других целей.

Удаление узла в середине списка после узла Old

Рассмотрим удаление узла в середине списка после узла Old.

```
procedure  DelAfter (var  Old, A : TLink);
begin
     A         := Old^.Next;             {(1)}
     Old^.Next := Old^.Next^.Next;       {(2)}
end;
```

Далее в вызывающей программе можно уничтожить (Dispose(A)) объект, на который ссылается указатель A.

Удаление текущего узла в середине списка

Труднее удалить сам элемент Old, а не следующий за ним, так как обращение к узлу, который предшествует Old, невозможно. Если узел, на который указывает Old, последний в списке, то удалить его тоже невозможно, потому что нужно знать ссылку на предпоследний узел. Если же узел, на который указывает Old, не последний в списке, то удалить его можно: запомним в A местоположение следующего узла, перешлем содержимое следующего узла вместо содержимого узла, который удаляется, и получим решение задачи.

```
procedure  DelCurrent (var  Old, A : TLink);
begin
      A    := Old^.Next;              {(1)}
      Old^ := Old^.Next^;             {(2)}
end;
```

Далее в вызывающей программе можно уничтожить (Dispose) объект, на который ссылается указатель A.

Удаление последнего узла списка

Рассмотрим удаление последнего узла из однонаправленного списка.

```
procedure DelLast(var Old, Last, A : TLink);
begin
   A         := Old^.Next;
   Old^.Next := Old^.Next^.Next;
   Last      := Old;
end;
```

Далее в вызывающей программе можно уничтожить (Dispose) объект, на который ссылается указатель A.

Задачи

1. **Задача о считалочке с индульгенцией.** Пусть в круг становятся n человек и получают номера 1, 2, ..., n, считая по часовой стрелке. Поскольку люди стоят по кругу, то после последнего сразу идет первый. Затем, начиная с первого, тоже по часовой стрелке отсчитывается m-й человек и выводится из круга. После этого, начиная со следующего, снова отсчитывается m-й человек и выводится из круга. Это повторяется $n - 1$ раз. Победитель – тот, кто останется последним. Найдите его номер. Запрограммируйте вариант считалочки, если у какого-то из участников существует на один раз «индульгенция» на невыбывание.

2. Запрограммируйте вариант считалочки, если у некоторых из участников существует на несколько раз «индульгенция» на невыбывание. Получите номера выбывающих кандидатов в порядке их выбывания.

3. Напишите функцию, которая по списку L строит два новых списка: L1 – из положительных элементов и L2 – из отрицательных элементов списка L.

4. Сформируйте список целых чисел, вводимых пользователем из файла, в том порядке, в котором вводятся эти числа, но без повторений элементов.

5. Напишите функцию, которая оставляет в списке L только первые вхождения одинаковых элементов.

6. Имеется список целых чисел. Продублируйте в нем все четные числа.

7. Напишите функцию, которая по двум данным линейным спискам формирует новый список, состоящий из элементов, одновременно входящих в оба списка.

8. Напишите функцию, которая по двум линейным спискам L1 и L2 формирует новый список L, состоящий из элементов, входящих в L1, но не входящих в L2.

9. Определите, есть ли в списке действительных чисел элементы, превосходящие сумму всех элементов списка.

10. Напишите функцию, которая в линейном списке из каждой группы подряд идущих одинаковых элементов оставляет только один.

11. Напишите функцию, которая удаляет из списка элементы, входящие в него только один раз.

12. Пусть имеется список L1 действительных чисел. Запишите в список L2 все элементы списка L1 в порядке возрастания их значений.

13. Сформируйте список действительных чисел. Затем преобразуйте его, прибавив к положительным числам максимальный элемент.

14. Удалите из списка все элементы, встречающиеся более одного раза.

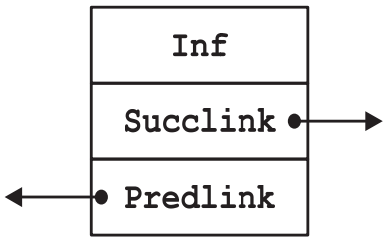
15. Дан список целых чисел. Продублируйте в нем все простые числа.

16. Удалите из списка действительных чисел все максимальные элементы.

17. Пусть имеются два списка, элементы которых упорядочены по возрастанию. Сформируйте новый список из элементов первого и второго списков, элементы которого будут упорядочены.

12.2. Двунаправленные связные списки

В двунаправленном списке каждый узел имеет два указателя: `Succlink` описывает связь узла со следующим, `Predlink` – с предыдущим.



Для работы с двунаправленным списком введем следующие типы данных:

```
type
        TInf   = record ...  end;
        TLink  = ^TZveno;
        TZveno = record
                   Inf      : TInf;
                   Succlink : Tlink;
                   Predlink : Tlink
                 end;
```

Ресурсы по работе с двунаправленным списком лучше объединить в отдельном модуле, образуя АТД «двунаправленный линейный список». Желательно предусмотреть следующие действия:

- размещение узла списка в динамической памяти;
- инициализация информационной части узла;
- включение узла в нужное место списка:
  - поиск определенного узла списка;
  - распечатка информационной части узла списка;
  - распечатка информационных частей всех узлов списка (распечатка списка);
  - нахождение и удаление конечного числа узлов списка;
  - сортировка узлов списка по некоторому ключу из информационной части.

Приведем основные операции для работы с двунаправленным списком.

Процедуры включения узла в двунаправленный список

Включение узла в список первым

```
procedure AddFirst_2(First, A : TLink);
begin
      A^.Succlink     := First;       {(1)}
      First^.Predlink := A;           {(2)}
      A^.Predlink     := nil;         {(3)}
      First           := A;           {(4)}
end;
```

Включение узла в список последним

Рассмотрим включение очередного узла в конец двунаправленного списка последним, но не единственным. Ссылка на последний узел находится в указателе `Last`.

```
procedure AddLast_2(var Last, A: TLink);
begin
      Last^.Succlink := A;                {(1)}
      A^.Predlink    := Last;             {(2)}
      A^.Succlink    := nil;              {(3)}
      Last           := A;                {(4)}
end;
```

Включение узла A после узла Old

Пусть `Old` – указатель, который указывает на место вставки. Рассмотрим процедуру включения нового узла `A` после узла, на который указывает `Old` (процедура `AddAfter_2`).

Для включения нового узла в двунаправленный список после узла, на который указывает `Old`, получим такую процедуру:

```
procedure AddAfter_2(var A, Old : TLink);
begin
   A^.Succlink           := Old^.Succlink; {(1)}
   A^.Predlink           := Old;           {(2)}
   Old^.Succlink         := A;             {(3)}
   A^.Succlink^.Predlink := A;             {(4)}
end;
```

Включение узла A перед узлом Old

Если нужно включить очередной узел перед узлом, на который указывает `Old`, то необходимо выполнить следующие действия.

```
procedure AddBefore_2(var A, Old : TLink);
begin
     A^.Succlink           := Old;             {(1)}
     A^.Predlink           := Old^.Predlink;   {(2)}
     Old^.Predlink         := A;               {(3)}
     A^.Predlink^.Succlink := A;               {(4)}
end;
```

Процедуры AddAfter_2 і AddBefore_2 не «подправляют» указатели, если включаемый узел станет первым или последним, так как для этого существуют отдельные процедуры.

Процедуры удаления узла из двунаправленного списка

Удаление первого узла

```
procedure DelFirst_2(var First, A : TLink);
begin
   A                          := First;
   First^.Succlink^.Predlink := nil;
   First                      := First^.Succlink;
end;
```

В указателе A содержится ссылка на элемент, который был выведен из списка. Потом можно освободить элемент с Heap (Dispose(A)).

Удаление последнего узла

```
procedure DelLast_2(var First, A : Link);
begin
      A                         := Last;
      Last^.Predlink^.Succlink := nil;
      Last                      := Last^.Predlink;
end;
```

В указателе A получили ссылку на элемент, который вывели из списка. Память, отведенную под него, можно освободить оператором Dispose(A).

Удаление узла после указателя на узел (середина списка)

```
procedure DelAfter_2(var A, Old : Link);
begin
      A                         := Old^.Succlink;
      Old^.Succlink             := A^.Succlink;
      Old^.Succlink^.Predlink := Old;
end;
```

Удаление узла перед указателем на элемент (середина списка)

```
procedure DelBefore_2(var A,Old:Link);
begin
   A                     := Old^.Predlink;
   A^.Predlink^.Succlink := Old;
   Old^.Predlink         := A^.Predlink;
end;
```

Удаление текущего узла (середина списка)

```
procedure  DelCurr_2(var Curr : Link);
begin
     Curr^.Predlink^.Succlink := Curr^.Succlink;
     Curr^.Succlink^.Predlink := Curr^.Predlink;
end;
```

После выполнения любой из пяти процедур удаления элемента из двунаправленного списка можно удалить элемент `A` или `Curr`.

Процедуры `DelAfter_2`, `DelBefore_2` и `DelCurr_2` предусматривают, что элемент удаляется строго с середины списка. Их следует усовершенствовать на те случаи, когда в удалении будут задействованы «крайние» узлы – `First` и `Last`.

Удаление узла после указателя на элемент

Объединяя рассмотренные алгоритмы, получим следующую процедуру удаления узла после указателя на элемент из двунаправленного списка:

```
procedure DelAfter_2_Best(var A, Old, Last : Link);
begin
 A := Old^.Succlink;
 if A^.Succlink = nil then
   begin
    Last          := Old;
    Old^.Succlink := nil;
   end
 else
   begin
    Old^.Succlink           := A^.Succlink;
    Old^.Succlink^.Predlink := Old
   end;
end;
```

Удаление узла перед указателем

Удаление узла перед указателем на элемент из двунаправленного списка фактически есть или удаление первого узла, или удаление узла перед указателем на элемент в середине списка.

Объединяя алгоритмы удаления первого узла и перед указателем на элемент в середине списка, получим для двунаправленного списка следующую процедуру удаления узла перед указателем на элемент:

```
procedure DelBefore_2_Best(var  A, Old, First : Link);
begin
 A := Old^.predlink;
 if A^.predlink = nil then
   begin
    First         := Old;
    Old^.Predlink := nil;
   end
 else
   begin
    A^.Predlink^.Succlink := Old;
    Old^.Predlink         := A^.Predlink;
   end;
end;
```

Удаление текущего элемента

Удаление текущего узла из двунаправленного списка – это фактически реализация одного из следующих случаев: 1) удаление первого узла, но не единственного; 2) удаление последнего узла, но не единственного; 3) удаление текущего узла в середине списка; 4) удаление текущего узла в списке, который содержит только один узел.

```
procedure DelCurr_2_Best(var Curr, First,
                                   Last : Link);
begin
 if  Curr^.Succlink = nil  then
    begin
     Last                     := Last^.Predlink;
     Curr^.Predlink^.Succlink := nil
    end
 else
    if Curr^.Predlink = nil  then
       begin
        First                    := First^.Succlink;
        Curr^.Succlink^.Predlink := nil;
```

```
        end
      else
        begin
         Curr^.Predlink^.Succlink := Curr^.Succlink;
         Curr^.Succlink^.Predlink := Curr^.Predlink;
        end;
end;
```

Прохождение связного списка не представляет трудностей. Фактически нужно переходить от узла к узлу по указателю из ссылочной части до достижения указателя, равного `nil`, который свидетельствует об окончании списка.

Задачи

1. **Задача о считалочке.** Пусть в круг становятся *n* человек и получают номера 1, 2, ..., *n*, считая по часовой стрелке. Затем, начиная с первого, тоже по часовой стрелке отсчитывается *m*-й человек и выводится из круга. После этого, начиная со следующего, снова отсчитывается *m*-й человек и выводится из круга. Это повторяется *n* – 1 раз. Победитель – тот, кто останется последним. Найдите его номер. Далее выполните движение против часовой стрелки и найдите номер оставшегося человека. Есть ли стратегии, когда ответом будет один и тот же номер?

2. Пусть имеется односвязный список действительных чисел, каждый элемент которого содержит дополнительное (нереализованное) ссылочное поле `prev`. Преобразуйте исходный односвязный список в двусвязный, в котором каждый элемент связан не только с последующим (с помощью поля `next`), но и с предыдущим (с помощью поля `prev`).

3. Напишите функцию, которая по произвольному указателю на один из элементов двусвязного списка подсчитывает количество элементов в списке.

4. Напишите функцию, которая, получив в качестве параметра указатель на один из элементов двусвязного списка действительных чисел и два числа, добавляет первое число в начало списка, а второе – в его конец.

5. Заданы два многочлена высокой степени, одночлены которых хранятся как звенья двунаправленного списка, т. е. имеются два упорядоченных по степеням двунаправленных списка, содержащие пары чисел – коэффициент и степень. Найдите многочлен как сумму заданных.

6. Имеется двусвязный список действительных чисел. Продублируйте все положительные числа в списке.

7. Пусть имеется некоторый двусвязный список действительных чисел и некоторое число *C*. Переставьте значения элементов таким образом, чтобы сначала следовали (в произвольном порядке) элементы, меньшие числа *C*, а затем элементы, не меньшие числа *C*.

8. Определите, расположены ли элементы в двусвязном списке симметричным образом.

9. Осуществите циклический сдвиг элементов двусвязного списка на k позиций вправо.

12.3. Стеки

Организация стека последовательным методом хранения

Стеки – это особый случай списка и простейшая динамическая структура данных. Добавление элементов в стек и выборка из него выполняются с одного конца, который называется *вершиной стека*. Другие операции со стеком не определены. Стеки широко используются в системном программировании, компиляторах, в различных рекурсивных алгоритмах.

Обычно стек обозначается английской аббревиатурой `LIFO – Last In First Out`, что можно перевести как «последний вошел, первый вышел». Это правило передает механизм работы со стеком.

Для стека нужна операция добавления элемента в стек (`AddFirst`), но для него такую операцию называют `Push` – «затолкнуть». Вторая операция (`DelFirst`) – «вытолкнуть» элемент из стека, для него такую операцию называют `Pop`. Указатель на вершину стека называют `Top`.

Поскольку в переполненный стек нельзя «затолкнуть» очередной элемент, а из пустого стека нельзя «вытолкнуть» элемент, то при работе со стеками нужны дополнительные подпрограммы:

- **`function`** `Empty` – возвращает значение `true`, если стек пуст, и значение `false` в противном случае;
- **`function`** `Full` – выдает значение `true`, если стек полностью заполнен, и `false` в противном случае;
- **`procedure`** `Mistake` – обрабатывает ошибочную ситуацию, параметр – код ошибки;
- **`procedure`** `Clear` – «очищает» стек, параметр – указатель на вершину стека.

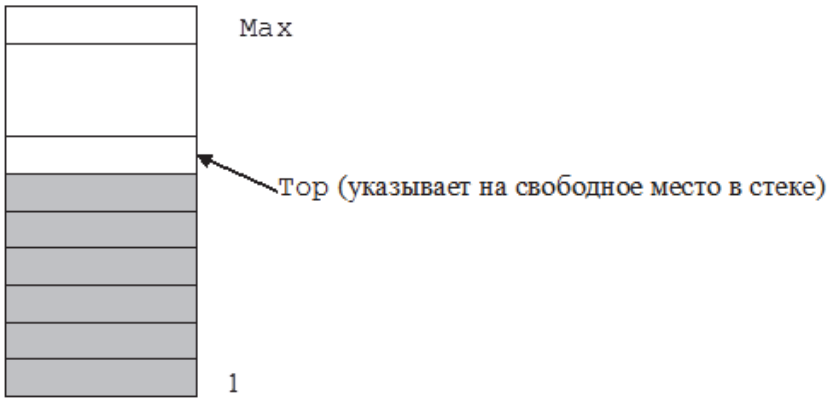
Поскольку для таких видов списка доступ односторонний, то их можно образовывать не только при помощи связных списков, используя разработанные выше процедуры, но и при помощи массивов.

Чем хорошо описывать работу со стеком отдельным модулем? Во-первых, это отдельная часть программы, которую можно модифицировать независимо от программы. Во-вторых, если все ресурсы по работе со стеками собрать в модуль, то реализацию стека можно «сохранить» в части `implementation`, создав АТД «стек».

Работу со списками в динамической памяти мы уже усвоили, а теперь рассмотрим организацию стека в виде массива.

Количество элементов в массиве – глубину стека – объявляется константой `max`. Индекс элемента массива, через который совершается доступ, – это указатель на вершину `Top`.

Если указатель `Top` равен единице, то стек «пустой» и в него можно затолкнуть очередной элемент, но нельзя из него вытолкнуть элемент. Если указатель `Top = Max + 1`, то стек «полный» и в него нельзя затолкнуть элемент, но можно вытолкнуть.



Далее опишем модуль, в котором соберем ресурсы по работе со стеком целых чисел.

```
unit StackInt;
interface
type
     TItem = Integer;
procedure Push    (x  : TItem);
function  Pop          : TItem;
procedure Mistake (Kod : Byte);
procedure Clear;
function  Empty        : Boolean;
function  Full         : Boolean;

implementation

const Max = 100;
var
      Stack : array[1..Max] of TItem;
      Top   : Word;

function  Empty;
begin
 Empty := (Top = 1);
end;
```

```
function  Full;
begin
 Full := (Top > Max);
end;

procedure  Clear;
begin
 Top := 1;
end;

procedure Push;
begin
 if Full then Mistake (1)
 else
   begin
    Stack[Top] := x;
    Inc(Top);
   end;
end;

function Pop;
begin
 if Empty then Mistake(2)
 else
    begin
     Dec(Top);
     Pop := Stack[Top]
    end;
end;

procedure Mistake;
begin
 case Kod of
   1:
  Write('Стек переполнен, увеличьте константу Max');
   2:
  Write('Стек пуст');
 else
  Write('Другие ситуации');
 end;
 Halt;
end;

begin
end.
```

Задачи

1. Напишите функцию, которая распечатывает слова из текстового файла в обратном порядке. По файлу можно пройти только один раз.

2. Заданы два многочлена высокой степени, одночлены которых хранятся в стеках, т. е. имеются два упорядоченных по степеням стека, содержащих пары чисел – коэффициент и степень. Найдите многочлен как сумму заданных.

3. В некотором текстовом файле хранится программа на некотором алгоритмическом языке. Проверьте ее на сбалансированность круглых, квадратных и фигурных скобок.

12.4. Очереди

Очередь – это отдельный случай списка и простая динамическая структура данных. Добавление элементов в очередь выполняется в конец очереди, а выборка – с начала очереди. При выборке элемент исключается из очереди. Другие операции с очередью не определены. Очереди широко используются в программировании.

Очередь реализует принцип `FIFO - First In First Out`, что можно перевести как «первый вошел, первый вышел». Это правило напрямую передает механизм работы с очередью. Для очереди нужна операция `Push` – затолкать элемент в очередь, операция `Pop` – вытолкнуть элемент из очереди. Указатель на начало очереди называют `Front`, на конец – `Rear`.

Поскольку в переполненную очередь нельзя затолкать очередной элемент и с пустой очереди нельзя вытолкнуть элемент, то при работе с очередью нужны дополнительные подпрограммы:

- **`function`** `Empty` – выдает значение `true`, когда очередь пустая, и значение `false` в противном случае;
- **`function`** `Full` – выдает значение `true`, когда очередь переполнилась, и значение `false` – в противном случае;
- **`procedure`** `Mistake` – обрабатывает ошибочную ситуацию, параметр – код ошибки;
- **`procedure`** `Clear` – очищает очередь, параметр – указатели `Front` и `Rear`.

Для такого вида списков доступ к элементу происходит на концах, поэтому их также можно реализовывать не только при помощи динамических структур данных, но и массивов.

Для моделирования очереди при помощи массива можно применить два подхода.

1. Массив с фиксированным количеством элементов, в котором элементы очереди занимают группу соседних компонент от `Front` до `Rear`. Когда очередь достигает правого края массива, то все ее элементы сразу сдвигаются к левому краю (пересылка элементов будет занимать какое-то время).

2. Представление аналогично предыдущему, но массив как бы склеивается в кольцо. Если элементы очереди достигают правого края массива, тогда новые элементы записываются на свободное место в начало массива. Второй подход более эффективен по времени работы программы.

Когда очередь пуста, индекс `Front` устанавливается в `1` – первый незанятый элемент массива, за которым можно обслуживать, а индекс `Rear` устанавливается в `0` – последний элемент очереди, за которым можно писать в очередь. Опишем переменную `Size`, в которой будем хранить текущую длину очереди.

Для работы с несколькими очередями напишем модуль, где будут описаны выше заявленные ресурсы без использования динамических переменных. Получим следующую АТД «очередь»:

```
unit  TurnReal;
interface
const
       TurnMax  = 100;
       TurnMin  = 1;
       TurnSize = TurnMax - TurnMin + 1;
type
       TItem = Real;
       TMas  = array[TurnMin..TurnMax] of  TItem;
       Turn  = record
                 Front : Word;   {Начальный  индекс}
                 Rear  : Word;   {Конечный   индекс}
                 Size  : Word;   {Длина     очереди}
                 A     : TMas;   {          Очередь}
               end;
procedure  Push       (var Q : Turn;  x : TItem);
function   Pop        (var Q : Turn)    : TItem;
procedure  Mistake    (Kod   : Byte);
procedure  Clear      (var Q : Turn);
function   Empty      (var Q : Turn)    : Boolean;
function   Full       (var Q : Turn)    : Boolean;
procedure  CheckBounds(var  index       : Word);

implementation

function Empty;
```

```
begin
 Empty := Q.Size = 0;
end;

function Full;
begin
 Full := Q.Size = TurnSize;
end;

procedure Clear;
begin
 Q.Front := TurnMin;
 Q.Rear  := TurnMin - 1;
 Q.Size  := 0;
end;

procedure CheckBounds;
begin
 if index < TurnMin then index := TurnMax
 else
    if index > TurnMax then index := TurnMin;
end;

procedure Push;
begin
 if Full (Q) then Mistake(1)
 else
   begin
    Inc(Q.Rear);
    Inc(Q.Size);
    CheckBounds(Q.Rear);
    Q.A[Q.Rear] := x;
   end;
end;

function Pop;
begin
 if Empty(Q) then Mistake(2)
 else
   begin
    Pop := Q.A[Q.Front];
    Inc(Q.Front);
    CheckBounds(Q. Front);
    Dec(Q.Size);
   end;
```

```
 end;

procedure Mistake;
begin
   case Kod of
 1:
  Write('Очередь переполнена,увеличьте константу Max');
 2:
  Write('Очередь пустая');
 else
  Write('Другие ситуации');
   end;
 Halt;
end;

begin
end.
```

Очередь с двусторонним доступом (дек)

Двусторонняя очередь обладает свойствами как стека, так и простой очереди. Данные в нее могут добавляться в любой конец и обслуживаться с любого конца. Обычно в программах, реализующих двустороннюю очередь, предусмотрены процедуры добавления данных в начало и в конец очереди, а также обслуживания как с начала, так и с конца. Программную реализацию можно выполнять как на базе массива, так и при помощи двусвязных списков.

Очередь с приоритетом

Очередь с приоритетом – это очередь, в которой каждому элементу присвоен некоторый приоритет. Приоритет задается номером. Обычно самый высокий приоритет имеет номер 1, более низкий – 2 и т. д. Однако бывает и наоборот.

Для очереди с приоритетом предусмотрены две операции: добавление нового элемента в очередь согласно его приоритету и обслуживание элемента, который имеет самый высокий приоритет. Для такой очереди при реализации процедуры `Push` необходимо сначала найти место вставки элемента в очередь, ориентируясь на его приоритет. Тогда процедура `Pop` будет правильно обслуживать очередь с приоритетом.

Задачи

1. За какое минимальное количество ходов конем можно попасть из заданной начальной точки в конечную, посещая шахматное поле один раз? По возможности укажите последовательность действий.

Алгоритм. Запомним начальный ход в очереди ходов, указав, что он нулевого порядка.

Повторяем действие, пока не попадаем в конечную точку.

Извлечем из очереди координаты очередного хода.

Организуем на следующем проходе добавление в очередь всех возможных перемещений коня согласно таблице перемещения, если ход приемлем и клетка не занята.

Рассмотрим правила перемещения коня. Если задана начальная пара координат (x, y), то в наилучшем случае имеется восемь возможных координат (u, v) следующего хода. Отобразим их в таблице.

	1		8	
2				7
		x, y		
3				6
	4		5	

Из этой таблицы видно, что изменения координат в соответствии с номером хода будут следующими:

k	Δx	Δy
1	–1	2
2	–2	1
3	–2	–1
4	–1	–2
5	1	–2
6	2	–1
7	2	1
8	1	2

2. В привилегированную очередь помещаются некоторые случайные целые числа. Признаком привилегии является старшая цифра числа: чем она больше, тем выше привилегия. Получите список случайных чисел и распечатайте его. Обслужите очередь, распечатав ее элементы.

3. Даны две непустые очереди, которые содержат одинаковое количество элементов. Объедините очереди в одну, в которой элементы исходных очередей чередуются.

4. Даны две непустые очереди. Элементы каждой из очередей упорядочены по возрастанию. Объедините очереди в одну с сохранением упорядоченности элементов.

5. Пусть имеется файл действительных чисел и некоторое число *C*. Используя очередь, напечатайте сначала все элементы, меньшие числа *C*, а затем все остальные элементы.

6. «Символы-убийцы». Отредактируйте строку символов по правилу: если встречается символ #, то предыдущий символ стирается; когда встречается символ @, то все предыдущие символы стираются.

12.5. Деревья

Основные понятия и определения

Отношения между объектами могут носить нелинейный характер, например, определяться логическими условиями типа «один ко многим» или «многие ко многим».

Отношение «один ко многим» носит иерархический характер и отображается древовидными структурами (структура вуза, УДК и др.).

Дерево – это набор узлов, между которыми установлены родительско-дочерние отношения.

На самом верхнем уровне такой иерархии всегда имеется только один узел, называемый *корнем* дерева.

Каждый узел, кроме корневого, связан только с одним узлом более высокого уровня, называемым *узлом-предком*. При этом каждый узел дерева может быть соединен с произвольным количеством узлов более низкого уровня, которые для данного узла являются *дочерними узлами* (*узлами потомками*). В некоторых источниках узлы-потомки называют *сыновьими узлами*.

Считается, что корень дерева размещен на уровне 0, остальные узлы имеют следующую иерархию: если узел x имеет уровень i, то его непосредственный потомок – уровень $i + 1$.

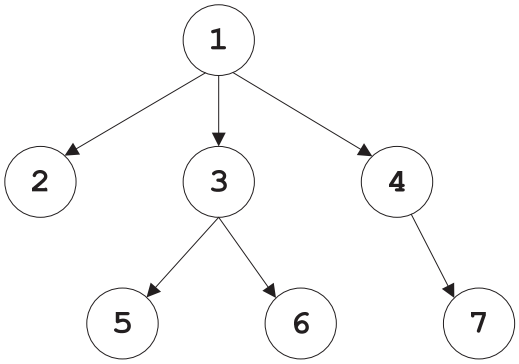
Любой узел дерева с его потомками также образует дерево, называемое *поддеревом* (относительно исходного дерева).

Основная терминология

Дерево – это конечное множество узлов с одним выделенным узлом B_0, который называется *корнем* дерева, а остальные узлы разбиты на $M > 0$ множеств B_1, B_2, ..., B_M. Эти множества не пересекаются, каждое из них является *поддеревом*.

Соотношения между узлами дерева обладают следующими особенностями:

- существует узел (корень дерева), который не имеет предшественника;
- любой другой узел имеет одного предшественника.



B_1 ={2};
B_2 ={3, 5, 6};
B_3 ={4, 7}.

Узлы дерева, которые не имеют потомков, являются *листьями* дерева (нетерминальные элементы), остальные – внутренние (терминальные элементы).

От корня до любого узла всегда существует только один путь. Максимальная длина пути от корня до листьев называется *высотой* дерева.

Глубина узла – длина пути от корня до этого узла.

Степень узла n – это число его потомков. Наибольшая из степеней всех узлов считается степенью дерева.

Дерево степени *n* называется *полным*, когда степень любого его узла равна *n* или 0.

Дерево, каждый узел которого представлен одним и тем же типом записи, называется *однородным*.

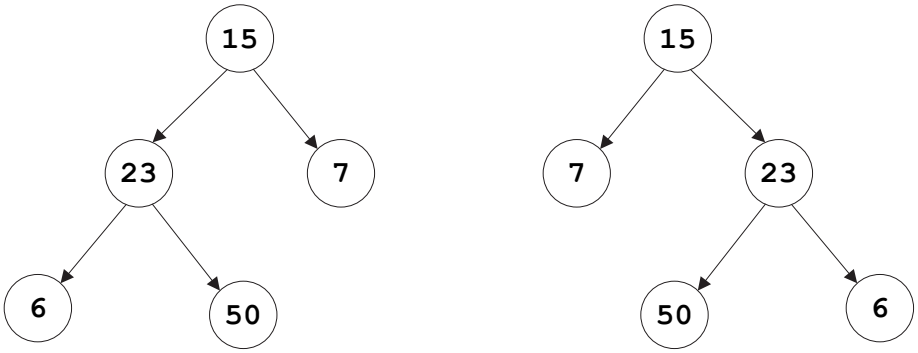
Если узлы дерева представлены разными типами записей, то оно называется *неоднородным*.

Типы деревьев

В зависимости от количества потомков (ветвистости) деревья разделяют на *бинарные* (*binary trees*) – не больше двух поддеревьев у произвольного элемента и *сильноветвистые* (*multiway trees*) – есть элементы, которые имеют больше двух поддеревьев.

Дерево *упорядочено*, когда для каждого потомка k' узла k степени n указано его положение: первый, второй, ..., n-й (левый, средний, ..., правый).

Ниже показано одно и то же полное неупорядоченное дерево степени 2. Если считать их упорядоченными, то это разные деревья.



Бинарные деревья

Прямой и симметричный обходы деревьев

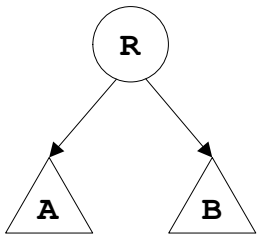
Рассмотрим упорядоченное дерево. Дочерние узлы обычно упорядочиваются слева направо – левостороннее упорядочивание (если не оговорено правостороннее).

При прохождении узлов дерева в *прямом порядке* сначала посещается корень, затем узлы самого левого поддерева, далее узлы следующих поддеревьев.

При *обратном обходе* узлов дерева сначала посещаются все узлы поддеревьев в обратном порядке, затем корень.

Обход упорядоченного бинарного дерева

Рассмотрим упорядоченное бинарное дерево:



Обход (прохождение) упорядоченного бинарного дерева проводят шестью методами:

а) при левостороннем упорядочивании:

- сверху вниз: `R`, `A`, `B` (префиксная форма обхода – `PreOrder`);
- слева направо: `A`, `R`, `B` (инфиксная форма обхода – `InOrder`);
- снизу вверх: `A`, `B`, `R` (постфиксная форма обхода – `Postorder`);

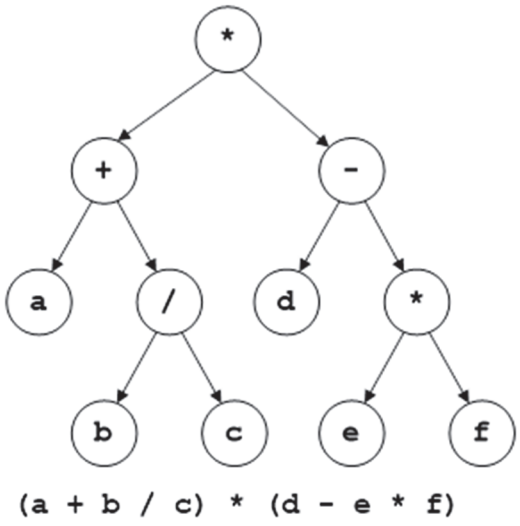
б) правостороннем упорядочивании:

- сверху вниз: `R`, `B`, `A`;
- слева направо: `B`, `R`, `A`;
- снизу вверх: `B`, `A`, `R`.

Более очевидными становятся три первых варианта обхода на примере деревьев-формул.

Дерево-формула получается в соответствии с арифметическим выражением, содержащим переменные величины, арифметические операции и скобки. Согласно такому алгоритму имеем: в корень дерева помещается операция, операнды же направляются в левое и правое поддеревья.

Очевидно, что арифметическому выражению $(a + b / c) * (d - e * f)$ соответствует рассмотренное далее дерево.



При выполнении префиксного обхода R, A, B, получаем следующую последовательность: *, +, a, /, b, c, –, d, *, e, f.

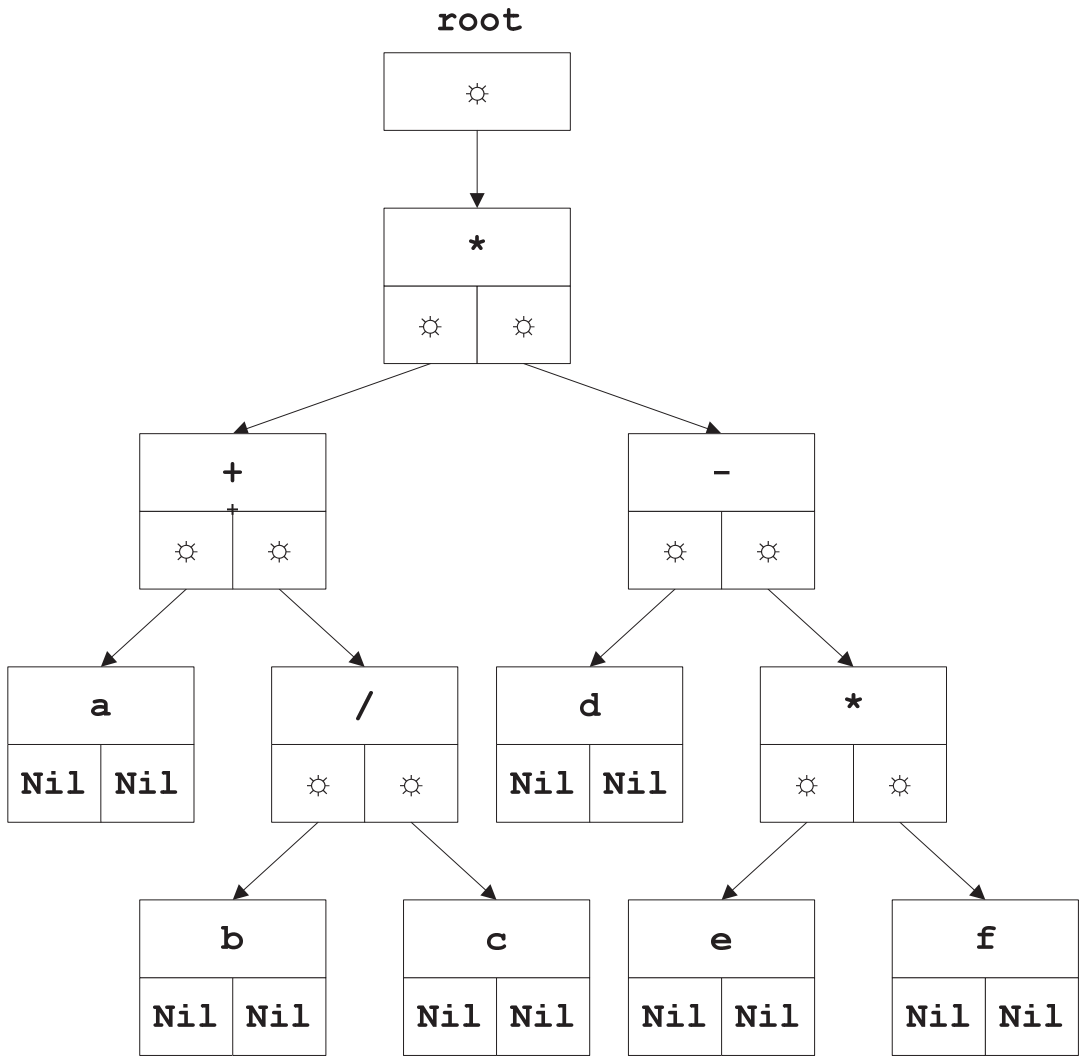
Если же выполнить инфиксный обход A, R, B, получим такую последовательность: a, +, b, /, c, *, d, –, e, *, f.

Если тут своевременно проставить круглые левые и правые скобки, получим полную скобочную запись выражения, в котором потом в соответствии с приоритетом операций некоторые скобки можно опустить.

Постфиксный обход A, B, R дает последовательность: a, b, c, /, +, d, e, f, *, –, *.

Представление деревьев

Представляя данное дерево с помощью ссылок, будем иметь следующее дерево:



Введем определения:

```
Type
         TRef  = ^TNode;
         TNode = record
                   op          : char;
                   left, rigth : TRef;
                 end;
var      root : TRef;
```

Над деревьями обычно выполняются следующие операции:

- добавить узел в дерево;
- исключить из дерева определенный узел;
- пройти все узлы дерева в заданном порядке;
- найти узел с заданным свойством;
- определить родительский узел заданного узла;
- определить дочерние узлы заданного узла и др.

Надо заметить, что само дерево определяется в терминах рекурсивных соотношений, поэтому действия над деревьями лучше всего задавать с помощью рекурсии.

Работа с бинарными деревьями

Рассмотрим процедуры левостороннего обхода упорядоченного дерева.

Способы левостороннего обхода упорядоченного дерева

```
procedure PreOrder(t : TRef);
begin
 if t <> nil then
   begin
 { Обработка узла, например,  Write(t^.op : 4);}
    PreOrder(t^.left);
    PreOrder(t^.right);
   end;
end;
```

```
procedure InOrder(t : TRef);
begin
 if t <> nil then
   begin
    InOrder(t^.left);
 { Обработка узла, например,  Write(t^.op : 4);}
    InOrder(t^.right);
   end;
```

```
end;
procedure Postorder(t : TRef);
begin
 if t <> nil then
   begin
    Postorder(t^.left);
    Postorder(t^.right);
 { Обработка узла, например,  Write(t^.op : 4);}
   end;
 end;
```

Напишем процедуру распечатки схемы дерева. Во-первых, представим дерево, повернув его на 90° против часовой стрелки. Поскольку каждый новый уровень будет отступать от предыдущего на несколько позиций, то в процедуру построения добавим параметр – номер уровня. Получим рекурсивный алгоритм:

- пустое дерево не печатается для поддерева уровня *h*;
- печатаем правое поддерево;
- печатаем узел, который выделяется предыдущими пробелами, соответствующими по количеству уровню *h*;
- печатаем левое поддерево.

Процедура распечатки схемы дерева

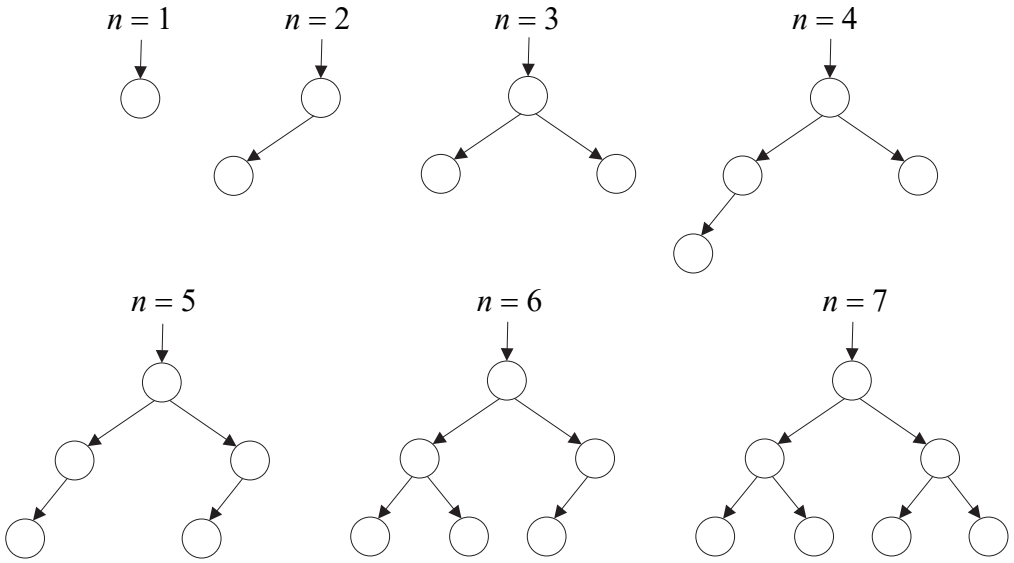
```
procedure PrintTree(t : TRef; h : Byte);
var   i : Byte;
begin
  if  t = nil then Exit;
  PrintTree(t^.right, h + 1);
  for i := l to h do Write('    ');
  Writeln(t^.op);
  PrintTree(t^.left, h + 1);
end;
```

Задание. Напишите процедуру распечатки схемы дерева с использованием графического режима.

Некоторые типы деревьев

Идеально сбалансированные деревья

Ранее рассматривались деревья-формулы. Существуют также *идеально сбалансированные* деревья, которые имеют вид



Такие деревья дают минимальную глубину. Чтобы достичь наименьшей глубины при данном числе узлов, нужно на всех уровнях, кроме самого нижнего, распределить максимально возможное количество узлов.

Дерево *идеально сбалансированное*, если для каждого его уровня число узлов в левом и правом поддереве отличается не более чем на один.

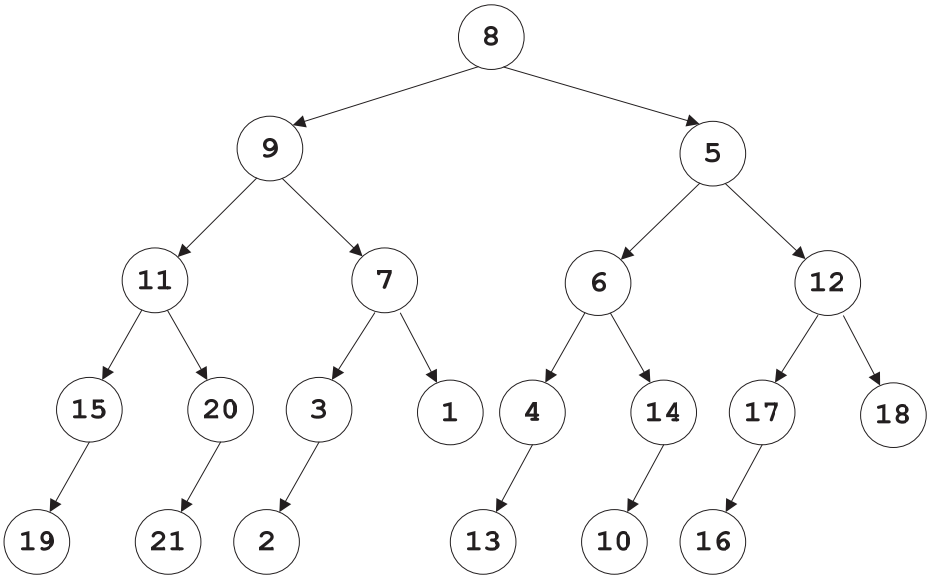
Это правило легко сформулировать при помощи рекурсии:

- взять один узел в качестве корня;
- построить левое поддерево с $n_л = n$ div 2;
- построить правое поддерево с $n_п = n - n_л - 1$.

Рассмотрим построение такого дерева на примере задачи.

Задача. Построить идеально сбалансированное дерево из заданных чисел, находящихся в типизированном файле.

Построим сначала самостоятельно дерево из следующей последовательности чисел: 8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 14, 10, 12, 17, 16, 18 ($n = 21$):



Далее опишем рекурсивную функцию

```
function Tree (n:Integer):TRef,
```

которая строит идеально сбалансированное дерево.

Результатом работы функции будет указатель на корень дерева.

Если подключим рассмотренные ранее подпрограммы, получим следующую программу.

Построение
идеально сбалансированного дерева

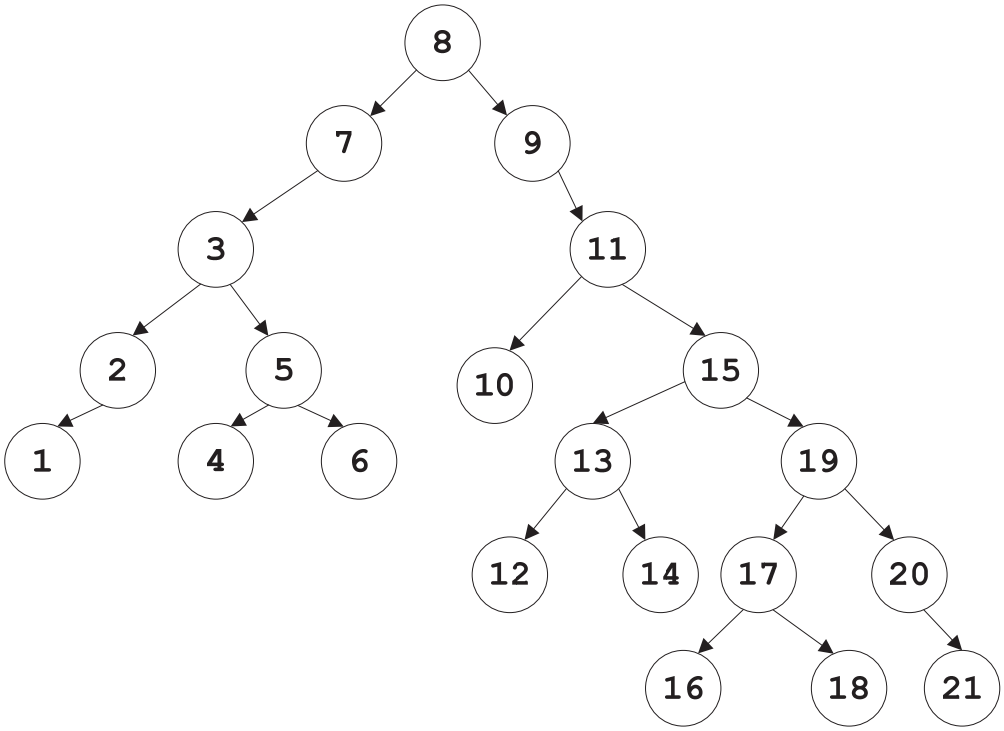
```
Program Build_Tree;
uses crt;
type
    TInf  = Integer;
    TRef  = ^TNode;
    TNode = record
              op         : TInf;
              left,right : TRef;
            end;
var  f    : file  of  TInf;
     root : TRef;
     n    : Integer;

function Tree(n : Integer) : TRef;
                 {Ссылка на корень}
var
      Newnode : TRef;
      nl,nr   : Integer;
      x       : TInf;
begin
 if n = 0 then  tree := nil
 else
   begin
    nl := n div 2;
    nr := n - nl - 1;
    New(Newnode);
    Read(f, x);
    Newnode^.op    := x;
    Newnode^.left  := tree(nl);
    Newnode^.right := tree(nr);
    tree           := Newnode;
   end;
end;
```

```
procedure PreOrder(t:TRef);
begin
 if t = nil then Exit;
 Write(t^.op : 4);
 PreOrder(t^.left);
 PreOrder(t^.right);
end;

procedure PrintTree(t:TRef; h : Byte);
var    i : Integer;
begin
 if t = nil then Exit;
 PrintTree(t^.right, h + 1);
 for i := 1 to h do Write('   ');
 Writeln(t^.op:10);
 PrintTree(t^.left, h + 1);
end;

begin
 ClrScr;
 Assign(f, 'c:\Pascal\DataWord.dat');
 Reset(f);
 n := FileSize(f);
 Root := Tree(n) ;
 PrintTree(root, 0);
 PreOrder(root);
 { другая работа }
end.
```

Дерево поиска

Двоичное дерево поиска (англ. *binary search tree*, *BST*) – это двоичное дерево, для которого выполняются следующие дополнительные условия: значения всех элементов, расположенных в левом поддереве, меньше значения в корне дерева, а значения всех элементов, расположенных в правом поддереве, больше либо равны значению в корне дерева.

Для последовательности 8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 14, 10, 12, 17, 16, 18 получим следующее дерево:



Программу построения дерева поиска напишем далее.

Следующая задача получения частотного словаря строит дерево поиска, которое называется *лексикографическим* деревом.

Лексикографическое дерево поиска

Задача. Имеется массив слов (чисел). Подсчитайте, сколько раз встречается каждое слово (число) в массиве. Распечатайте слова (числа) в алфавитном порядке (порядке возрастания).

Решение выполним для целых чисел, хранящихся в файле. Начинаем с пустого дерева. Затем каждое число ищется (просеивается) в дереве по принципу дерева поиска (меньше, чем в вершине, – идем на левое поддерево; в противном случае – на правое). Если число найдено, увеличиваем счетчик его появлений; если числа нет в дереве, оно вставляется в дерево, причем счетчик его становится равным 1.

Распечатка чисел в порядке возрастания

```
Program Build_Tree_Praseivanne;
type
       TInf  = Integer;
       TRef  = ^TNode;
       TNode = record
                key        : TInf;
                count      : Integer;
                left,right : TRef;
               end;
```

```
var      f    : file  of  Integer;
         root : TRef;
         k    : TInf;
         P    : Pointer;
procedure Praseivanne(x : TInf; var p : TRef);
begin
 if p = nil  then
  begin
   New(p);
   P^.key   := x;
   P^.count := 1;
   P^.left  := nil;
   P^.right := nil;
  end
 else
    if P^.key>x then Praseivanne(x, P^.left)
    else
        if P^.key<x then Praseivanne(x,P^.right)
        else Inc(p^.count);
end;

procedure InOrder(t:TRef);
begin
 if t <> nil   then
   begin
    InOrder(t^.left);
    Write  (t^.key:4);
    InOrder(t^.right);
   end;
end;

procedure PrintTree(t : TRef; h : Byte);
var    i : Byte;
begin
if t = nil then Exit;
   PrintTree(t^.right, h + 1);
   for i := 1 to h do   Write('   ');
   Writeln(t^.key);
   PrintTree (t^.left, h + 1);
end;

begin
 Mark  (P);
 Assign(f, 'Z:\Home\ffff.dat');
 Reset (f);
```

```
Root := nil;
while not Eof(f) do
    begin
      Read(f, k);
      Praseivanne(k, root);
    end;
PrintTree(root, 0);
InOrder  (root);
{іншая работа}
Readln;
Release(P);
end.
```

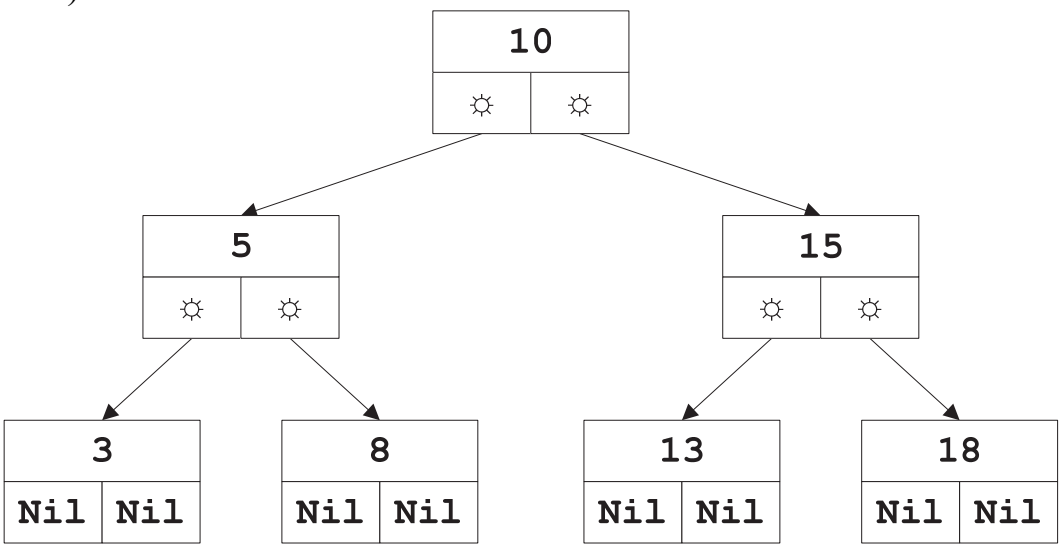
Симметричный обход дерева, построенного слева направо, даст отсортированный по возрастанию массив чисел, справа налево – по убыванию.

Удаление из бинарного дерева поиска

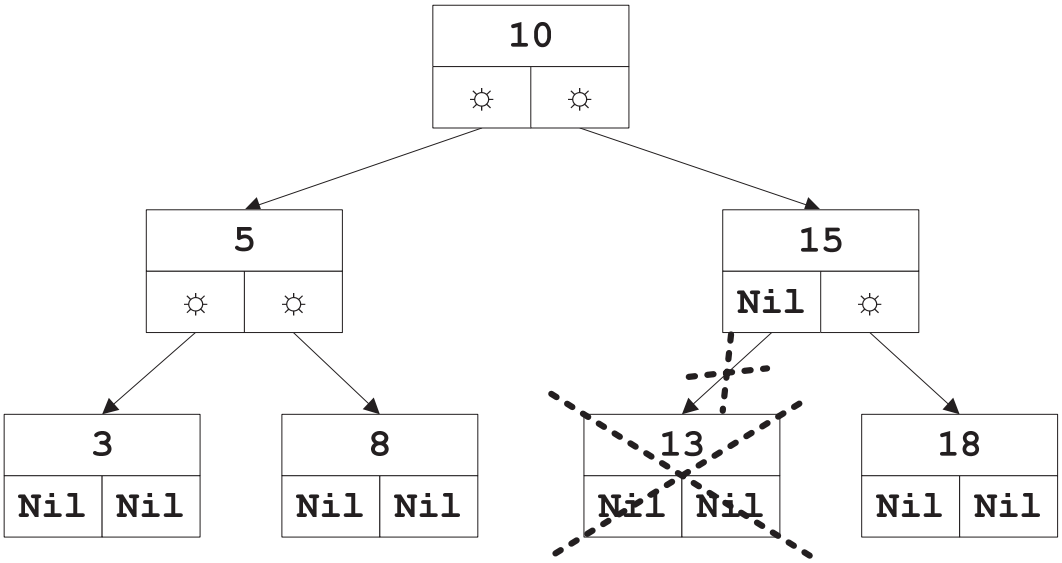
Когда встает задача удаления узла из дерева, то могут возникнуть различные обстоятельства.

Рассмотрим следующее дерево поиска и обсудим ситуации удаления узла так, чтобы после операции осталось также дерево поиска.

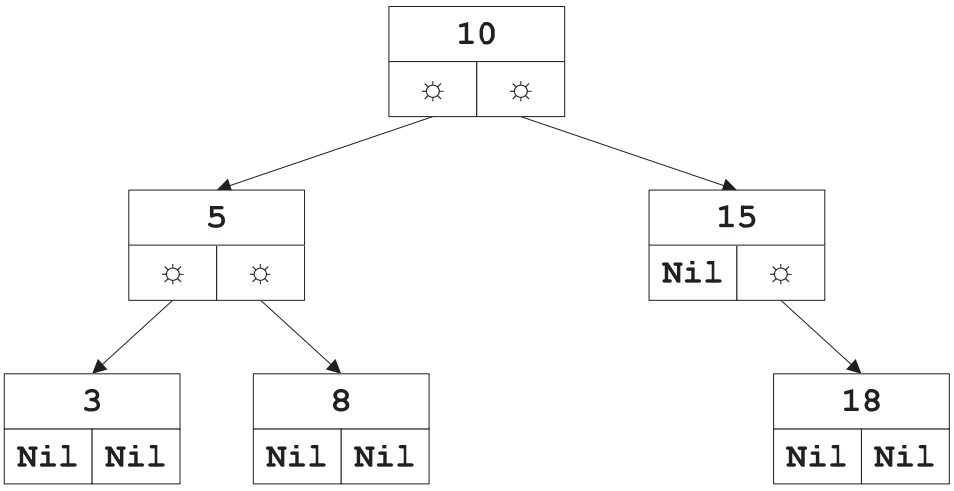
1. Пусть нужно удалить лист следующего дерева (например, элемент «13»):



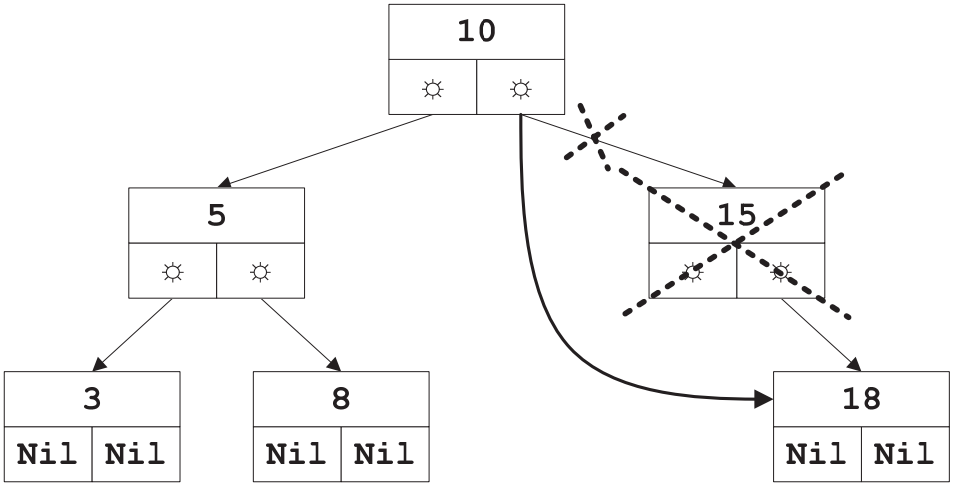
Тогда родительский узел «15» вместо ссылки на дочерний узел должен записать **`nil`**:



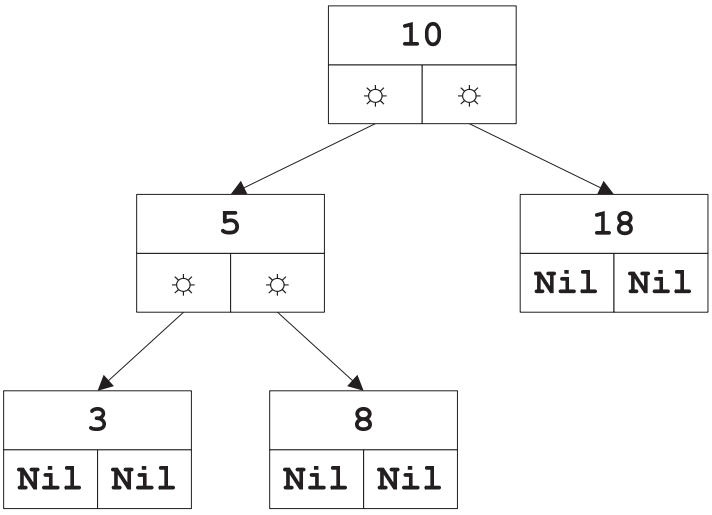
2. Пусть в полученном дереве нужно удалить узел, который имеет одного потомка (например, элемент «15»):



Тогда на место родительского узла нужно переместить дочерний узел:

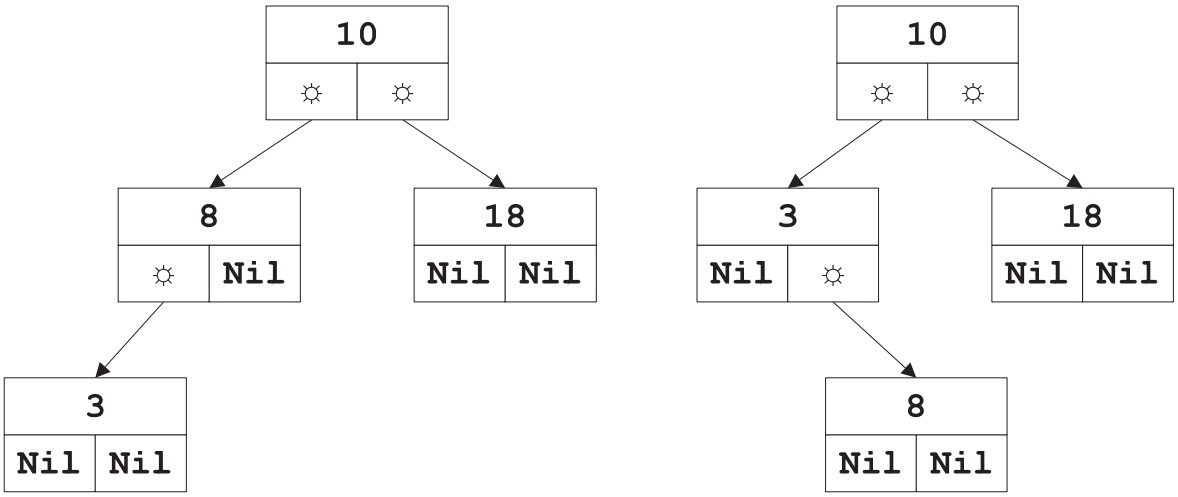


3. Пусть в последнем дереве нужно удалить узел, имеющий двоих потомков (например, элемент «5»):



Это возможно сделать двумя способами, так как на место родительского узла можно подставить левый (вариант 1) или правый (вариант 2) дочерний узел.

Получим следующие варианты схем удаления:

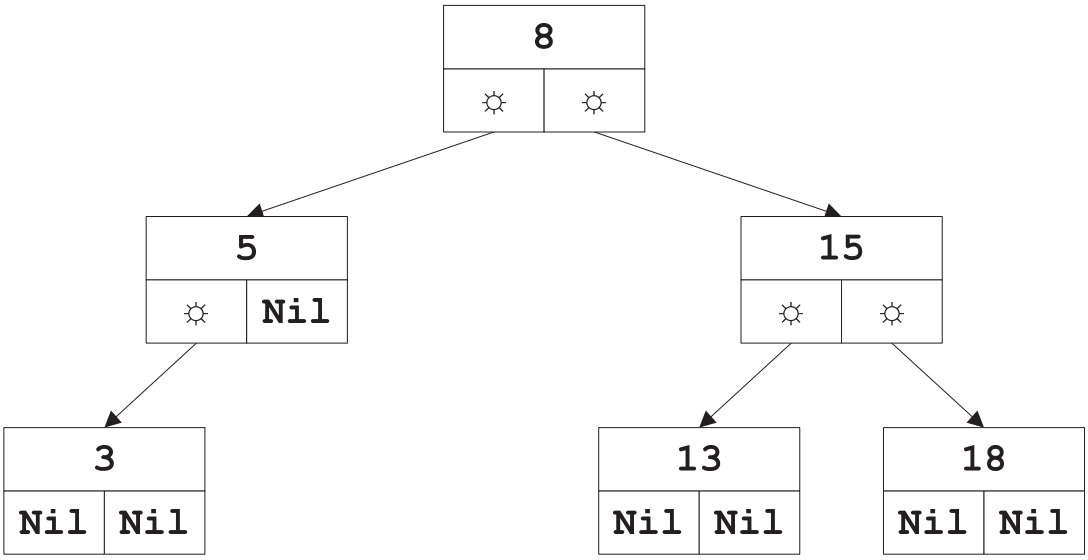


Вариант 1 Вариант 2

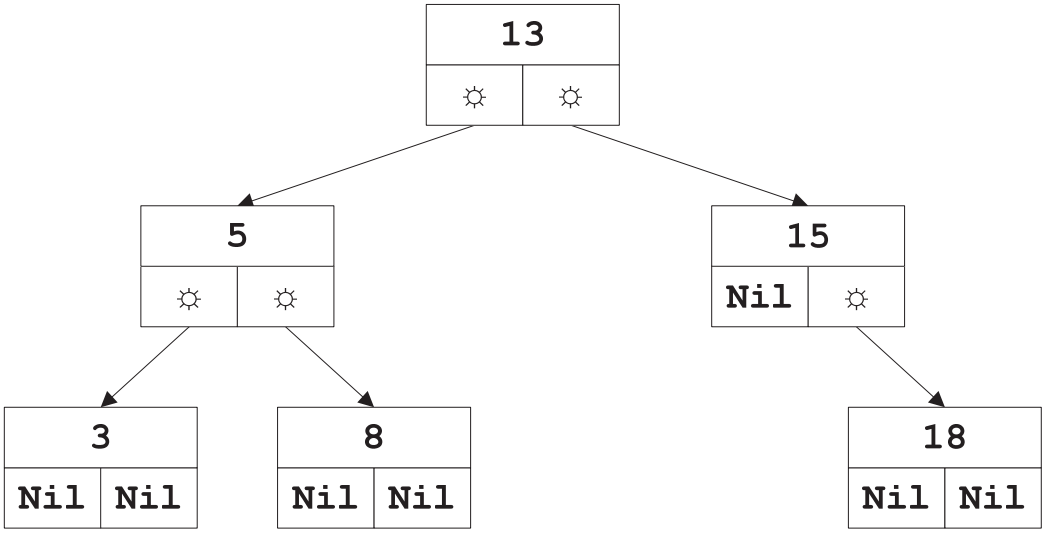
Заметим, что ситуация 3 является частным случаем следующей ситуации 4.

4. Пусть нужно удалить узел, имеющий много потомков (например, элемент «10» первоначального дерева).

Это можно сделать двумя способами: заменить (листом) узлом «8» (шаг влево и потом до конца вправо):



или узлом «13» (шаг вправо и потом до конца влево):



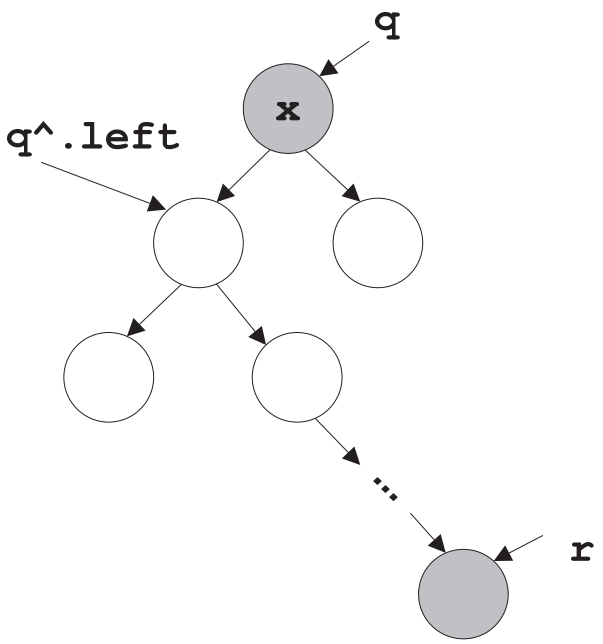
Задача. Постройте алгоритм для удаления узла с ключом, равным *x*, из дерева поиска.

Решение. Напишем процедуру, которая различает три случая:
1) узел с ключом *x* не найден;
2) узел с ключом *x* имеет одного наследника;
3) узел с ключом *x* имеет двух наследников.

Удаление узла с ключом, равным x, из дерева поиска

```
procedure Delete(x : item; var p : TRef);
var
     q : TRef;     {глобальная переменная }
procedure Del(var r : TRef);
begin
```

```
 if r^.rigth<>nil then Del(r^.rigth)
 else
   begin
    q^.key   := r^.key;
    q^.count := r^.count;
    q := r;       {ссылка на узел, который удаляется}
    r := r^.left;
   end;
end;

begin
 if p = nil then Writeln('элемент не нашли')
 else
    if x < p^.key then Delete(x, p^.left)
    else
      if x > p^.key then Delete(x, p^.rigth)
      else     {      x  =  p^.key}
         begin
          q := p;{ссылка на узел, который удаляется}
          if q^.rigth = nil then p := q^.left
          else
              if q^.left = nil then p := q^.rigth
              else Del(q^.left);
          Dispose(q);
         end;
end;
```



Задачи

1. Найдите сумму элементов бинарного дерева.

2. Имеется текстовый файл. Подсчитайте число появлений каждого слова и создайте новый текстовый файл, каждая строка которого имеет вид "<слово>– <число его появлений>". Слова должны располагаться в лексикографическом порядке.

3. Напишите функцию подсчета количества терминальных элементов бинарного дерева.

4. Напишите функцию подсчета количества листьев бинарного дерева.

5. В заданном бинарном дереве (дерево поиска, сбалансированное дерево) подсчитайте количество элементов, которые равны максимальному.

6. Определите количества элементов на *k*-м уровне дерева.

7. Выведите все листья дерева поиска в порядке возрастания.

8. Найдите максимальный элемент идеально сбалансированного бинарного дерева и количество повторений максимального элемента в данном дереве.

9. Напишите функцию, которая определяет, является ли бинарное дерево деревом поиска.

10. Найдите суммы элементов всех нечетных уровней.

11. Имеются два дерева поиска, содержащие целые числа. Постройте лексикографическое дерево поиска как объединение данных двух.

12. Найдите вершины, у которых количество потомков в левом поддереве не равно количеству потомков в правом поддереве.

13. Найдите вершины, для которых высота левого поддерева не равна высоте правого поддерева.

Разные задачи

Пусть имеют место описания

```
type
       TInf  = Integer;
       TRef  = ^TNode;
       TNode = record
                 key        : TInf;
                 left,right : TRef;
               end;
```

Задача 1. Напишите рекурсивную функцию копирования двоичного дерева.

Решение.

```
function TreeCopy(A:Tref):Tref;
var
   Tree : Tref;
begin
If A<>nil then
   begin
    New(Tree);
    Tree^.key   := A^.key;
    Tree^.left  := TreeCopy(A^.left);
    Tree^.right := TreeCopy(A^.right);
   end
else Tree := nil;
  TreeCopy := Tree;
end;
```

Задача 2. Напишите рекурсивную функцию определения высоты двоичного дерева.

Решение.

```
function TreeHeight(A:Tref):Integer;
var
   Lh, Rh : Integer;
begin
If A=nil then TreeHeight := 0
else
    begin
     Lh := TreeHeight(A^.left);
     Rh := TreeHeight(A^.right);
   If Lh > Th  then TreeHeight := Lh + 1
               else TreeHeight := Rh + 1;

     end;
end; {TreeHeight }
```

Тема 13
МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ
(12 часов)

13.1. Алгоритмы «разделяй и властвуй»

Возможно, самым важным и наиболее распространенным методом проектирования эффективных алгоритмов является метод декомпозиции (метод «разделяй и властвуй», или метод разбиения). Этот метод предусматривает такую декомпозицию (разбиение) задачи на более мелкие задачи, что на основе их решения можно легко получить решение первоначальной задачи. Этот метод хорошо подходит при программировании рекурсивных алгоритмов. Рассмотрим далее несколько практических задач и заметим, что таким способом можно решить множество других задач.

Перестановки чисел

Задача получения всех перестановок из заданных чисел является важной для большого количества комбинаторных задач. Рассмотрим ее решение методом «разделяй и властвуй».

Задача. Имеется *n* разных чисел. Напечатайте все возможные перестановки этих чисел.

Решение. Перестановки индексов элементов массива будем получать при помощи рекурсивной процедуры и затем печатать перестановки собственно элементов массива.

```
Program perestanowki;
uses Crt;
const  n = 4;
type
    Item   = integer;
    mas    = array [1..n]  of Byte;
    masitm = array [1..n]  of  item;
var
    i : integer;
    A : mas;
    C : masitm;

procedure  WriteMas (A : mas);
  var  i : Byte;
```

```
  begin
   for  i := 1  to n  do
          Write (C[A[i]]: 4);
   Write('    ');
  end;

procedure  Perestanowka (A : mas; m : Byte);
  procedure  Swap (m, j : Byte);
    var  B : item;
    begin
     B := A[m];  A[m] := A[j];  A[j] := B;
    end;
  var
    j : Byte;


 begin
   for  j :=  m  to  n  do
      begin
       if  j <> m  then
          begin
           Swap (m, j); WriteMas (A);
          end;
       Perestanowka (A, m + 1);
      end;
  end;

begin
 clrscr;
 for  i := 1  to  n  do
    begin
     A[i] := i;
     C[i] := i;           {Random(100)}
    end;
 Writeln;
 Writeln('Maciў A: ');
 WriteMas(A);
 Perestanowka(A, 1);
 Readln;
end.
```

Результат

Массив A:

1	2	3	4	1	2	4	3	1	3	2	4	1	3	4	2
1	4	2	3	1	4	3	2	2	1	3	4	2	1	4	3
2	3	1	4	2	3	4	1	2	4	1	3	2	4	3	1
3	1	2	4	3	1	4	2	3	2	1	4	3	2	4	1
3	4	1	2	3	4	2	1	4	1	2	3	4	1	3	2
4	2	1	3	4	2	3	1	4	3	1	2	4	3	2	1

Задачи

1. Имеется пять коробочек. На каждой из них написана одна из цифр: 1, 2, 3, 4, 5. Внутри коробочек лежит различное количество драгоценных камней: 1, 2, 3, 4, 5. Число, написанное на коробочке, не равно числу камней, которые лежат в ней, более того, сумма цифр, написанных на любых двух коробочках, не равна числу камней, которые в них лежат.

Указание. Можно воспользоваться одним из следующих алгоритмов.

Алгоритм 1. Решение можно получить используя перестановки чисел от 1 до 5 и выполняя далее проверку по условию задачи.

Алгоритм 2. Решение можно получить, если построить дерево вариантов и выполнить его обход.

Ответ:

2	3	4	5	1		3	4	5	1	2		4	5	1	2	3		4	5	2	3	1
5	1	2	3	4		5	3	4	1	2												

2. Если натуральное число имеет не менее трех разрядов, то фрагментом этого числа назовем любое число, образованное тремя подряд идущими его цифрами. Например, фрагментами чисел 14314377 будут ровно шесть чисел: 143, 431, 314, 143, 437 и 377, – из которых пять разные. Из цифр 1, 2, 3 и 4 (можно использовать каждую цифру по несколько раз) записали число, такое, что в любом его фрагменте все цифры разные, и любые два его фрагмента различны (различные как числа).

Какое наибольшее количество разрядов может иметь такое число? Напечатайте все такие числа.

3. Среди цифр *x*, *y*, *z*, *t*… поставьте знаки арифметических операций +, -, *, /, чтобы получить заданное значение.

13.2. «Жадные» алгоритмы

На каждом отдельном этапе любой «жадный» алгоритм выбирает тот вариант, который является *локально оптимальным* в том или ином смысле.

Когда единственным способом получения оптимального решения некоторой задачи является использование метода полного перебора, но для этого тре-

буется слишком много времени, тогда «жадный» алгоритм, или другой эвристический метод получения приемлемого (не обязательно оптимального) решения, может оказаться единственным реальным способом его получения.

Однако не всякий «жадный» алгоритм допускает оптимальное решение в целом, так как на каждом этапе гарантируется мгновенная выгода, но при этом общий результат может оказаться неприемлемым.

Если же нас удовлетворяет «почти оптимальный» результат, то «жадные» алгоритмы часто оказываются самыми быстрыми методами получения решения.

Рассмотрим эти подходы при решении различных задач.

Счастливые билеты

Задача. Талон для проезда в городском транспорте имеет шестизначный номер от 000000 до 999999.

Будем считать талон счастливым, если сумма первых трех цифр равна сумме последних трех.

Подсчитайте количество счастливых номеров среди шестизначных номеров.

Алгоритм 1. Эвристический подход приводит к следующему:

- перебираем все номера от 000000 до 999999;
- выделяем цифры в каждом числе;
- проверяем, является ли билет счастливым.

Алгоритм 2. Во втором варианте формируем сразу все шесть цифр числа, так как в предыдущем варианте какое-то время тратится на выделение цифр.

Алгоритм 3. Будем считать, сколько раз выйдет сумма цифр, равная 0, 1, 2, …, 3 * 9 в одной половине и во второй.

Очевидно, что количество «счастливых» билетов с определенной суммой первых трех цифр равно $a[sc]^2$.

Найдем сумму первых трех цифр и увеличим соответствующий элемент массива на единицу.

В этом варианте сэкономлено время (примерно в 20 раз), но незначительно увеличено использование памяти. Существуют и другие способы усовершенствования этого подхода, которые оптимизируют циклы, но они нас не очень интересуют, так как рассчитаны на шестизначные номера билетов.

И это существенный недостаток рассмотренных алгоритмов.

Если необходимо совершить подсчет для номеров, разрядность которых будет варьироваться или может стать известной, например, только во время выполнения программы, то необходимо иначе запрограммировать наши вложенные циклы.

Алгоритм 4. Рассмотрим следующий неожиданный подход, который можно применять в процессе выполнения программы, там, где необходимо

моделировать десятичные алгоритмы, и там, где нужно подсчитать значение вложенных сумм в не заданном наперед количестве.

Есть механизм, который называется «одометр». Проимитируйте работу одометра, например четырехразрядного. Заметили, что вы имеете дело со схемой работы вложенных циклов? Применим этот принцип и улучшим предыдущую программу.

```
Program Happy_ticket_4;

const
      k = 4;     {2 * k разрядное число}

var
     i          : Byte;
     Sum_Digit  : Byte;
     Count      : Longint;
     a          : array[0..9 * k]  of  Longint;
     Cp         : array[0..k] of  Byte;

begin
 for i := 0 to k * 9 do a[i] := 0;
    {               количество возможных сумм цифр}
 for i := 0 to k  do Cp[i] := 0; {Обнулили одометр}
   {одометр переполняется при Cp[0]=1. Конец вычислений}
 repeat
      Sum_Digit := 0;
      for  i := 1  to  k  do
           Sum_Digit := Sum_Digit + Cp[i];
           {    подсчитали текущую сумму цифр
                                 на одометре}
      Inc(a[Sum_Digit]);
      i := k;
           {         переходим на следующую
                  комбинацию цифр одометра}
       while  Cp[i] = 9  do
              begin
               Cp[i] := 0;
               Dec(i);
              end;
      {Сбросили все цифры 9 в конце одометра
             и прибавили 1 в нужной позиции}
       Cp[i] := Cp[i] + 1;
 until  Cp[0] = 1;
 Count := 0;
```

```
 for  i := 0  to  k * 9    do
      count := count + a[i] * a[i];
 Writeln('количество = ', Count);
 Readln;
end.
```

Задача получения сдачи эвристическим методом

Рассмотрим упрощенную задачу выдачи сдачи монетами следующего номинала (копеек): 25, 10, 5 и 1. Пусть сдача составляет 63 копейки. Тогда можно сдачу выдать следующим образом:

$$63 = 2 \cdot 25 + 1 \cdot 10 + 3 \cdot 1.$$

Количество монет: $2 + 1 + 3 = 6$.

Оказывается, что «жадные» алгоритмы не всегда дают оптимальный вариант.

Пример. Рассмотрим случай, где монеты номинала 11, 5 и 1, остаток – 15 копеек.

«Жадный» алгоритм даст результат: $15 = 1 \cdot 11 + 4 \cdot 1$.

Количество монет – 5.

Но очевидно, что остаток в 15 копеек таким номиналом монет можно отсчитать и так: $15 = 3 \cdot 5$. Количество монет – 3.

Значит, мало того, что «жадный» алгоритм в целом не всегда обеспечивает оптимальное решение, но бывают ситуации, когда он вообще не дает решения. Например, когда ограничено количество монет определенного номинала.

Задачи

1. Посчитайте процент счастливых номеров среди $2k$-значных номеров талонов. Будем считать талон счастливым, если сумма первой половины цифр равна сумме последней (использовать принцип работы одометра).

2. Пусть количество вложенных сумм задается во время выполнения программы. Напишите программу, которая подсчитывает сумму

$$S = \sum_{i_k=1}^{n} \cdots \sum_{i_1=1}^{n} \frac{1}{i_1 + \cdots i_k}.$$

3. Пусть количество вложенных сумм задается во время выполнения программы. Напишите программу, которая подсчитывает сумму

$$S = \sum_{i_k=a_k}^{b_k} \cdots \sum_{i_1=a_1}^{b_1} u(i_1, \ldots, i_k)$$

для заданных чисел $a_1, ..., a_k, b_1, ..., b_k$.

13.3. Алгоритмы на полный перебор

Нужно заметить, что при каждом добавлении к выплате сдачи очередной монеты возникает такая же задача, но с уменьшенной суммой сдачи и с другим набором монет $q_i,\ i=\overline{1,n}.$

Определим алгоритм задачи получения остатка как булевскую функцию «остаток возможен» следующим образом:

$$\begin{aligned}
RM(x;\ q_1,q_2,q_3,q_4,q_5,q_6,q_7,q_8) := (x=0 \vee \\
\vee (x\ge 50 \wedge q_1\ge 1 \wedge RM(x-50;\ q_1-1,q_2,q_3,q_4,q_5,q_6,q_7,q_8)) \\
\vee (x\ge 20 \wedge q_2\ge 1 \wedge RM(x-20;\ q_1,q_2-1,q_3,q_4,q_5,q_6,q_7,q_8)) \\
\vee (x\ge 15 \wedge q_3\ge 1 \wedge RM(x-15;\ q_1,q_2,q_3-1,q_4,q_5,q_6,q_7,q_8)) \\
\vee (x\ge 10 \wedge q_4\ge 1 \wedge RM(x-10;\ q_1,q_2,q_3,q_4-1,q_5,q_6,q_7,q_8)) \\
\vee (x\ge 5\ \wedge q_5\ge 1 \wedge RM(x-5;\ \ q_1,q_2,q_3,q_4,q_5-1,q_6,q_7,q_8)) \\
\vee (x\ge 3\ \wedge q_6\ge 1 \wedge RM(x-3;\ \ q_1,q_2,q_3,q_4,q_5,q_6-1,q_7,q_8)) \\
\vee (x\ge 2\ \wedge q_7\ge 1 \wedge RM(x-2;\ \ q_1,q_2,q_3,q_4,q_5,q_6,q_7-1,q_8)) \\
\vee (x\ge 1\ \wedge q_8\ge 1 \wedge RM(x-1;\ \ q_1,q_2,q_3,q_4,q_5,q_6,q_7,q_8-1))).
\end{aligned}$$

Если в некоторый момент $RM(y;q_1,q_2,q_3,q_4,q_5,q_6,q_7,q_8)$ будет иметь $y=0$, тогда решение $RM(\ldots)$ будет *true* (поскольку так работает $a\vee b$), иначе будет *false*.

В таком рекурсивном варианте, если решение работы функции *false*, это лишь означает, что на данном наборе $q_i,\ i=\overline{1,n},$ остаток получить невозможно.

Задачи

1. Используя алгоритм получения сдачи полным перебором при некотором заданном количестве монет, постройте список возможных вариантов. Найдите варианты выплаты сдачи наименьшим количеством монет.

2. Учитель математики написал на доске трехзначное число, вызвал Васю к доске и предложил ему записать несколько делителей этого числа. Вася записал следующие числа: 7, 10, 27, 35, 45, 63, на что учитель ответил: «Среди шести записанных чисел четыре действительно являются делителями написанного мною числа, а два – нет». Какое число написал на доске учитель?

3. Существует *n* предметов, масса которых и стоимость известны. Определите, какие предметы нужно положить в рюкзак, чтобы общая масса не превышала заданной границы, а общая стоимость была наибольшей.

4. Разработайте алгоритм игры в «крестики-нолики».

13.4. Поиск с возвратом. Задачи искусственного интеллекта

Особенно интересный раздел программирования – задачи из области «искусственного интеллекта».

Здесь нужно строить алгоритмы, которые находят решение не по фиксированным правилам, а методом проб и ошибок.

Процесс проб и ошибок можно рассматривать в общем виде как поисковый процесс, который постепенно строит и просматривает (а также обрезает) деревья подзадач.

Во многих случаях такие деревья поиска растут очень быстро, обычно экспоненциально, в зависимости от заданного параметра.

Соответственно увеличивается стоимость поиска. Часто деревья поиска можно обрезать, используя только эвристические соображения, и тем самым сводить к количеству подсчетов в разумных пределах.

Рассмотрим общий принцип разбивки таких задач на подзадачи и использования в них рекурсии.

Задача обхода конем шахматной доски

Задана доска, $n \times n$, которая содержит n^2 полей. Конь, который ходит соответственно шахматным правилам, помещается на поле с заданными начальными координатами. Нужно покрыть всю доску ходами коня, т. е. найти обход доски, если он существует, за $n^2 - 1$ ходов такой, что в каждое поле конь заходит ровно один раз.

Очевидно, что эта задача разбивается на две более простые задачи:

- выполнить очередной ход;
- установить, что никакой ход невозможен.

Здесь программа:

```
Program khightstour;
uses  Crt;
const   n = 8;   nsqr = n * n;
type
    Ta = array[1..8] of shortint;
    Index = 1..n;
const
      a : Ta = (-1,-2,-2,-1, 1, 2, 2, 1);
      b : Ta = (2, 1,-1,-2,-2,-1, 1, 2);
var
   i, j : Index;
   q    : Boolean;
   S    : set of Index;
```

```
   H    : array[Index, Index] of Byte;
   C    : Longint;

procedure Try(i:Integer;x,y:Index;var q:Boolean);
    var
        k, u, v : Integer;
        q1      : Boolean;
 begin
  k := 0;
  repeat
  Inc(k);
  q1 := false;
  u  := x + a[k];
  v  := y + b[k];
  if (u in S) and (v in S)  then
    if H[u,v] = 0  then
      begin
       H[u,v] := i;
       Inc(C);
       if  i < nsqr then
          begin
           Try(i + 1, u, v, q1);
           if not q1 then H[u,v] := 0
          end
        else  q1 := true
       end
  until  q1  or (k = 8);
  q := q1;
  end;     {Try}
begin
 TextBackGround(15);
 TextColor(0);
 ClrScr;
 C := 0;
 S := [];
 for  i := 1  to  n  do  S  := S + [i];
 for  i := 1  to  n do
    for  j := 1  to  n  do   H[i,j] := 0;
 H[1,1] := 1;
 Try(2, 1, 1, q);
 Writeln('Число проб: ', C:16);
 if  q  then
    for  i := 1  to  n do
        begin
```

```
          for  j := 1  to  n  do
               Write (H[i,j]:5);
          Writeln
         end
 else
    Writeln (' not equation ');
 Readln;
end.
```

Результат

```
Число проб:         27241112
    1   10   23   64    7    4   13   18
   24   63    8    3   12   17    6   15
    9    2   11   22    5   14   19   32
   62   25   40   43   20   31   16   51
   39   44   21   58   41   50   33   30
   26   61   42   47   36   29   52   55
   45   38   59   28   57   54   49   34
   60   27   46   37   48   35   56   53
```

Задание. В графическом режиме на шахматной доске совершите движение коня по стратегии, когда ход коня выбирается не детерминированно, а случайно (из восьми позиций).

Задачи

1. Задача обхода конем шахматной доски. Подсчитайте количество попыток и определите оптимальную стратегию.
2. Задача о восьми ферзях.
3. Задача о восьми ладьях.
4. Задача про устойчивые браки.

13.5. Поиск с возвратом и локальный поиск

Существует целый класс задач, которые попадают в категорию теоретически разрешимых, но практически неосуществимых. Нет особого различия между программой, которая не заканчивает своей работы никогда, и программой, которая работает годами. Этот класс – класс переборных задач. Приведем пример такой задачи.

Задача о «рюкзаке». Существует набор грузов известного веса и машина известной грузоподъемности. Требуется определить, можно ли полностью загрузить машину, поместив в нее некоторые из грузов.

Теоретическое решение этой задачи очевидно. Если у нас есть n грузов, то возможных комбинаций этих грузов будет 2^n. Перебрав все комбинации, можно дать положительный или отрицательный ответ. Оценим количество операций, необходимых для выполнения этого алгоритма. Например, когда грузов $n = 100$, то $2^n = 2^{100} = (2^{10})^{10} = 1024^{10} \approx 1000^{10} = 10^{30}$, требуется перебрать примерно 10^{30} вариантов. Сколько же времени будет работать машина, выполняя этот полный перебор? Пусть мы имеем компьютер, который выполняет миллиард (10^9) операций в секунду. На проверку одного набора идет никак не меньше одной операции. Значит, требуется не менее $2^{30} / 10^9 = 10^{21}$ секунд > > 10^{13} лет (!). Очевидно, что с любой практической точки зрения этот алгоритм непригоден.

Единственный выход – это всеми возможными способами уменьшить количество вариантов перебора.

Следует отметить, что существует ряд задач, которые называются полными и требуют выполнения перебора всех возможных ситуаций.

Приведем примеры полных задач.

Задача о «раскраске графа». Задан граф – набор из нескольких вершин, некоторые из которых соединены линиями (дугами), и число h. Требуется определить, можно ли так раскрасить вершины графа в h цветов, чтобы любые две вершины, соединенные одной дугой, были разукрашены в разные цвета.

Задача о «разбиении». Задано множество S и некоторое семейство его подмножеств. Требуется узнать, можно ли из этого семейства подмножеств выбрать несколько множеств, которые попарно не пересекаются и объединение которых есть S.

Для этих трех задач показано, что если одна из них может быть быстро решена, то и другая – тоже. Но дальше показано, что быстрого алгоритма не существует и нужно перебирать все варианты. Значит, в решении задач такого рода надо уметь отвергать так называемые тупиковые варианты.

Имея задачу, сначала нужно решить, к какому классу она принадлежит – полных задач или нет. Далее нужно или искать быстрый алгоритм, или выполнять полный перебор с улучшением, который называется *поиск с возвратом*.

Те задачи, к которым подходит идея «разделяй и властвуй», сводят задачу к нескольким подзадачам того же типа, но меньшего размера – имеют быстрые алгоритмы. Обычно это рекурсивный подход. Однако существуют и нерекурсивные алгоритмы с возвратом.

Поиск с возвратом

Остановимся более подробно на задачах перебора, при решении которых можно реализовать нерекурсивные алгоритмы с возвратом.

Рассмотрим некоторый массив a_1, ..., a_n. Он обладает 2^n подмассивами: пустым, массивами по одному элементу, по два элемента и, вообще, подмассивами a_i, a_j, ..., a_p.

Задача. В заданном массиве натуральных чисел a_1, ..., a_n нужно выбрать подмассив a_i, a_j, ..., a_p такой, что $a_i + a_j + ... + a_p = z$, где z – заданное число.

Данная задача принадлежит к полным, поэтому решается перебором всех возможных ситуаций. При ее решении возникнет ряд подзадач, которые понадобятся в дальнейшем.

Надо научиться фиксировать определенные подмассивы и потом проверять их элементы на удовлетворение поставленному условию.

Во многих случаях нужно решать задачу выбором из заданного массива одного или всех подмассивов, которые удовлетворяют определенному условию.

Для этого заведем вспомогательный массив b_1, ..., b_n из нулей и единиц, в котором будем хранить такую информацию: если $b_i = 0$, тогда a_i не включается в подмассив, и если $b_j = 1$, тогда a_j включается в подмассив.

Например, при $n = 6$ массив $b = (0, 1, 0, 1, 1, 0)$ зафиксирует подмассив a_2, a_4, a_5, массив $b = (0, 0, 0, 0, 0, 0)$ – пустой подмассив, а массив $b = (1, 1, 1, 1, 1, 1)$ – исходный массив.

Поэтому далее будем работать не с условием $a_i + a_j + ... + a_p = z$, а с условием $z = \sum_{i=1}^{n} b_i a_i$.

Рассмотрим, как можно перебрать все 2^n подмассивов для решения поставленной задачи. Для этого будем строить массив b_1, ..., b_n из нулей и единиц в порядке, который называется словарным.

Если рассматривать массивы как слова в алфавите из двух цифр 0 и 1, считая, что 0 предшествует 1, то именно в таком порядке массивы попадут в словарь.

При $n = 3$ получим подмассивы:

(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1).

Продумали, как строить подмножество. Теперь рассмотрим, как сократить перебор.

Чтобы определить n-элементный подмассив b (из нулей и единиц), который фиксирует текущий подмассив массива a_1, ..., a_n, надо о каждом элементе $a_i (1 \le i \le n)$ решить, принимается ли он в подмассив. Может возникнуть следующая ситуация: относительно элементов a_1, ..., a_k, $(k < n)$ приняты некоторые решения, но после этого оказалось, что как бы мы не распоряжались остальными элементами, не удастся получить подмассив, который даст решение поставленной задачи. Это так называемые *тупиковые* ситуации.

В таком случае можно исключить из рассмотрения все подмассивы, первые элементы которых выбраны из числа a_1, ..., a_k $(k < n)$ в соответствии с принятыми ранее относительно их решениями.

Например, в условиях задачи известно, что все a_1, ..., a_n – положительные числа. Тогда, если зафиксированы начальные элементы a_i, a_j, ..., a_r этого массива и их сумма уже больше z, то попытка подобрать к ним еще несколько составляющих из оставшихся элементов a_{r+1}, ..., a_n будет бессмысленной (ведь от этого сумма только увеличится). Это приводит к необходимости распознавать тупиковые подмассивы в массиве b и отбрасывать их без дальнейшего просмотра.

Для этого рассмотрим более подробно построение вспомогательного массива b. Поскольку используется словарный метод построения, то нужно в нем хранить и более короткие слова (как в обычном словаре, где встречаются и слова на одну, две и более букв). Тогда легче будет отказаться от «удлинения» подмассива a_1, ..., a_k ($k < n$), когда обнаружена тупиковая ситуация.

Будем считать, что нам известен алгоритм `Impasse`, позволяющий распознавать тупиковые ситуации и возвращать значение $t = 1$, если подмассив a_1, ..., a_k ($k < n$) тупиковый, $t = 0$, если нет. Используем алгоритм `Impasse` для как можно большего уменьшения вариантов перебора.

Проанализируем задачу построения вспомогательного массива b.

В общем случае массив b будет содержать все возможные как односимвольные, так и двух-, трех- и т. д. символьные «слова». Короткие массивы важны, поскольку к ним как раз и нужно применять алгоритм распознавания тупиков. Для $n = 3$ получим следующую последовательность слов:

0; 0, 0; 0, 0, 0; 0, 0, 1; 0, 1; 0, 1, 0; 0, 1, 1; 1; 1, 0; 1, 0, 0; 1, 0, 1; 1, 1; 1, 1, 0; 1, 1, 1.

Анализируя данную последовательность, получим следующий алгоритм построения вспомогательного массива b:

1) $b_1 = 0$, $k := 1$;

2) если массив b_1, ..., b_k не тупиковый, тогда продлеваем его, дописывая справа нулевые элементы;

3) если зашли в тупик или получили массив длины n, тогда надо рассматривать массив, который:

а) не является удлинением рассматриваемого массива;

б) расположен в словаре дальше, чем рассматриваемый;

в) из всех массивов, удовлетворяющих а), б), встречается в словаре первым.

Такой массив, когда он существует, может быть построен из b_1, ..., b_k просмотром элементов в обратном порядке, т. е. в порядке b_k, b_{k-1}, ... до первого элемента $b_m = 0$; тогда берем $b_m = 1$, $k = m$. После этого группа элементов b_1, ..., b_k удовлетворяет всем условиям.

Если получим массив $b_1 = b_2 = ... = b_n = 1$, тогда перебор закончен.

Задачи

1. **Задача о «рюкзаке».** Существует набор грузов известного веса и стоимости за условную единицу веса (грамм, килограмм, штука и т. п.), а также машина известной грузоподъемности. Определите, можно ли полностью загрузить машину, поместив в нее некоторые из грузов. Найдите вариант загрузки с максимальной общей ценой.

2. Существует *n* предметов, масса которых известна. Разделите эти предметы на две группы так, чтобы общая масса первой группы была максимально близкой к общей массе второй группы.

13.6. Фракталы

В настоящее время есть несколько определений термина «фрактал»:

1) фрактал – это геометрическая фигура, в которой один и тот же фрагмент повторяется при каждом уменьшении масштаба. Это конструктивные фракталы, получаемые рекурсивной процедурой, объединяющей сжимающие отображения подобия;

2) фракталом называется структура, состоящая из частей, в каком-то смысле подобных целому;

3) фрактал – это такое множество с фрактальной размерностью, превышающей топологическую.

В первом определении слово «фрактал» происходит от латинского *fractus*, означающее «изломанный». Во втором и третьем определении оно связано с английским *fractional* – «дробный».

Первые фракталы появились в математике в конце XIX начале XX в. К их числу относятся такие удивительные конструкции, как канторовское множество (Г. Кантор, 1883), или, например, заметающая целый квадрат, кривая Гильберта-Пеано (Д. Гильберт – Дж. Пеано, 1890), множества Серпинского (В. Серпинский, 1916) и другие типичные фракталы (термин «фрактал» в то время еще не был введен). Эти конструкции были в свое время открыты математиками для того, чтобы показать, насколько наивными и хрупкими могут быть наши представления о столь знакомых, казалось, объектах, как функция и кривая.

Существуют разные *классификации фракталов*. По одной из них фракталы делятся на группы:

- алгебраические;
- геометрические;
- системы итерируемых функций;
- стохастические.

По другой классификации фракталы делятся на:

- детерминированные;
- алгебраические,
- геометрические;
- недетерминированные:
- стохастические.

Так же фракталы делят на:

- динамические:
- алгебраические;
- конструктивные:
  - системы итерируемых функций,
  - геометрические,
  - стохастические.

Задания

Множество Мандельброта

Задание 1. Выполните эксперименты на компьютере, изменяя значения параметров `P_min, P_max, Q_min, Q_max.`

Решение. Итерационный процесс

$$z_{k+1} = z_k^2 + c,\ k = 0,\ 1,\ 2,\ \ldots,\ \ z_0 = c,$$

при различных значениях комплексных чисел c можно выполнять как с комплексной арифметикой, так и с арифметикой для действительных чисел (x_k, y_k), т. е. представляя $z_k = x_k + iy_k,\ c = c_x + ic_y$, легко получить

$$x_{k+1} = x_k^2 - y_k^2 + c_x,\ y_{k+1} = 2x_k y_k + c_y,\ k = 0,\ 1,\ 2, \ldots,\ x_0 = c_x,\ y_0 = c_y.$$

Множество Мандельброта получается так: выбирается фиксированная точка (x, y) и просчитывается ее путь при разных значениях параметров (c_x, c_y). Результаты наносятся, точка за точкой, на плоскости (c_x, c_y). Таким образом получается рисунок типа множества Мандельброта.

Алгоритм.

1. Задаются отрезки изменения действительной части c_x и мнимой части c_y комплексной переменной c.
2. На экране формируется прямоугольник, параметры которого определяются разрешением экрана.
3. Подсчитывается шаг изменения переменных c_x и c_y.
4. Задается максимальное количество итераций $k_{\max}$ и радиус круга сходимости $r_{\min}$ ($r_{\min} = 2$).

5. Образуется цикл на количество пикселей по оси Ox $(i = \overline{1, n})$.

6. Образуется цикл на количество пикселей по оси Oy $(j = \overline{1, m})$.

7. Каждому пикселю экрана сопоставляется текущая точка комплексной плоскости C (фактически по оси Ox задается значение действительной части c_x, по оси Oy – мнимой части c_y).

8. Запускается итерационный процесс $z_{k+1} = z_k^2 + c,\ k = 0, 1, 2, \ldots,\ z_0 = c.$

9. Если за максимальное число итераций значение $|z_k| \le r_{\min}$, то заданная точка $c(c_x, c_y)$ принадлежит множеству Мандельброта. На экране такие точки обозначаются черным цветом. Если же на некотором шаге k итерации значение $|z_k| > r_{\min}$, то считается, что значение идет к бесконечности и цвет такого пикселя выбирается по формуле $k \bmod 256$ (или $k \bmod 16$) в зависимости от количества цветов палитры.

10. Конец цикла по оси Oy.

11. Конец цикла по оси Ox.

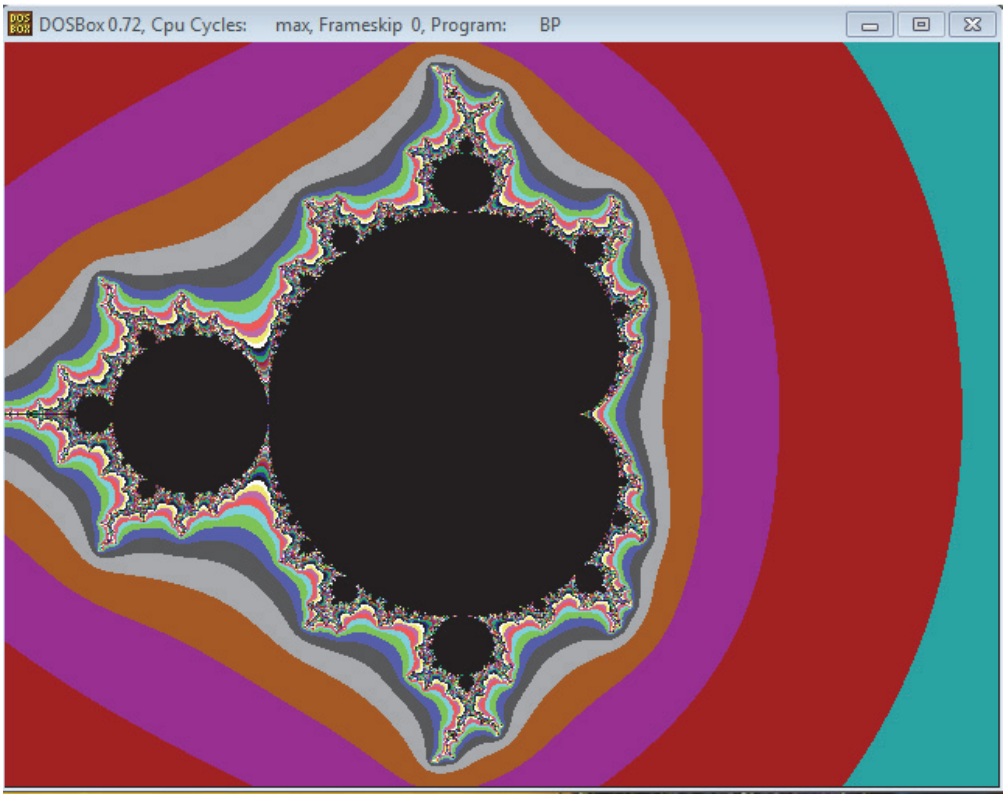
Время подсчетов можно уменьшить в 2 раза благодаря симметрии процесса. Точки (x, y) и $(-x, -y)$ имеют одну судьбу.

Ниже приведен код программы построения классического множества Мандельброта с использованием комплексной арифметики.

```
Program Maldelbrot_1;
uses Graph, Crt;
type
     Complex = record
                Re, Im : Real;
               end;
const
     MaxIter = 100;
     BailOut = 16;
var
   W1, W2, Backup      : Complex;
   X, Y, Count, dr, dm : Integer;
   Max_X, Max_Y, Color : Integer;

procedure Sqr_C(a : Complex; var r : Complex);
begin
with a  do
    begin
     r.re := sqr(re) - sqr(im);
     r.im := 2 * re * im;
    end;
```

```
end;

procedure Add_C(a,b : Complex; var r : Complex);
begin
with r  do
    begin
     re := a.re + b.re;
     im := a.im + b.im;
    end;
end;

procedure Init_C(a, b  : Real; var r : Complex);
begin
with r  do
    begin
     re := a;
     im := b;
    end;
end;

begin
 dr:=Detect;
 InitGraph(dr,dm, '');
 Max_X := GetMaxX  div  2;
 Max_Y := GetMaxY  div  2;
 for Y := - Max_Y  to  0  do
    for X := -Max_X   to  Max_X    do
       begin
        Count := 0;
        Init_C(X / 200, Y / 200, Backup);
        Init_C(0, 0, W1);
        while (Sqr(W1.re)+Sqr(W1.im)<BailOut) and
                    (Count < MaxIter)  do
              begin
               W2 := W1;
               Sqr_C(W2, W1);
               Add_C(W1, Backup, W2);
               W1 :=W2;
               Inc(Count);
              end;
        if count<>MaxIter then Color:=Count mod 256)
```

```
          else Color :=0;
              PutPixel(Max_X+X, Max_Y+Y,Color);
              PutPixel(Max_X+X,Max_Y+Abs(Y),Color);
         end;
  Readln;
end.
```

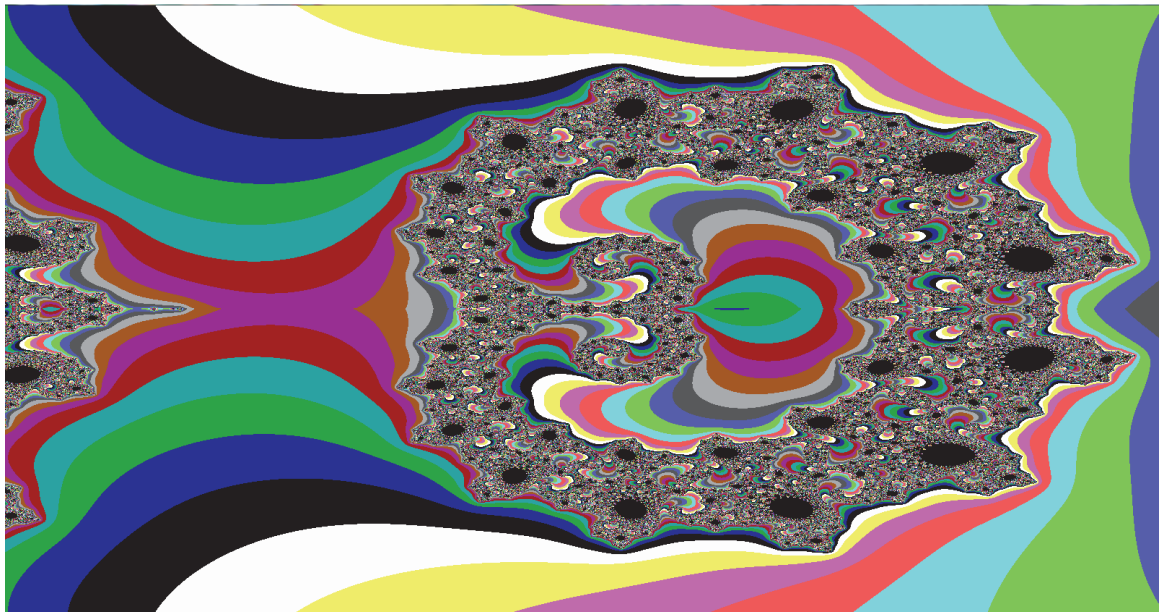


Задание 2. Постройте множества Мандельброта с использованием вещественной арифметики.

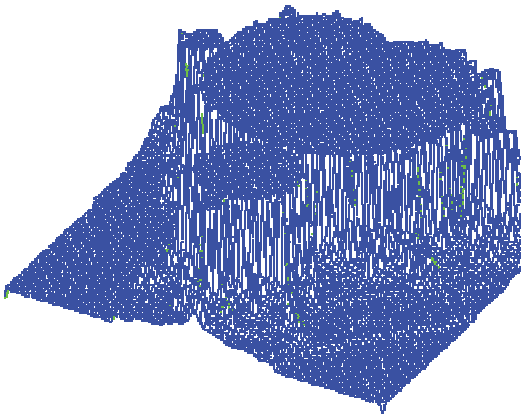
```
Program Maldelbrot_2;
uses  Graph, Crt;
var
   gd, gm : Integer;

function f(x, y, p : Real) : Real;
  begin
   f := x * x – y * y + p;
  end;

function g(x, y, q : Real) : Real;
  begin
   g := 2 * x * y + q;
  end;
```

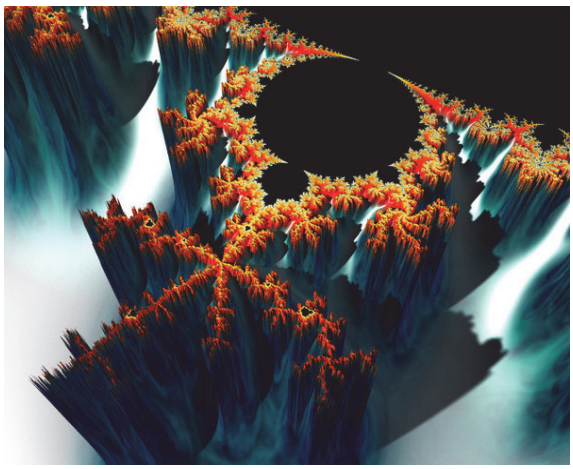
```
procedure Mandelbrot(P_min, P_max, Q_min, Q_max,
                     L : Real; MaxIter : Integer);
var
     Max_x, Max_y   : Integer;
     r, t, xk, yk   : Real;
     p, q, dp, dq   : Real;
     color, i, j, k : Integer;
     flag           : Boolean;
begin
 gd := detect;
 InitGraph(gd, gm, '');
 max_x := GetMaxX;
 max_y := GetMaxY;
 dp := (p_max - p_min) / max_x;
 dq := (q_max - q_min) / max_y;
 p  := p_min;
for i := 0  to  max_x  do
   begin
    q := q_min;
    for j := 0  to  max_y div 2  do
       begin
        k    := 0;
        flag := true;
        xk   := p;
        yk   := q;
        while flag  and  (k<=maxiter)  do
             begin
              Inc(k);
              t    := xk;
              xk   := f(xk, yk, p);
              yk   := g(t , yk, q);
              r    := xk * xk + yk * yk;
              flag := r < L;
             end;
         if  flag  then color := 0
                   else  color := k mod 256;
         PutPixel(i, j, color);
         PutPixel(i, max_y - j, color);
         q := q + dq;
        end;
    p := p + dp;
   end;
end;
```

```
begin
 Mandelbrot(-0.74591,      -0.74448,      0.11196,      0.11339,
              225, 255);
 Readln;
end.
```



Задание 3. Для каждой точки экрана $c_{ij} \in C$, $i = \overline{1,n}$, $j = \overline{1,m}$, можно запустить итерационный процесс, сформировать матрицу $M = (m_{ij})$, элемент $m_{ij} \in [1, k_{\max}]$, равный номеру итерации k, на которой процесс остановлен. Полагая числа m_{ij} вершинами некоторой поверхности в точках c_{ij}, можно получить объемный образ множества Мандельброта или его части, которая при специально подобранном освещении может выглядеть и как скала с плоской вершиной, и как водопад, и как горная вершина. Примеры приведены ниже.





Задание 4. Фрактальные множества можно получить для итерационных процедур общего вида $z_{k+1} = F(z_k, c),\ k = 0, 1, 2, \ldots$, которые порождают разные рисунки. Например, если взять $F(z_k, c) = z_k^n + c$ с целым числом n, то получим симметричное фрактальное множество, объединяющее $n - 1$ множество Мандельброта. При $n = 7$ получается почти настоящая шестилучевая снежинка.

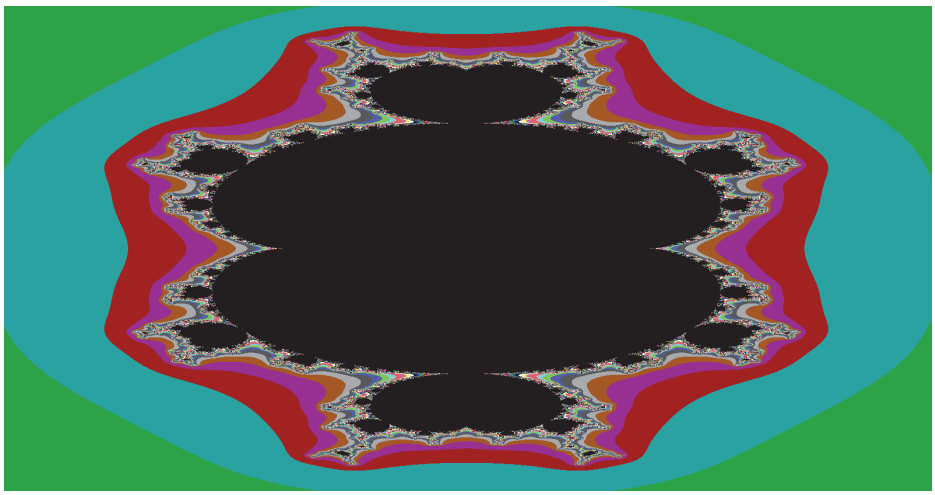
Рассмотрим сначала $n = 3$, $c = p + iq$.

Получим итерационный процесс:

$$\begin{cases} x_{n+1} = x_n^3 - 3x_n y_n^2 + p, \\ y_{n+1} = 3x_n^2 y_n - y_n^3 + q. \end{cases}$$

```
Program Raznoe_3;
uses  Graph, Crt;
var
   gd, gm : Integer;

function f(x, y, p : Real) : Real;
  begin
   f := x * x * x - 3 * x * y * y + p;
  end;

function g(x, y, q : Real) : Real;
  begin
   g := 3 * x * x * y - y  * y * y + q;
  end;

procedure Raznoe(P_min, P_max, Q_min,  Q_max,
                 L : Real; MaxIter : Integer);
var
     Max_x, Max_y   : Integer;
     r, t, xk, yk   : Real;
     p, q, dp, dq   : Real;
     color, i, j, k : Integer;
     flag           : Boolean;
begin
 gd := detect;
 InitGraph(gd, gm, '');
 max_x := GetMaxX;
 max_y := GetMaxY;
 dp := (p_max - p_min) / max_x;
 dq := (q_max - q_min) / max_y;
```

```
 p  := p_min;
for i := 0  to  max_x  do
   begin
    q := q_min;
    for j := 0  to  max_y div 2  do
       begin
        k    := 0;
        flag := true;
        xk   := p;
        yk   := q;
        while flag  and  (k <= maxiter)  do
             begin Inc(k);
              t    := xk;
              xk   := f(xk, yk, p);
              yk   := g(t , yk, q);
              r    := xk * xk + yk * yk;
              flag := r < L;
             end;
        if  flag  then  color := 0
                  else  color := k mod 16;
        PutPixel(i, j, color);
        PutPixel(i, max_y - j, color);
        q := q + dq;
       end;
    p := p + dp;
   end;
end;

begin
 Raznoe(-1.0, 1.0, -1.5, 1.5, 96, 100);
 Readln;
end.
```



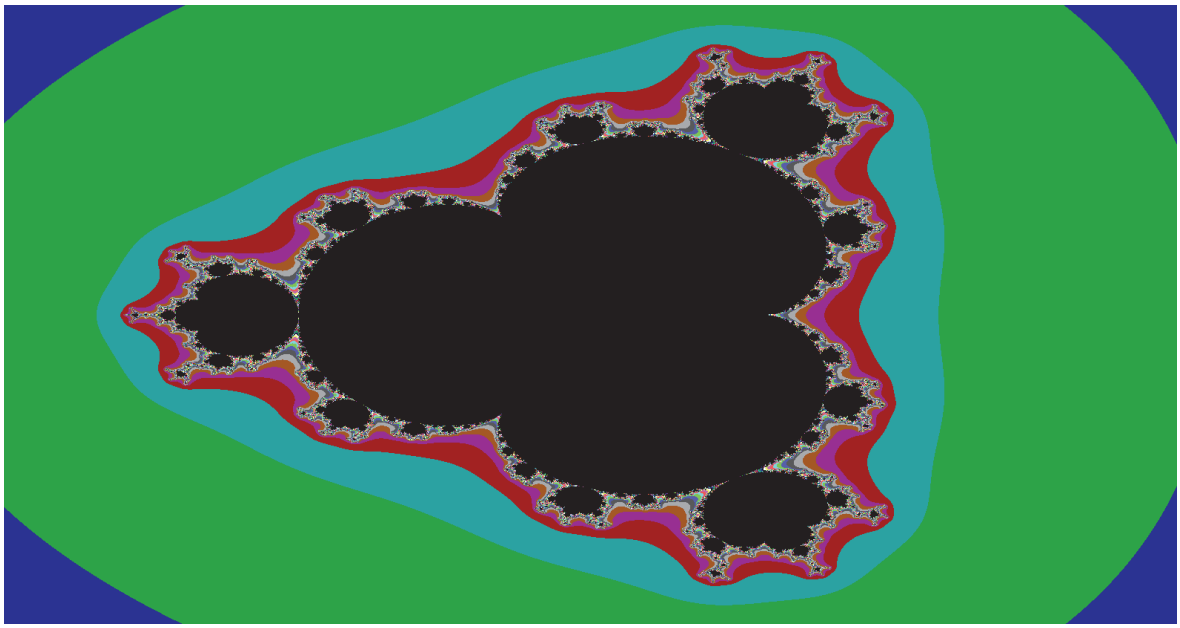
Задание 5. Для $n = 4$, изменяя в предыдущей программе функции

```
function f(x, y, p : Real) : Real;
  begin
   f := x*x*x*x - 6 *x*x*y*y + y*y*y*y + p;
  end;

function g(x, y, q : Real) : Real;
  begin
   g := -4*x*y*y*y + 4 *x*x*x*y + q;
  end;
```

и вызов процедуры

```
Raznoe(-1.6, 1.6, -1.3, 1.3, 96, 100);
```

Получим следующий рисунок.



Задание 6. Для $n = 7$, изменяя в предыдущей программе функции

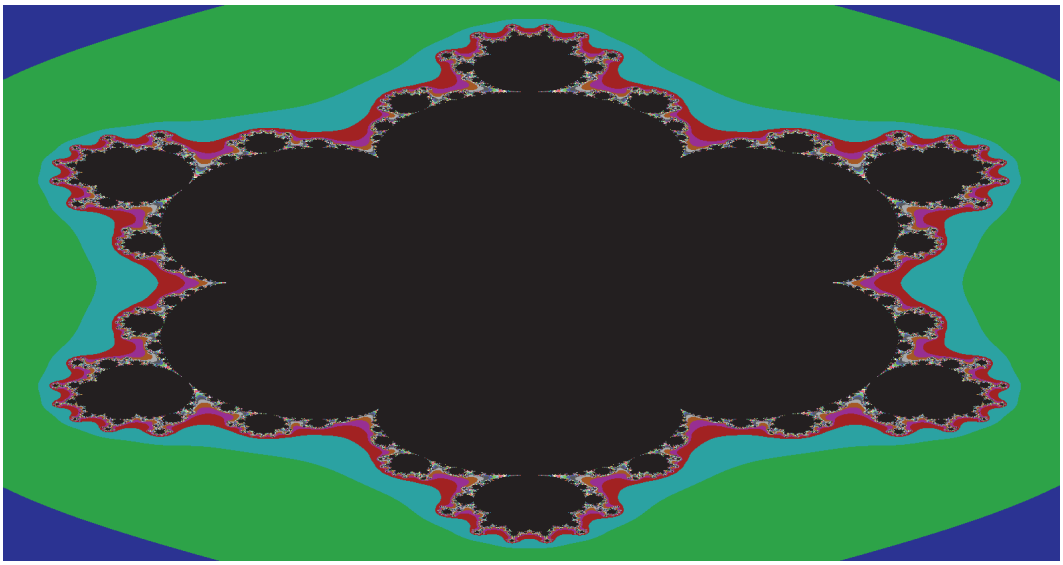
```
function f(x, y, p : Real) : Real;
 begin
  f:=x*x*x*x*x*x*x-21*x*x*x*x*x*y*y
     +35*x*x*x*y*y*y*y-7*x*y*y*y*y*y*y*y+p;
 end;

function g(x, y, q : Real) : Real;
  begin
   g := -y*y*y*y*y*y*y+21*x*x*y*y*y*y*y
        -35*x*x*x*x*y*y*y+7*x*x*x*x*x*x*y+q;
```

```
    end;
```

и вызов процедуры

```
Raznoe(-1.1, 1.1, -1.2, 1.2, 96, 100);
```

получим следующий рисунок (снежинка!).



Модели магнетизма

Задание 7. Рассмотрите следующие процессы.

1. Первая модель магнетизма $z = \left(\dfrac{z^2 + q - 1}{2z + q - 2} \right)^2$,

где q – комплексный параметр.

1. Вторая модель магнетизма $z = \left(\dfrac{z^3 + 3(q-1)z + (q-1)(q-2)}{3z^3 + 3(q-2)z + q^2 - 3q + 3} \right)^2$,

где q – комплексный параметр.

Постройте для вышеуказанных моделей множество типа множеств Мандельброта – рисунок на плоскости z – для зафиксированного значения q.

В обеих моделях точки $z = 1$ и $z = \infty$ являются сверхустойчивыми аттракторами. Кроме того, могут быть еще один или два дополнительных аттрактора, так как есть два критические значения $z = 0$ и $z = (1 - q)^2$.

Проведите исследование параметров по подсчету времени, которое необходимо для того, чтобы эти критические точки приблизились к $z = 1$ (один основной цвет) или к $z = \infty$ (второй основной цвет). Когда q лежит в черных областях, критические точки сходятся к одному из дополнительных аттракторов.

Тема 14
СОВРЕМЕННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ
(4 часа)

14.1. Стиль программирования

Стиль программирования – это способ построения программ, основанный на определенных принципах программирования и выборе подходящего языка, который делает понятными программы, написанные в этом стиле.

Каждый стиль программирования имеет свою концептуальную базу.

Процедурное (императивное) программирование (Basic, Pascal) является отражением архитектуры традиционных ЭВМ, которая была предложена Дж. фон Нейманом в 1940-х гг.

Программа на процедурном языке состоит из последовательности операторов (инструкций), задающих процедуру решения задачи. Основным является оператор присваивания, служащий для изменения содержимого областей памяти.

Концепция памяти как хранилища значений, содержимое которого может обновляться операторами программы, является фундаментальной в императивном программировании.

Таким образом, с точки зрения программиста, имеются программа и память, причем первая последовательно обновляет содержимое последней.

Процедурные языки характеризуются следующими особенностями:

- необходимостью явного управления памятью, в частности описанием переменных;
- малой пригодностью для символьных вычислений;
- отсутствием строгой математической основы;
- высокой эффективностью реализации на традиционных ЭВМ.

Функциональное программирование (Clojure, Common Lisp, F#) предполагает вычисление результатов функций от исходных данных и результатов других функций и не предполагает явного хранения состояния программы. Программа представляет собой совокупность описаний функций и выражений, которые необходимо вычислить. Функциональное программирование не использует концепцию памяти как хранилища значений переменных.

Логическое программирование (Prolog) основано на теории и аппарате математической логики. Логические программы, в принципе, имеют небольшое быстродействие, так как вычисления осуществляются методом проб и ошибок, поиском с возвратами к предыдущим шагам. Программы пишутся не в виде

последовательности инструкций, а в виде множества фактов и правил, а процесс выполнения программы сводится к выводу нужных результатов из этого множества. Логическое программирование относится к декларативному программированию, поскольку программа на нем скорее описывает свойство задачи, нежели алгоритм ее решения.

Объектно ориентированное программирование (C++ и Java) основано на концепциях объектов и классов. Для описания объектов служат классы. Класс определяет свойства и методы объекта, принадлежащего этому классу. Соответственно, любой объект можно определить как экземпляр класса. Программирование заключается в выборе имеющихся или создании новых объектов и организации взаимодействия между ними. При создании новых объектов свойства объектов могут добавляться или наследоваться от объектов-предков. В процессе работы с объектами допускается полиморфизм – возможность использования методов с одинаковыми именами для обработки данных разных типов.

Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные, в отдельно компилируемые модули (библиотеки подпрограмм), например модуль графических ресурсов. Связи между модулями осуществляются через специальный интерфейс, в то время как доступ к реализации модуля запрещен. Эту технологию поддерживают современные версии языков Pascal и C (C++), языки Ада и Modula.

Параллельное программирование – это процесс разделения задачи на множество более мелких и независимых подзадач, которые могут быть выполнены одновременно на разных вычислительных устройствах или на многопроцессорном компьютере. Применяется для выполнения задач, в которых автоматические системы должны обрабатывать много событий одновременно, а также в случаях, когда потребность в вычислительной мощности превышает ресурсы одного процессора.

Наряду с указанными выше выделяют следующие методы.

Декларативное программирование – метод, предназначенный для решения задач искусственного интеллекта, когда программа описывает логическую структуру решения задачи, указывая, что нужно сделать, не вдаваясь в детали. Используются языки программирования типа Prolog.

Эвристическое программирование – метод, основанный на моделировании мыслительной деятельности человека.

Сложность выбора первого языка программирования

Разработчики на Python утверждают, что быстро пишут код. Программисты на C++ – что их код очень производительный. Те, кто используют Java, говорят, как важна кроссплатформенность. У каждого языка есть свои преимущества и недостатки. Один язык не может быть хорош для всего.

Языки программирования похожи друг на друга, поэтому чем больше вы их знаете, тем проще учить новые. Однако всегда важна цель – для чего каждый из них осваивается. Как и любым инструментом, языком нужно пользоваться на практике, иначе знания быстро забудутся. Сам процесс изучения нового порой помогает лучше понять другие технологии.

Новички еще слишком мало знают, чтобы понять, что им нужно от языка. Поэтому выбирать нужно не язык, а то, *чем* вы хотите заниматься.

Многие языки в первую очередь затачиваются под решение определенных проблем или под определенные сферы:

- быстро создать сайт – PHP или Python;
- создать игру – C++ или C#;
- веб-систему для банка – Java, C# или C++;
- красивый интерфейс для сайта – HTML, CSS и JavaScript;
- приложение для Android – Java или Kotlin;
- приложение для iOS или MacOS – Objective-C или Swift.

Парадигмы программирования

Простыми словами *парадигма* – это давно признанные истины, не подлежащие никакому сомнению.

Парадигма программирования – система идей и понятий, определяющих фундаментальный стиль программирования.

Императивное программирование – это парадигма, в которой задается последовательность действий, необходимых для получения результата. В нем используются переменные, операторы присваивания и составные выражения.

Первые языки программирования компьютеров (машинный, ассемблер, фортран, алгол, кобол) были императивными в силу простоты подхода.

Декларативное программирование – парадигма программирования, в которой задается спецификация решения задачи: описывается, что представляет собой проблема и ожидаемый результат, но без описания способа достижения этого результата.

Отличия императивного программирования от декларативного

Императивный стиль – это стиль программирования, при котором вы описываете, как добиться желаемого результата. Например, пишем:

- поставь сковородку на огонь;
- возьми два яйца (куриных);
- нанеси удар ножом по каждому яйцу;
- вылей содержимое на сковородку;
- выкинь скорлупу и т. д.

Это что ни на есть декларативный стиль, но при этом с примесью императивного. Ну хотя бы потому, что получатель должен знать, что такое сковорода и яйца.

Декларативный стиль – это стиль, в котором вы описываете, какой именно результат вам нужен. Тут просто пишем: приготовь яичницу. И получатель такого сообщения уже сам разбирается, какие шаги для этого надо предпринять. Это пример императивного интерпретатора декларативного языка, где способ получения результата не важен. Вы показываете пальцем и говорите: здесь должна быть яичница. Интерпретатор, допустим официант, может заказать ее в другом ресторане, на кухне, сделать сам, собрать из атомов, разогреть вчерашнюю, главное, чтобы она соответствовала декларативному описанию.

Практически все современные языки программирования общего назначения высокого уровня, за исключением некоторых функциональных, относятся к императивным языкам.

Императивные языки, такие как Java, Python, JavaScript, C, C++, занимают доминирующее положение в индустрии ПО, соответственно императивное программирование – самое распространенное. Смысл его в том, что императивная программа содержит прямые указания, что должен сделать компьютер и в каком порядке должны выполняться инструкции. Этот подход легко понять программисту, а компилятору – легко породить достаточно эффективный код.

Самый популярный язык РСУБД SQL так же является декларативным. На нем описывается конечный результат, а способ его получения генерируется сервером СУБД исходя из множества факторов.

Задание

Напишите реферат на тему «Сравнительный анализ языков программирования», выбрав два разных языка программирования.

14.2. О средах языка Pascal

Free Pascal

Язык Pascal был первоначально разработан Н. Виртом в 1970 г. С того дня он значительно эволюционировал, с большим количеством вкладов различными конструкциями компилятора (особенно Borland).

Основные элементы были сохранены на протяжении многих лет:

- легкий синтаксис, довольно многословный, но все же легкий для чтения, идеально подходящий для обучения;
- строго типизированный;
- процедурный;
- отсутствие чувствительности к регистру;

• возможность вложенных процедур;
• наличие встроенных процедур для обработки ввода/вывода.

Язык Pascal признан многими российскими преподавателями как один из лучших именно для начального обучения. Однако среда Borland Pascal, ориентированная на MS DOS, устарела, а среда Borland Delphi с ее богатыми возможностями сложна для начинающего программиста.

Отличительные особенности Free Pascal

Существенные отличия диалекта Turbo Pascal и Free Pascal наблюдаются в основном при работе с динамическими массивами и, в некоторой мере, при работе с графикой, потому что Free Pascal поддерживает большее количество видеоадаптеров и может работать с гораздо лучшим разрешением экрана.

К значительным отличиям следует также отнести поддержку Free Pascal перегрузки арифметических операторов (+, -, *, **, /, div, mod), операторов сравнения (<, >, =, >=, <=) и оператора присваивания :=, поддержку операторов присваивания с выполнением арифметической операции в стиле Си (+=, -=, *=, /=).

Компиляторы Turbo Pascal и Delphi ввели различные особенности в язык Pascal, наиболее заметными из них являются более легкая строковая обработка и объектная ориентированность. Компилятор Free Pascal первоначально эмулировал большую часть Turbo Pascal, а позже и Delphi. Он эмулирует поведение этих компиляторов в соответствующих режимах: некоторые функции доступны только в том случае, если компилятор переключается на соответствующий режим. Когда для определенных функций это необходимо, нужно использовать переключатель командной строки -M или директиву {$MODE}, указанную в исходном тексте. Более подробную информацию о различных режимах можно найти в руководстве пользователя и руководстве программиста.

Pascal ABC

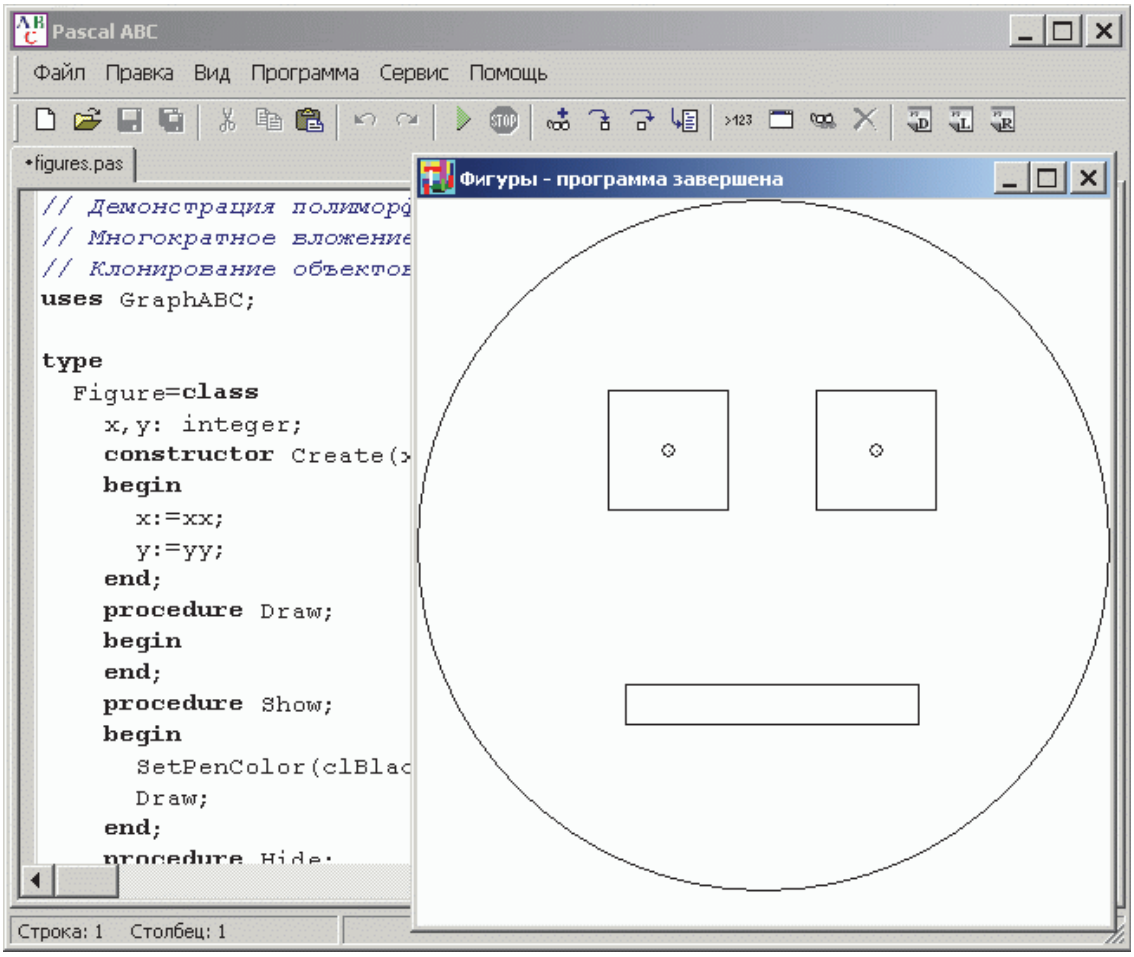
Система Pascal ABC предназначена для обучения программированию школьников и студентов младших курсов на языке Pascal. По мнению авторов, первоначальное обучение программированию должно проходить в достаточно простой и дружественной среде, в то же время эта среда должна быть близка к стандартной по возможностям языка программирования и иметь достаточно богатые и современные библиотеки стандартных подпрограмм.

Система Pascal ABC основана на языке Delphi Pascal и призвана осуществить плавный переход от простейших программ к модульному, объектно ориентированному, событийному и компонентному программированию. Многие концепции в Pascal ABC сознательно упрощены, что позволяет использо-

вать их на более ранних этапах обучения. Например, модуль графики обходится без объектов, хотя его возможности практически совпадают с графическими возможностями Borland Delphi.

Простейшие событийные программы также можно писать без объектов, пользуясь лишь процедурными переменными. Даже в консольных программах можно создавать таймеры и звуки, которые реализованы без использования объектов. Модули устроены практически так же, как и основная программа: отсутствует разделение на секцию интерфейса и секцию реализации. Тела методов можно определять непосредственно внутри классов, что позволяет создавать классы практически сразу после изучения записей, процедур и функций. Имеется модуль контейнерных классов (динамические массивы, стеки, очереди, множества), а также библиотека визуальных компонентов.

Компилятор Pascal ABC является компилятором переднего плана (front-end). Это означает, что он не генерирует исполняемый код в виде .exe-файла, а создает в результате компиляции дерево программы в памяти, которое затем выполняется с помощью встроенного интерпретатора. В итоге скорость работы программы примерно в 20 раз медленнее скорости работы этой же программы, откомпилированной в среде Borland Pascal, и в 50 раз медленнее этой программы, откомпилированной в среде Borland Delphi.



С сентября 2007 г. система Pascal ABC не поддерживается. Ей на смену пришла система программирования PascalABC.NET, основанная на платформе Microsoft.NET и позволяющая генерировать .exe-файлы.

Ключевые особенности PascalABC.NET:

• высокая совместимость с Borland Pascal 7.0 и Delphi;

• генерация кода для платформы .NET;

• возможность доступа к огромному количеству .NET-библиотек от контейнерных классов до средств работы с сетью;

• самые современные средства языков программирования: обобщенные классы и подпрограммы, интерфейсы, перегрузка операций, исключения, сборка мусора;

• ряд расширений языка Pascal, в числе которых оператор foreach, внутриблочные описания переменных, автоопределение типа при описании, встроенные множества произвольных типов, упрощенный синтаксис модулей, методы в записях, операция new для создания объектов, определение тел методов внутри классов;

• среда разработки с встроенным отладчиком, обеспечивающая подсказки по коду, переход к определению и реализации подпрограммы, шаблоны кода, автоформатирование кода;

• простая и эффективная растровая графическая библиотека;

• встроенный электронный задачник Programming Taskbook;

• модули исполнителей «Робот» и «Чертежник», используемых в школьной информатике;

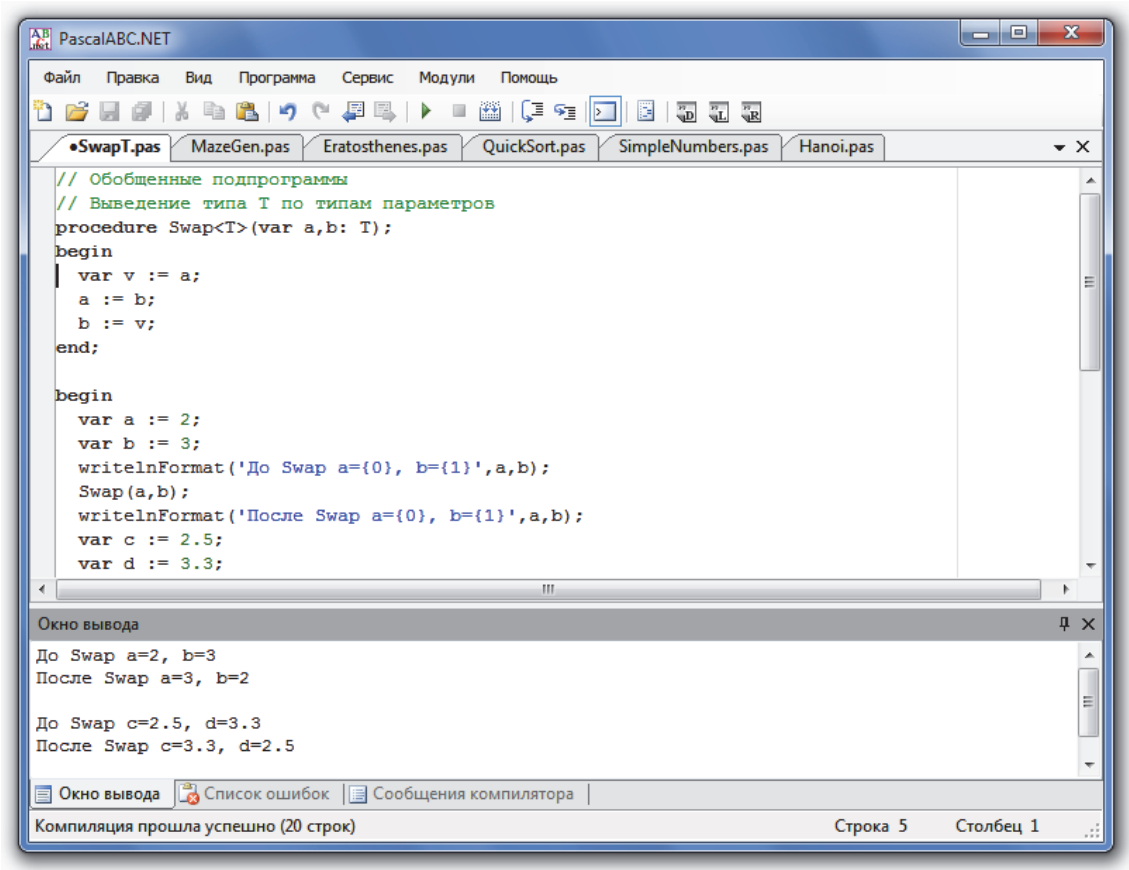
• механизм проверяемых заданий, обеспечивающий автоматическую постановку и проверку заданий;

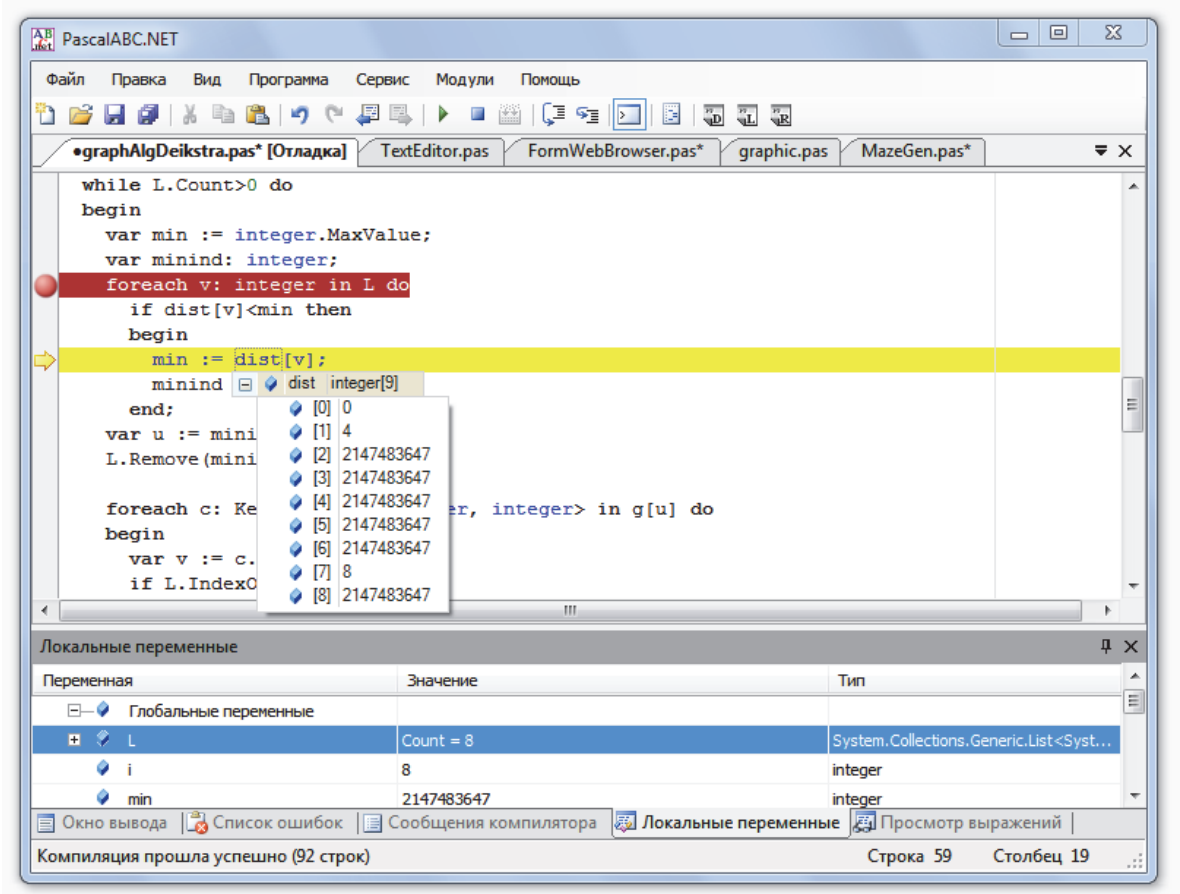
• наличие Web-среды разработки WDE, позволяющей запустить программу прямо из окна браузера;

• возможность опубликовать в интернете ссылку на файл, сохраненный в Web-среде разработки;

• использование многолетнего опыта обучения программированию при создании языка и среды;

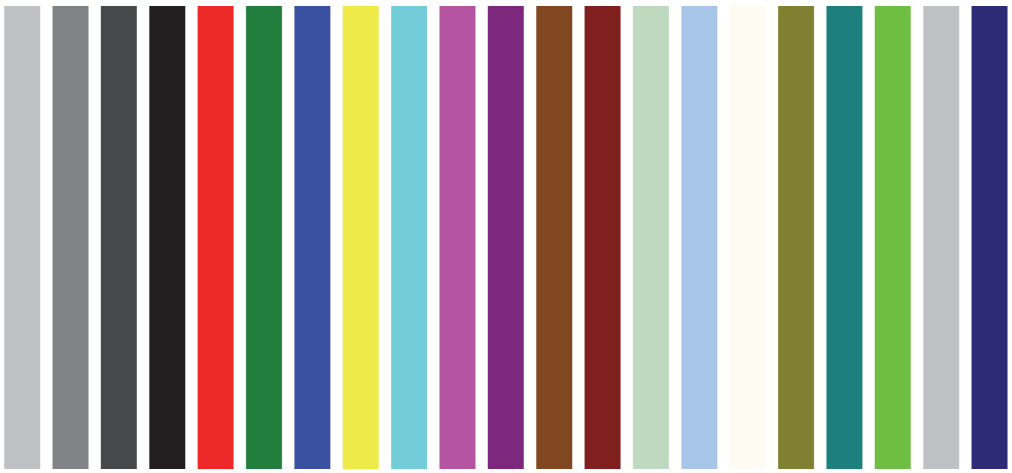
• простота, современные возможности, бесплатность – вот главные достоинства PascalABC.NET!


PascalABC.NET
Файл Правка Вид Программа Сервис Модули Помощь
•SwapT.pas MazeGen.pas Eratosthenes.pas QuickSort.pas SimpleNumbers.pas Hanoi.pas
// Обобщенные подпрограммы
// Выведение типа T по типам параметров
procedure Swap<T>(var a,b: T);
begin
var v := a;
a := b;
b := v;
end;
begin
var a := 2;
var b := 3;
writelnFormat('До Swap a={0}, b={1}',a,b);
Swap(a,b);
writelnFormat('После Swap a={0}, b={1}',a,b);
var c := 2.5;
var d := 3.3;
Окно вывода
До Swap a=2, b=3
После Swap a=3, b=2
До Swap c=2.5, d=3.3
После Swap c=3.3, d=2.5
Окно вывода Список ошибок Сообщения компилятора
Компиляция прошла успешно (20 строк)
Строка 5 Столбец 1


PascalABC.NET
Файл Правка Вид Программа Сервис Модули Помощь
•graphAlgDeikstra.pas* [Отладка] TextEditor.pas FormWebBrowser.pas* graphic.pas MazeGen.pas*
while L.Count>0 do
begin
var min := integer.MaxValue;
var minind: integer;
foreach v: integer in L do
if dist[v]<min then
begin
min := dist[v];
minind
dist integer[9]
[0] 0
[1] 4
[2] 2147483647
[3] 2147483647
[4] 2147483647
[5] 2147483647
[6] 2147483647
[7] 8
[8] 2147483647
end;
var u := mini
L.Remove(mini
foreach c: Ke er, integer> in g[u] do
begin
var v := c.
if L.IndexO
Локальные переменные
Переменная Значение Тип
Глобальные переменные
L Count = 8 System.Collections.Generic.List<Syst...
i 8 integer
min 2147483647 integer
Окно вывода Список ошибок Сообщения компилятора Локальные переменные Просмотр выражений
Компиляция прошла успешно (92 строк)
Строка 59 Столбец 19

Задачи

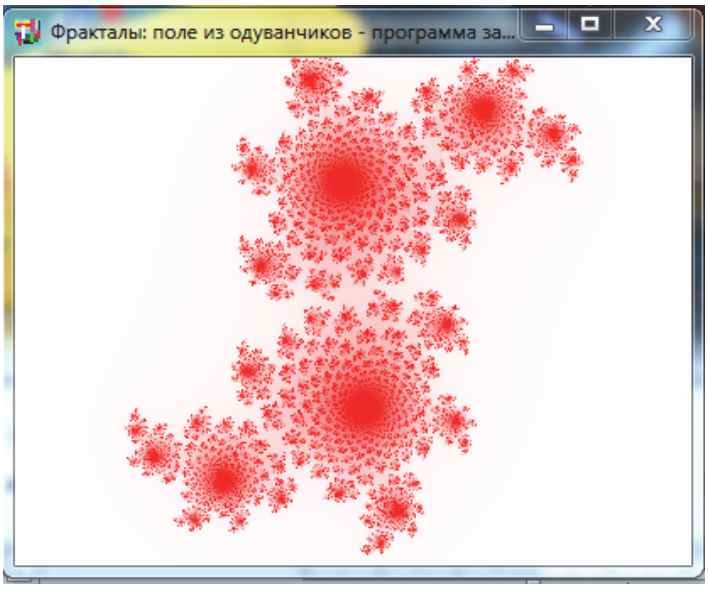
1. Напишите программу получения стандартных цветов в разных языковых средах по принципу следующей картинки.



```
// Демонстрация стандартных цветов
uses GraphABC;

const n=22;

var
  x,y,y1,w,ww,i: integer;
  Colors: array [1..n] of integer;

begin
  SetWindowCaption('Стандартные цвета');
  SetPenStyle(psClear);
  Colors[1]:=clWhite;
  Colors[2]:=clLightGray;
  Colors[3]:=clGray;
  Colors[4]:=clDarkGray;
  Colors[5]:=clBlack;
  Colors[6]:=clRed;
  Colors[7]:=clGreen;
  Colors[8]:=clBlue;
  Colors[9]:=clYellow;
  Colors[10]:=clAqua;
  Colors[11]:=clFuchsia;
  Colors[12]:=clPurple;
  Colors[13]:=clBrown;
  Colors[14]:=clMaroon;
  Colors[15]:=clMoneyGreen;
```

```
  Colors[16]:=clSkyBlue;
  Colors[17]:=clCream;
  Colors[18]:=clOlive;
  Colors[19]:=clTeal;
  Colors[20]:=clLime;
  Colors[21]:=clSilver;
  Colors[22]:=clNavy;

  x:=10; y:=20;
  y1:=400;
  w:=30;
  ww:=9;

  SetWindowSize(x+n*(w+ww),y1+y);

  for i:=1 to n do
  begin
    SetBrushColor(Colors[i]);
    Rectangle(x,y,x+w,y1);
    x:=x+w+ww;
  end;
end.
```

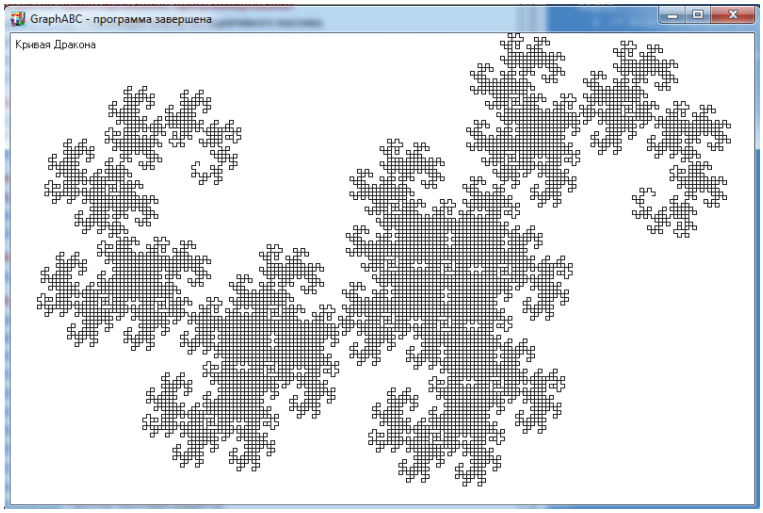
2. Напишите программу получения следующей картинки в разных языковых средах.




```
// 'Фракталы: поле из одуванчиков'
uses Utils,GraphABC;
```

```
const
  n=255;
  max=10;

var
  z,z1,c: complex;
  i,ix,iy: integer;
// z=z^2+c
begin
  cls;
  SetWindowCaption('Фракталы: поле из одуванчиков');
  SetWindowSize(400,300);
  c:=(0.6,0.9);
  for ix:=0 to WindowWidth-1 do
  for iy:=0 to WindowHeight-1 do
  begin
    z:=0.015*(ix-200,iy-140);
    for i:=1 to n do
    begin
      z1:=0.5*z*z+c;
      if abs(z1)>max then break;
      z:=z1;
    end;
    if i>=n then SetPixel(ix,iy,clRed)
      else SetPixel(ix,iy,RGB(255,255-i,255-i));
  end;
  writeln(Время расчета= ',Milliseconds/1000,' ñ');
end.
```

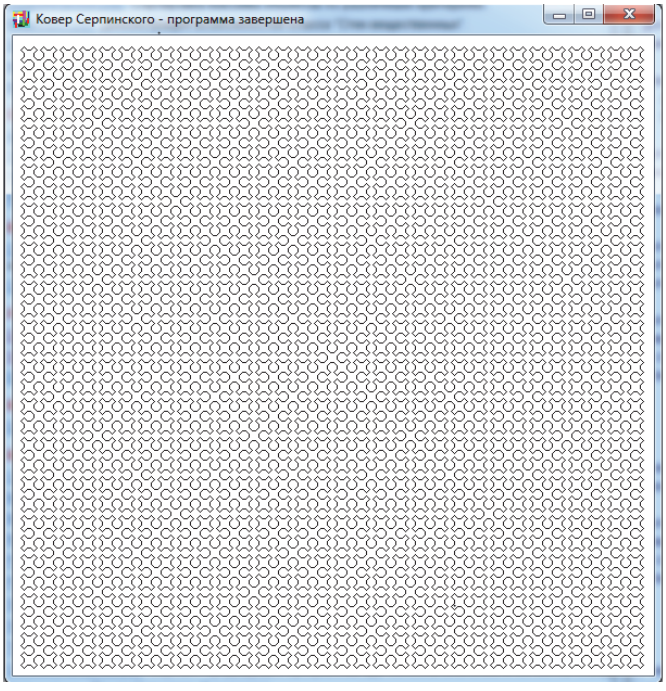
3. Напишите программу получения следующей картинки в разных языковых средах.



```
// Дракон  KuMir/PMir
program dracon;

uses GraphABC;

var
  x,y : integer;
  dx,dy: integer;
  turn: array [1..1000] of Boolean;
  a,b,d,t: integer;
  f: Boolean;
  i: integer;

begin
  SetWindowSize(790,500);
  TextOut(5,5,'Êðèâàÿ Äðàêîíà');
  f:=true;
  for a := 1 to 64 do
  begin
    turn[2*a-1]:=f;
    f:=not f;
    turn[2*a]:=turn[a];
  end;
  x:=200; dx:=0;
  y:=140; dy:=-4;
  b:=0;
  d:=1;
  f:=false;
  MoveTo(x,y);
  for a:=1 to 128 do
  begin
    for i:=1 to 127*4 do
    begin
      b := b+d; x:=x+dx; y:=y+dy;
      LineTo(x,y);
      if f and not turn[b] or not f and turn[b] then
      begin
        t:=dy;
        dy:=-dx;
      end
      else
      begin
        t:=-dy;
        dy:=dx;
```

```
      end;
      dx:=t;
    end;
    b:=b+d; d:=-d;
    f:=not f;
    x:=x+dx; y:=y+dy;
    LineTo(x,y);
    if turn[a] then
    begin
      t:=dy;
      dy:=-dx;
    end
    else
    begin
      t:=-dy;
      dy:=dx;
    end;
    dx:=t;
  end;
end.
```

4. Напишите программу получения следующей картинки в разных языковых средах.

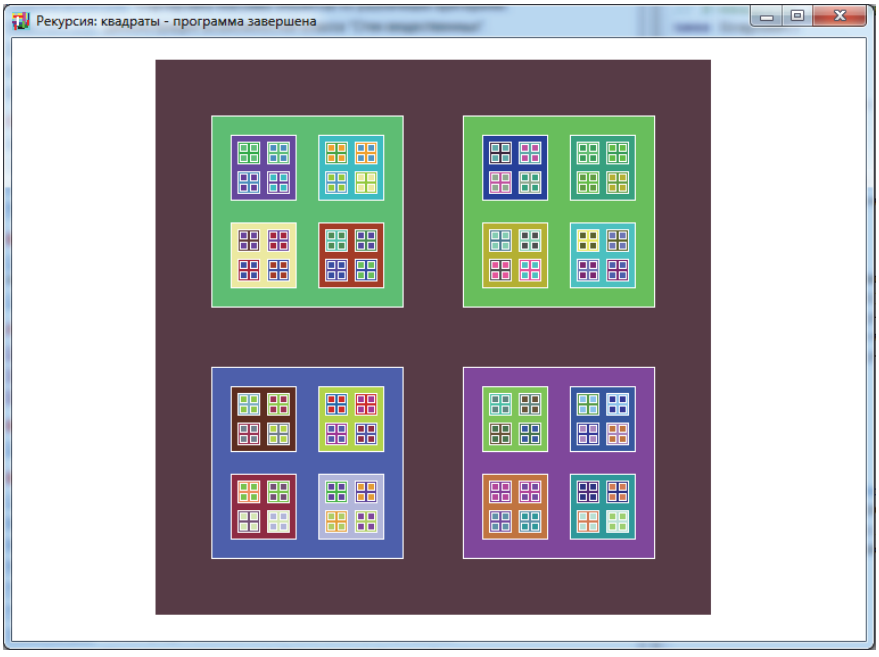



```
// Ковер Серпинского. Иллюстрация forward-объявлений
program Serp;
uses GraphABC;
```

```
var
  a,b,x,y: integer;
  N: integer;

procedure lrel (dx,dy: integer);
begin
  x:=x+dx; y:=y+dy; LineTo(x,y);
end;

procedure BB(k: integer); forward;
procedure CC(k: integer); forward;
procedure DD(k: integer); forward;
procedure AA(k: integer);
begin
 if k>0 then
 begin
   AA(k-1); lrel(a,b);
   BB(k-1); lrel(a,0);
   DD(k-1); lrel(a,-b);
   AA(k-1);
 end;
end;

procedure BB(k: integer);
begin
 if k>0 then
 begin
   BB(k-1); lrel(-a,b);
   CC(k-1); lrel(0,b);
   AA(k-1); lrel(a,b);
   BB(k-1);
 end;
end;

procedure CC(k: integer);
begin
 if k>0 then
 begin
   CC(k-1); lrel(-a,-b);
   DD(k-1); lrel(-a,0);
   BB(k-1); lrel(-a,b);
   CC(k-1);
 end;
end;
```

```
procedure DD(k: integer);
begin
 if k>0 then
 begin
   DD(k-1); lrel(a,-b);
   AA(k-1); lrel(0,-b);
   CC(k-1); lrel(-a,-b);
   DD(k-1);
 end;
end;

begin
  N:=6;
  a:=3;
  b:=a;
  x:=10;
  y:=10;
  SetWindowCaption('Ковер Серпинского');
  SetWindowSize(590,590);
  MoveTo(x,y);
  AA(N); lrel(a,b);
  BB(N); lrel(-a,b);
  CC(N); lrel(-a,-b);
  DD(N); lrel(a,-b);
end.
```

5. Напишите программу получения следующей картинки в разных языковых средах.



```
// E-квадраты. Демонстрация рекурсии
uses GraphABC;

const mw = 2.9;

procedure EKv(n,x,y,w: integer);
var w1,h: integer;
begin
  w1:=round(w/mw);
  h := (w-2*w1) div 3;
  Rectangle(x,y,x+w,y+w);
  if n>0 then
  begin
//    Sleep(1);
    SetBrushColor(clRandom);
    EKv(n-1,x+h,y+h,w1);
    EKv(n-1,x+w-h-w1,y+h,w1);
    EKv(n-1,x+h,y+w-h-w1,w1);
    EKv(n-1,x+w-h-w1,y+w-h-w1,w1);
  end;
end;

var s: string;
    r: integer;
begin
  SetWindowCaption(' E-квадраты ');
  SetWindowSize(750,530);
  SetPenColor(clWhite);
  SetBrushColor(clRandom);
  EKv(4,125,18,490);
end.
```

СПИСОК ЛИТЕРАТУРЫ

Аляев, Ю. А. Практикум по алгоритмизации и программированию на языке Pascal : учеб. пособие / Ю. А. Аляев, В. П. Гладков, О. А. Козлов. – М. : Финансы и статистика, 2004.

Долинский, *М. С*. Алгоритмизация и программирование на Turbo Pascal: от простых до олимпиадных задач / М. С. Долинский. – СПб. : Питер, 2005.

Расолько, *Г. А.* Теория и практика программирования на языке Pascal / Г. А. Расолько, Ю. А. Кремень. – Минск : Выш. шк., 2015.

Расолько, *Г. А.* Теория и практика программирования на языке Pascal / Г. А. Расолько, Ю. А. Кремень. – Минск : Выш. шк., 2022.