БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

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

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

В двух частях

Часть 1

ТЕХНОЛОГИИ РЕАЛИЗАЦИИ АЛГОРИТМОВ И ОБРАБОТКА СТРУКТУР ДАННЫХ

Рекомендовано
Учебно-методическим объединением
по естественно-научному образованию в качестве
учебно-методического пособия для студентов,
обучающихся по специальности «математика (по направлениям)»,
направление специальности «математика
(научно-педагогическая деятельность)»

Учебное электронное издание

Минск, БГУ, 2022

ISBN 978-985-881-165-5 (ч. 1)
ISBN 978-985-881-169-3

УДК 004.42.045(075.8)
ББК 32.973.26-018.2я73-1

Р е ц е н з е н т ы:
кафедра математического анализа дифференциальных уравнений
и их приложений Брестского государственного
университета им. А. С. Пушкина (заведующий кафедрой
кандидат физико-математических наук, доцент *Н. Н. Сендер*);
кандидат технических наук *О. Г. Смолякова*

Расолько, Г. А. Технологии программирования [Электронный ресурс] : учеб.-метод. пособие. В 2 ч. Ч. 1. Технологии реализации алгоритмов и обработка структур данных / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2022. – 1 электрон. опт. диск (CD-ROM). – ISBN 978-985-881-165-5.

Рассмотрены основные технологии реализации алгоритмов. Большое внимание уделено овладению объектно ориентированной технологией программирования и формированию практических знаний и умений использования современных методов программирования на примерах базовых алгоритмов внутренней и внешней сортировок данных, работы с абстрактными типами данных.

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

Оригинал-макет подготовлен в программе MicroSoft Word.

В авторской редакции

Ответственный за выпуск *Е. В. Кремень*. Дизайн обложки *В. П. Явуз*.
Технический редактор *В. П. Явуз*. Компьютерная верстка *Е. В. Кремень*.

Подписано к использованию 04.02.2022. Объем 2,09 МБ.

Белорусский государственный университет.
Управление редакционно-издательской работы.
Пр. Независимости, 4, 220030, Минск.
Телефон: (017) 259-70-70.
e-mail: urir@bsu.by
http://elib.bsu.by/

СОДЕРЖАНИЕ

ТЕМА 1
ТЕХНОЛОГИИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

Содержание темы

- Технологии структурного программирования:
- Этапы создания структурной программы.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

ТЕХНОЛОГИИ СТРУКТУРНОГО ПРОГРАММИРОВАНИЯ

Знакомство с возможностями языка программирования Pascal позволяет писать довольно сложные программы, однако профессиональный коммерческий программный продукт можно создать, если придерживаться жёстких принципов.

Основное требование, которое предъявляется в настоящее время к программе, – её *надёжность*.

Программа также должна владеть *расширяемостью*, это значит допускать оперативное внесение необходимых изменений и дополнений без остановки и сбоев в программном продукте.

Кроме того, программа должна быть *выпущена* (написана) *до заявленного времени*. Это все означает, что весь процесс производства программы должен четко планироваться и контролироваться.

На другой план отходят такие критерии качества программы, как *эффективность* и требуемые *ресурсы*, например, объем оперативной и внешней памяти. Этим критериям также нужно уделять много внимания.

После многочисленных проб и ошибок научный подход к программированию позволил разработать технологию структурного программирования, которая охватывает все этапы разработки программы: спецификацию, проектирование, собственно программирование и тестирование.

Структурное программирование – это способ создания программ, позволяющий путем определенных приемов уменьшить время разработки и облегчить возможность модификации программы.

Технология структурного программирования позволяет создавать программы, которые имеют простую структуру. Это достигается при помощи проектирования «сверху вниз» и применения при написании программы фиксированного множества базовых конструкций, что позволяет уменьшить не только количество ошибок в программе, но и их стоимость: она тем выше, чем позже в процессе разработки найдена ошибка.

Важность защиты от ошибок

Термин синтаксис означает набор жёстких правил, которому должна удовлетворять запись кода, а *семантика* – значение кода или соотношение кода с задачей, которая решается.

В компьютерных программах выделяют три вида ошибок: ошибки времени выполнения, синтаксические и логические. *Синтаксические ошибки* – это ошибки в коде программы (в написании служебных слов, грамматике или пунктуации). *Логические ошибки* – это ошибки в логике программы или в значении переменных. Обычно логические ошибки приводят к ошибкам времени выполнения. Логические ошибки связаны с семантикой кода. При работе программы компьютер выполняет инструкции (операторы) исходного кода и делает конкретно то, что ему предписано делать. Однако бывает, что действия, которые заложены в исходный код, существенно отличаются (намеренно или нет) от того, что программист предполагает получить в итоге. Например, неправильное условие завершения цикла, несовершенное условие для разветвления и т.д.

Транслятор обнаруживает только синтаксические ошибки, остальные должен находить программист.

ЭТАПЫ СОЗДАНИЯ СТРУКТУРНОЙ ПРОГРАММЫ

Понятно, что на создание больших программ и программ небольшого объёма нужно затратить разное количество времени, однако содержание и последовательность этапов создания программы остаются неизменными. Напомним их.

1. Постановка задачи

Когда между заказчиком и исполнителем установилось взаимопонимание, определяется среда, в которой будет выполняться программа. Постановка задачи завершается созданием технического задания, а затем внешней спецификацией программы, включающей в себя:

- описание исходных данных и результатов, которые должны быть получены (типы, форматы, точность, способ передачи, ограничения);

- описание задачи, реализуемой программой;
- способ обращения к программе;
- описание возможных аварийных ситуаций и ошибок пользователя.

2. Выбор модели и метода решения задачи

Постановка задачи формализуется, и на этом основании определяется общий метод ее решения. Если существует несколько методов, наилучший выбирается исходя из критериев сложности, эффективности, точности и так далее в зависимости от конкретных обстоятельств.

3. Разработка внутренних структур данных

Начинать проектирование надо не с алгоритмов, а с разработки структур данных, которые могут быть статическими или динамическими. Нужно дать себе ответ на следующие вопросы:

- Какая точность представления данных необходима?
- В каком диапазоне лежат значения данных?
- Ограничено ли максимальное количество данных?
- Обязательно ли хранить их в программе одновременно?
- Какие действия требуется выполнять над данными?

После ответа на эти вопросы станет понятно, какие структуры данных надо выбрать для использования – статические или динамические и нужно ли использовать абстрактные типы данных (списки, деревья).

4. Проектирование

Под проектированием программы понимается определение общей структуры и взаимодействие модулей (подпрограмм). Одна задача может реализовываться с помощью нескольких модулей и, наоборот, в одном модуле могут решаться несколько задач. Главным критерием разбивки задачи на модули является минимизация их взаимодействия.

На этом этапе применяется технология проектирования сверху вниз, основная идея которого теоретически проста: разбивка задачи на подзадачи меньшей сложности, пригодные для рассмотрения по отдельности.

Сразу же определяются способы взаимодействия подзадач (спецификация интерфейсов).

Представление алгоритма решения задачи в виде последовательности подзадач называется *процедурной декомпозицией*, а вся технология структурного программирования относится к процедурной парадигме программирования, в отличие от объектно ориентированной.

5. Структурное программирование

Структурное программирование здесь понимается как структурное кодирование. Оно опять же организуется по принципу «сверху вниз»: сначала кодируются модули самого верхнего уровня и составляются тестовые примеры для их отладки, при этом вместо еще не написанных модулей следующего уровня кодируются так называемые *заглушки* – временные подпрограммы. Таким образом создается логический каркас программы, который уже можно анализировать на сложность проектирования подпрограмм низшего уровня.

Этапы проектирования и кодирования обычно совмещаются по времени: сначала проектируется и кодируется верхний уровень, затем – следующий и т. д. Такая стратегия снижает цену ошибки, потому что в процессе кодирования может возникнуть необходимость внести изменения, которые отражаются на модулях более низкого уровня.

6. Тестирование

И проектирование, и программирование обязательно должны сопровождаться написанием набора тестов – проверочных исходных данных и соответствующих им наборов эталонных реакций.

Тестирование – это процесс, при помощи которого проверяется правильность работы программы (подпрограммы). Цель тестирования – показать, что программа работает правильно и удовлетворяет всем проектным спецификациям. *Отладка* – процесс исправления ошибок в программе, которые обнаружены при тестировании.

7. Правила программирования

Все технологии программирования направлены на достижение цели – получить эффективную и надежную, легко читаемую программу, как можно более простой структуры.

Если следовать некоторым простым рекомендациям, то можно избежать многих распространенных ошибок. Приведем некоторые из этих рекомендаций, которые вытекают из многолетней практики программирования:

- программа должна состоять из максимально отдельных частей, связанных друг с другом только через интерфейсы;
- каждое законченное действие оформляется в виде подпрограммы;
- все величины, которыми подпрограмма обменивается с вызывающей программой, должны передаваться через параметры;
- входные параметры лучше передавать как параметры-константы;

- в подпрограмме полезно предусмотреть реакцию на неправильные входные параметры и аварийное завершение (формировать код результата, который необходимо анализировать в вызывающей программе);
- величины, которые используются только в подпрограмме, нужно описывать внутри ее как локальные переменные;
- имена переменных должны отражать их смысл.

Правильно выбранные имена могут сделать программу в некоторой степени самодокументированной. Для счетчиков коротких циклов, наоборот, лучше обойтись однобуквенными именами.

Нужно избегать использования в программе чисел (кроме 0 и 1) в явном виде.

Константы должны иметь осмысленные имена и задаваться в разделе `const`. Тогда при необходимости изменения константы правки нужно будет выполнять в одном месте.

Придерживайтесь ясности кода программы.

Комментарии, размещенные в начале программы, должны содержать сведения о названии программы, ее назначении, фамилии программиста, дате создания программы.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 2
ОБЪЕКТНО ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Содержание темы

- ООП.
- Совместимость объектных типов.
- Виртуальные методы. Позднее связывание.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

ОБЪЕКТНО ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Структурная программа состоит из совокупности подпрограмм, связанных с помощью интерфейса. Подпрограммы работают с данными, которые или являются локальными, или передаются им в качестве параметров.

Ошибки часто связаны с тем, что в подпрограмму передаются неверные данные. Естественный путь избежать таких ошибок – связать в единое целое данные и все подпрограммы, которые предназначены для их обработки, и назвать эту совокупность новым типом – объектом.

Объектно ориентированное программирование основано на трех важнейших принципах, придающих объектам новые свойства. Этими принципами являются инкапсуляция, наследование и полиморфизм.

Инкапсуляция есть объединение в единое целое данных и алгоритмов обработки этих данных.

Инкапсуляция позволяет при необходимости изменять реализацию объекта без модификации основной части программы, если его интерфейс остался тем же. Простота же модификации программы – важный критерий качества программы.

Наследование есть свойство объектов порождать потомков. Объект-потомок автоматически получает от родителя все поля и методы, и может дополнять их.

Наследование позволяет создать иерархию объектов. В языке Pascal любой объект может иметь одного родителя и произвольное количество

потомков. Иерархия представляется в виде дерева, в котором общие объекты располагаются ближе к корню, а специализированные – на ветках и листьях.

В рамках ООП поведенческие свойства объекта определяются набором методов, входящих в объект. Изменяя алгоритм того или иного метода в потомках объекта, программист может дать этим потомкам специфические свойства, отсутствующие у родителя. Для изменения метода нужно перекрыть его в потомке, т. е. объявить в потомке одноименный метод и реализовать в нем нужные действия. В результате в объекте-родителе и объекте-потомке будут действовать два одноименных метода, имеющие разную алгоритмическую основу и, таким образом, дающие объектам различные свойства.

Полиморфизм – это свойство родственных объектов (т. е. объектов, имеющих одного общего родителя) решать схожие по смыслу проблемы разными способами.

В Borland Pascal полиморфизм расширяется виртуализацией метода (`virtual` – эффективный), когда из родительских методов обращаются к методу потомков. Объявление метода виртуальным дает возможность дочернему классу произвести замену виртуального метода своим собственным. Этот механизм рассмотрим позже.

Применение ООП позволяет писать гибкие программы, которые легко расширяются и читаются. Во многом это обеспечивается благодаря изменчивости объектов, т. е. такому свойству как полиморфизм, когда имеется возможность при наследовании менять реализацию методов так, что они будут иметь один и тот же интерфейс и разную алгоритмическую реализацию.

Механизм объявления объектов

Объектный тип может быть определен как полный, самостоятельный тип, подобный описанию записей в Pascal, но он может определяться и как потомок существующего типа объекта.

Объект – это тип данных, поэтому он определяется в разделе описания типа с использованием служебного слова `object`. В других языках объектный тип называют классом. Объект похож на тип `record`, но кроме *полей данных*, в нем можно описывать *заголовки методов*.

Методами называются подпрограммы, предназначенные для работы с полями объекта.

Поля объекта описываются аналогично обычным переменным: для каждого поля задается его имя и тип. Значения полей определяют состояние объекта.

Поля и ***методы называются элементами объекта***.

Для реализации принципа инкапсуляции, т. е. ограничения видимости элементов, объекты обычно описывают в модулях. В интерфейсной части модуля описывается тип объекта, а тела методов объектов описываются в разделе реализации.

Хранение описаний в объектах

Согласно принципу инкапсуляции во многих случаях требуется ограничение доступа к составляющим объект компонентам (методам и переменным). Доступ к полям объектов напрямую нежелателен. Например, объект описывает ноутбук и содержит в том числе поля характеризующие его размер. Очевидно, что значения длины, ширины и толщины корпуса должны быть положительны, и более того находиться в некотором диапазоне, поскольку не существует ноутбуков толщиной 2 метра или шириной 3 миллиметра. Для того, чтобы пользователь мог изменить размеры устройства должны быть реализованы методы для изменения параметров, внутри которых новое значение должно проверяться на пригодность, т. е. валидироваться. В таких случаях предусматриваются объекты, в которых есть открытые (публичные, доступные) поля и методы и такие поля и методы, прямой доступ к которым запрещен, – закрытые (тайные, личные) поля и методы.

Видимостью элементов объекта управляют директивы `public` и `private`. В объекте может быть произвольное количество разделов `public` и `private`. По умолчанию все элементы объекта считаются видимыми извне, т. е. `public`. Действие директивы распространяется до другой директивы или до конца объекта.

Поля и методы, которые идут сразу за заголовком объектного типа или за директивой `public`, не имеют никаких ограничений на область действий. Поля и методы, объявленные после директивы `private`, считаются закрытыми (собственными, личными) и ограничены использованием в пределах модуля.

Полное описание объекта примерно следующее:

```
type
      NEWObject = object
        Поля;   {Открытые}
        Методы; {Открытые}
      private
        Поля;   {Закрытые}
        Методы; {Закрытые}
      public
```

```
    Поля;   {Открытые}
    Методы; {Открытые}
              end
```

С наследованием:

```
type
     NEWObject=object(Имя_объекта_родителя)
       Поля;   {Открытые}
       Методы; {Открытые}
      private
       Поля;   {Закрытые}
       Методы; {Закрытые}
      public
       Поля;   {Открытые}
       Методы; {Открытые}
      end
```

Каждое действие, которое должен выполнять объект, оформляется в виде отдельной процедуры или функции, т. е. метода.

Описания реализаций методов (текст подпрограмм) размещаются вне объекта в разделе описания процедур и функций.

При определении содержимого методов исходят из нужного поведения объекта.

При объявлении объектов должны соблюдаться следующие требования:

- описание типа объект может происходить только в секции `type` основной программы или в разделах модуля; нельзя описывать локальные объекты в подпрограммах;
- при описании типа объекта в каждом разделе все поля данных должны находиться перед описаниями методов;
- компоненты объекта не могут быть файлами, а файлы, в свою очередь, не могут содержать компоненты типа «объект»;
- при наследовании полей в дочерних типах уже нельзя объявлять их идентификаторы, определенные в одном из родительских типов, однако дочерние объекты могут переопределять любой из наследуемых методов.

Область действия полей данных объекта неявно распространяется на тело процедур и функций, реализующих методы этого объекта.

В случае если в дочернем типе описывается новая процедура инициализации, в ней обычно сначала вызывается процедура родительской инициализации. Это самый естественный способ проинициализировать поля, полученные в наследство.

Механизм определения метода

В объектах (в разделах `private` или `public`) сначала определяются поля объекта, а затем методы.

Параметры метода имеют уникальные имена, которые не совпадают с именами всех полей, как собственных, так и наследованных. Сами методы описываются вне определения объекта как отдельная процедура или функция, при этом имя метода дополняется именем типа объекта, которому принадлежит метод, с последующей точкой, список параметров можно опускать.

Замечание. Фактически имя метода – сложное:

```
имя_объекта.имя_метода
```

Значит, в программе может быть объявлена другая подпрограмма, имя которой будет совпадать с именем метода, и здесь не будет ошибки.

Переопределение методов

Наследование дочерними типами информационных полей и методов их родительских типов выполняется соответственно следующим правилам:

Правило 1. Информационные поля и методы родительского типа наследуются всеми его дочерними типами независимо от количества промежуточных уровней иерархии.

Правило 2. Доступ к полям и методам родительских типов в рамках описания любых дочерних типов выполняется так, будто они описаны в самом дочернем типе.

Правило 3. Нельзя в дочерних типах использовать идентификаторы полей, совпадающие с идентификаторами полей какого-либо родительского типа.

Правило 4. Дочерний тип может доопределять любое количество собственных методов и информационных полей.

Правило 5. Любое изменение кода в родительском методе автоматически влияет на все методы порожденных дочерних типов, которые его вызывают.

Правило 6. Идентификаторы метода в дочерних типах могут совпадать с именами метода в родительских типах. Тогда дочерний метод подавляет тождественный ему родительский, и в рамках дочернего типа при указании имени такого метода будет вызываться именно дочерний метод. Для вызова родительского метода перед именем метода нужно написать

```
<имя_родительского_объекта.метод>
```

или использовать конструкцию `inherited` (наследуемый) метод.

Важным аспектом наследственности являются *правила вызова наследуемых методов*:

1. При вызове метода компилятор сначала ищет метод, имя которого определено внутри типа объекта.

2. Если в типе объекта не определен метод с указанным в операторе вызова именем, тогда компилятор в поисках метода с таким именем поднимается выше к родительскому или прародительскому типу. Вызовы метода с ниже размещенных по иерархии типов (дочерних) не разрешаются.

3. Если наследуемый метод найден и его адрес подставлен, нужно помнить, что метод, который вызывается, будет работать так, как он определен в родительском типе. И если этот наследуемый родительский метод вызывает еще и другие методы, тогда вызываться уже будут только родительские или методы предков.

Раннее связывание

После компиляции и запуска программы ее исполняемые операторы в виде инструкций – команд процессору, находятся в сегменте кода. Каждая подпрограмма имеет точку входа. *При компиляции вызов подпрограммы заменяется последовательностью команд, передающих управление в эту точку входа*.

Этот механизм называется *ранним связыванием*, так как все ссылки на подпрограммы, компилятор строит до выполнения программы.

Очевидно, что с помощью раннего связывания не удается обеспечить возможность вызова из одной и той же подпрограммы метода то одного объекта, то другого.

В Pascal существует и другой механизм вызова методов, когда ссылки определяются уже на этапе выполнения программы в момент вызова метода. Такой механизм реализуется с помощью, так называемых, виртуальных методов и называется ***поздним связыванием***.

Возможность использования методов то одного объекта, то другого нарушает правило соответствия типа по присваиванию, которое мы изучали до этого. Для объектов понятие совместимости расширено.

Экземпляры объектов

Переменная объектного типа называется *экземпляром* объекта.

Часто экземпляры просто называют объектами.

Экземпляры объектов можно создавать и в статической и в динамической памяти, можно определять массивы экземпляров объектов, или указателей на объекты, или другие структуры данных.

Доступ к элементам экземпляра объекта осуществляется либо с использованием составного имени, когда указывается *имя экземпляра объекта* и *через точку имя поля* или *имя метода*, либо с помощью оператора присоединения `with`.

Перед использованием элементов объекта требуется проинициализировать его поля и, возможно, выполнить другие подготовительные действия. Метод, выполняющий такие действия, обычно называют именем `Init`.

Если объект описан в модуле, получить или изменить значения элементов с директивой `private` в программе можно только через обращение к соответствующим методам.

При создании каждого экземпляра объекта выделяется память, достаточная для хранения всех его полей.

Коды методов объекта хранятся в одном экземпляре. Для того чтобы методу было известно, с данными какого экземпляра объекта он работает, при вызове ему в неявном виде передается параметр `self`, который и определяет место размещения данных этого объекта.

Фактически внутри метода обращение к полю «x» объекта имеет вид `self.x` (`@self` представляет собой адрес начала области оперативной памяти, в которой хранятся поля объекта).

Экземпляры объектов одного типа *можно присваивать друг другу, при этом выполняется копирование всех полей*. Позже будем рассматривать правила расширенной совместимости типов объектов.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?

ТЕМА 3
СОВМЕСТИМОСТЬ ОБЪЕКТНЫХ ТИПОВ

Содержание темы

- Совместимость объектных типов.
- Виртуальные методы. Конструкторы.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

СОВМЕСТИМОСТЬ ОБЪЕКТНЫХ ТИПОВ

Наследование несколько изменяет правила совместимости типов в Turbo Pascal: порожденный тип наследует совместимость со всеми родительскими типами. Другими словами: объекты дочерних типов могут свободно использоваться вместо объектов родительских, но не наоборот.

Совместимость объектных типов бывает трех видов:

- между экземплярами объектов;
- между указателями на экземпляры объектов;
- между формальными и фактическими параметрами.

Во всех трех случаях совместимость односторонняя: *родительскому экземпляру объекта может быть присвоен экземпляр такого же типа или любого из его потомков*.

Совместимость между экземплярами объектов

Смысл совместимости объектов в том, что в результате присваивания значения все информационные поля объекта приемника, который стоит в левой части оператора присвоения значения, должны получить значения. Объект же потомка может иметь и свои собственные поля.

Из источника в приемник будут копироваться *только те поля*, которые являются общими и для объекта родительского типа, и для объекта дочернего типа.

Совместимость между указателями на экземпляры объектов

Указатель на объект дочернего типа может быть присвоен указателю на родительский тип, т. е. совместимость типов работает также для указателей на типы объектов, *при этом тип вызываемого метода объекта соответствует типу указателя*, а не типу того объекта, на который он ссылается.

Если известно, что указатель на предка в самом деле хранит ссылку на потомка, можно обратиться к элементам, определенным в потомке, с помощью такого приема как явное преобразование типа.

Совместимость между формальными и фактическими параметрами

Если экземпляр объекта является параметром подпрограммы, ему может соответствовать *аргумент того же типа или любого из его наследников*, но есть разница между передачей экземпляров объекта по значению (параметр-значение) и по адресу (параметр-переменная).

Другими словами, формальный параметр (параметр-значение, параметр-переменная) данного объектного типа может принимать в качестве фактического параметра объект своего же типа или объекты всех своих дочерних типов. Но вспомним механизм передачи параметров.

Важно помнить, что параметр, передаваемый по *значению*, представляет собой *копию* экземпляра объекта-аргумента, *приведенную к типу параметра*. Копия содержит только те поля данных и методы, которые соответствуют типу параметра.

При передаче экземпляра объекта по адресу подпрограмме передается указатель (адрес) на фактический экземпляр объекта, т. е. *приведение типов не выполняется*.

Поэтому в подпрограмме тип экземпляра объекта, передаваемого по адресу, может изменяться в зависимости от аргумента.

Замечание. Процитируем еще раз описанное выше свойство: «Указатель на объект дочернего типа может быть присвоен указателю на родительский тип, *при этом тип вызываемого метода соответствует типу указателя*, а не типу того объекта, на который он ссылается».

Это все объясняется тем, что методы являются статическими. Для реализации полиморфизма в Turbo Pascal существуют *виртуальные* методы, которые динамически связываются с экземплярами объектов во время выполнения программы.

ВИРТУАЛЬНЫЕ МЕТОДЫ. КОНСТРУКТОРЫ

Экземпляры объектов, фактический тип которых может изменяться во время выполнения программы, называются полиморфными. Исходя из предыдущего, можно сделать вывод, что полиморфным может быть экземпляр объекта, определенный через указатель или переданный в подпрограмму по адресу.

Полиморфные объекты обычно применяются вместе с виртуальными методами.

Метод становится виртуальным, если за его определением в типе объекта относится служебное слово `virtual`:

```
procedure имя_метода (параметры);                virtual;
function имя_метода (параметры) : тип_вызова; virtual;
```

При описании реализации метода слово `virtual` уже не ставится.

Различие между вызовом статического и динамического методов заключается в том, что в первом случае компилятору сразу известна связь объекта с методом, и он устанавливает ее на этапе компиляции (*раннее связывание)*. Во втором случае компилятор как бы откладывает решение связи до момента выполнения программы (*позднее связывание*).

Для реализации позднего связывания необходимо, чтобы адреса виртуальных методов хранились там, где ими можно будет воспользоваться в любой момент во время выполнения программы. Поэтому *компилятор формирует* для этих методов внутреннюю таблицу виртуальных методов – ТВМ (VMT). В первое поле этой таблицы записывается размер объекта, а затем идут *адреса кодов* процедур или функций, реализующих каждый из его виртуальных методов, как описанных в объекте, так и унаследованных. Такая таблица – *одна для каждого объектного типа*.

Каждый экземпляр объекта *во время выполнения программы* при своем реальном создании должен иметь доступ к ТВМ. Эта связь экземпляра объекта с ТВМ устанавливается с помощью специального метода, называемого *конструктором*.

Значит, объект, имеющий хотя бы один виртуальный метод, должен содержать конструктор – это специальный метод, который инициирует объект. В нем может выполняться выделение памяти под динамические переменные или структуры, если они есть в объекте, и присваиваться начальные значения. Обычно ему дают имя `INIT`.

В этом методе служебное слово `procedure` в объявлении и реализации заменяется словом `constructor`, обозначающее особый вид процедуры – конструктор, который помимо своей собственной работы

(которой может вообще и не быть) выполняет установочную работу для механизма виртуальных методов. По ключевому слову `constructor` компилятор вставляется в начало метода фрагмент, который записывает ссылку на ТВМ в специальное поле экземпляра объекта (память под это поле выделяется компилятором).

Конструктор всегда должен вызываться до первого вызова виртуального метода, потому что конструктор устанавливает связь между экземпляром объекта, который вызывает конструктор, и таблицей виртуальных методов данного объектного типа. Конструктор должен быть *вызван для каждого создаваемого экземпляра объекта*. Присваивание одного экземпляра объекта другому возможно только после конструирования обоих. Если же конструктор не будет вызван перед обращением к виртуальному методу, тогда компьютер не будет знать, где искать этот метод. Это и приведет к фатальной ситуации.

Если программу компилировать с директивой {`$R+`}, тогда программа сама проверяет корректность объектов, анализируя их размеры. При этом генерируется вызов подпрограммы проверки правильности ТВМ перед каждым вызовом виртуального метода. Если контрольные значения размера в ТВМ указывают на сбой, происходит фатальная ошибка 210. После отладки директиву можно отключить.

Сами *конструкторы* могут быть только *статическими*, хотя внутри конструктора могут вызываться и виртуальные методы. Если в конструкторе есть поля, которые также являются объектами с виртуальными методами, то в теле конструктора вызываются конструкторы этих объектов.

В объекте может быть определено несколько конструкторов. Повторный вызов конструктора вреда программе не принесет.

Правила описания виртуальных методов

Когда объявляется виртуальный метод в каком-нибудь родительском типе, то это накладывает следующие ограничения на его дочерние типы:

- все методы дочерних типов одноименные с виртуальными родительскими также должны быть виртуальными. Статический метод не может подавить виртуальный;
- после того как метод стал виртуальным, его *заголовок не может меняться в объектах более низкого уровня иерархии*. Заголовки всех реализаций одного и того же виртуального метода должны быть идентичными (одинаковое число параметров, порядок их следования, их типы и т. д.). Статический же метод, переопределив другой статический метод, может иметь другую совокупность параметров;

- переопределять виртуальный метод в каждом из потомков нужно только по необходимости (если хочется изменить действия, описанные в наследуемом методе);
- объект, имеющий хотя бы один виртуальный метод, должен содержать конструктор.

Каждый вызов виртуального метода проходит через обращение к ТВМ, а статические методы вызываются «напрямую», вот почему вызов статического метода происходит быстрее, чем виртуального. Поэтому при выборе метода следует оценивать гибкость, которую дает виртуальный метод, и некоторое увеличение скорости подсчетов, которое дают статические методы.

О ВНУТРЕННЕМ ПРЕДСТАВЛЕНИИ ОБЪЕКТОВ

Поля объекта записываются в порядке их описаний как непрерывная последовательность переменных. Если объектный тип определяет виртуальные методы, то компилятор размещает в нем дополнительное 16-ти битовое поле, называемое полем таблицы виртуальных методов. Оно используется для запоминания адреса смещения таблицы виртуальных методов в сегменте данных. Если объект унаследовал родительские поля, то его собственные поля записываются после родительских полей. Если объектный тип наследует виртуальные методы, то он также наследует и поле таблицы виртуальных методов, благодаря чему новое 16-ти битовое поле не размещается.

Таблица виртуальных методов автоматически создается компилятором для объекта, имеющего виртуальные методы, и программа никогда не манипулирует ими непосредственно.

Заполнение полей таблицы виртуальных методов экземпляра объекта осуществляется конструктором объектного типа. Программа никаким другим способом не имеет к таблице виртуальных методов прямого доступа.

Первое слово (16 бит) таблицы виртуальных методов содержит размер экземпляра соответствующего объектного типа. Эта информация используется конструктором и деструктором для определения количества байтов, которое затем передается операторам `New` и `Dispose` при использовании расширенного их синтаксиса.

Второе слово таблицы виртуальных методов содержит отрицательный размер экземпляра соответствующего объектного типа. Эта информация используется программой контроля вызовов виртуальных методов для выявления неинициализированных экземпляров объектов в случае включения директивы {`$R+`}.

Третье и четвертое слова таблицы виртуальных методов содержит 0. Далее следует список 32-разрядных указателей методов – адресов точки входа в порядке их описания. Каждый указатель ссылается на свой виртуальный метод.

Для непосредственной работы с ТВМ используются две функции.

Стандартная функция

```
TypeOf (ИмяТипаИлиИмяЭкземпляра): Pointer
```

возвращает указатель на ТВМ для конкретного экземпляра или самого типа объекта и применяется только к объектам, имеющим ТВМ, иначе будет ошибка.

Функция `TypeOf` может быть использована для проверки фактического экземпляра по схеме

```
if TypeOf (Self) = TypeOf (ObjVAR) then ...
```

Стандартная функция `SizeOf` при применении к экземпляру типа объект, который имеет связь с таблицей ТВМ, возвращает размер, который хранится в ТВМ. Для типов объектов, имеющих ТВМ, функция `SizeOf` всегда возвращает фактический размер экземпляра, который может отличаться от заявленного в описании типа.

Задание 1. Технологией ООП написать программу для решения уравнений до третьей степени включительно $a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 0$ с вещественными коэффициентами на множестве комплексных чисел, если степень уравнения и коэффициенты уравнения – последовательность некоторых случайных чисел.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 4
ЭКЗЕМПЛЯРЫ ОБЪЕКТОВ В ДИНАМИЧЕСКОЙ ПАМЯТИ

Содержание темы

- Экземпляры объектов в динамической памяти.
- Освобождение динамических экземпляров объектов. Деструкторы.
- Обработка ошибок при работе с динамическими объектами.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

Экземпляры объектов в динамической памяти

Все приведенные до сих пор экземпляры объектов были статическими, они описывались в секции `var`, им присваивались имена, и сами экземпляры объектов размещались в сегменте данных программы (глобальные переменные) или в стеке (локальные переменные).

Но экземпляры объектов могут быть размещены и в динамической памяти. Подобно любым другим динамическим структурам данных динамические экземпляры объектов объявляться как ссылки:

```
var
    ИмяССылкиНаОбъект : ^Тип_Объекта;
```

Дальнейшее обращение к экземплярам объектов, их полям и методам тоже будет обычным:

- `ИмяССылкиНаОбъект – ссылка на экземпляр объекта;`
- `ИмяССылкиНаОбъект^ – экземпляр объекта в целом;`
- `ИмяССылкиНаОбъект^.ИмяПоля – поле экземпляра объекта;`
- `ИмяССылкиНаОбъект^.ИмяМетода – метод экземпляра объекта.`

Для выделения памяти под динамические экземпляры объекта используется стандартная процедура `New`:

```
New(ИмяССылкиНаОбъект)
```

Процедура `New`, как и обычно, выделяет в динамической памяти область, достаточную для хранения экземпляра типа, определяемого

указателем, и возвращает адрес этой области в указателе. Если же динамический объект содержит виртуальные методы, то он должен быть инициализирован посредством вызова конструктора до вызова всех его остальных методов:

```
ИмяССылкиНаОбъект^.ИмяКонструктора(параметры)
```

В Turbo Pascal процедура New расширена и позволяет *в одной операции выделить память под объект и вызвать конструктор*:

```
New(ИмяССылкиНаОбъект, ИмяКонструктора(параметры))
```

Сначала выполняется New для выделения памяти для объекта в Heap. Когда она закончилась без ошибок, то вызывается конструктор объектов. Компилятор определяет правильность вызова конструктора, проверяя тип указателя, передаваемого в качестве первого параметра.

Оператор New может вызываться и как функция:

```
ИмяССылкиНаОбъект:=
New(ТипОбъекта,ИмяКонструктора(Параметры))
```

Использование оператора New как функции применимо ко всем типам данных, а не только к объектам.

Когда поля объектов, в свою очередь, динамические, тогда выделение памяти в Heap для них должно происходить в конструкторе.

В конструкторе должно происходить необходимая инициализация полей данных (в том числе и вызовы конструкторов для унаследованных полей).

Освобождение динамических экземпляров объектов. Деструкторы

Для освобождения памяти от динамических объектов применяется стандартная процедура

```
Dispose (ИмяССылкиНаОбъект)
```

Такой вызов освободит динамический экземпляр объекта в целом. При выполнении этой процедуры освобождается количество байтов, равное размеру объекта, соответствующего типу указателя.

Но когда поля данного объекта были динамическими, и под них выделялась дополнительная память при выполнении конструктора или иной процедуры инициализации, тогда их нужно освободить до уничтожения самого объекта.

Для корректного освобождения памяти из-под полиморфных объектов, для которых создавалась ТВМ, вводится специальный вид метода – *деструктор*.

Деструктор объявляется среди других методов служебным словом `destructor` вместо `procedure`. Обычно деструктору дается имя `Done` («Завершено»).

Исполняемый код деструктора никогда не бывает пустым, потому что компилятор по служебному слову `destructor` вставляет в конец тела метода операторы получения размера объекта из ТВМ.

Смысл и необходимость введения деструктора заключается в том, что его можно использовать в расширенной процедуре `Dispose` так же, как конструктор в `New`:

```
Dispose (ИмяСсылкиНаОбъект, ИмяДеструктора);
```

Действует такой вызов следующим образом: сначала вызывается деструктор и выполняются описанные в нем завершающие действия как обычного метода. Далее, если объект содержит виртуальные методы, тогда деструктор осуществляет поиск размера объекта в ТВМ и передает размер процедуре `Dispose`, которая освобождает правильное количество байт памяти.

Объект может иметь деструкторы даже в том случае, если все его методы статические.

Поэтому для динамических объектов всегда имеет смысл объявлять виртуальный деструкторы, даже и пустой, который нужен для нормальной работы процедуры `Dispose`:

```
destructor ИмяТипоОбъекта.Done; virtual;
 begin
 end;
```

Обработка ошибок при работе с динамическими объектами

При работе с динамическими объектами в программе могут возникать следующие нештатные ситуации:

- при размещении динамического экземпляра объекта в `Heap` при помощи указателя не хватает памяти для экземпляра объекта. Тогда указатель получает значение `nil`. Инициализация дальше не возможна и нужно было бы прекратить работу конструктора;
- успешно выполнилось выделение памяти для экземпляра объекта в динамической памяти, однако поля объекта – большие динамические массивы или другие динамические объекты, для выделения памяти под которые места не хватает.

Правильное размещение данных в динамической области контролируется системной стандартной функцией, имеющей заголовок

```
function HeapFunc (Size:Word) : Integer;    Far;
```

которая возвращает следующие коды завершения операций `New` и `GetMem`:

0: попытка выделить блок необходимого размера закончилась неудачно. Происходит аварийное завершение программы;

1: попытка выделить блок необходимого размера закончилась неудачно, но вместо ошибки процедуры `GetMem` и `New` возвращают указатель со значением `nil`;

2: попытка выделить блок необходимого размера закончилась удачно.

Turbo Pascal позволяет установить пользовательскую функцию обработки ошибок динамической памяти при помощи переменной `HeapError`, являющейся стандартной и не требующей описания в разделе переменных. Она содержит адрес стандартной функции обработки ошибок, которая может быть замещена на пользовательскую функцию обработки ошибок формата:

```
function HeapFunc (Size:Word) : Integer;    Far;
```

путем присвоения адреса пользовательской функции переменной `HeapError` таким образом:

```
HeapError := @HeapFunc
```

Данная возможность полезна при использовании конструкторов, если, например, при выделении памяти под динамические поля произошел сбой (не хватает памяти). Будет разумно, если в подобной ситуации конструктор отменит все предыдущие действия по выделению памяти для экземпляра объекта.

Для этой цели введена стандартная процедура `Fail` (неуспешно), которая может быть вызвана только из конструктора.

Вызов этой процедуры освобождает память, которая была выделена для экземпляра объекта еще до входа в конструктор, и возвращает в ссылке значение nil. Получение nil означает неудачное выделение памяти.

Ситуация, когда недостаточно памяти возможна и в случае статических объектов с динамическими полями. Так, при выделении памяти конструктором для динамических полей в куче может возникнуть ситуация нехватки памяти. Но поскольку объект статический, нельзя передать значение `nil` в ссылку – ее просто нет. Вместо этого предлагается использовать имя конструктора как логическую функцию.

Если внутри конструктора была вызвана процедура `Fail`, то вернется значение `true`, в остальных случаях – `false`. Подобным способом анализа можно пользоваться и для проверки работы наследственных конструкторов.

Напомним, что если при попытке поместить динамический экземпляр объекта свободной памяти будет недостаточно, то вызов расширенной процедуры `New` сгенерирует код фатальной ошибки выполнения `202`. А если переписать системную функцию `HeapFunc` таким образом:

```
function HeapFunc(Size:Word):Integer;  Far;
 begin
  HeapFunc := 1
 end;
```

чтобы она возвращала значение 1 во всех случаях, то в ссылочную переменную в случае ошибки вернется значение `nil`, а программа не прервется.

Далее эту ситуацию можно программно проанализировать и как-то отреагировать. Значит, если при запросе памяти для объекта процедурой

```
New(ИмяССылкиНаОбъект, ИмяКонструктара(параметры))
```

функция `HeapFunc` выдаст значение 1, то конструктор не будет выполняться, а в `ИмяССылкиНаОбъект` запишется `nil`.

Если же начинает выполняться тело конструктора, то для экземпляра объекта уже гарантированно выделено место.

Но сам конструктор также может выполнять действия по выделению памяти для динамических полей.

Опишем функцию `Check`, которую будем использовать при выделении памяти.

```
procedure Check (var P: Pointer; Size: Word);
 var
     OldHeapError: Pointer;
 begin
    OldHeapError := HeapError;
    HeapError    := Addr (HeapFunc);
    GetMem          (P, Size);
    HeapError    := OldHeapError;
 end;
```

Перед тем как использовать стандартные процедуры выделения динамической памяти `GetMem` или `New`, можно было бы проверить функцией `MaxAvail`, которая возвращает размер в байтах наибольшего непрерывного участка памяти в `Heap` области, имеется ли в `Heap` требуемая память.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 5
ООП. ПРАКТИКА ИСПОЛЬЗОВАНИЯ

Содержание темы

- Примеры учебных задач по ООП:
  - задача 1. Модуль по работе с векторами и матрицами,
  - задача 2. Модуль по решению некоторых задач линейной алгебры.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

Примеры учебных задач по ООП

Задача 1. Модуль по работе с векторами и матрицами

Технологией ООП разработать модуль по работе с векторами и матрицами, размещенными в динамической памяти, размеры которых становятся известными во время выполнения программы, для последующего использования их при решении задач линейной алгебры.

Алгоритм. При размещении вектора в динамической памяти используем тип – указатель на вектор, сделав следующие определения:

```
Item          = Real;
VectorType    = array [1..1] of Item;
VectorTypePtr = ^ VectorType;
```

и далее выделим память под вектор такого типа процедурой `GetMem`.

При размещении матрицы в динамической памяти используем тип – указатель на вектор указателей на строку матрицы:

```
MatrixType    = array [1..1] of VectorTypePtr;
MatrixTypePtr = ^ MatrixType;
```

В модуле `unit V_OOP` опишем два независимых объектных типа `Vector` и `Matrix` и соберем простейшие ресурсы по работе с вектором и матрицей, полями которых будут не только переменные типа `VectorType` и `MatrixType`, но и их текущие размеры:

- размещение вектора и матрицы в динамической памяти `(GetMemory);`

- освобождение вектора и матрицы из динамической памяти (FreeMemory);
- инициализация вектора и матрицы действительными случайными числами (Init_R);
- инициализация вектора и матрицы информацией из типизированных файлов (Init_File);
- распечатка вектора и матрицы (Show);
- обработка вектора и матрицы (Work).

Описание объектов по работе с массивами, размещенными в динамической памяти

```
unit V_OOP;
interface {$R-}
type
     Item          = Real;
     TFile         = Text;
     VectorType    = array[ 1..1]   of   Item;
     VectorTypePtr = ^VectorType;
     MatrixType    = array[1..1] of VectorTypePtr;
     MatrixTypePtr = ^MatrixType;
     Vector        = object
            V : VectorTypePtr;
            n : Word;
             procedure    Init_n(n_:Word);
             procedure    GetMemory;
             procedure    FreeMemory;
             procedure    Init_R;
             procedure    Init_File(name: String);
             procedure    Show;
                     end;
     Matrix        = object
            A : MatrixTypePtr;
            n, m : Word;
             procedure    Init(n_, m_ : Word);
             procedure    GetMemory;
             procedure    FreeMemory;
             procedure    Init_R;
             procedure    Init_File(name: String);
             procedure    Show;
                     end;
     implementation
```

```
procedure  Vector.Init_n;
  begin
   n := n_;
  end;
procedure  Vector.GetMemory;
  begin
   GetMem(V, n * SizeOf(Item));
  end;
procedure  Vector.FreeMemory;
  begin
   FreeMem(V, n * SizeOf(Item));
  end;
 procedure   Vector.Init_R;
  var  i : Word;
   begin
     for  i := 1  to  n  do
          V^[i] := 10 * Random - 5;
   end;
procedure   Vector.Init_File;
 var  i : Word;
      F : TFile;
  begin
      Assign(F, name);
      Reset (F);
      for  i := 1  to  n  do  Read(F,V^[i]);
      Close(F);
  end;
procedure   Vector.Show;
 var i : Integer;
  begin
   for  i := 1  to  n  do
     Write (V^[i]:8:2);
   Writeln;
   Writeln;
  end;
procedure  Matrix.Init;
  begin
   n := n_;
   m := m_;
  end;
procedure  Matrix.GetMemory;
 var  i : Word;
```

```
  begin
    GetMem(A, n * SizeOf(Pointer));
    for  i := 1  to n do
       GetMem ( A^[i], m * SizeOf(item) );
  end;
procedure  Matrix.FreeMemory;
 var  i : Word;
  begin
      for  i := 1  to n do
        FreeMem ( A^[i], m * SizeOf(item));
      FreeMem(A, n * SizeOf(Pointer));
  end;
 procedure   Matrix.Init_R;
  var  i, j : Word;
   begin
    for  i := 1  to n do
      for  j := 1  to  m do
         A^[i]^[j] := 10 * Random - 5;
   end;
 procedure   Matrix.Init_File;
  var  i, j : Word;
       F    : TFile;
   begin
     Assign(F, name);
     Reset(F);
     for  i := 1  to n do
      for  j := 1  to  m do
         Read(F, A^[i]^[j]);
     Close(F);
   end;
 procedure   Matrix.Show;
  var
     i, j : Integer;
   begin
     Writeln;
     for  i := 1  to n do
      begin
       for  j := 1  to  m do
         Write ( A^[i]^[j] :8:2 );
       Writeln;
      end;
     Writeln;
```

```
        end;
end.
```

Для отладки методов напишем следующую главную программу.

```
Program OOP00;
 uses CRT, V_OOP;
 var
    Vec     : Vector;
    Matr    : Matrix;
 {$R+}
 begin
        ClrScr;
        Vec.Init_n(6);
        Vec.GetMemory;
        Vec.Init_R;
        Vec.Show;
        Vec.Init_File('Tv.txt');
        Vec.Show;
        Readln;
        Matr.Init(5,4);
        Matr.GetMemory;
        Matr.Init_R;
        Matr.Show;
        Matr.Init_File('TM.txt');
        Matr.Show;
        Readln;
 end.
```

Задача 2. Модуль по решению некоторых задач линейной алгебры

Используя ресурсы модуля из предыдущей задачи, технологией ООП разработать модуль по решению некоторых задач линейной алгебры.

Алгоритм. Разработаем модуль `VLin_alg`, в котором подключим предыдущий модуль `V_OOP`. Опишем объектный тип `Lin_Alg`, согласно последующему объявлению, в котором в разделе `public` объявим методы для решения классических задач линейной алгебры, назначение которых видно из их названий. В разделе `private` объявим поля объектных и простых типов, позволяющие работать с векторами и матрицами разных размерностей. Еще объявим методы по их инициализации не только случайными числами, но и данными из соответствующих файлов. Далее объявим методы для получения произведения двух матриц, умножения матрицы на вектор, суммы двух векторов.

```
unit VLin_alg;
interface
uses V_OOP;
{$R-}
type
Lin_Alg = object
          public
  procedure Show_Mult_Matr_F(nameA1, nameA2 : String;
                             nA1_, mA1_, nA2_, mA2_:Word);
  procedure Show_Summ_Vect_F(nameV1,nameV2:String;
                             nV1_:Word);
  procedure Show_Mult_Matr_R(nA1_, mA1_,
                             nA2_, mA2_:Word);
  procedure Show_Summ_Vect_R(nV1_:Word);
  procedure Show_Mult_Matr_Vect_R(nA1_, mA1_:Word);
  procedure Show_Mult_Matr_Vect_F
           (nameA1, nameV1:String; nA1_, mA1_:Word);
          private
     OV1, OV2, OV3                : Vector;
     nV1, nV2, nV3                : Word;
     OA1, OA2, OA3                : Matrix;
     nA1, mA1, nA2, mA2, nA3, mA3 : Word;
 procedure Init_Vect_R(var V : Vector; nV   : Word );
 procedure Init_Matr_R(var A:Matrix; nA, mA : Word );
 procedure Init_Vect  (var V:Vector; nV     : Word );
 procedure Init_Matr  (var A:Matrix; nA, mA : Word );
 procedure Init_Vect_F(name:String;
                       var V:Vector;nV:Word);
 procedure Init_Matr_F(name:String; var A:Matrix;
                       nA,mA:Word);
 procedure Mult_Matr;
 procedure Mult_Matr_Vect;
 procedure Summ_Vect;
 end;

implementation

 procedure Lin_Alg.Show_Mult_Matr_Vect_R;
     begin
         nA1 := nA1_;
         mA1 := mA1_;
         Init_Matr_R ( OA1, nA1, mA1 );
```

```
        Init_Vect_R ( OV1, mA1 );
        Init_Vect   ( OV2, nA1 );
        Mult_Matr_Vect;
        OV2.Show;
        OA1.FreeMemory;
        OV1.FreeMemory;
        OV2.FreeMemory;
   end;

procedure  Lin_Alg.Show_Mult_Matr_Vect_F;
   begin
        nA1 := nA1_;
        mA1 := mA1_;
        Init_Matr_F ( nameA1, OA1, nA1, mA1 );
        Init_Vect_F ( nameV1, OV1, mA1 );
        Init_Vect   ( OV2, nA1 );
        Mult_Matr_Vect;
        OV2.Show;
        OA1.FreeMemory;
        OV1.FreeMemory;
        OV2.FreeMemory;
   end;

procedure Lin_Alg.Mult_Matr_Vect;
 var i, j, k : Word;
       t       : item;
   begin
   for i := 1  to  nA1 do
     begin
      t := 0;
      for  j := 1  to mA1 do
          t := t + OA1.A^[i]^[j] * OV1.V^[j];
      OV2.V^[i] := t;
     end;

   end;

procedure Lin_Alg.Mult_Matr;
 var i, j, k : Word;
       t       : item;
   begin
   for i := 1  to nA1 do
    for  j:= 1  to mA2 do
     begin
```

```
      t := 0;
      for  k := 1  to mA1 do
         t := t + OA1.A^[i]^[k] * OA2.A^[k]^[j];
      OA3.A^[i]^[j] := t;
     end;
   end;
procedure  Lin_Alg.Show_Mult_Matr_R;
   begin
        nA1 := nA1_;  mA1 := mA1_;
        nA2 := nA2_;  mA2 := mA2_;
        nA3 := nA1_;  mA3 := mA2_;
        Init_Matr_R ( OA1, nA1, mA1 );
        Init_Matr_R ( OA2, nA2, mA2 );
        Init_Matr   ( OA3, nA3, mA3 );
        Mult_Matr;
        OA3.Show;
        OA1.FreeMemory;
        OA2.FreeMemory;
        OA3.FreeMemory;
   end;
procedure  Lin_Alg.Show_Mult_Matr_F;
   begin
        nA1 := nA1_;  mA1 := mA1_;
        nA2 := nA2_;  mA2 := mA2_;
        nA3 := nA1_;  mA3 := mA2_;
        Init_Matr_F ( nameA1, OA1, nA1, mA1 );
        Init_Matr_F ( nameA2, OA2, nA2, mA2 );
        Init_Matr   ( OA3, nA3, mA3 );
        Mult_Matr;
        OA3.Show;
        OA1.FreeMemory;
        OA2.FreeMemory;
        OA3.FreeMemory;
   end;
procedure  Lin_Alg.Summ_Vect;
 var i : Word;
   begin
     for i := 1  to  nV1  do
       OV3.V^[i] := OV1.V^[i] + OV2.V^[i];
   end;
```

```
procedure  Lin_Alg.Show_Summ_Vect_R;
    begin
      nV1 := nV1_;
      nV2 := nV1_;
      nV3 := nV1_;
      Init_Vect_R (OV1, nV1);
      Init_Vect_R (OV2, nV2);
      Init_Vect   (OV3, nV3);
      Summ_Vect;
      OV3.Show;
      OV1.FreeMemory;
      OV2.FreeMemory;
      OV3.FreeMemory;
    end;
 procedure  Lin_Alg.Show_Summ_Vect_F;
    begin
      nV1 := nV1_;
      nV2 := nV1_;
      nV3 := nV1_;
      Init_Vect_F (nameV1, OV1, nV1);
      Init_Vect_F (nameV2, OV2, nV2);
      Init_Vect   (OV3, nV3);
      Summ_Vect;
      OV3.Show;
      OV1.FreeMemory;
      OV2.FreeMemory;
      OV3.FreeMemory;
    end;
 procedure Lin_Alg.Init_Vect_R;
   begin
    V.Init_n(nV);
    V.Getmemory;
    V.Init_R;
    V.Show;
   end;
 procedure Lin_Alg.Init_Matr_R;
   begin
    A.Init(nA, mA);
    A.Getmemory;
    A.Init_R;
    A.Show;
```

```
  end;
procedure Lin_Alg.Init_Vect;
  begin
   V.Init_n(nV);
   V.Getmemory;
  end;
procedure Lin_Alg.Init_Matr;
  begin
   A.Init(nA, mA);
   A.Getmemory;
  end;
procedure Lin_Alg.Init_Vect_F;
  begin
   V.Init_n(nV);
   V.Getmemory;
   V.Init_File(name);
   V.Show;
  end;
procedure Lin_Alg.Init_Matr_F;
  begin
   A.Init(nA, mA);
   A.Getmemory;
   A.Init_File(name);
   A.Show;
  end;
end.
```

В главной программе подключим предыдущий модуль и выполним модельные расчеты согласно следующей программе.

Проверка объектов по решению задач линейной алгебры

```
Program OOP11;
  uses CRT, VLin_Alg; {$R-}
  var
        D    : Lin_Alg;
  begin
      ClrScr;
      Writeln('random-chisla');
      Writeln('MULT Matrises');
      D.Show_Mult_Matr_R (5, 4, 4, 5);
      Readln;
```

```
      Writeln('ADD Vectors');
      D.Show_Summ_Vect_R (5);
      Readln;
      Writeln('Iz file''s');
      Writeln('MULT Matrises');
      D.Show_Mult_Matr_F('TM2.txt','TM1.txt',5,4,4,5);
      Readln;
      Writeln('ADD Vectors');
      D.Show_Summ_Vect_F ( 'TV1.txt', 'TV2.txt',5 );
      Readln;
      Writeln( 'Iz file''s' );
      Writeln( 'MULT Matrisa_Vector' );
      D.Show_Mult_Matr_Vect_F('TM1.txt',
                              'TV1.txt',5,4);
      Readln;
      Writeln( 'random-chisla');
      Writeln( 'MULT Matrisa_Vector');
      D.Show_Mult_Matr_Vect_R( 5, 4 );
      Readln;
  end.
```

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 6
МЕТОДЫ СОРТИРОВКИ ДАННЫХ

Содержание темы

- Сортировка данных:
  - оценка алгоритма сортировки,
  - свойства алгоритма и классификация.
- Сортировка массивов.
- Первый тип. Сортировка обменом.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

СОРТИРОВКА ДАННЫХ

Под *сортировкой* понимают процесс целенаправленной перестановки элементов данных в указанном порядке по возрастанию или убыванию согласно определенным линейным отношениям их порядка, таким как отношения «≥» (больше или равно) или «≤» (меньше или равно) для чисел.

Цель сортировки – облегчить потом *поиск* элемента в отсортированной (упорядоченной) совокупности элементов. Аналогом является поиск элемента в словаре, телефонном справочнике, ведомостях и др.

Сортировка – одна из распространенных задач в информатике. Алгоритмы сортировки являются одним из наиболее изученных направлений информатики. Сортировка является идеальным примером огромного разнообразия алгоритмов, выполняющих одну и ту же задачу.

Объемы информационных массивов непрерывно и стремительно растут, соответственно возрастают и требования к скорости сортировки, что обусловливает актуальность оптимизации используемых алгоритмов.

Реализация алгоритмов сортировки имеет большое количество разного рода ухищрений. На примере сортировки можно убедиться в необходимости сравнительного анализа алгоритмов, так как при помощи усложнения алгоритма можно получить существенное увеличение эффективности при сравнении с более простыми и очевидными алгоритмами. Во многих случаях определить характеристики выполнения сортировки довольно просто.

Существующие алгоритмы сортировки значительно различаются по уровню сложности, скорости, устойчивости, требованиям к памяти и другим параметрам. Однако практически каждый алгоритм оказывается наиболее удобным в какой-либо конкретной ситуации. Востребованными являются даже очень медленные алгоритмы, которые из-за своей простоты находят применение в образовательных целях.

Зависимость выбора алгоритма от структуры данных – явление довольно частое, и в случае сортировки она очень сильная, поэтому методы сортировки обычно разделяют на две категории:

- сортировка данных типа массив, сортировка данных типа связанные списки;
- сортировка последовательных файлов.

Эти два класса часто называют *внутренней* и *внешней* сортировками.

Внутренняя сортировка. Массивы и списки располагаются в оперативной памяти быстрого доступа к каждому элементу. Данные обычно упорядочиваются на том же месте без дополнительных затрат. В современных архитектурах персональных компьютеров широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с применяемыми алгоритмами кэширования и подкачки.

Внешняя сортировка оперирует запоминающими устройствами большого объёма, но не с произвольным доступом, а последовательным: в каждый момент времени можно считать или записать только элемент, следующий за текущим. Объём данных не позволяет им разместиться в оперативной памяти. Это накладывает некоторые дополнительные ограничения на алгоритм и приводит к специальным методам упорядочения, обычно использующим дополнительное дисковое пространство. Отметим, что доступ к данным во внешней памяти производится намного медленнее, чем операции с оперативной памятью.

Сортировка строк – это ранжирование данных строкового типа по алфавиту. Сортировка строк напоминает во всем сортировку массивов, так как доступ имеем к любому элементу строки, т. е. символу, который в памяти хранится своим кодом. Происходит сравнивание строковых данных по закону «≥» (больше или равно) или «≤» (меньше или равно). Следует учитывать, что применение сравнения к строкам, представляющим собой числа в естественной записи, выдаёт контр интуитивные результаты: например, «9» оказывается больше, чем «11», так как первый символ первой строки имеет большее значение, чем

первый символ второй строки. Для исправления этой проблемы алгоритм сортировки может преобразовывать сортируемые строки в числа и сортировать их как числа. Такой алгоритм называется «числовой сортировкой», а описанный ранее – «строковой сортировкой».

Сортировка элементов файла с прямым доступом (а не последовательным) также напоминает во всём сортировку массивов, так как доступ имеем к любому элементу, однако нужно помнить, что работа с файлами происходит при помощи буферов обмена, в который идет подкачка информации, и это накладывает свои негативные отпечатки.

Оценка алгоритма сортировки

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти:

- **время** – основной параметр, характеризующий быстродействие алгоритма, называемый также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах размера n входной последовательности A. Для типичного алгоритма хорошее поведение – это $O(n \log n)$ и плохое поведение – это $O(n^2)$. Идеальное поведение для упорядочения – $O(n)$;
- **память** — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы (так как всё это потребляет $O(1)$). Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к сортировкам на месте.

Свойства алгоритма и классификация

- **Устойчивость** (англ. *stability*). При использовании алгоритмов устойчивой (стабильной) сортировки при наличии в наборе данных нескольких равных элементов в отсортированном наборе они сохраняются в том же порядке, в котором были в исходном наборе. Таким образом, устойчивая сортировка не меняет относительный порядок сортируемых элементов, имеющих одинаковые ключи. Сохранение взаимного расположения равных элементов важно при сортировке по одному полю данных, состоящих из нескольких полей.
- **Естественность поведения** – эффективность метода при обработке уже упорядоченных или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

• **Использование операции сравнения**. Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях.

• **Алгоритмы, не основанные на сравнениях.** Как следует из самого их названия, такие алгоритмы совсем не используют сравнений сортируемых элементов.

• **Прочие алгоритмы сортировки.**

СОРТИРОВКА МАССИВОВ

Введём систему обозначений, которую будем использовать далее. Пусть нам дан массив – последовательность элементов $a_1, a_2, \ldots, a_n$.

Сортировка по возрастанию (убыванию) означает перестановку этих элементов в порядке $a_{k_1}, a_{k_2}, \ldots, a_{k_n}$ так, что при заданной функции упорядочения f справедливы отношения $f(a_{k_1}) \le f(a_{k_2}) \le \ldots \le f(a_{k_n})$ (или, наоборот, $f(a_{k_1}) \ge f(a_{k_2}) \ge \ldots \ge f(a_{k_n})$).

Обычно функция упорядочения не подсчитывается по какому-то специальному правилу, а присутствует в каждом элементе в виде явной компоненты (поля элемента), которую называют *ключом* элемента. Таким образом, для представления элемента a_i особенно хорошо подходит структура записи.

Для дальнейшего изложения материала определим тип `item` [`aitem`], который будет использоваться в алгоритмах сортировки:

```
const   n = 100;
type
      inf = record
    ... {описание полей записи без ключевого поля}
      end;
type
     item = record
         key   : Integer;  {описание ключевого поля}
         zmest : inf;      {«другие компоненты»   }
            end;
type
     index = 1..n;
```

«Другие компоненты» – это все существенные данные элемента. Поле `key` – ключ служит только для идентификации элементов. Но если мы говорим об алгоритмах сортировки, ключ для нас – единственная существенная компонента и нам нет необходимости сейчас определять

остальные. Тип ключа может быть любым типом, для которого заданы отношения всеобщего порядка «больше или равно» («меньше или равно»).

Сделаем еще такое определение:

```
var  A: array [1..n] of item;
```

Разберем подробно некоторые интересные алгоритмы сортировки. Полный обзор методов сортировки можно найти в книгах [2, 7–9] и др.

Основное требование к методам сортировки массивов – экономное использование памяти. Это значит, что сортировку элементов нужно выполнять на том же месте (*in situ*), и поэтому методы, которые пересылают элементы из массива A в массив B, для нас на данном этапе не представляют интереса.

Таким образом, выбирая метод сортировки и руководствуясь критериями экономии памяти, классификацию алгоритмов мы будем проводить в соответствии с их *эффективностью*, это значит *экономией* времени и *быстротой действия*. При этом мы будем подсчитывать *C* – *необходимое количество сравнений ключей* и *M* – *перенаправления* (*перестановку*) *элементов* (что отнимает много времени). Это будут функции от *n* – числа элементов, которые сортируются.

Исторически повелось, что методы, которые сортируют элементы *in situ*, можно разбить на три основных класса в зависимости от заложенного в их основе базового алгоритма:

- сортировка обменом;
- сортировка включением;
- сортировка выбором.

Рассмотрим методы сортировки массивов в соответствии с этой классификацией и сразу будем искать те подходы, которые позволят улучшать базовый алгоритм. Не снижая общности, сортировку элементов будем проводить в *порядке возрастания ключа*.

При различных методах сортировки интерес будут вызывать такие вопросы:

1) как ведет себя алгоритм в крайних ситуациях:
   - массив упорядочен в нужном порядке;
   - массив упорядочен в обратном порядке;
   - массив не упорядочен;
2) на каких массивах алгоритм дает количество действий:
   - минимальную;
   - максимальную;
3) чему равняется среднее количество действий.

Первый тип. Сортировка обменом

Первый метод. Сортировка простым обменом

Это один из простейших методов сортировки, который обычно входит в школьный курс информатики.

Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок упорядочения в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются n-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает – массив отсортирован.

Если выполнять проходы справа налево, то при каждом проходе алгоритма по внутреннему циклу очередной наименьший элемент массива ставится на своё место в начале массива рядом с предыдущим «наименьшим элементом», а наибольший элемент перемещается на одну позицию к концу массива.

Это *сортировка простым обменом*, или *«пузырьком»*. Минимальный элемент как бы всплывает (как *пузырёк*) в начале массива. Вместо поднятия «самого легкого» можно «топить» самый «тяжелый».

Сначала рассмотрим работу такого алгоритма на примере: A = {44, 55, 12, 42, 94, 18, 06, 67} (n = 8).

На Рис. 1 в строках отражены промежуточные состояния массива A при сортировке простым обменом (n = 8).

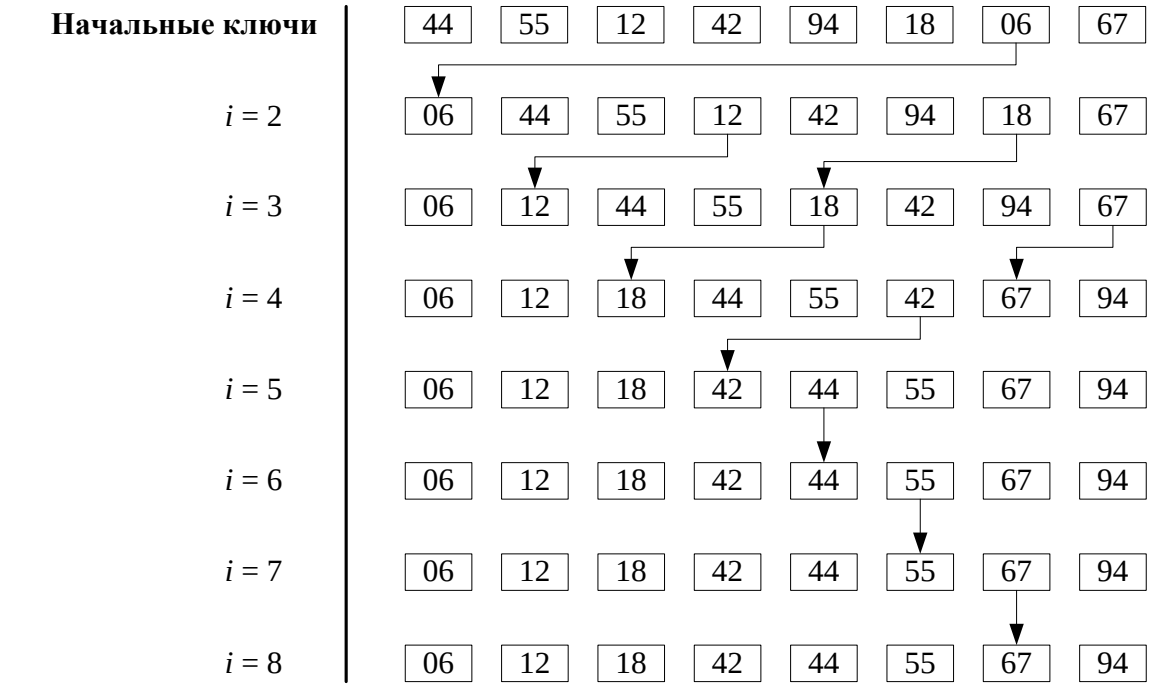


Рис. 1

Фрагмент кода, который реализует сортировку простым обменом, приведён в листинге.

```
procedure BubbleSort;      {проход справа налево}
var
   i, j : index;
   temp : item;
begin
for i := 2  to  n  do
    for j := n   downto   i do
        if a[j-1].key > a[j].key then
           begin
           temp   := a[j-1];
           a[j-1] := a[j];
           a[j]   := temp;
           end;
end; {BubbleSort}
```

Подсчитаем количество сравнений:

$$C = (n-1) + (n-2) + ... + 2 + 1 = \frac{n-1+1}{2}(n-1) = \frac{n^2 - n}{2} = O(n^2).$$

Количество обменов:

$$M_{\min} = 0;\ M_{\max} = \frac{3}{2}(n^2 - n);\ M = \frac{3}{4}(n^2 - n)\ \text{(среднее)}.$$

Таким образом, $M_{\max} = 3C$.

Можно заметить, что массив был отсортирован на каком-то промежуточном шаге ($i = 5$), но алгоритм построен так, что такое явление в нём не анализируется.

Второй метод. Сортировка методом пузырька с преждевременным выходом

Этот метод – оптимизация предыдущего метода: нужно запомнить, проводился ли на данном проходе какой-либо обмен, и если нет, то можно преждевременно закончить работу.

Сортировка простым обменом с преждевременным выходом

```
procedure BubbleSort_2;
var
      i, j : index;
      temp : item;
      flag : Boolean;
begin
```

```
 for i := 2  to  n do
   begin
    flag := true;
    for j := n  downto  i do
        if a[j - 1].key > a[j].key then
          begin
           temp    := a[j - 1];
           a[j-1]  := a[j];
           a[j]    := temp;
           flag    := false;
          end;
     if flag then exit;
    end
end; {BubbleSort2}
```

Третий метод. Оптимальная обменная сортировка

В этом методе сортировки нужно запомнить не только факт обмена, но и индекс последнего обмена. Очевидно, что все пары соседних элементов с индексами, меньшими чем этот индекс, уже расположены в нужном порядке. Поэтому проход можно заканчивать на этом индексе, вместо того, чтобы двигаться до конца. Программу алгоритма напишите самостоятельно.

Четвертый метод. Алгоритм шейкер-сортировки

Анализируя метод пузырьковой сортировки, можно отметить два обстоятельства:

- если при движении по части массива перестановки не происходят, то эта часть массива уже отсортирована и, следовательно, её можно исключить из рассмотрения;
- при движении от конца массива к началу минимальный элемент «всплывает» на первую позицию, а максимальный элемент сдвигается только на одну позицию вправо.

Например:

а) если A = {94, 06, 12, 18, 42, 44, 55, 67}, то просмотр слева направо сортирует массив за один проход, а справа налево – за семь;

б) если A = {12, 18, 42, 44, 55, 67, 94, 06}, то просмотр слева направо сортирует массив за семь проходов, а справа налево – за один.

Это приводит к следующей модификации метода пузырьковой сортировки:

- границы сортируемой части устанавливаются в месте последнего обмена на каждом проходе;
- массив просматривается поочередно справа налево и слева направо.

Шейкер-сортировка (с запоминанием места последнего обмена)

```
procedure ShakerSort;
var
    j, k, L, r : index;
          temp : item;
begin
 L := 2; r := n; k := n;
 repeat
{1}   for j := r downto L do                        {←}
          if a[j - 1].key > a[j].key then
             begin
             temp := a[j - 1]; a[j - 1] := a[j];
             a[j] := temp ; k := j; {последний обмен}
             end;
      L := k + 1;
 {2}  for j := L to r do                          {→}
          if a[j - 1].key > a[j].key then
             begin
             temp := a[j - 1]; a[j - 1] := a[j];
             a[j] := temp ; k := j;  {последний обмен}
             end;
          r := k - 1;
     {когда в циклах не было больше обмена, то
                 L = k + 1; r = k - 1 i L > r - выход!}
 until L > r;
end; {ShakerSort}
```

Схема работы алгоритма. Пусть $A=\{a_1, a_2, \ldots, a_n\}$ – исходный массив. Выполним первый проход справа налево для $j := n, n-1, \ldots, 2$ и запомним номер k последней перестановки. Заметим, что элементы $\{a_1, a_2, \ldots, a_k\}$ упорядочены. Обозначим $l := k+1$. Выполнив второй проход слева направо для $j := l, l+1, \ldots, n$ и запомнив номер k последней перестановки, заметим, что элементы $\{a_k, a_{k+1}, \ldots, a_n\}$ упорядочены. Обозначим $r := k-1$. Осталось упорядочить элементы $\{a_l, \ldots, a_k\}$. Для их сортировки повторяем чередование просмотров справа налево и слева направо (рис. 2).

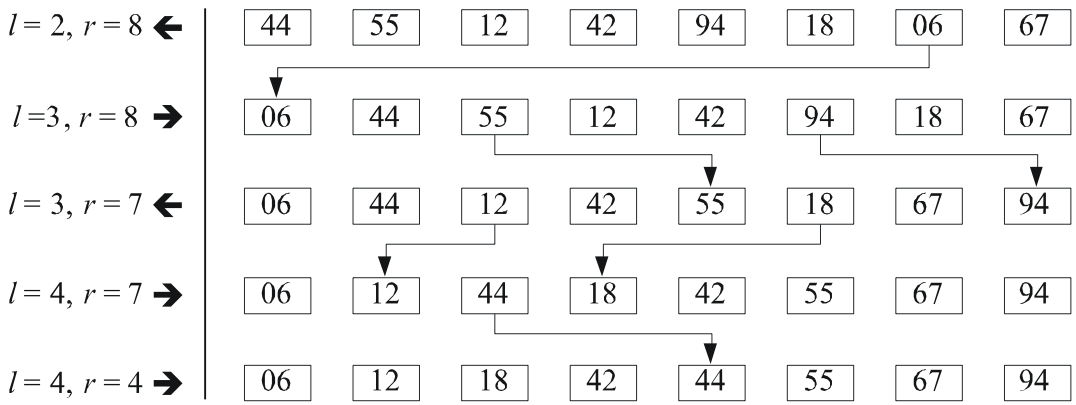


Рис. 2

Лучший случай для этой сортировки – отсортированный массив ($O(n)$), худший – отсортированный в обратном порядке ($O(n^2)$).

Отметим, что сортировка пузырьком массива A = {44, 55, 12, 42, 94, 18, 06, 67} требует $4 \cdot 7 = 28$ просмотров, пузырьком с преждевременным выходом – $7 + 6 + 5 + 4 + 3 = 25$ просмотров, шейкер-сортировкой – $7 + 6 + 5 + 4 + 1 = 23$ просмотра, а количество перестановок одинакова – $O(n^2)$.

Шейкер-сортировку (англ. *Cocktail sort*) в литературе еще называют сортировкой *перемешиванием*, или *двунаправленной*.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 7
МЕТОДЫ СОРТИРОВКИ МАССИВОВ

Содержание темы

- Первый тип. Обменная сортировка с разделением.
- Второй тип. Сортировка включением.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

Первый тип.
Пятый метод. Обменная сортировка с разделением

Сортировка с разделением – это ещё одно улучшение метода, основанного на принципе обмена. Алгоритм настолько хорош, что его автор английский информатик К. Хоар назвал *быстрой сортировкой* (англ. *quicksort*). Ранее рассмотренные методы обменивали соседние элементы. Быстрая сортировка находит элементы, расположенные не на своих местах: «тяжелый» – в «легком» конце, а «легкий» – в «тяжелом» конце, а потом обменивает их.

Быстрая сортировка относится к алгоритмам «разделяй и властвуй».

Алгоритм состоит из трёх шагов:

1. По вспомогательному алгоритму в массиве выбирается какой-то элемент x – *опорный элемент*.
2. Выполняется разбиение: перераспределение элементов в массиве таким образом, что элементы меньшие опорного помещаются перед ним, а большие или равные после.

Это происходит следующим образом. Движемся по совокупности элементов слева направо, пока не встретим элемент $a_i \geq x$, а затем – справа налево, пока не найдем элемент $a_j < x$. Если $i < j$, обменяем a_i с a_j. Продолжаем просмотр дальше от места $i+1$ до $j-1$. Выполняем такие поиски до тех пор, пока два просмотра не встретятся.

3. Рекурсивно применяются первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

Рассмотрим данный алгоритм на следующем примере. Пусть для массива A = {55, 12, 42, 94, 06, 18, 44, 67} опорный элемент x равен 42 (средний среди первых трех неравных). Получим такую последовательность перемещений:

A = {55, 12, 42, 94, 06, 18, 44, 67};
A = {18, 12, 42, 94, 06, 55, 44, 67};
A = {18, 12, 06, 94, 42, 55, 44, 67}.

Числа, меньшие чем 42 (42 – опорны элемент) Числа, не меньше чем 42

Просмотр закончился слева направо на элементе 94, а справа налево – на элементе 06. Они не обмениваются. Получили две части последовательности: в одной числа, меньшие, чем опорный элемент x, во второй числа, не меньшие чем x.

От выбора опорного элемента многое зависит. Заметим, что если бы мы выбрали элемент, стоящий в середине последовательности на 4-ом или 5-ом месте, а это соответственно наибольший и наименьший элементы массива, то на первом шаге разбиения получились бы неудачные подмассивы. Проследите этот момент самостоятельно.

Сейчас это разделение исходного массива представим в виде процедуры.

Вспомогательный алгоритм для быстрой сортировки

```
procedure Partition; {разделение}


var
     w, x : item;
begin
 i := 1;
 j := n;
         {          выбрать случайно x:
     можно посередине или средний среди неравных }
repeat
   while a[i].key < x.key do Inc(i);
   while a[j].key > x.key do Dec(j);
   if i <= j then
     begin
      w    := a[i];
      a[i] := a[j];
      a[j] := w;
      Inc(i);
```

```
      Dec(j)
     end;
 until i > j;
end; {разделение}
```

Разделив массив опорным элементом на две части, нужно сделать то же самое с каждой из них, затем с частями этих частей и так далее, пока каждая часть не будет содержать только один или ни одного элемента. Значит, процедура разделения будет сама себя вызвать. Получим следующую процедуру сортировки.

Быстрая сортировка

```
procedure QuickSort;
   procedure Partition(L, r : index);
   var temp, x : item;
   begin
       {Дополнить нахождением опорного элемента x}
    i := L; j := r;
    repeat
      while a[i].key < x.key do Inc(i);
      while a[j].key > x.key do Dec(j);
      if i <= j then
        begin
         temp := a[i]; a[i] := a[j];
         a[j] := temp; Inc(i); Dec(j)
        end;
    until i>j;
    if  L<j  then Partition(L, j);
    if  i<r  then Partition(i, r);
   end; {Sort}

begin
 Partition(1, n);
end; {QuickSort}
```

Существует нерекурсивный вариант алгоритма быстрой сортировки [7].

Анализ быстрой сортировки. Анализ методов сортировки – не всегда простая задача. Вероятностный подход позволил получить такие оценки:

- ожидаемое число обменов ~n / 6;
- общее количество сравнений есть $O(n \log n)$.

Надо отметить, что рекурсивная версия программы требует для рекурсии дополнительную память и время, но для больших n это оправдано.

Это один из известных универсальных алгоритмов сортировки массивов. Из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

Алгоритмы выбора опорного элемента

Для выполнения быстрой сортировки необходимо знать два алгоритма более низкого уровня: как выбирать опорный элемент и как наиболее эффективно переставить элементы последовательности таким образом, чтобы получить два набора элементов – со значениями, меньшими чем значения опорного элемента, и со значениями, большими чем значение опорного элемента.

Рассмотрим некоторые алгоритмы выбора опорного элемента на примере сортировки чисел.

Было бы идеально, если бы выбирался средний элемент последовательности, и тогда слева и справа относительно опорного элемента получалось примерно одинаковое количество элементов. Подсчет среднего элемента последовательности (или медианы последовательности) представляет собой достаточно сложный процесс, к тому же стандартный алгоритм его определения использует метод разделения быстрой сортировки, которую мы сейчас обсуждаем.

Худший случай имеет место, если в качестве опорного элемента выбирается элемент с максимальным или минимальным значением. Тогда фактически никакого существенного разделения на две части не будет и глубина рекурсии будет наибольшей.

Таким образом, после рассмотрения этих двух предельных случаев можно сказать, что желательно выбирать опорный элемент, который был бы как можно ближе к среднему элемента и как можно дальше от минимального и максимального.

В литературе встречаются следующие подходы к выбору опорного элемента последовательности $a_l, a_{l+1}, ..., a_r$ $(1 \le l \le r \le n)$:

- выбирается средний элемент последовательности $a_{(l+r)/2}$;
- выбирается средний элемент среди трех первых неравных;
- выбирается случайный элемент среди элементов последовательности, который потом обменивается со средним $k = l + \text{random}(r - l + 1);\ m = (l + r)\ \text{div}2;\ a_k \Leftrightarrow a_m$;

• выбирается элемент с индексом $k = (xl + yr)/(x + y)$, где x и y – произвольные два числа;

• выбирается средний элемент среди a_l, a_r, $a_{(l+r)/2}$. Затем элемент с наименьшим значением попадает в позицию l, средний – в середину последовательности, а элемент с наибольшим значением – в позицию r. Таким образом, при выборе опорного элемента размер частей последовательности сокращается на два элемента, так как уже известно, что они находятся в правильных частях последовательности относительно опорного элемента.

Замечание. Еще одна модификация метода быстрой сортировки. Если сначала количество элементов последовательности велико, то выполняется рекурсивный алгоритм разделения последовательности на две последовательности относительно опорного элемента. Однако если получили последовательность небольшого размера, то уже сортировать можно более медленными методами (метод пузырька, метод вставки, метод выбора).

Второй тип. Сортировка включением

Первый метод. Сортировка простым включением

Сортировка вставками (англ. *Insertion sort*) – алгоритм сортировки, в котором элементы массива $A = \{a_1, a_2, \ldots, a_n\}$ условно делятся на две части: отсортированную

$$\{a_1, a_2, \ldots, a_{i-1}\}$$

и неотсортированную

$$\{a_i, a_{i+1}, \ldots, a_n\}.$$

Сначала отсортированная часть состоит только из одного элемента $\{a_1\}$. На каждом шаге, начиная с $i = 2, 3, \ldots$, берут очередной элемент a_i и вставляют его на соответствующее место в отсортированную часть массива.

```
for i := 2 to n do
  begin
   x := a[i];  {вставить его на подходящее место}
  end;
```

В отсортированную часть массива $\{a_1, a_2, \ldots, a_{i-1}\}$ нужно вставить элемент x, не нарушив порядок упорядочивания. Сделать это возможно, используя просеивание, которое может закончиться при двух условиях:

• найден элемент a_j такой, что $a_j \le x \le a_{j+1}$, $1 \le j \le i-1$;

• достигли левого конца отсортированной части массива.

Цикл на просеивание можно усовершенствовать, если ввести фиктивный элемент («барьер») $a_0 = x$. Тогда окончание просеивания будет всегда только при выполнении первого условия, и второе условие можно не проверять.

Если уже ясно, куда вставить элемент x, тогда нужно освободить для него место, сдвигая элементы a_{j+1}, ..., a_{i-1} на шаг вправо. Однако можно уже при сравнении делать это перемещение. Заметим, что массив $A = \{a_1, a_2, ..., a_n\}$ с учетом необходимости хранения барьера нужно описать так:

```
var  a : array[0..n]  of  item;
```

Фрагмент кода, который реализует сортировку простым включением, приведенный в листинге.

Сортировка простым включением

```
procedure StraightInsertion;
var     i, j : index;
        x    : item;
begin
 for i := 2 to n do
    begin
     x      := a[i];
     a[0]   := x;
     j      := i - 1;
     while x.key < a[j].key do
         begin
          a[j + 1]   := a[j];
          j          := j - 1
         end;
     a[j + 1] := x
    end
end;
```

Анализ сортировки простым включением. Число сравнений ключей C_i на i-ом просеивания составляет самое большее $C_{\max} = i - 1$, а меньшее $C_{\min} = 1$, в среднем $C_c = i/2$. Число пересылок M_i (придание значений) равно $C_i + 2$ (учтем барьер). Поэтому M_i принимает наименьшее значение, если элементы изначально упорядочены, а наибольшее – когда элементы расположены в обратном порядке.

$$C_{\min} = \sum_{i=2}^{n} 1 = n - 1, \quad M_{\min} = 3(n-1), \quad C_c = \sum_{i=2}^{n} \frac{i}{2} = \frac{1}{4}(n+2)(n-1).$$

$$M_{\text{с}} = \sum_{i=2}^{n}(\frac{i}{2}+2) = \frac{1}{4}(n+2)(n-1)+2(n-1) = \frac{1}{4}(n-1)(n+10).$$

$$C_{\max} = \sum_{i=2}^{n}(i-1) = \frac{(n-1)n}{2}, \quad M_{\max} = \sum_{i=2}^{n}(i-1+2) = \frac{(n-1)(n+4)}{2}.$$

Второй метод. Сортировка простым включением с улучшением

Улучшение можно получить, если поиск места вставки элемента *x* выполнять *бинарными делениями*. Этим уменьшается количество сравнений. Затем сдвигаем часть элементов в массиве, освобождая место для вставки. Количество перемещений *M* в таком случае остается прежним, а пересылки занимают гораздо больше времени, чем сравнения. Программу напишите самостоятельно.

Третий метод. Сортировка Шелла

Сортировка Шелла была названа в честь её изобретателя – Дональда Шелла, который опубликовал этот алгоритм в 1959 году.

Сортировка Шелла (англ. *Shell sort*) – алгоритм сортировки, являющийся усовершенствованным вариантом сортировки вставками. Идея метода Шелла состоит в сравнении элементов, стоящих не только рядом, но и на определённом расстоянии друг от друга. Иными словами – это сортировка вставками с предварительными «грубыми» проходами. Аналогичный метод усовершенствования пузырьковой сортировки называется *сортировка расчёской* (*методом прочесывания*).

Это сортировка включениями с приращением, которое уменьшается. Рассмотрим ее на примере сортировки массива из восьми элементов $A = \{a_1, a_2, \ldots, a_8\}$.

Сначала отдельно группируются и сортируются все элементы, которые отстают друг от друга на четыре позиции. Таким образом сортируются четыре пары элементов массива: $\{a_1, a_5\}$; $\{a_2, a_6\}$; $\{a_3, a_7\}$; $\{a_4, a_8\}$.

Далее группируются и сортируются элементы, которые отстают друг от друга на две позиции. На этом этапе сортируются две группы элементов массива: $\{a_1, a_3, a_5, a_7\}$; $\{a_2, a_4, a_6, a_8\}$.

На последнем этапе выполняется обычная сортировка всех элементов массива.

На первый взгляд, добавились два лишних шага (сортировка групп элементов, которые отстают друг от друга на четыре и две позиции). Но

на каждом этапе сортируется относительно мало элементов или элементы уже довольно хорошо упорядочены.

Пример. Над исходным массивом $A = \{44, 55, 12, 42, 94, 18, 06, 67\}$ выполним сортировку каждого четвертого элемента (рис. 3).

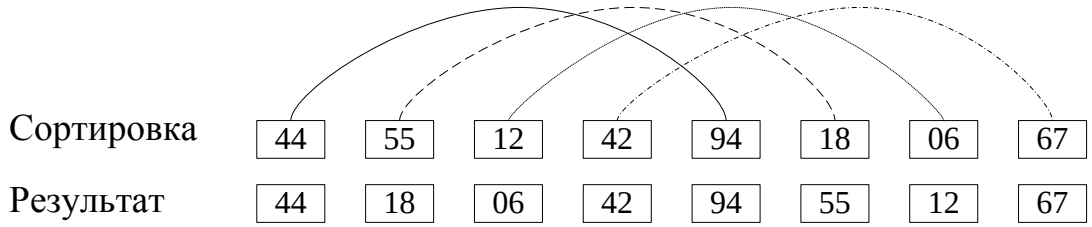


Рис. 3. Четверная сортировка

Результатом будет массив $A = \{44, 18, 06, 42, 94, 55, 12, 67\}$. Выполним в нем сортировку каждого второго элемента (рис. 4).

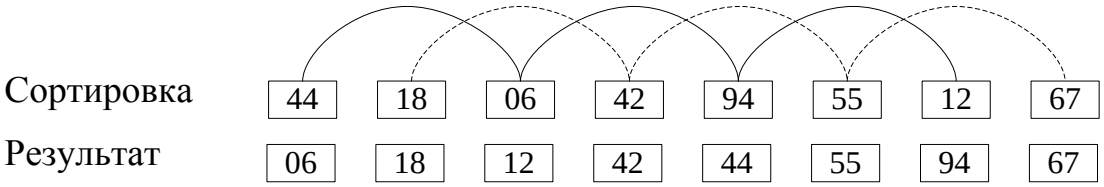


Рис. 4. Двойная сортировка

Получим следующий массив: $A = \{06, 18, 12, 42, 44, 55, 94, 67\}$. Выполним сортировку и получим полностью отсортированный массив $A = \{06, 12, 18, 42, 44, 55, 67, 94\}$ (рис. 5).

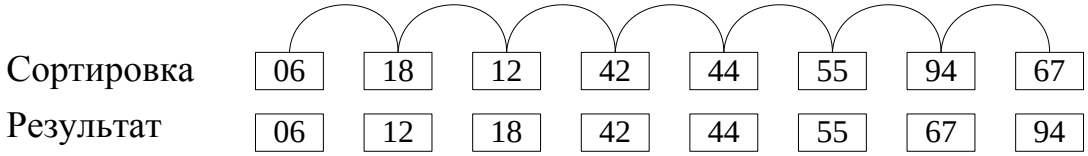


Рис. 5. Одинарная сортировка

Продвижение легкого элемента вперед идет быстрыми темпами.

Очевидно, что этот метод в итоге дает упорядоченный массив, и также ясно, что каждый проход будет использовать результаты предыдущего прохода, поскольку каждая i-ая сортировка объединяет две группы, отсортированные предыдущей сортировкой. Анализ показал, что допустимо произвольная последовательность приращений, важно чтобы последнее было равное 1, так как в худшем случае вся работа будет выполнена на последнем проходе и, как показывает практика, даже лучшие результаты получаются, если приращение не является степенью двойки.

Обозначим все t приращения через h_1, h_2, ..., h_t с условиями: $h_t = 1$, $h_{i+1} < h_i$. Каждая h_i-сортировка программируется как сортировка простыми включениями, при этом, для того, чтобы условие окончания поиска места включения была простой, используется барьер. Поэтому массив a нужно определить так:

```
var   a : array[-h1..n]  of  Integer;
```

Фрагмент кода, который реализует сортировку Шелла, приведенный в листинге.

Сортировка Шелла

```
procedure ShellSort;
const
      t = 4;
      h : array[1..t] of  Byte = (9, 5, 3, 1);
var
      i, j, k, s : Integer;
      x          : item;
      m          : 1..t;
begin
 for m := 1 to t do
   begin
    k := h[m];
    s := -k;
    for i := k + 1  to   n   do
       begin
        x := a[i];
        j := i - k;
        if s = 0  then  s := -k;
        s := s + 1;     a[s] := x;   {  барьер  }
        while x.key < a[j].key do
             begin
              a[j + k] := a[j];
              j        := j - k;
             end;
        a[j + k] := x;
       end
   end
end;
```

Анализ сортировки Шелла

До сих пор неизвестно, какая последовательность приращений дает наилучший результат, но оказалось, что лучше, если приращения не кратны друг другу. В противном случае каждый проход сортировки объединяет две цепочки, которые ранее не взаимодействовали, а лучше, чтобы взаимодействие происходило как можно чаще. Имеет место следующее утверждение: если *k*-отсортированная последовательность *i* сортируется, то она остается *k*-отсортированной.

Чем больше будут взаимодействовать последовательности, которые сортируются, тем быстрее получится результат. Например, в 1969 Г. Д. Кнут предложил последовательность приращений (используется в обратном порядке) 1, 4, 13, 40, 121, ... ($h_{i+1} = 3h_i + 1$), которая дает в среднем случае скорость $O(n^{5/4})$, в худшем случае – $O(n^{3/2})$. Известно, что самая быстрая последовательность, разработанная Р. Седжвиком: 1, 5, 19, 41, 109 и т. д., дает в среднем случае $O(n^{7/6})$, а в худшем – $O(n^{4/3})$. Хороший результат дает последовательность: 1, 3, 7, 15, 31, ..., где $h_{k-1} = 2 \cdot h_k + 1$ и $t = [\log_2 n] - 1$.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 8
МЕТОДЫ СОРТИРОВКИ МАССИВОВ ВЫБОРОМ

Содержание темы

- Третий тип. Сортировка выбором:
  - сортировка простым выбором,
  - сортировка при помощи дерева,
  - пирамидальная сортировка.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

ТРЕТИЙ ТИП. СОРТИРОВКА ВЫБОРОМ

Первый метод. Сортировка простым выбором

Шаги алгоритма:

- Среди *n* элементов выбирается элемент с наименьшим ключом и меняется местами с первым.
- Потом действие повторяется с остальными *n-1* элементами, *n-2* элементами и т д., пока не останется один последний элемент.

Рассмотрим работу такого алгоритма при сортировке массива
A = {44, 55, 12, 42, 94, 18, 06, 67}.

На Рис. 6 в строках показано промежуточное состояние массива при сортировке простым выбором.

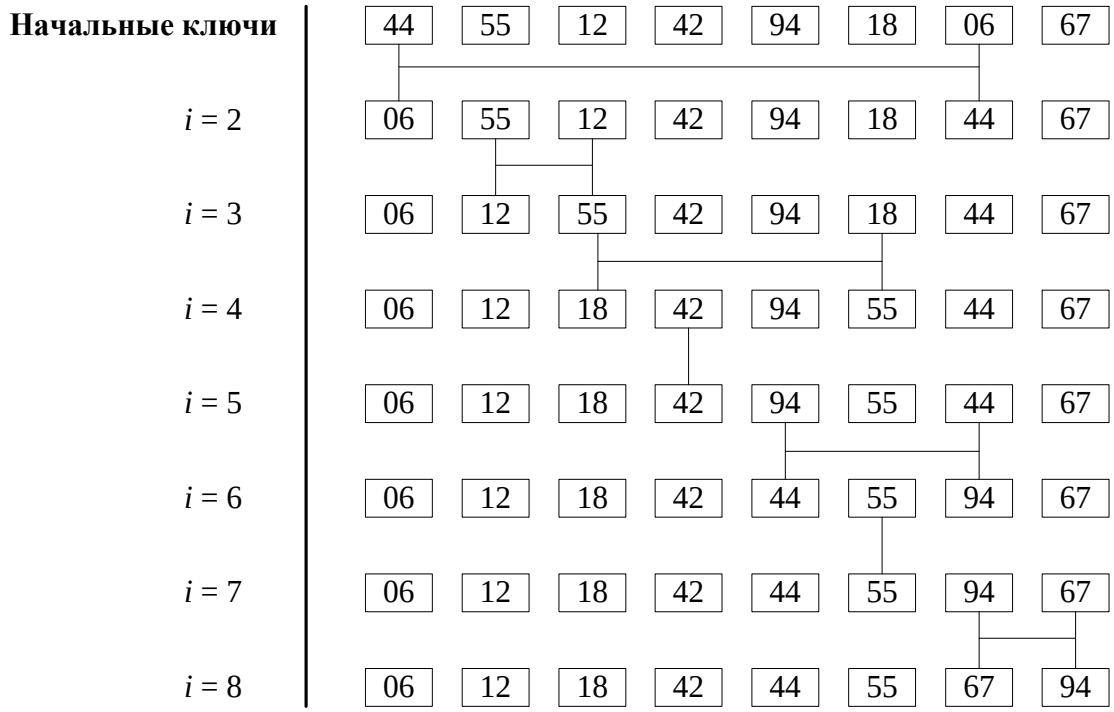


Рис. 6

Фрагмент кода, который реализует сортировку простым выбором, приведенный в листинге.

Сортировка простым выбором

```
Program StraightSelection;
var     i, j, k : index;
        x       : item;
begin
 for  i := 1  to  n - 1  do
   begin
    k := i;
    x := a[i];
    for  j := i + 1  to  n  do
         if  a[j].key < x.key  then
            begin
             k := j;
             x := a[j];
            end;
    a[k] := a[i];
    a[i] := x;
   end;
 end;
```

Анализ сортировки простым выбором. Сравнение:

$$C = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}.$$

Минимальное число пересылок: $M_{\min} = 3(n-1)$.

Максимальное число пересылок получается, если ключи расположены в обратном порядке:

$$M_{\max} = \sum_{i=1}^{n-1} (n-(i+1)+1) + 3(n-1) = n(n-1) - \frac{n}{2}(n-1) + 3(n-1).$$

Сортировка простым выбором (англ. *Straight Selection sort*) на массиве из n элементов имеет время выполнения в худшем, среднем и лучшем случае $O(n^2)$, если сравнения делаются за постоянное время.

Второй метод. Сортировка при помощи дерева

Улучшить предложенный выше метод сортировки простым выбором можно, если после каждого сравнения хранить больше информации об элементах. Например, на первом шаге при помощи $n/2$ сравнений можно определить наименьший ключ из каждой пары; на втором шаге при помощи $n/4$ сравнений из выбранных ключей можно определить наименьший из каждой пары и т. д. Получим наименьший из элементов в корне дерева (рис. 7).

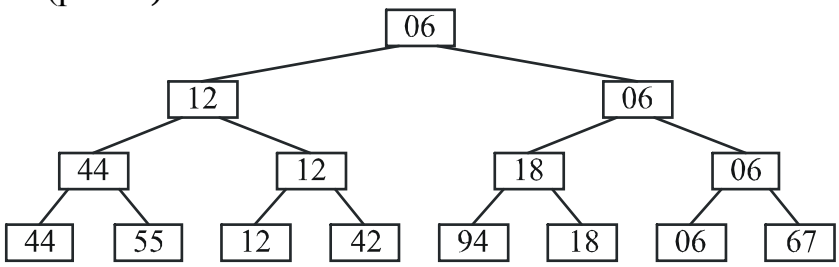


Рис. 7

На втором этапе мы спускаемся по пути, указанному наименьшим ключом, и исключаем его в исходной последовательности, заменяя, например, на ∞ (рис. 8).

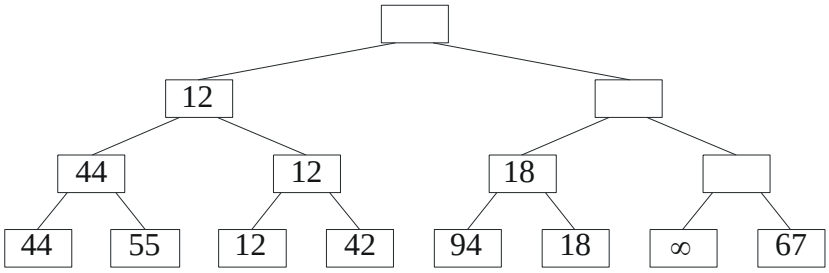


Рис. 8

Далее по такой схеме находим наименьший среди $n-1$ элементов (рис. 9).

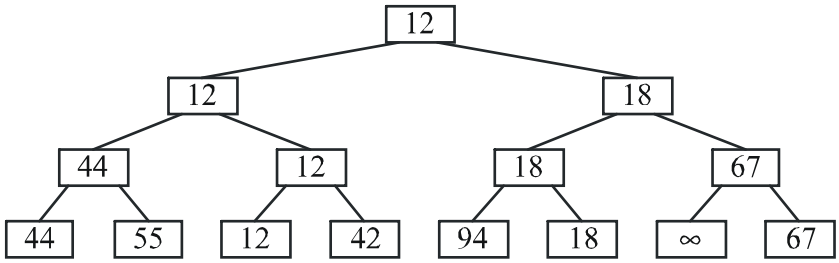


Рис. 9

После n таких шагов последовательность состоит только из ∞, значит, процесс сортировки закончился.

Каждый из n шагов требует $\log_2 n$ сравнение (так как уменьшение шло каждый раз в 2 раза), тогда за n шагов получаем $n\log_2 n$ сравнений. Сортировки же простым выбором требует n^2 сравнений.

Но реализация этого метода требует дополнительную память для хранения дерева – информации после каждого из сравнений. Нужно найти более удобный метод хранения дерева и каким-то образом освободиться от необходимости хранить ∞.

Оригинальный метод изобрел Дж. Вильямс и назвал его *пирамидальной сортировкой.*

Третий метод. Пирамидальная сортировка

Пирамида определяется как последовательность ключей $h_l, h_{l+1}, \ldots, h_r,$ такая, что выполняются условия

$$\begin{cases} h_i \le h_{2i}, \\ h_i \le h_{2i+1} \end{cases} \qquad (*)$$

для любого $i = l, ..., r/2.$

Отсюда $h_1 = \min(h_1, ..., h_n).$

Замечание. Условия

$$\begin{cases} h_i \ge h_{2i}, \\ h_i \ge h_{2i+1} \end{cases} \qquad (**)$$

для любого $i = l, ..., r/2,$ так же определяют пирамиду, у которой $h_1 = \max(h_1, ..., h_n).$

Фактически получается следующее дерево (рис. 10):

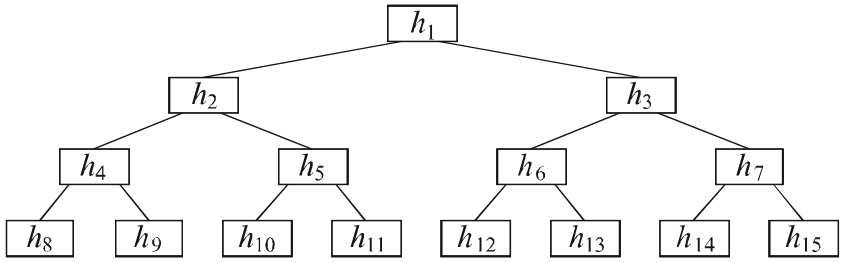


Рис. 10

Встают вопросы: как строить и сохранять пирамиду. Оказывается, это можно делать *in situ*.

Допустим, что дана пирамида с элементами $h_{l+1}, ..., h_r$ (l, r – зафиксированы), удовлетворяющими условию (*). Нужно добавить новый элемент x так, чтобы сформировать расширенную на этот элемент пирамиду $h_l, h_{l+1}, \ldots, h_r$.

Если новый элемент сначала добавить в вершину дерева, а затем «просеивать» по пути, на котором находятся меньшие по сравнению с ним элементы, которые одновременно поднимаются вверх, тогда сформируется новая (расширенная на один элемент) пирамида, так как при этом выполняется условие (*).

Рассмотрим пример.

Пусть имеется пирамида: h_2, ..., h_7: $h_2 = 42$, $h_3 = 6$, $h_4 = 55$, $h_5 = 94$, $h_6 = 18$, $h_7 = 12$.

Нужно добавить в нее новый элемент 44. Берем $h_1 = 44$ (рис.11).

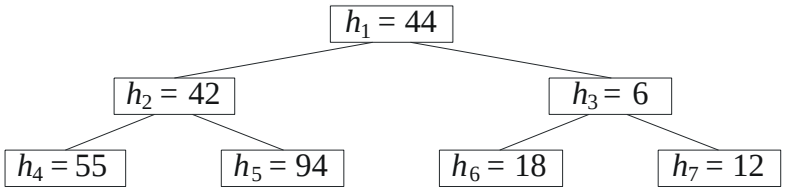


Рис.11

По условию (*) $h_1 = 44$ сравнивается с меньшим среди $h_2 = 42$ и $h_3 = 6$. Это элемент $h_3 = 6$, и он обменивается с $h_1 = 44$.

Получится: $h_1 = 6$, $h_3 = 44$.

Далее $h_3 = 44$ сравнивается с меньшим среди $h_6 = 18$ и $h_7 = 12$. Это $h_7 = 12$, и он обменивается с $h_3 = 44$.

Получилась расширенная пирамида, 42, 12, 55, 94, 18, 44 (рис.12).

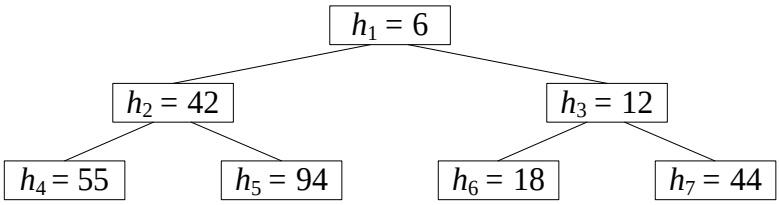


Рис.12

Основываясь на этом приеме просеивания, красивый способ построения пирамиды in situ предложил Р.В. Флойд. Рассмотрим этот способ.

Пусть задан массив $h_1, h_2, \ldots, h_n$. Ясно, что элементы $h_{n/2+1}, \ldots, h_n$ уже образуют пирамиду, потому что для $n/2+1 \le i \le n$ здесь не существует элементов с номерами 2i, 2i + 1. Фактически эти элементы используют как самый нижний ряд бинарного дерева (как на 10).

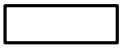
Первый этап. Начиная с элемента i = n / 2 с шагом от -1 до 1, будем просеивать через предыдущую пирамиду очередной h_i элемент. В конце этой процедуры в вершину пирамиды перенесется самый маленький элемент. Но последовательность еще не отсортирована. Сделаем следующее.

Второй этап. Обменяем первый элемент с последним и просеем его через пирамиду, не задевая последнего. Новый первый элемент обменяем с предпоследним и просеем его через пирамиду, не задевая двух последних. И так сделаем n-1 раз. Получим упорядоченный массив по убыванию значений.

Отобразим первый этап построения пирамиды для массива A = {44, 55, 12, 42, 94, 18, 06, 67}. Далее на 13-16 введены следующие обозначения (табл. 1).

Таблица 1

Обозначения на рисунках пирамидальной сортировки

Элемент	Обозначение
	Элемент массива h_i, i = 1, 2, ..., 8
	Линия, справа от которой элементы удовлетворяют определению пирамиды (*), слева - еще не просеянные элементы
	Элемент массива h_i, который просеивается
	Элементы массива h_{2i} и h_{2i+1}, с которыми сравнивается просеиваемый элемент h_i; наименьший из двух обведен штриховой линией, если он меньше чем элемент, который просеивается

Элемент	Обозначение
	Обмен элемента h_i, который просеивается, с наименьшим элементом из пары h_{2i} и h_{2i+1}

Элементы h_i, $i = 5, 6, 7, 8$, уже образуют пирамиду, поскольку в массиве A не существует элементов h_{2i} h_{2i+1}, а значит, условие (*) выполнено. Просеивание начнем с элемента $h_4 = 42$. Элемент $h_4 = 42$ сравнивается с $h_8 = 67$ (элемента h_9 не существует); поскольку $h_4 < h_8$, то h_4 остается на месте и элементы h_i, $i = 4, 5, 6, 7, 8$, уже образуют пирамиду (рис.13).

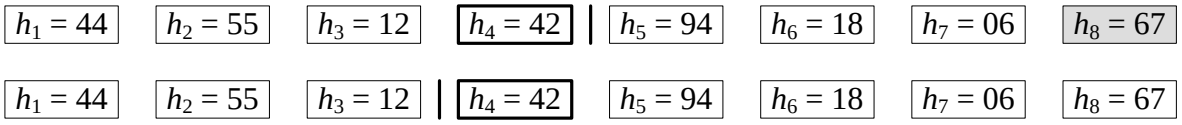


Рис.13

Просеиваем элемент $h_3 = 12$. Он сравнивается с меньшим среди $h_6 = 18$ и $h_7 = 06$; поскольку $h_3 > h_7$, то h_3 обменивается с h_7. В итоге элементы h_i, $i = 3, 4, 5, 6, 7, 8$, уже образуют пирамиду (рис.14).

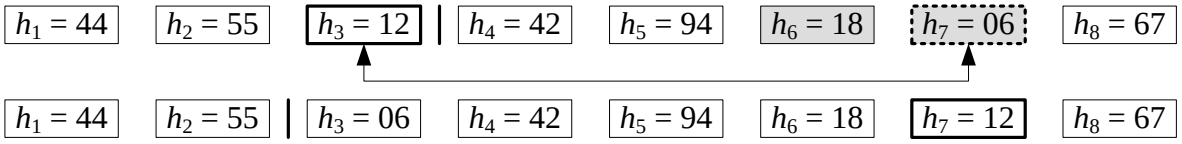


Рис.14

Просеиваем элемент $h_2 = 55$. Он сравнивается с меньшим среди $h_4 = 42$ и $h_5 = 94$; поскольку $h_2 > h_5$, то h_2 обменивается с h_5. Потом элемент $h_4 = 55$ сравнивается з $h_8 = 67$ (элемента h_9 не существует); поскольку $h_4 < h_8$, то h_4 остается на месте и элементы h_i, $i = 2, 3, 4, 5, 6, 7, 8$, уже образуют пирамиду (рис. 15).

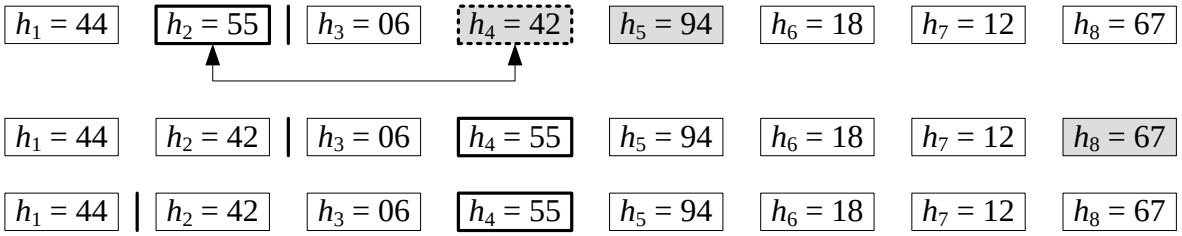


Рис.15

Просеиваем элемент $h_1 = 44$. Он сравнивается с меньшим среди $h_2 = 42$ и $h_3 = 06$; поскольку $h_1 > h_3$, то h_1 обменивается с h_3. Потом элемент $h_3 = 44$ сравнивается с меньшим среди $h_6 = 18$ и $h_7 = 12$; поскольку $h_3 > h_7$, то h_3 обменивается с h_7 и все элементы h_i, $i = 1, 2, 3, 4, 5, 6, 7, 8$, образуют пирамиду (рис. 16).

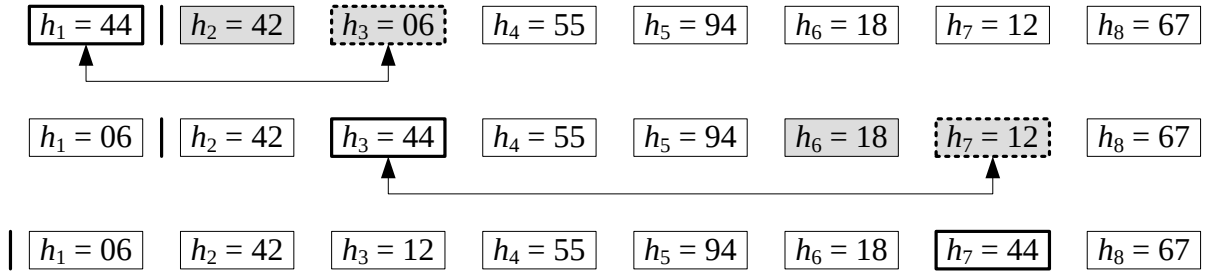


Рис. 16

Первый этап построения пирамиды для массива $A = \{44, 55, 12, 42, 94, 18, 06, 67\}$, который отражен на 13–16, компактно представлен на Рис. 17.

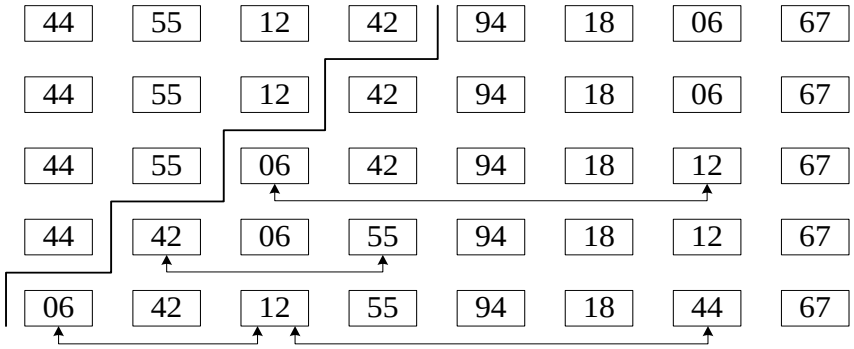


Рис. 17

Этим мы сформировали пирамиду, и наименьший элемент переместился наверх. Далее будем обменивать его согласно этапу 2. Обменяем первый элемент с последним и просеем его через пирамиду, не затрагивая последний (рис. 18).

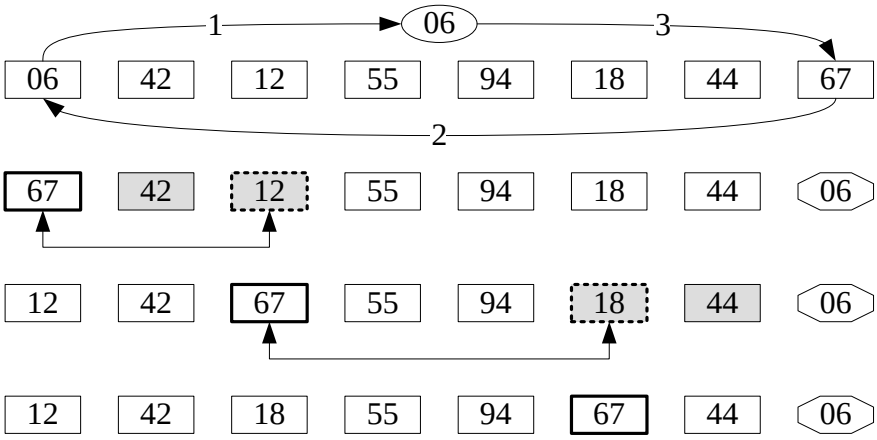


Рис. 18

Обозначения на 18–24 совпадают с приведенными в табл. 1. Дополнительно используются ⬭ – для вспомогательного элемента,

который используется для обмена первого элемента массива и последнего неотсортированного, ⬡ – для отсортированных элементов массива *A*.

Новый первый элемент обменяем с предпоследним и просеем его через пирамиду, не затрагивая двух последних (рис. 19).

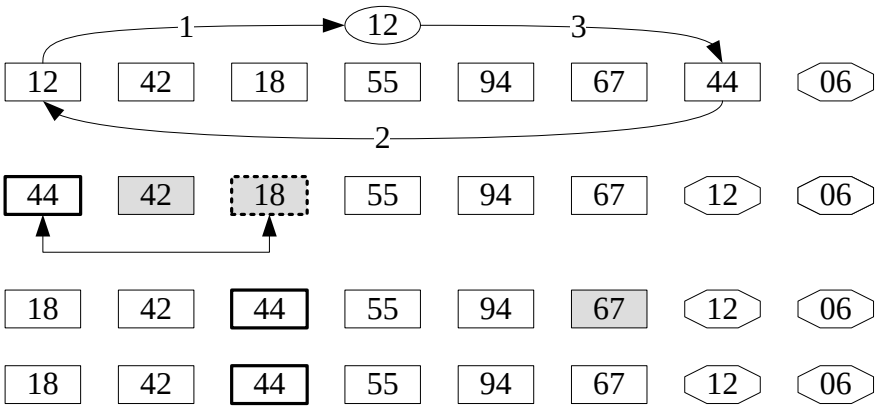


Рис. 19

Новый первый элемент обменяем с последним неотсортированным (с шестым) и просеем его через пирамиду, не затрагивая трех последних (рис. 20).

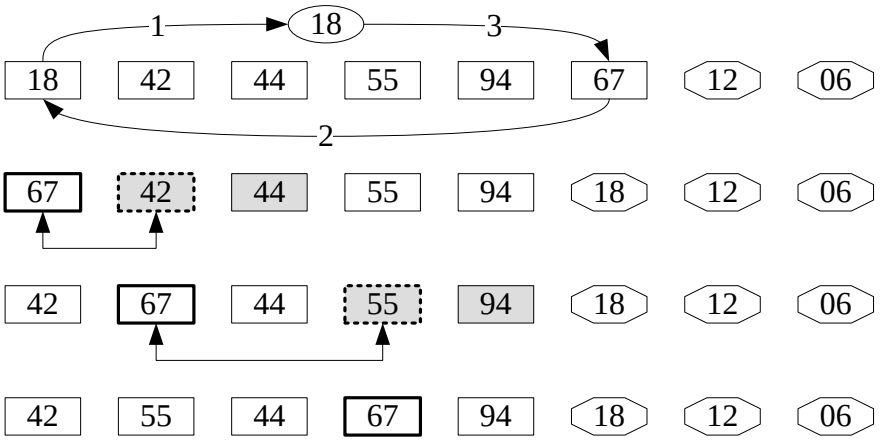


Рис. 20

Новый первый элемент обменяем с последним неотсортированным (с пятым) и просеем его через пирамиду, не затрагивая четырех последних (рис. 21).

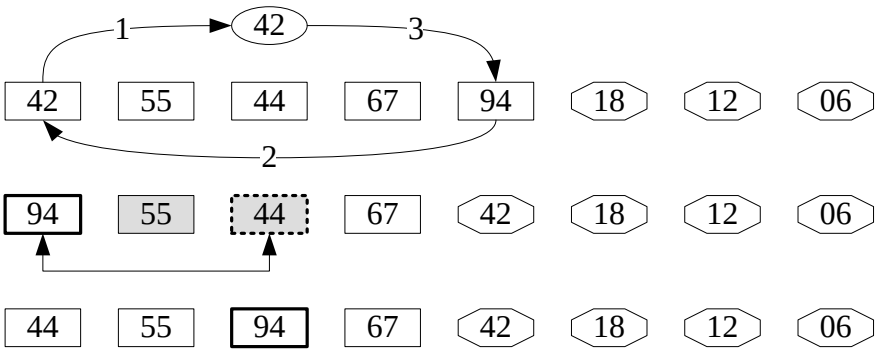


Рис. 21

Новый первый элемент обменяем с последним неотсортированным (с четвертым) и просеем его через пирамиду, не затрагивая пяти последних (рис. 22).

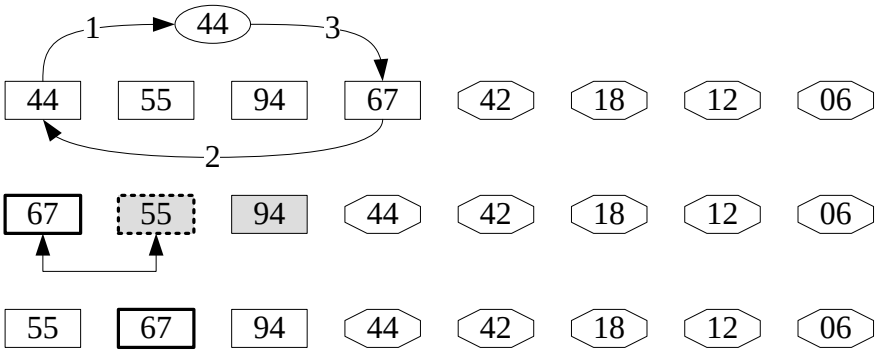


Рис. 22

Новый первый элемент обменяем с последним неотсортированным (с третьим) и просеем его через пирамиду, не затрагивая шести последних (рис. 23).

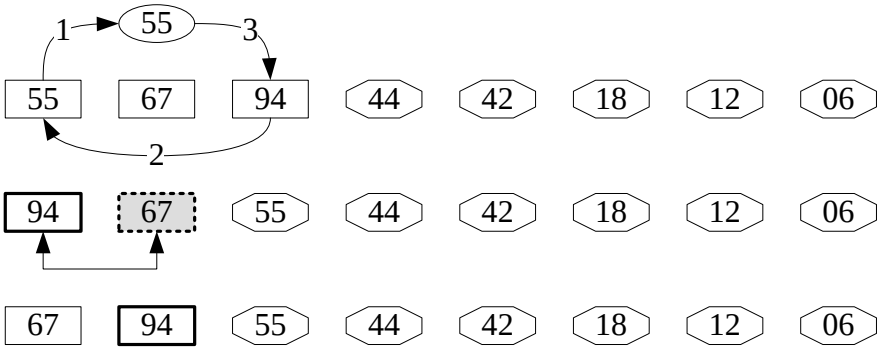


Рис. 23

Новый первый элемент обменяем с последним неотсортированным (со вторым) и получим отсортированный массив (рис. 24).

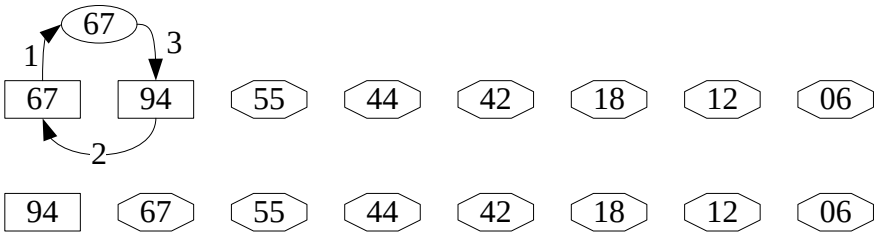


Рис. 24

Получили упорядоченный массив по убыванию значений. Значит, чтобы получить упорядочение в другую сторону, нужно изменить условие (*) на условие (**).

Напишем процедуру просеивания. Заданы: a_l, $a_{l+1}, ..., a_r$. Просеем a_l через пирамиду $a_{l+1}, ..., a_r$.

```
procedure Sift(L,r:index);
var
      j,i : index;
        x : item;
begin
 x := a[L]; i := L; j := 2 * i;
 if (j < r) and (a[j + 1].key < a[j].key)
 then j := j + 1;
             {чтобы дальше сравнивать с меньшим}
 while (j <= r) and (x > a[j].key) do
                    {  подъем вверх}
     begin
      a[i] := a[j]; i := j; j := i * 2;
      if (j < r) and (a[j + 1].key < a[j].key)
      then j := j + 1;
     end;
 a[i] : = x;
end; {Sift}
```

Мы умеем строить пирамиду из n элементов, при этом наименьший элемент перемещается алгоритмом в a_1.

Напишем фрагмент программы построения пирамиды.

```
L := (n div 2) + 1;
 while (L > 1) do
      begin
       L := L - 1;
```

```
        Sift(L, n);
      end;
```

Для того чтобы получить не только частичную, но и полную упорядоченность среди элементов, нужно сделать *n* шагов, причем тот элемент, который переносится каждый раз на вершину дерева, есть очередной наименьший элемент. Мы их будем отправлять в конец последовательности и не рассматривать в очередной проход, а очередной последний в свою очередь просеивать через пирамиду.

Получим такой алгоритм сортировки по убыванию элементов.

Пирамидальная сортировка

```
procedure HeapSort;
var
      L, r : index;
      y    : item;

  procedure  Sift(L, r: index);
  var
      j, i : index;
      x    : item;
  begin
   x := a[L]; i := L; j := 2 * i;
   if (j < r) and (a[j + 1].key < a[j].key)
   then j := j + 1;
   while (j <= r) and (x > a[j].key) do
     begin
      a[i] := a[j];
      i    := j;
      j    := i * 2;
      if (j<r) and (a[j+1].key<a[j].key) then Inc(j);
     end;
   a[i] := x;
  end; {Sift}

begin
 L := (n div 2) + 1;
 while (L > 1) do
      begin
       L := L - 1;
       Sift(L,n);
      end;                  {построили пирамиду}
```

```
    r := n;
    while r >1 do
        begin
         y    := a[1];
         a[1] := a[r];
         a[r] := y;
         r    := r - 1;
         Sift(1,r);
        end;
  end;  {HeapSort}
```

Анализ пирамидальной сортировки. Пирамидальную сортировку (англ. *Heapsort*, «Сортировка кучей») рекомендуется использовать для больших n. Среднее число пересылок приблизительно равно $1{,}5 \cdot n \cdot \log_2 n$.

Недостатки алгоритма:

- Неустойчив – для обеспечения устойчивости нужно расширять ключ.
- На почти отсортированных массивах работает столь же долго, как и на хаотических данных.
- На одном шаге выборку приходится делать хаотично по всей длине массива – поэтому алгоритм плохо сочетается с кэшированием и подкачкой памяти.
- Методу требуется «мгновенный» прямой доступ; не работает на связанных списках и других структурах памяти последовательного доступа.

Из-за сложности алгоритма выигрыш получается только на больших n. На небольших n (до нескольких тысяч) быстрее сортировка Шелла.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 9
НЕКОТОРЫЕ ДРУГИЕ МЕТОДЫ СОРТИРОВОК

Содержание темы

- «Карманная» сортировка:
- Сортировка элементов методом подсчета.
- Introsort или интроспективная сортировка и некоторые другие.
- Сравнение методов сортировки массивов *in situ*.
- Классификация алгоритмов сортировки.
- Эволюция способов и алгоритмов сортировки.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

«Карманная» сортировка

Это сортировка массивов за линейное время.

Если исходная последовательность чисел $a_1, a_2, ..., a_n$ является последовательностью натуральных чисел, и они не повторяются, то можно предложить для их сортировки по возрастанию следующий подход. Берется вспомогательный массив $b_1, \ldots, b_n$ и организуется такой цикл:

```
for i := 1 to n do
    B[A[i]] := A[i];
```

Этот код подсчитывает, где в массиве *B* должен находиться элемент `A[i]`, и помещает его туда. Весь этот цикл требует времени порядка $O(n)$ и работает корректно только тогда, когда значения всех ключей различны и являются натуральными числами из интервала от 1 до n.

Если же не разрешается использовать вспомогательный массив *B*, то организовывается такой цикл:

```
for i := 1 to n do
    while A[i] <> i do
         begin
          обмен A[i] и A[A[i]]
         end;
```

Этот алгоритм также требует времени порядка $O(n)$. По очереди проверяем элементы $a_1, a_2, \ldots, a_n$. Если в a_i стоит число $j \neq i$, то меняются местами элементы a_i и a_j. Если после этой перестановки новое число в a_i имеет значение $k \neq i$, то совершается перестановка a_i и a_k элемента. Каждая перестановка смещает хотя бы один элемент в нужное место.

Сортировка элементов методом подсчета

Это сортировка массивов за линейное время.

Каждый элемент сравнивается с остальными элементами. Если какой-то элемент больше, чем k элементов этого же массива, то после упорядочения он должен занимать $(k + 1)$ место. Значит, необходимо сравнивать попарно все элементы и подсчитывать в отдельном массиве, сколько из них меньше за каждый текущий элемент. После этого необходимо переставить элементы в соответствии с вычисленными местами.

Introsort или интроспективная сортировка

Introsort или **интроспективная сортировка** – алгоритм сортировки, предложенный Дэвидом Мюссером в 1997 году. Он использует быструю сортировку и переключается на пирамидальную сортировку, когда глубина рекурсии превысит некоторый заранее установленный уровень (например, логарифм от числа сортируемых элементов). Этот подход сочетает в себе достоинства обоих методов с худшим случаем $O(n \log n)$ и быстродействием, сравнимым с быстрой сортировкой. Так как оба алгоритма используют сравнения, этот алгоритм также принадлежит классу сортировок на основе сравнений.

Мюссер выяснил, что на худшем наборе данных для алгоритма быстрой сортировки «медиана из трёх» (рассматривался массив из 100 тысяч элементов) *introsort* работает примерно в 200 раз быстрее.

Timsort

Timsort – гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием, опубликованный в 2002 году Тимом Петерсом.

Основная идея алгоритма в том, что в реальном мире сортируемые массивы данных часто содержат в себе упорядоченные подмассивы. На таких данных Timsort существенно быстрее многих алгоритмов сортировки.

Основная идея алгоритма:

• По специальному алгоритму входной массив разделяется на подмассивы.

• Каждый подмассив сортируется сортировкой вставками.

• Отсортированные подмассивы собираются в единый массив с помощью модифицированной сортировки слиянием.

Принципиальные особенности алгоритма в деталях, а именно в алгоритме разделения и модификации сортировки слиянием.

Терпеливая сортировка

Терпеливая сортировка (англ. *patience sorting*) – алгоритм сортировки с худшей сложностью $O(n \log n)$. Позволяет также вычислить длину наибольшей возрастающей подпоследовательности данного массива. Алгоритм назван по одному из названий карточной игры "Солитёр" – "Patience".

Алгоритм:

Имеем массив *A,* элементы которого нужно отсортировать по возрастанию. Разложим элементы массива по стопкам: для того чтобы положить элемент в стопку, требуется выполнение условия – новый элемент меньше элемента, лежащего на вершине стопки; либо создадим новую стопку справа и сделаем наш элемент её вершиной. Используем жадную стратегию: каждый элемент кладётся в самую левую стопку из возможных, если же таковой нет, справа от существующих стопок создаётся новая. Для получения отсортированного массива сначала построим массив стопок, затем выполним n шагов (здесь и далее нумерация шагов начинается с единицы): на i-м шаге выберем из всех вершин стопок наименьшую, извлечём её и запишем в массив B на i-ю позицию.

Плавная сортировка

Плавная сортировка (англ. *Smoothsort*) – алгоритм сортировки выбором, разновидность пирамидальной сортировки, разработанная Э. Дейкстрой в 1981 году. Как и пирамидальная сортировка, имеет сложность в худшем случае равную $O(n \log n)$. Преимущество плавной сортировки в том, что её сложность приближается к $O(n)$, если входные данные частично отсортированы, в то время как у пирамидальной сортировки сложность всегда одна, независимо от состояния входных данных.

Общее представление:

Как и в пирамидальной сортировке, в массиве накапливается куча из данных, которые затем сортируются путём непрерывного удаления максимума из кучи. В отличие от пирамидальной сортировки, здесь используется не двоичная куча, а специальная, полученная с помощью чисел Леонардо. Куча состоит из последовательности куч, размеры которых равны одному из чисел Леонардо, а корни хранятся в порядке возрастания. Преимущества таких специальных куч перед двоичными состоят в том, что если последовательность отсортирована, её создание и разрушение займёт $O(n)$ времени, что будет быстрее.

Гномья сортировка

Гномья сортировка (англ. *Gnome sort*) – алгоритм сортировки, похожий на сортировку вставками, но в отличие от последней перед вставкой на нужное место происходит серия обменов, как в сортировке пузырьком. Название происходит от предполагаемого поведения садовых гномов при сортировке линии садовых горшков.

Параллельная сортировка Бэтчера

Параллельно в первой и во второй половинах выполняется сортировка простым обменом, а затем идет слияние отсортированных массивов.

Сортировка методом прочесывания

В сортировке пузырьком или шейкер-сортировке, когда сравниваются два элемента, промежуток (расстояние друг от друга) равен 1. Основная идея метода прочесывания в том, чтобы первоначально брать достаточно большое расстояние между сравниваемыми элементами и по мере упорядочивания массива сужать это расстояние вплоть до минимального. Таким образом, мы как бы причёсываем массив, постепенно разглаживая на всё более аккуратные пряди. Сортировка также основана на этой идее, но она является модификацией сортировки вставками, а не сортировки пузырьком.

Сортировка методом прочесывания или расчёской (англ. *comb sort*) – это довольно упрощённый алгоритм сортировки, изначально спроектированный В. Добосевичем в 1980 г.

Позднее он был переоткрыт и популяризован в статье Стивена Лэйси и Ричарда Бокса в 1991 г.

Сортировка расчёской улучшает сортировку пузырьком, и конкурирует с алгоритмами, подобными быстрой сортировке. Основная идея – устранить черепах, т. е. маленькие значения в конце списка, которые крайне замедляют сортировку пузырьком (кролики, большие значения в начале списка, не представляют проблемы для сортировки пузырьком).

Сравнение методов сортировки массивов *in situ*

Мы получили следующие алгоритмы.

I. Обменные сортировки.
   1. Метод простого обмена.
   2. Метод простого обмена и выход, если нет обмена.
   3. Метод простого обмена не с начала, а от последнего обмена.
   4. Шейкер-сортировка.
   5. Быстрая сортировка.
   6. Гномья.
   7. Расчёской.

II. Сортировка вставками.
   1. Сортировка простыми вставками.
   2. Сортировка простыми вставками с бинарным поиском.
   3. Сортировка Шелла.

III. Сортировка выбором.
   1. Сортировка простым выбором.
   2. Пирамидальная сортировка.
   3. Плавная.

IV. Сортировка слиянием.

V. Без сравнений.
   1. Подсчётом.
   2. Поразрядная.
   3. Блочная.

VI. Гибридные.
   1. Introsort.
   2. Timsort.

Попробуем сравнить их эффективность. Для усовершенствованных методов нет сколько-нибудь осмысленных простых и точных оценок (формул). Существенно, что для сортировки Шелла вычислительные затраты составляют $O(n^{1,2})$, а для *QuickSort* (быстрой сортировки) и

HeapSort (пирамидальной) – $O(n \log_2 n)$. Эксперимент показывает, что *Quicksort* лучше в 2-3 раза по сравнению с *HeapSort*. Эти оценки позволяют разбить алгоритмы на методы продуктивности порядке $O(n^2)$ и на усложнённые порядка $O(n \log n)$.

Для сравнения алгоритмов сортировки удобно использовать матричный метод, который предполагает сравнение по двум параметрам. Так, в зависимости от устойчивости и скорости алгоритмы сортировки можно разделить на шесть групп.

	$<n^2$	**n^2**	**$> n^2$**
Устойчивые	• Сортировка с помощью двоичного дерева. • Сортировка слиянием. • Сортировка методом Тима Петерса.	• Сортировка методом простого обмена. • Сортировка методом простого обмена и выход, если нет обмена. • Сортировка методом простого обмена не с начала, а от последнего обмена. • Шейкер-сортировка. • Сортировка вставками. • Сортировка выбором.	• Гномья сортировка.
Неустойчивые	• Сортировка Шелла. • Сортировка расчёской. • Пирамидальная сортировка. • Быстрая сортировка. • Интроспективная сортировка. • Терпеливая сортировка.	• Сортировка выбором.	

Классификация алгоритмов сортировки

Алгоритмы устойчивой сортировки

- Сортировка пузырьком (англ. *Bubble sort*).
- Сортировка перемешиванием (англ. *Cocktail sort*).
- Сортировка вставками (англ. *Insertion sort*).
- Гномья сортировка (англ. *Gnome sort*); первоначально опубликована под названием «глупая сортировка» (англ. *stupid sort*) за простоту реализации.
- Сортировка слиянием (англ. *Merge sort*).

- Сортировка с помощью двоичного дерева (англ. *Tree sort*).
- Сортировка Timsort (англ. *Timsort*).
- Сортировка подсчётом (англ. *Counting sort*).

Алгоритмы неустойчивой сортировки

- Сортировка выбором (англ. *Selection sort*).
- Сортировка расчёской (англ. *Comb sort*).
- Сортировка Шелла (англ. *Shell sort*).
- Пирамидальная сортировка (англ. *Heapsort*) .
- Плавная сортировка (англ. *Smoothsort*).
- Быстрая сортировка (англ. *Quicksort*).
- Интроспективная сортировка (англ. *Introsort*).
- Терпеливая сортировка (англ. *Patience sorting*).

Эволюция способов и алгоритмов сортировки

Выделяют следующие пять этапов в эволюции способов и алгоритмов машинной сортировки данных в массивах.

Первый этап начался в 1870 году и длился до начала 1940-х годов. Его ознаменовал переход от ручной сортировки к сортировке с помощью статистических табуляторов. При этом использовался алгоритм поразрядной сортировки.

Второй этап – с начала 1940-х годов до середины 1950-х. На смену счетно-перфорационным машинам пришли ЭВМ первого поколения, для которых был разработан ряд новых алгоритмов сортировки.

Третий этап начался в середине 1950-х годов и продолжался до середины 1970-х. Активное развитие алгоритмов сортировки началось с разработкой ЭВМ второго поколения (увеличилась производительности компьютеров до 30 тысяч операций в секунду, резко уменьшились их габариты и стоимость). Разработка первых языков программирования высокого уровня (Фортран, Алгол, Кобол) создало предпосылки для ускорения написания программ для ЭВМ.

Четвертый этап продолжался с середины 1970-х до середины 1990-х годов. Появление вычислительных центров, объединяющих мощности отдельных вычислительных машин и позволяющих работать с разделением времени потребовало разработки новых алгоритмов сортировки и модификации существующих. Началось исследование задач

сортировки в классе параллельных алгоритмов, были достигнуты значительные успехи в увеличении скорости сортировки за счет повышения эффективности уже известных к тому времени алгоритмов путем их доработки или комбинирования.

Пятый этап начался с середины 1990-х годов и продолжается по настоящее время. Особую актуальность получило исследование задач сортировки на частично упорядоченных множествах. Актуальность этих задач объясняется появлением и широким распространением компьютеров на сверхсложных микропроцессорах с параллельно-векторной структурой, а также высокоэффективных сетевых компьютерных систем.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 10
СОРТИРОВКА ПОСЛЕДОВАТЕЛЬНЫХ ФАЙЛОВ

Содержание темы

- Сортировка последовательных файлов.
- Простое слияние.
- Естественное слияние.
- Сбалансированное многоленточное слияние.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

СОРТИРОВКА ПОСЛЕДОВАТЕЛЬНЫХ ФАЙЛОВ

Простое слияние

Когда происходит сортировка массивов в оперативной памяти, тогда имеется доступ к любому элементу массива. Что нельзя сказать в случае последовательного файла, так как в каждый момент времени существует доступ только к одному элементу. Такое ограничение строгое по сравнению с возможностями, которые дает массив, и поэтому здесь приходится применять другие методы сортировки. Основной метод – сортировка слиянием. Слияние означает объединение двух (или более) упорядоченных последовательностей в одну упорядоченную последовательность.

Рассмотрим один из методов сортировки слиянием – *простое слияние*.

1. Последовательность *A* разбивается на две половины – *B* и *C*.

2. Последовательности *B* и *C* сливаются при помощи объединения отдельных элементов в упорядоченные пары.

3. Полученной последовательности дается имя *A*, и повторяются шаги 1 и 2; на этот раз упорядоченные пары сливаются в упорядоченые четверки.

4. Предыдущие шаги повторяются: четверки сливаются в восьмерки, и весь процесс продолжается до тех пор, пока не будет упорядочена вся последовательность, так как длина последовательностей, которые сливаются, каждый раз удваивается.

Рассмотрим этот алгоритм на нашем примере:

44	55	12	42	|	94	18	06	67	{пополам и пары}
44	94	18	55	|	06	12	42	67	{пополам и четверки}
06	12	44	94	|	18	42	55	67	{пополам и восьмерки}
06	12	18	42		44	55	67	94	{все}

Получили 3 прохода (элементов $n = 8 = 2^3$).

Для выполнения сортировки требуются три последовательности, поэтому процесс называется *трехленточным слиянием*. Исторически последовательные файлы хранились на магнитных лентах.

Далее можно улучшать этот алгоритм и получать различные модификации.

Каждый проход состоит из фазы разделения и фазы слияния. Фазы разделения не относятся к сортировке непосредственно, потому что это просто переписывание элементов; в каком-то смысле они непродуктивны, хотя и составляют половину всех операций перезаписи.

Их предложили уничтожить, объединив фазы разделения и фазы слияния так: результат слияния сразу распределяется на две ленты, которые на следующем проходе будут входными. Это уже однофазное, или сбалансированное, слияние. Оно требует вдвое меньше операций перезаписи, но это достигается за счет использования четвертой ленты.

Анализ сортировки слиянием. Поскольку на каждом проходе p длина подпоследовательностей, которые объединяются слиянием, удваивается и сортировка заканчивается, как только $p >= n$, то понадобится $\log_2 n$ проходов. По определению при каждом проходе все множество из n элементов копируется только один раз, значит, общее количество пересылок равна $M = [\log_2 n] \cdot n$.

Количество сравнений примерно такое же, так как при копировании остатка последовательности (если есть остаток) сравнение не проводится.

Этот алгоритм хорошо переводится и на сортировку массива, который будет просматриваться строго последовательно.

Естественное слияние

В случае простого слияния ничего не выигрывается, когда данные уже частично отсортированы. Фактически можно было бы сразу сливать какие-либо упорядоченные подпоследовательности длин m и n в одну последовательность из $m + n$ элементов.

Метод сортировки, при котором каждый раз сливаются две самые длинные возможные подпоследовательности, называется *естественным слиянием.*

Упорядоченную подпоследовательность назовем *серией*. Это такие элементы a_i, a_2, ..., a_j, которые удовлетворяют условиям:

$$a_{i-1} > a_i; \quad a_k <= a_{k+1}, \quad k = i, \ldots, j-1; \quad a_j > a_{j+1}.$$

Значит, сортировка естественным слиянием сливает не последовательности фиксированной, ранее заданной длины, а максимальные серии. Таким образом, на каждом проходе общее число серий уменьшается вдвое, и число необходимых пересылок элементов в худшем случае $[\log_2 n] \cdot n$, а в обычном случае меньше.

Сбалансированное многоленточное слияние

Затраты на последовательную сортировку пропорциональны числу проходов, так как по определению на каждом проходе происходит переписывание всего множества данных. Один из способов уменьшить это число – распределять серии более чем на две ленты. Слияние r серий, которые поровну распределены на N лентах, дает в результате последовательность из r/N серий.

Здесь мы имеем массив файлов. За второй проход число серий уменьшается до r/N^2, и после k проходов остается r/N^k отрезков. Все они каждый раз распределяются на N лент.

Таким образом, общее число проходов при сортировке n элементов N-путевым слиянием $k = [\log_N n]$. Поскольку на каждом проходе происходит N операций перезаписи, то общее число операций перезаписи в худшем случае будет $M = [\log_N n] \cdot n$.

На Рис. 25 покажем наиболее известные алгоритмы внешних сортировок.

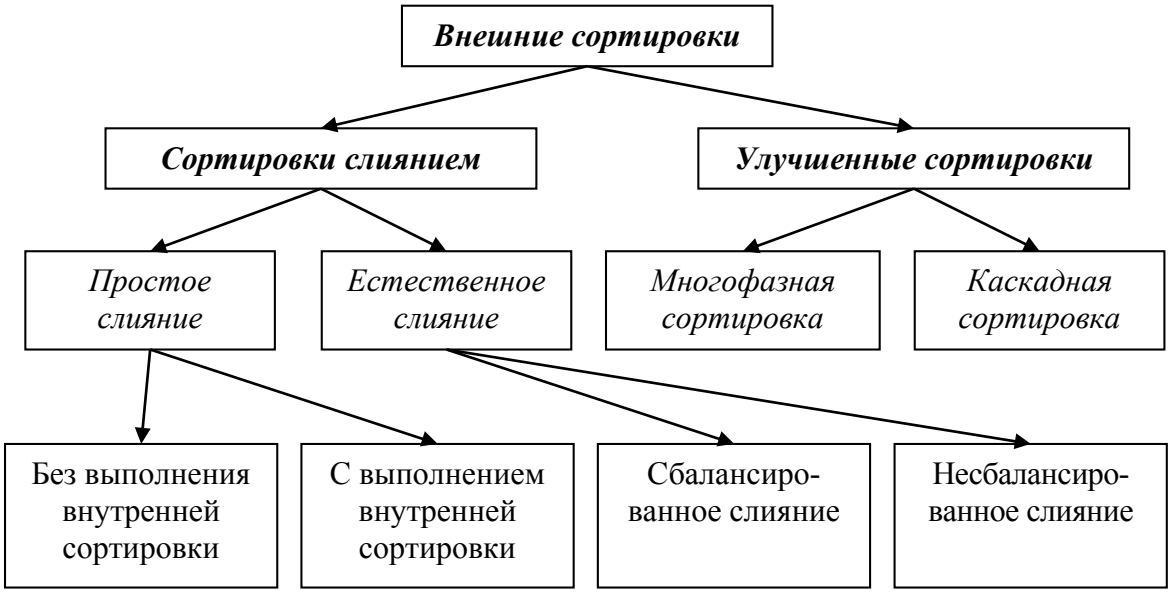


Рис. 25

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 11
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Содержание темы

- Абстрактные типы данных.
- Общие сведения о динамических структурах данных.
- Задача о считалочке.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

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

В модели АТД операндами операторов могут быть не только данные из данного АТД, но и других типов: стандартных типов языка программирования или из других АТД. Результат действия оператора также может иметь тип, отличающийся от типов данной модели АТД. Будем думать, что существует хотя бы один операнд или результат любого оператора, который имеет тип данных из модели АТД.

Заметим, что процедуры можно рассматривать как обобщение понятия оператора. В отличие от ограниченных по своим возможностям встроенных операторов языка программирования (сложения, умножения и т д.), при помощи процедур программист может создавать собственные операторы и применять их к операндам разных типов, не только базовых.

Примером такой процедуры-оператора может служить процедура перемножения матриц.

Если поместить (инкапсулировать) в отдельный модуль все операторы, отвечающие за определенный аспект функционирования программы, то их можно не только использовать в других программах, но и заменять в случае возникновения проблем при эксплуатации программы.

АТД можно рассматривать как обобщение простых типов данных (целых, действительных и т д.), также как процедура является обобщением простых операторов.

АТД инкапсулирует типы данных в том смысле, что определение типа и все операторы, которые выполняются над данными этого типа, содержатся в одном разделе программы.

Для АТД используются структуры данных, которые представляют собой набор переменных, объединенных определенным образом.

В качестве примера АТД можно взять файловый тип и множество операций, связанных с ним.

К основным АТД обычно относят наиболее общие абстрактные типы данных: списки (последовательности элементов); два специальных случая списков – стеки, где элементы добавляются и уничтожаются только на одном конце списка, и очереди, когда элементы добавляются на одном конце, а уничтожаются на втором; деревья, графы.

ОБЩИЕ СВЕДЕНИЯ О ДИНАМИЧЕСКИХ СТРУКТУРАХ ДАННЫХ

Между объектами материального мира, поведение которых моделируют программы, существуют разнообразные, постоянно изменяющиеся, связи. Одни объекты могут исчезать, другие – появляться. Понятно, что, когда в программе мы хотим моделировать группы с переменным числом объектов, между которыми связи могут быть изменчивыми, тогда нужны языковые средства для налаживания, изменения и расторжения связей между объектами, которые моделируется, а также для *рождения* и *уничтожения* объектов. Для этой цели в языке Pascal служат динамические переменные и указатели.

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

Динамические структуры данных появились тогда, когда использование регулярных структур (массивы, строки) начало тормозить написание программ несовершенным аппаратом.

Работа с массивами имеет свои преимущества и недостатки. Отметим их.

Преимущества:

- массивы помогают объединять совокупности сведений в осмысленные группы;
- элементы массива отличаются друг от друга только индексами;
- использование индексов обеспечивает непосредственный доступ к любому элементу массива;
- индексация позволяет проводить автоматическую, быструю, эффективную обработку элементов массива. Обработка – это инициализация, модификация, поиск и др.

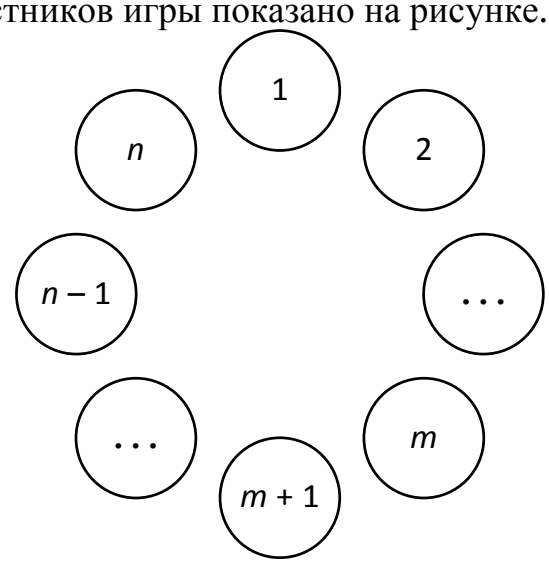
К *недостаткам* можно отнести операцию изменения чередования элементов массива, удаления или добавления элементов массива.

Значит, если какие-либо элементы нужно добавлять или удалять в совокупности, то такие абстрактные структуры данных плохо ложатся на массивы.

ЗАДАЧА О СЧИТАЛОЧКЕ

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-ый человек и выводится из круга. После этого, начиная со следующего, снова отсчитывается m-ый человек и выводится из круга. Это повторяется $n-1$ раз. Победитель – тот, кто останется последним. Найти его номер.

Расстановка участников игры показано на рисунке.



Обсудим возможные варианты решения задачи.

Вариант 1. Используем массив на n человек и присвоим каждому элементу массива значение, равное значению его индекса.

Запрограммируем проход по элементам массива на $n-1$ раз так, что, сумев попасть на последний элемент, мы переходим на первый.

Получив при просмотре очередной элемент массива, который нужно вывести из игры (m-й по счету), можно поступить двояко:

1) заменить значение элемента, который выводится, на некоторую оценку (например, отрицательное число или логическое значение), чтобы при дальнейшем просмотре этот элемент не участвовал в игре;
2) вычеркнуть элемент из массива, подтягивая следующие элементы на один шаг вперед.

Элемент, который останется последним, и даст решение этой задачи.

С какими препятствиями нам придется встретиться при программировании этих подходов?

В первом случае нужно отметить выбывшие элементы, при очередном просмотре пропускать отмеченные элементы и после последнего элемента переходить на первый.

Рассмотрим при $n = 7$, $m = 5$ подробнее работу первого варианта.

В первой строке таблицы разместим индексы элементов, во втором – признак наличия элемента, в третьем – признак удаления элемента и то, на каком проходе он удаляется.

1	2	3	4	5	6	7
true	true	true	true	true	true	true
false(6)	false(3)	false(2)	false(4)	false(1)	true	false(5)

С элементом массива связан индекс. Нас интересует индекс неотмеченного элемента. Ответом будет 6. Программу при желании напишите самостоятельно.

Во втором случае нужно своевременно делать перестановку элементов и изменять количество имеющихся элементов.

Очевидно, что второй вариант можно запрограммировать рекурсией, так как количество элементов в массиве уменьшается, но вычеркивать нужно, как и ранее.

Однако, чтобы свести задачу к первоначальной (предыдущей), требуется перестановка элементов массива, чтобы отсчет начинать с первого.

Тогда получится следующая программа.

```
Program Recursia_Schitalochka;
const
       n = 7;
       m = 5;
type
       TItem = Integer;
       TMas  = array[1..n] of TItem;
var    a      : TMas;
       kol, i : Integer;
 function  Last(n,m : Integer; a : TMas) : Integer;

   function  Rec(n,m:Integer) : Integer;

     procedure Zryx;
     var i,j : Integer;
         m1  : Integer;
         z   : TItem;
     begin           {procedure Zryx}
      m1 := m;
      if m > n then  m1 := (m - 1) mod n + 1;
        for i := 1 to n - m1 do
           begin
            z := a[n];
            for j := n - 1 downto 1 do
                a[j + 1] := a[j];
            a[1] := z;
          end;
     end; {procedure Zryx}

  begin             {function Rec}
   if n = 1 then Rec := a[n]
   else
      begin
       if n <> m then Zryx;
       Rec := Rec(n - 1, m);
      end;
  end; {function Rec}

 begin {function Last}
  Last := Rec(n, m);
 end;{function Last}
```

```
begin
 for i := 1 to n do a[i] := i;
 Writeln('зачеркивая ',m, '-го из ', n,
        ' участников, останется: ', Last(n, m, a));
 Readln;
end.
```

Мы знаем, что рекурсия требует много дополнительных ресурсов, и тут пришлось довольно часто выполнять перестановки элементов вспомогательного массива, поэтому рассмотрим другие подходы.

Эти препятствия отнимают время и создают определенные трудности при программировании.

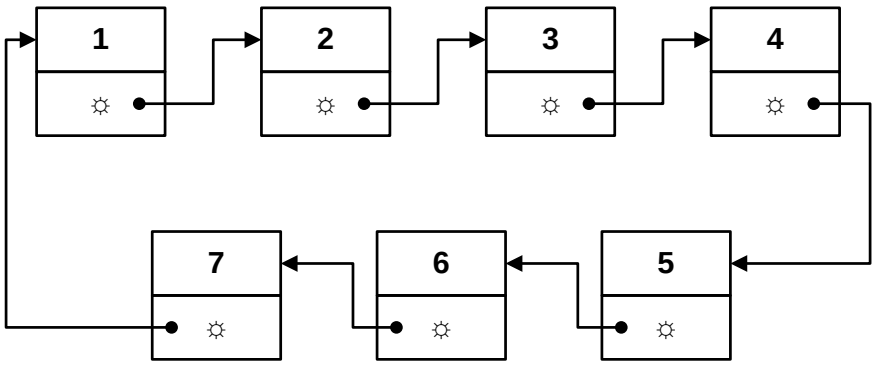
Вариант 2. Уже на этой задаче видно, что нужно иметь какие-то новые структуры данных, которые позволяли бы проще устанавливать и разрывать связи между элементами во время выполнения программы. А это уже динамические структуры данных и, следовательно, нужно использовать указатели, значениями которых являются адреса динамических переменных.

Введем элемент, объединяющий два поля: номер элемента и ссылку на следующий. Разместим элементы в круг и отобразим стрелками их связи.

Сейчас промоделируем ход считалочки, используя указатели. Начиная с первого, пропустим $m-1$ элемент и изменим связь: $(m-1)-$ый элемент соединим с $(m+1)$-ым. Продолжим $n-1$ раз процесс поиска и вычеркивания. В круге останется один элемент. Это и будет победитель.

Такое представление более точно отражает действительную ситуацию: люди стоят в кругу и между ними существуют соседские связи, а потом связи постепенно меняются, если кто-то выбывает из круга.

Упражнение. Переосмыслите ход считалочки на примере, явно разрывая связи с выбывшими кандидатами.



Чтобы запрограммировать последний вариант, определим тип Kandidat, характеризующий элемент-запись из считалочки и сохраним номер кандидата и адрес (указатель на) человека, который стоит следующим. Определим также тип – указатель на тип Kandidat. В первой части процедуры построим цепочку с участниками. Во второй части организуем цикл на удаление.

Получим следующую программу.

```
Program Schitalochka;

var
   m, n, l : Integer;
  procedure Last(n,m:Integer; var l:Integer);
  type
      Ptr_kand = ^Kandidat;
      Kandidat = record
                  Num  : Word;
                  Next : Ptr_kand;
                 end;

  var
     First, Current, Newitem : Ptr_kand;
     i, j                    : Integer;
  begin
   New(First);
   First^.Num := 1;
   Current    := First;
   for i := 2 to n do
               { создание ряда кандидатов}
               { и налаживание между ними связи}
     begin
      New(Newitem);
      Newitem^.Num  := i;
      Current^.Next := Newitem;
      Current       := Newitem;
     end;
  {после выхода из цикла указатели Current и Newitem
            сохраняют ссылку на последнего кандидата }
   Current^.Next := First;         {завершаем цепочку}
   for i := 2 to n do
     begin         {образуем отсчет (m-1)-го человека}
      for j := 1 to m - 1 do
            Current := Current^.Next;
```

```
      Current^.Next := Current^.Next^.Next;
            { этим вывели человека, который был m-м}
     end;
   l := Current^.Num;
  end;

begin
 n := 7;   m := 5;
 Last(n, m, l);
 Writeln(l);
end.
```

Задание 1. Запрограммируйте вариант считалочки, если у какого-то из участников существует на 1 раз «индульгенция» на не выбывание.

Задание 2. Запрограммируйте вариант считалочки, если у некоторых из участников существует на несколько раз «индульгенция» на не выбывание.

Задание 3. Получите номера выбывающих кандидатов в порядке их выбывания.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 12
СПИСКИ И ИХ КЛАССИФИКАЦИЯ

Содержание темы

- Основные положения:
  - связные списки,
  - действия со списками,
  - типы списков по методам доступа к узлам.
- Однонаправленные связные списки.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

ОСНОВНЫЕ ПОЛОЖЕНИЯ

Списком называют такую структуру данных, каждый элемент которой содержит ссылку, связывающую его с последующим элементом – *узлом* (*звеном*) списка.

Различают списки:

- линейные;
  - линейные с одной связью;
  - линейные с несколькими связями;
- кольцевые (циклические);
  - кольцевые с одной связью;
  - кольцевые с 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 мы видели 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

Связные списки обладают одним очень важным преимуществом: операции вставки и удаления принадлежат по времени обработки классу $O(1)$ ($O(1)$ – константа, не зависящая от количества внешних данных), так как для таких действий всегда требуется одно и то же время.

Основным недостатком связных списков является то, что получение доступа к их узлам принадлежит классу $O(n)$, так как при поиске n-го узла мы начинаем с некоторой позиции в списке и переходим по ссылке к искомому узлу. Чем больше узлов в списке, тем больше переходов нужно совершить.

Для увеличения скорости доступа к узлам в некоторых частных случаях используются стеки и очереди.

По сравнению с массивами списки требуют большего размера памяти, так как узел списка содержит указатель или указатели на следующий узел.

Как и массив, связный список 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 с некоторыми описаниями, образующими информационную часть узла списка. При этом каждый узел содержит указатель, который указывает на следующий узел в списке или равен `nil`, если он последний. В случае кольцевого списка последний должен ссылаться на первый.

Сам 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.

Чем же связный список отличается от массива?

Первое, что нужно отметить, – размер связного списка не устанавливается.

Для массива всегда надо знать заранее, сколько элементов будет в нем храниться (чтобы можно было статически выделить непрерывный участок памяти для его хранения), или разработать некоторую схему расширения массива (или его сокращения), чтобы массив смог поместить большее (или меньшее) количество элементов.

В массиве каждый следующий элемент находится рядом с предыдущим.

Выделение памяти под *n* элементов массива фактически представляет собой операцию класса *O*(1): все элементы должны находиться в одном непрерывном блоке памяти, поэтому одновременно выделяется сразу весь блок. В массиве доступ к *n*-ному элементу требует проведения простых арифметических подсчетов адреса памяти. Эта операция класса *O*(1).

Прежде чем начать описание операций со связным списком, рассмотрим, как каждый узел списка будет представляться в памяти. Знание структуры узла позволит нам более детально рассмотреть основные операции со связными списками.

Структура узла списка выглядит следующим образом: тип узла – это запись, в котором хранятся данные и указатель на следующий узел списка.

Действия со списками

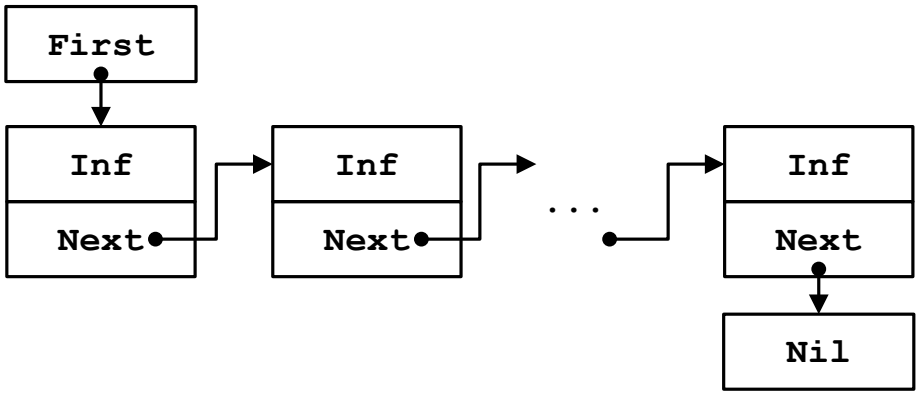
При работе со списками чаще всего приходится выполнять следующие действия:

- найти узел с заданным свойством;
- удалить узел;
- определить по номеру нужный узел;
- добавить узел;
- распечатать узлы списка;
- упорядочить узлы списка по некоторому ключу и прочее.

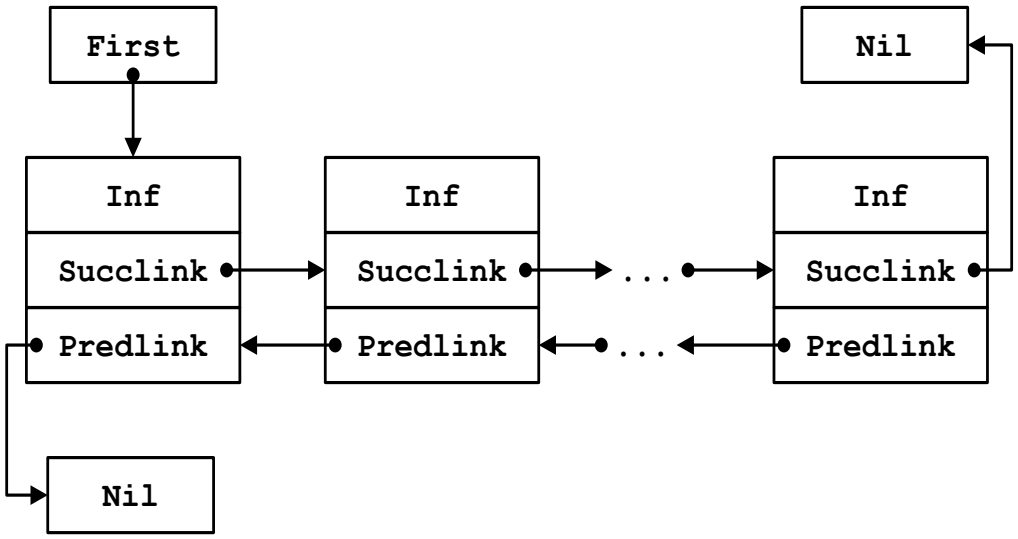
Типы списков по методам доступа к узлам

Приведем графическую интерпретацию методов связного хранения некоторых простых типов списков.

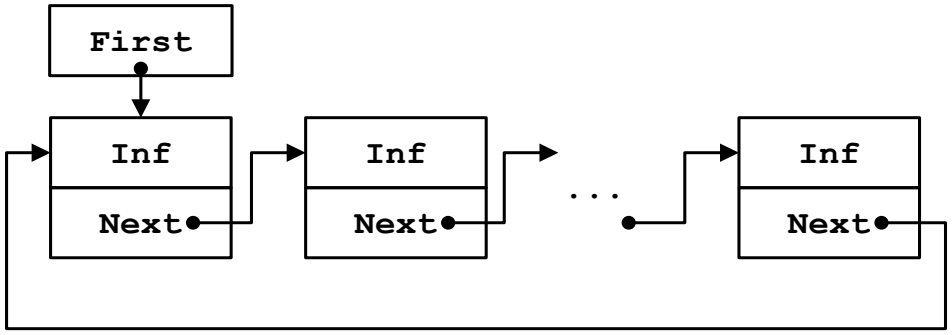
1. Обычный линейный список с одной связью:



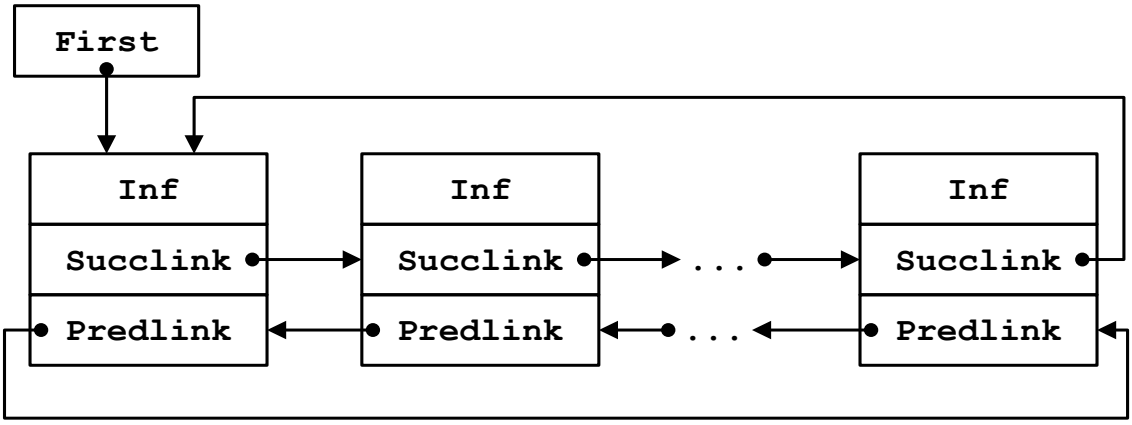
2. Обычный линейный список с двумя связями:



3. Обычный линейный циклический список с одной связью:



4. Обычный линейный циклический список с двумя связями:

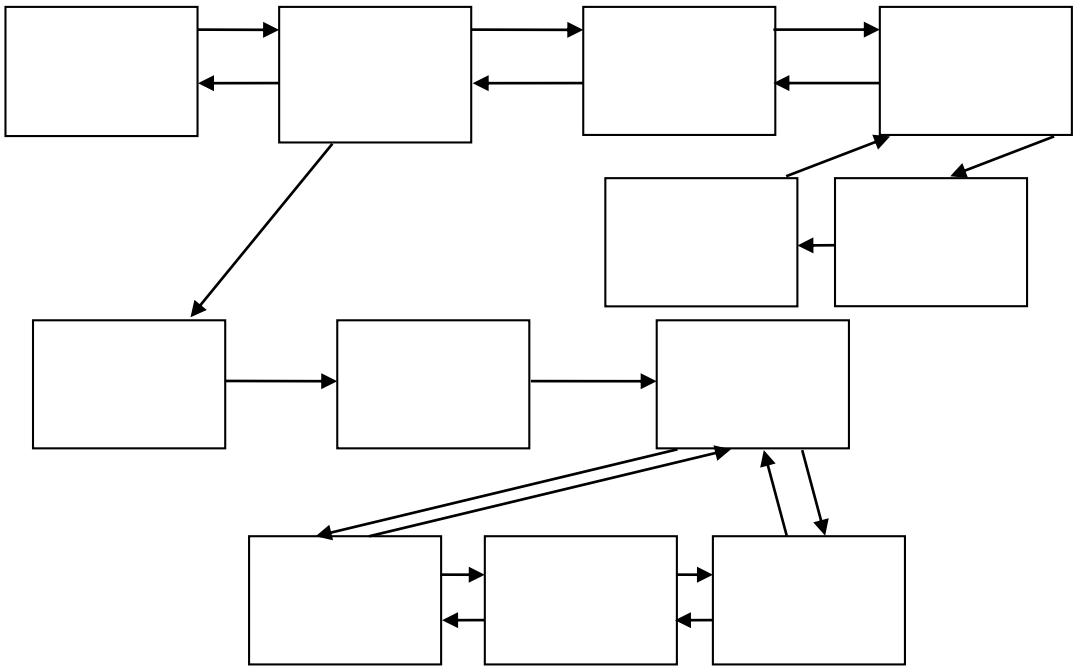


Во всех этих списках доступ возможен к любому узлу списка. Если в линейном списке добавление узла происходит в конце списка, а уничтожение – с начала списка, тогда такой список называется *очередью*.

Если же в линейном списке добавление узла происходит в начало списка и уничтожение идет с начала списка, тогда такой список называется *стеком*.

Ассоциативные списки организуются на одном общем наборе узлов, причем каждый из подсписков объединяет в определенном порядке только те узлы из этого набора, которые обладают заданным им характеристическим свойством. Значит, узел такого списка должен содержать столько ссылочных частей, сколько подсписков предполагается создать на базе общего набора записей.

Иерархические списки – более сложная структура. Их можно представить, например, следующей схемой:



Информационная часть в иерархических списках – тело узла списка – может образовывать группу записей с определенным внутренним порядком, например списком. Иерархическим списком можно представить список высших учебных заведений, каждое из которых имеет свои списки факультетов, а факультеты – списки курсов и т.д. Такая структура данных наиболее близко отражает связи в реальных задачах.

Ресурсы по работе со списками лучше объединить в отдельном модуле и этим образовать АТД «линейный список» и т. п.

Для работы со списками 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.

ОДНОНАПРАВЛЕННЫЕ СВЯЗНЫЕ СПИСКИ

Пусть надо работать со списком, который содержит сведения о каком-то автомобиле. Информационная часть узла такого списка будет содержать сведения о марке автомобиля, номере двигателя, шасси, государственном номере регистрации, владельце и т.д. Но если мы рассматриваем собственно работу с узлами списка, то эта информация для нас 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.
```

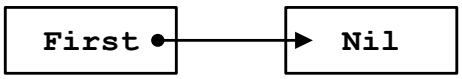
Здесь одно из полей записи предназначено для связывания узлов списка в единое целое – однонаправленный связный список. Это поле – `Next : TLink`. Структура связного списка предполагает, что очередной узел в списке через этот указатель связывается с последующим узлом. Последний узел списка имеет в поле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

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

Алгоритм создания списка

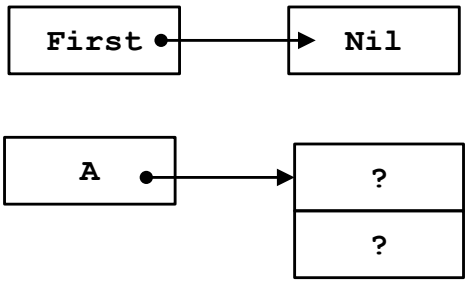
Это тривиальная задача. Добавлять узлы в создаваемый список можно перед первым или после последнего. Рассмотрим следующий алгоритм, основанный на добавлении нового узла первым.

1. Изначально, когда список пуст, указатель `First` устанавливается в `nil`.



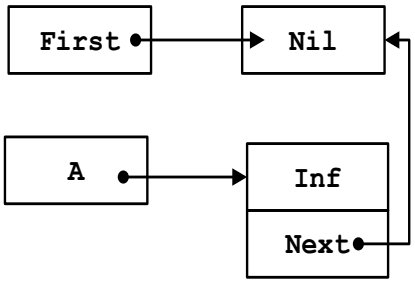
2. Затем в динамической области под указатель `A` выделяется память.

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



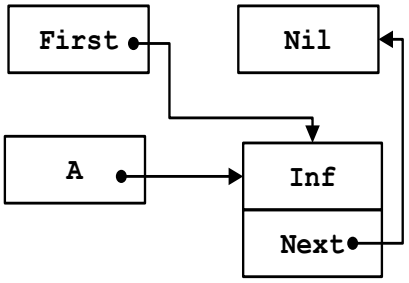
3. Информационная часть `A^,` например, при помощи процедуры `Init_inf,` наполняется информацией.

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 последним.

Выделим далее операции по работе с одиночным узлом. На основании предыдущих алгоритмов становится очевидным, что для осуществления таких действий требуется выполнить некоторую работу с указателями. Оформим нужные действия в виде процедур, которые далее можно использовать как операторы при работе с АТД «линейный список».

Вставка узла в список

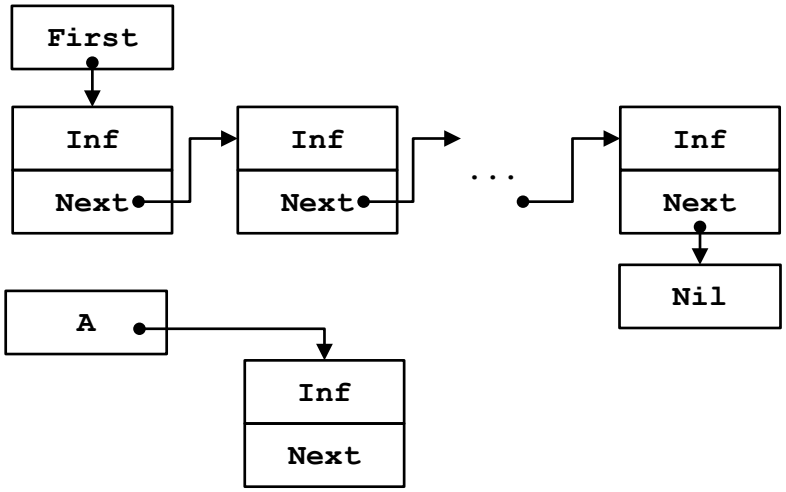
При вставке очередного узла в список могут возникнуть следующие ситуации:

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

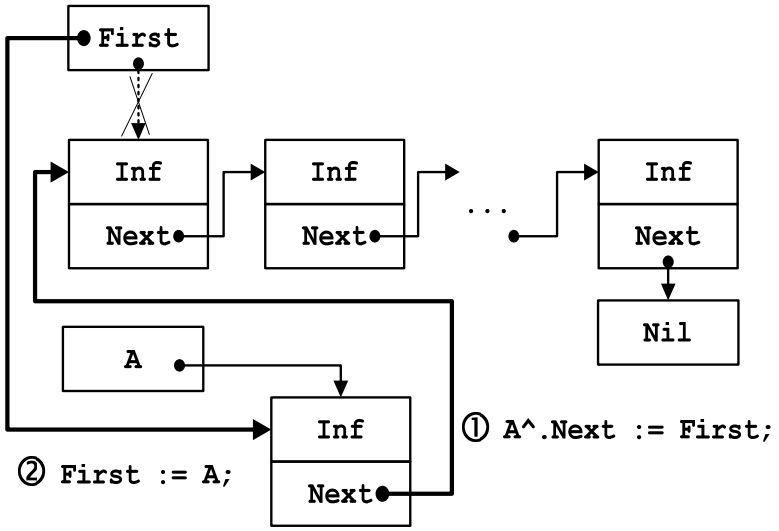
Этим случаям соответствуют следующие процедуры.

Включить узел в список первым

Рассмотрим включение очередного узла в начало списка. Исходное состояние будет следующим:



Этапы включения очередного узла A в начало списка приведены ниже:



Соответственно напишем процедуру AddFirst для включения очередного узла A в начало списка.

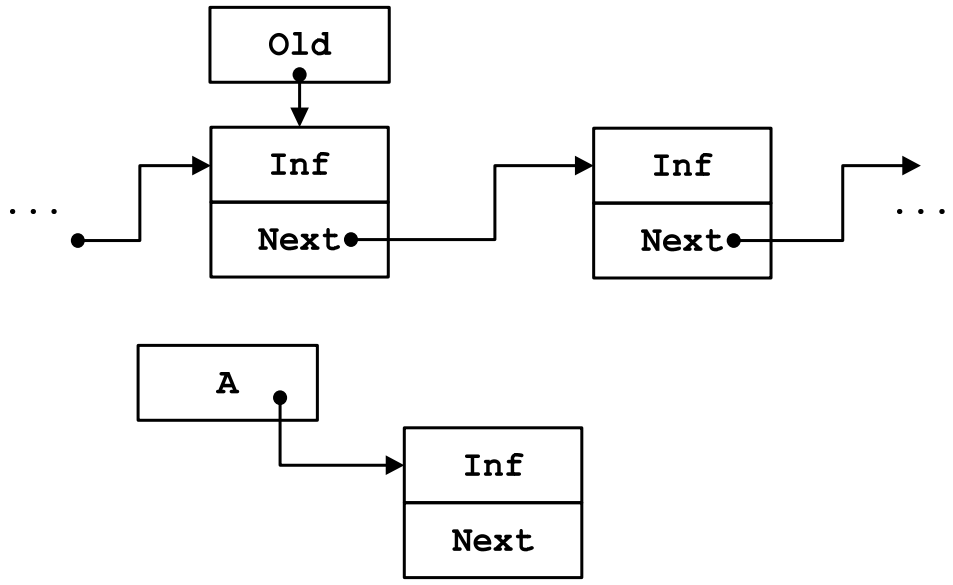
```
procedure AddFirst (var  A, First : TLink);
begin
     A^.Next := First;              {(1)}
     First   := A;                  {(2)}
end;
```

Включить узел 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).

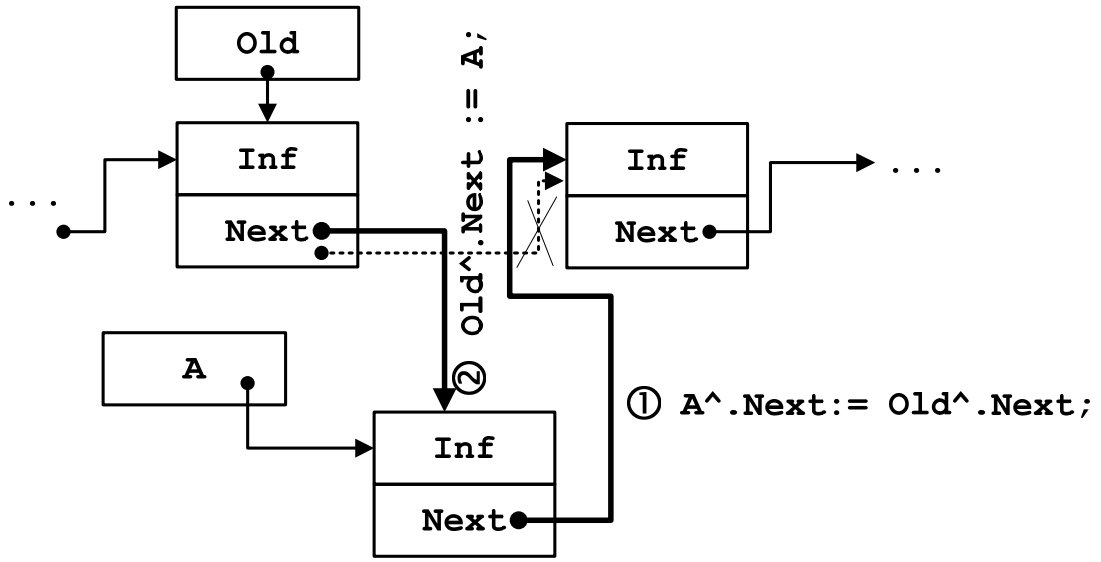
Исходное состояние для включения узла в середину списка:



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;
```

Результат выполнения операции включения в середину списка после узла, на который указывает указатель `Old`, будет следующий:



Если нужно включить очередной 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:

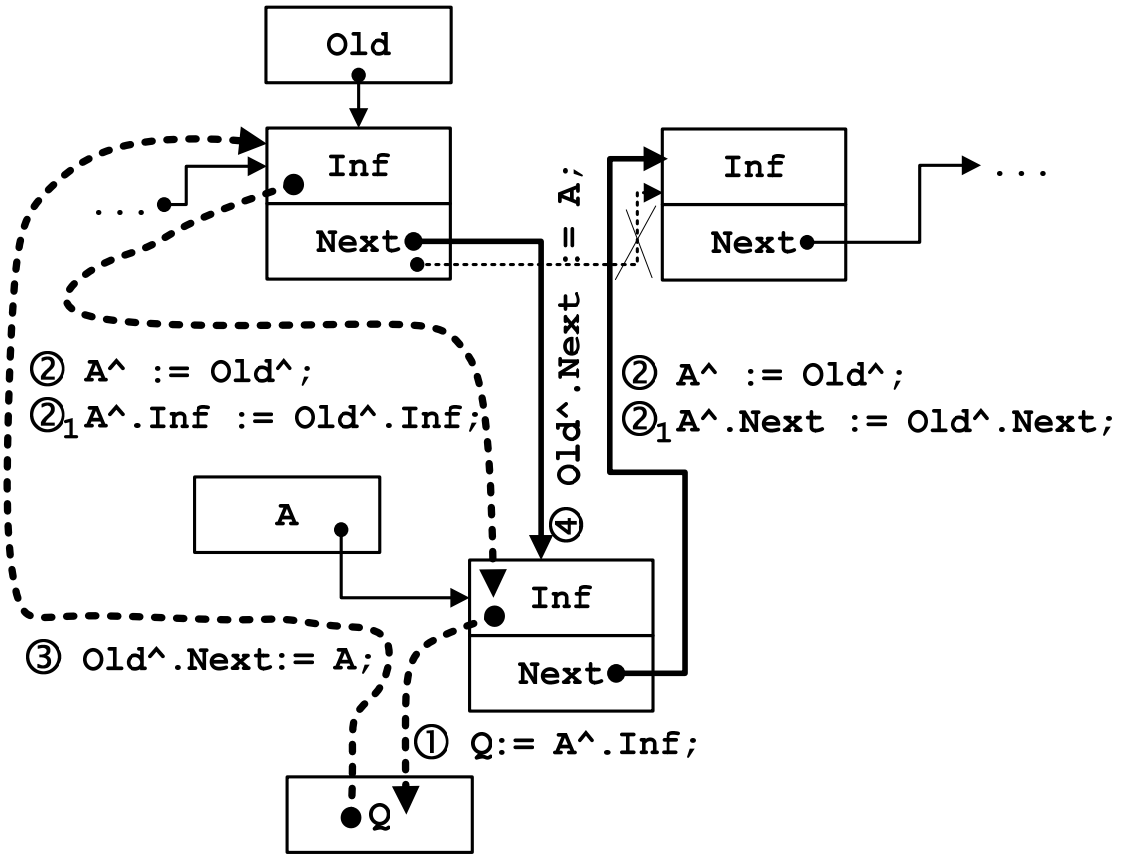
1) содержимое A^.Inf переслать во вспомогательный элемент Q;

2) на место содержимого A^ послать содержимое Old^;

3) на место содержимого Old^.Inf послать вспомогательный элемент Q;

4) установить ссылку Old^.Next = A.

Покажем это:



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

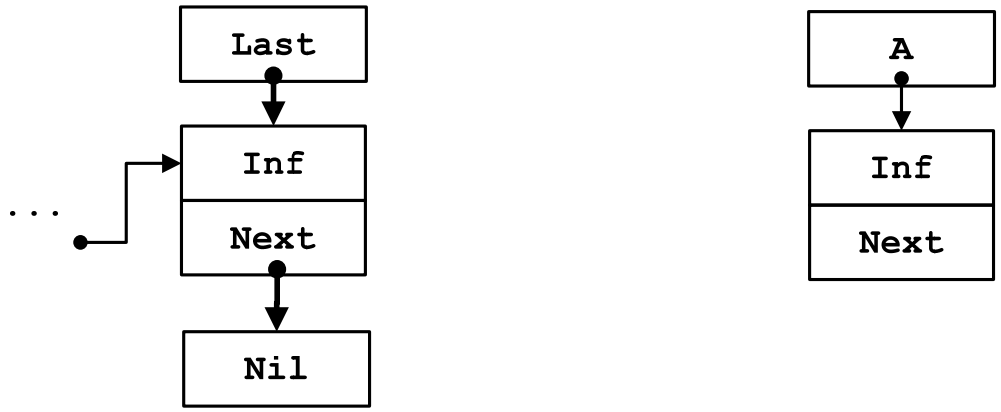
```
procedure AddBefore(var  A, Old : TLink);
var   Q : TInf;
begin
   Q         := A^.Inf;              {(1)}
   A^        := Old^;                {(2)}
   Old^.Inf  := Q;                   {(3)}
```

```
    Old^.Next := A;                    {(4)}
end;
```

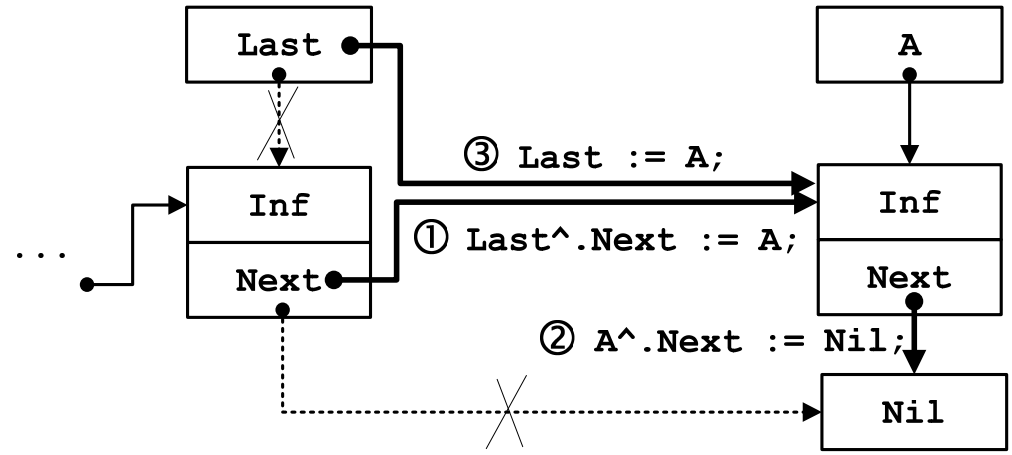
Поместить узел в конец списка последним, но не единственным

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

Исходное состояние:



Этапы включения очередного узла `A` в конец списка последним, но не единственным:



Соответственно получим следующую процедуру включения очередного узла `A` в конец списка последним, но не единственным.

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

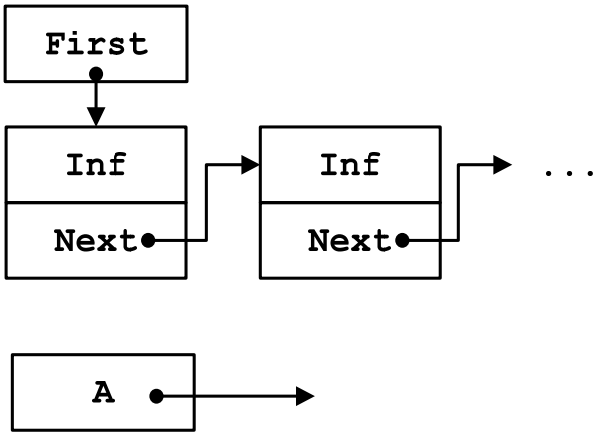
Удаление узла списка

При удалении узла списка также могут возникнуть различные ситуации:

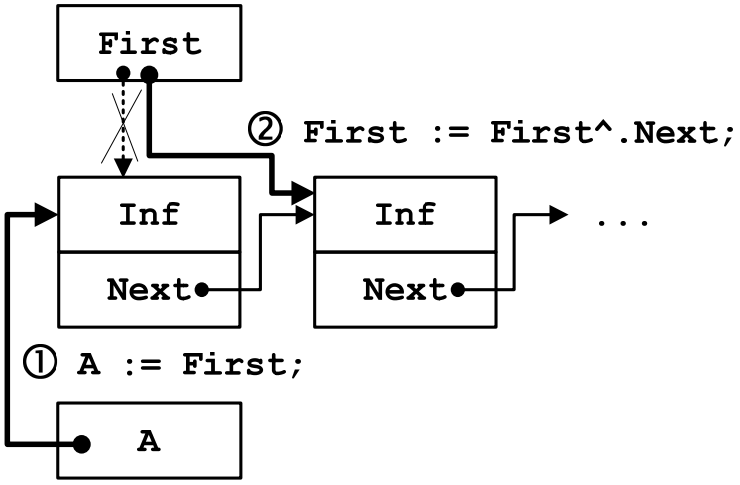
- удаление первого узла списка (`DelFirst`);
- удаление узла в середине списка:
  - узел, который удаляется, стоит после узла `Old` (`DelAfter`);
  - удаление самого `Old` (`DelCurrent`);
- удаление последнего узла (`DelLast`).

Удаление первого узла списка

Рассмотрим удаление первого узла списка. Исходное состояние:



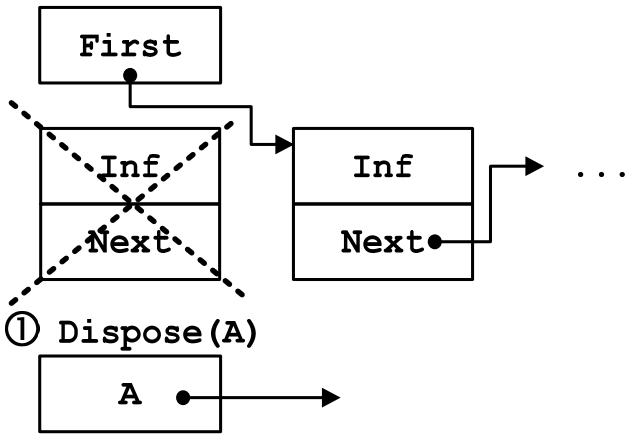
Этапы удаления первого узла списка:



Соответственно получим следующую процедуру удаления первого узла списка.

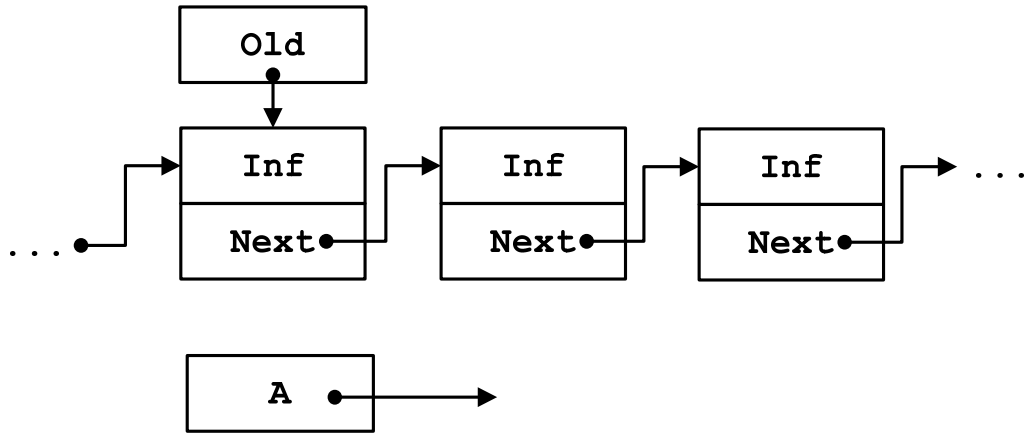
```
procedure  DelFirst(var First, A : TLink);
begin
      A     := First;                    {(1)}
      First := First^.Next;              {(2)}
end;
```

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

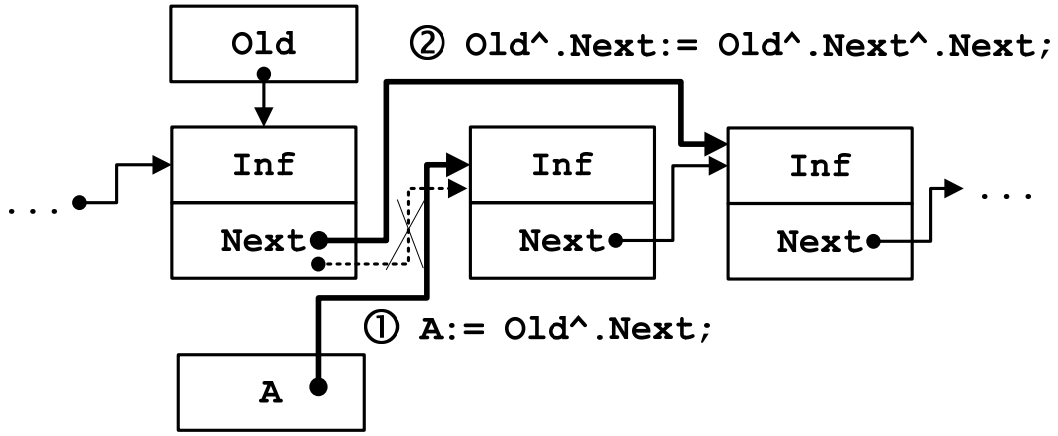


Удаление узла в середине списка после узла Old

Рассмотрим удаление узла в середине списка после узла `Old`. Исходное положение:



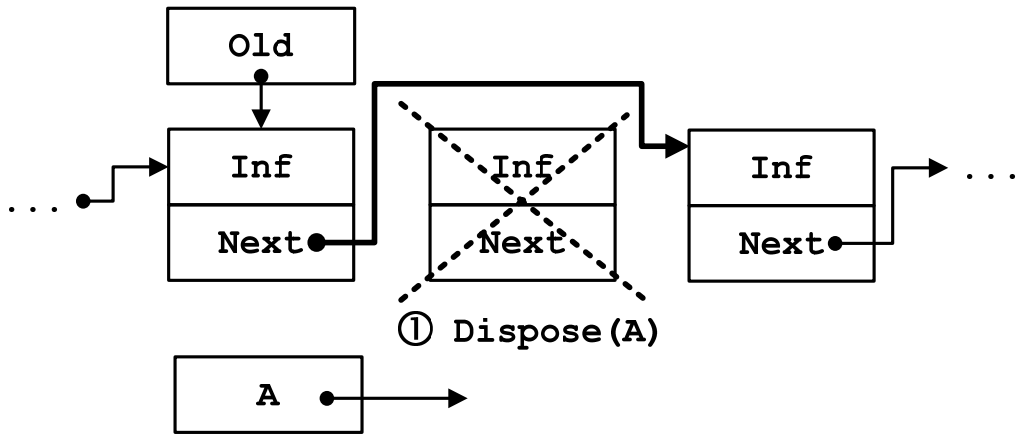
Этапы удаления узла в середине списка после узла `Old`:



Предложенную схему удаления узла в середине списка после узла `Old` опишем в процедуре `DelAfter`.

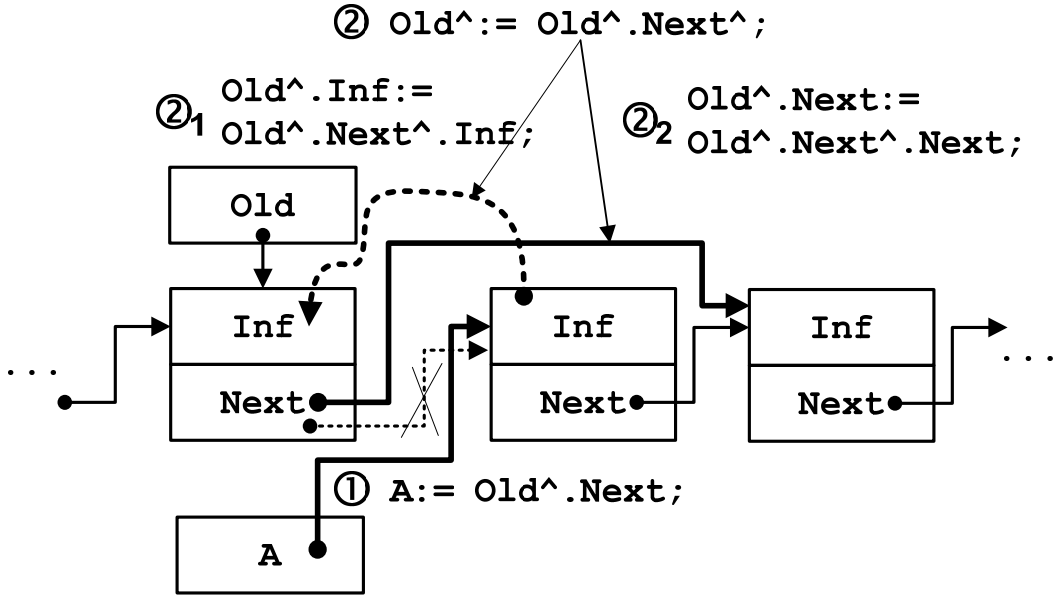
```
procedure  DelAfter (var  Old, A : TLink);
begin
     A          := Old^.Next;              {(1)}
     Old^.Next  := Old^.Next^.Next;        {(2)}
end;
```

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.



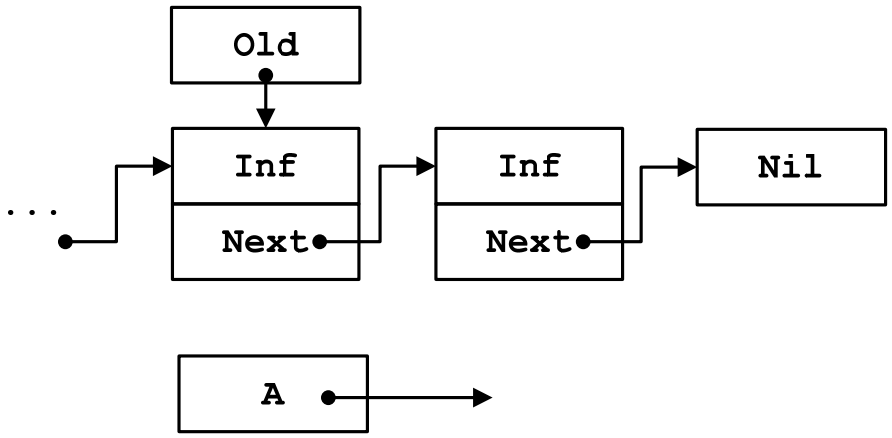
Получим такую процедуру.

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

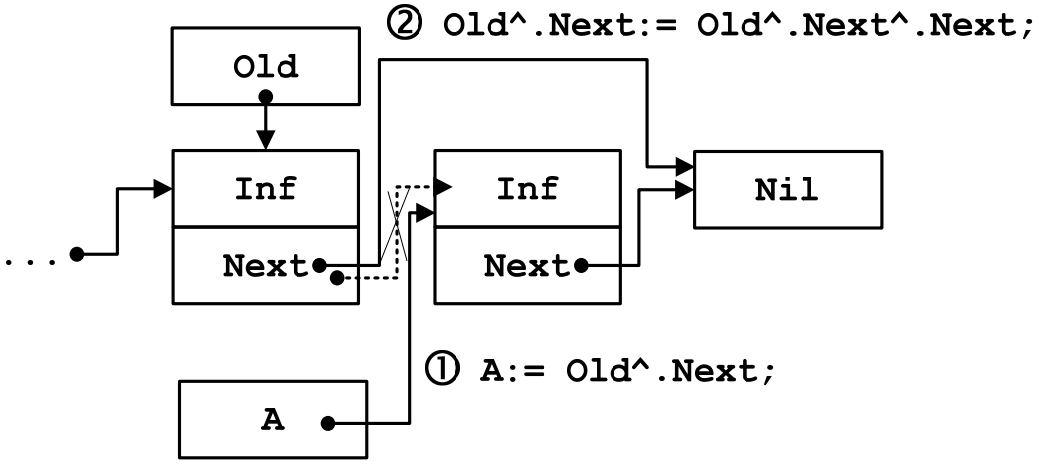
Далее в вызывающей программе можно уничтожить (`Dispose`) объект, на который ссылается указатель `A`.

Удаление последнего узла списка

Рассмотрим удаление последнего узла из однонаправленного списка. Исходное состояние:



Этапы удаления последнего узла списка:



Соответственно получим следующую процедуру удаления последнего узла 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`.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 13
ДВУНАПРАВЛЕННЫЕ СВЯЗНЫЕ СПИСКИ

Содержание темы

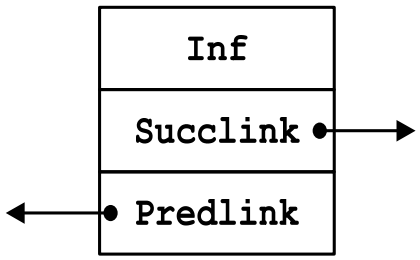
- Двунаправленные связные списки.
- Сортировка и слияние списков.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

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 и этим образовать 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.

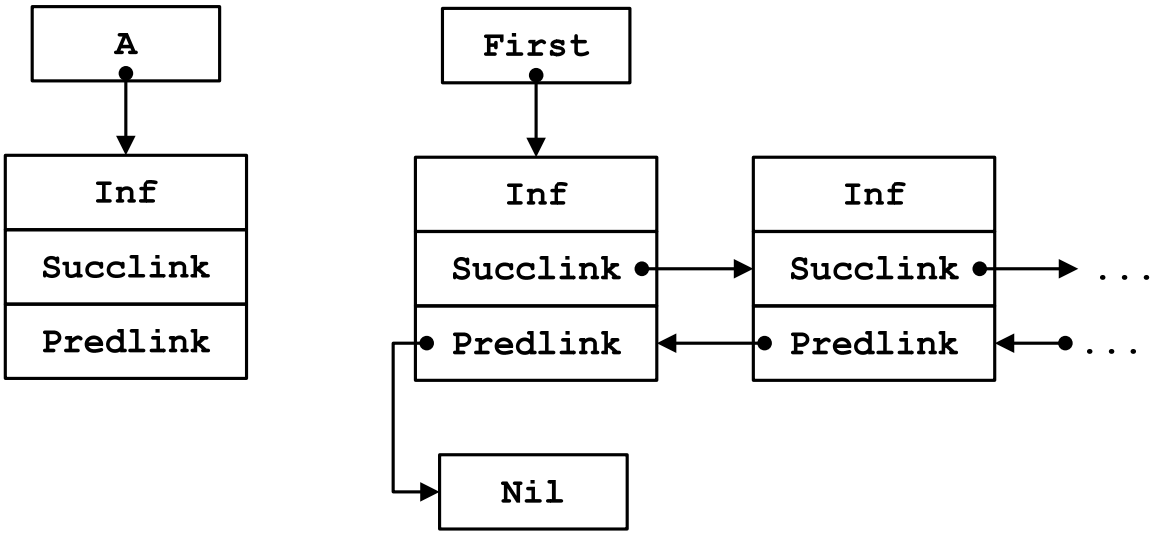
Приведём основные операции для работы с двунаправленным списком.

Процедуры включения узла в двунаправленный список

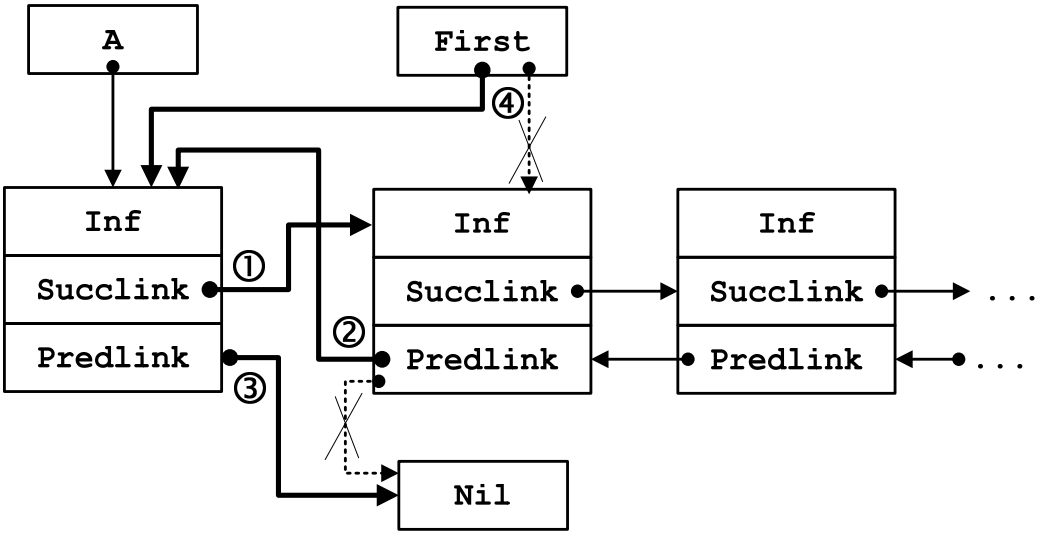
Рассмотрим процедуры *включения* узла в двунаправленный список.

Включение узла первым

Исходное состояние:



Этапы включения очередного узла A в начало двунаправленного списка:

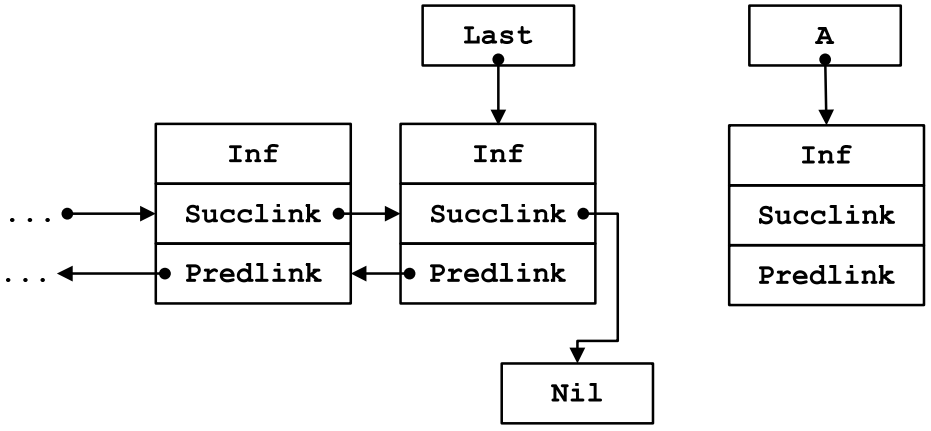


Соответственно напишем процедуру AddFirst_2 для включения очередного узла A в начало двунаправленного списка.

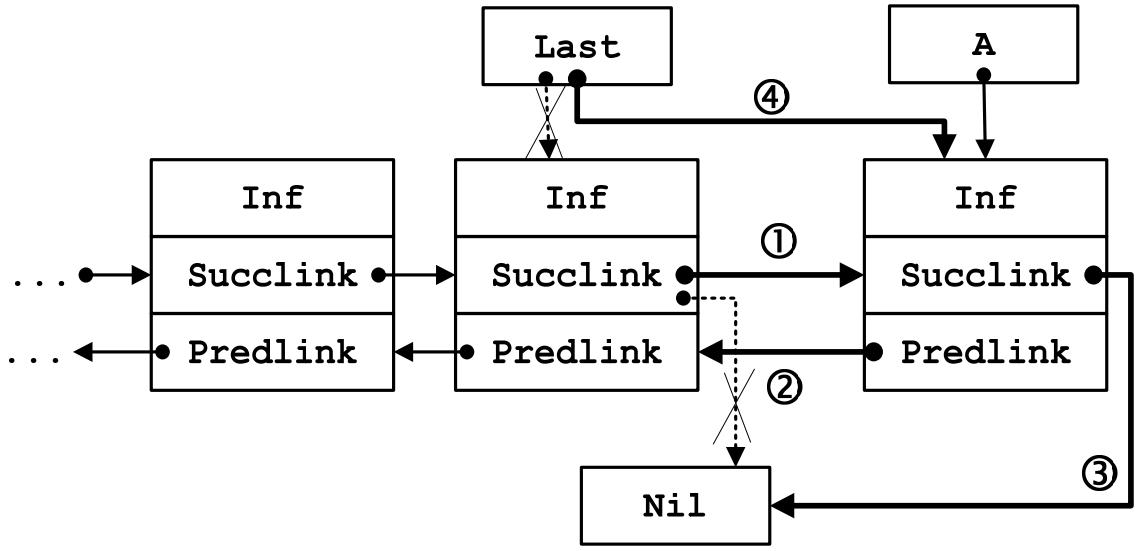
```
procedure AddFirst_2(First, A : TLink);
begin
      A^.Succlink     := First;       {(1)}
      First^.Predlink := A;           {(2)}
      A^.Predlink     := nil;         {(3)}
      First           := A;           {(4)}
end;
```

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. Исходное состояние:



Этапы включения очередного узла A в конец списка последним, но не единственным:



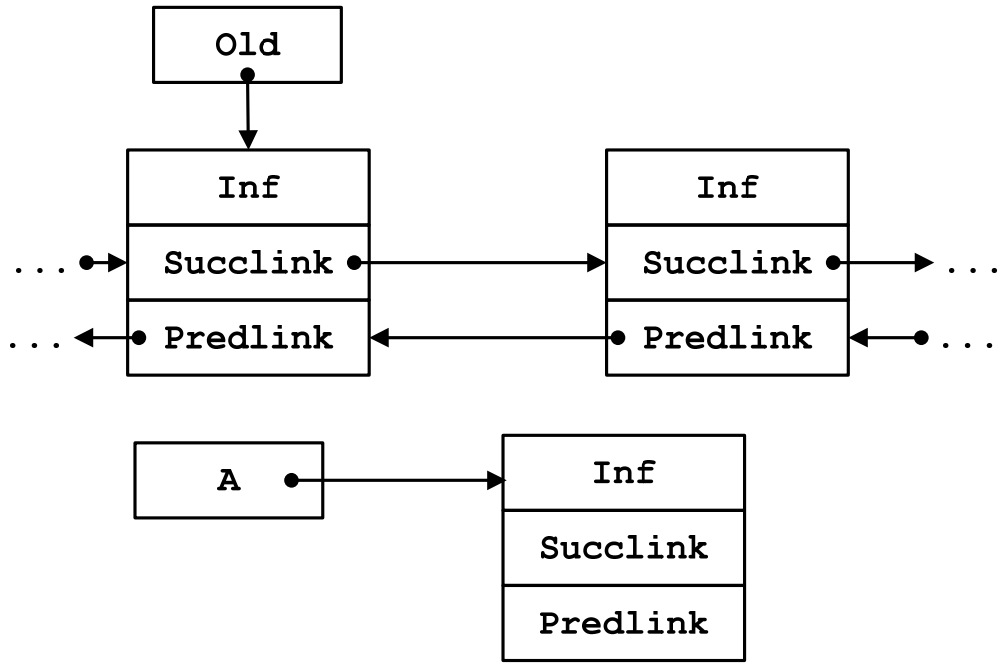
Соответственно получим следующую процедуру включения очередного узла A в конец двунаправленного списка последним, но не единственным.

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

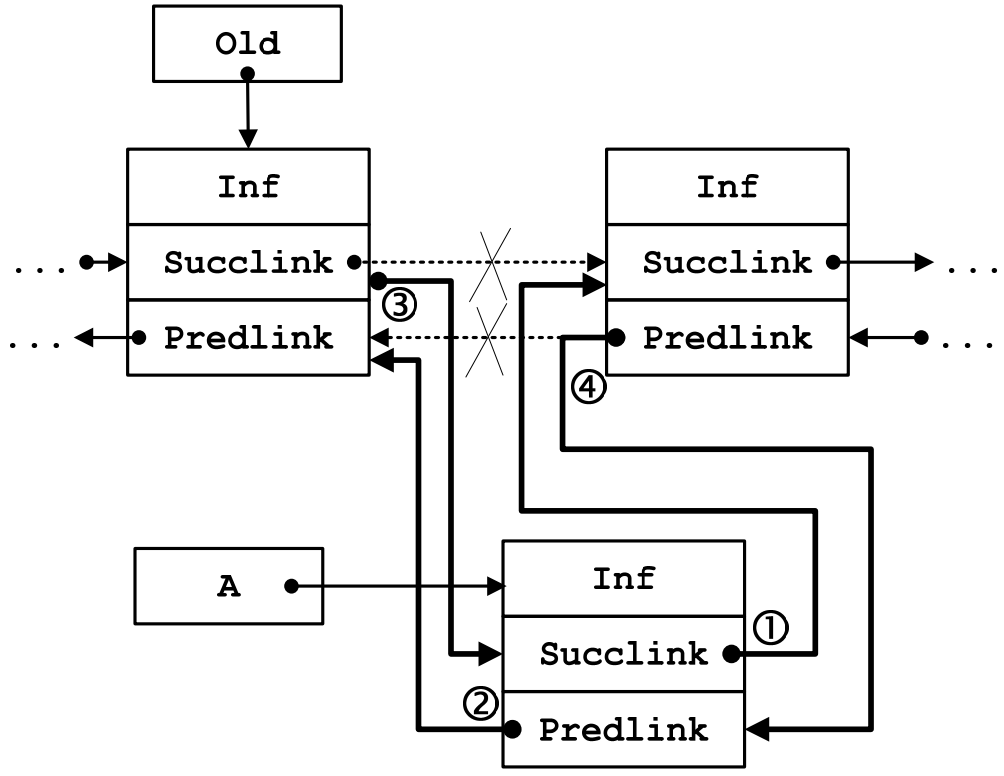
Включение узла A после узла Old

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

Рассмотрим исходную ситуацию:



Этапы включения очередного узла `A` в двунаправленный список после узла, на который указывает `Old`:

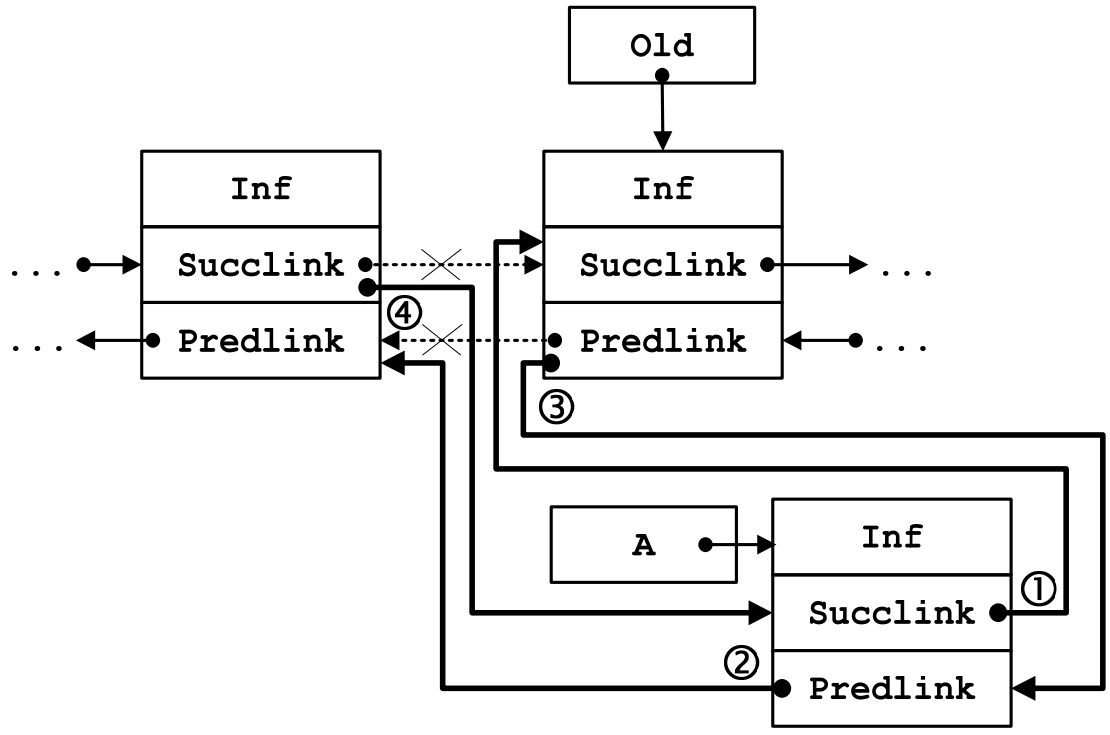


Для включения нового узла в двунаправленный список после узла, на который указывает `Old`, получим такую процедуру.

```
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.



Соответственно получим такую процедуру включения очередного узла `A` в середину списка перед узлом, на который указывает `Old`.

```
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;
```

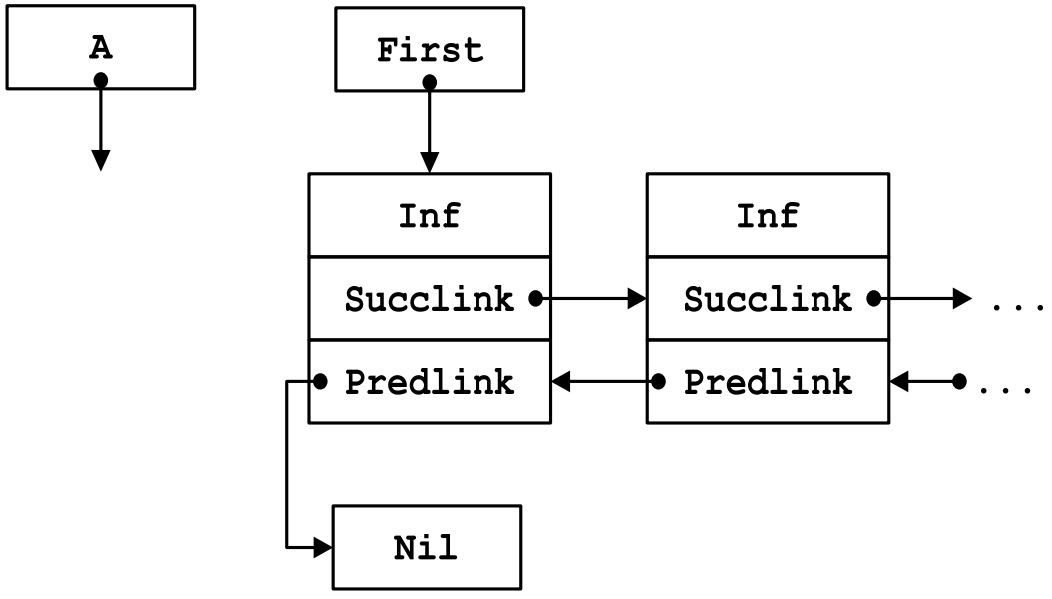
Процедуры AddAfter_2 і AddBefore_2 не «подправляют» указатели, если включаемый узел станет первым или последним, ведь мы такие процедуры сделали отдельно.

Процедуры удаления узла из двунаправленного списка

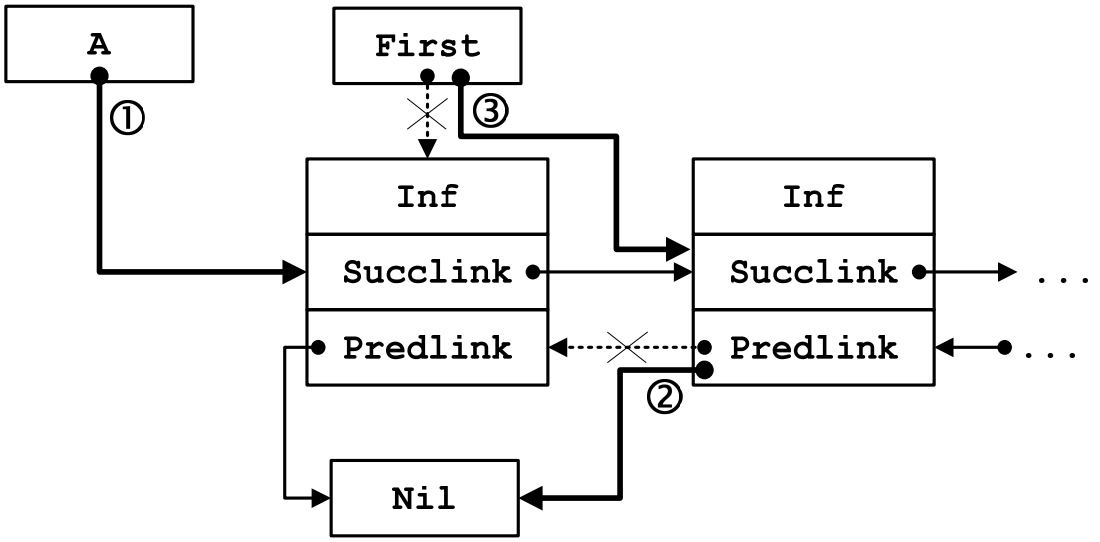
Рассмотрим ещё процедуры *удаления* узлов из двунаправленного списка.

Удалить первый узел

Исходное состояние удаления первого узла с двунаправленного списка:



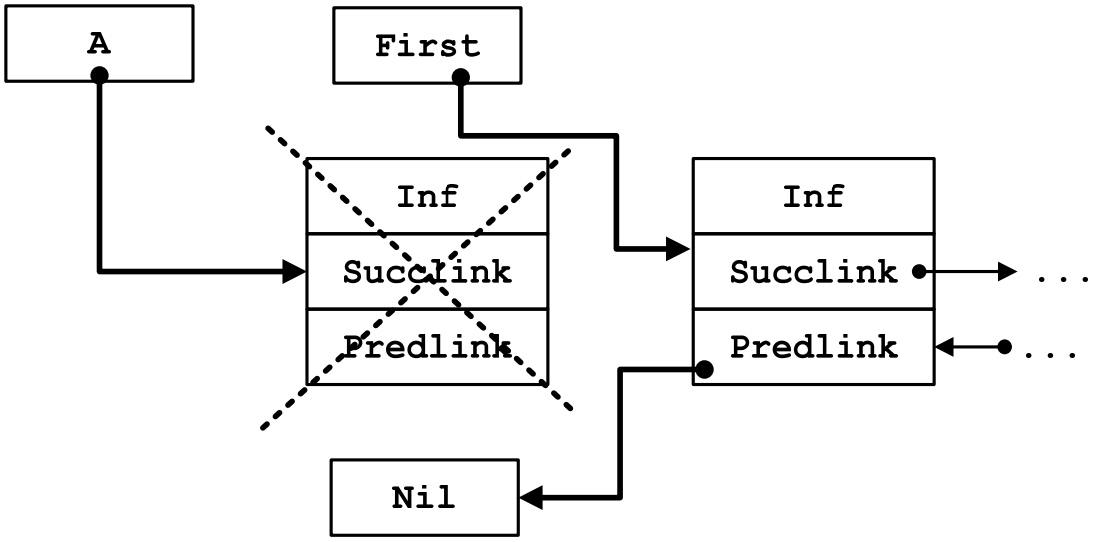
Этапы удаления первого узла списка:



Соответственно получим следующую процедуру удаления первого узла списка.

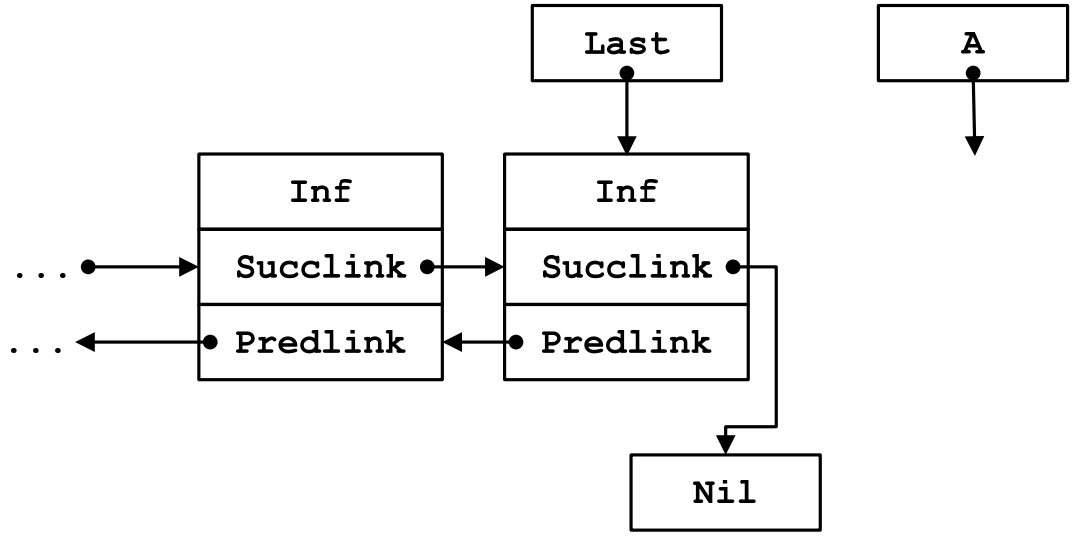
```
procedure DelFirst_2(var First, A : TLink);
begin
   A                          := First;
   First^.Succlink^.Predlink  := nil;
   First                      := First^.Succlink;
end;
```

В указателе A получили ссылку на элемент, который вывели из списка. Потом можно освободить элемент с Heap (Dispose(A)).

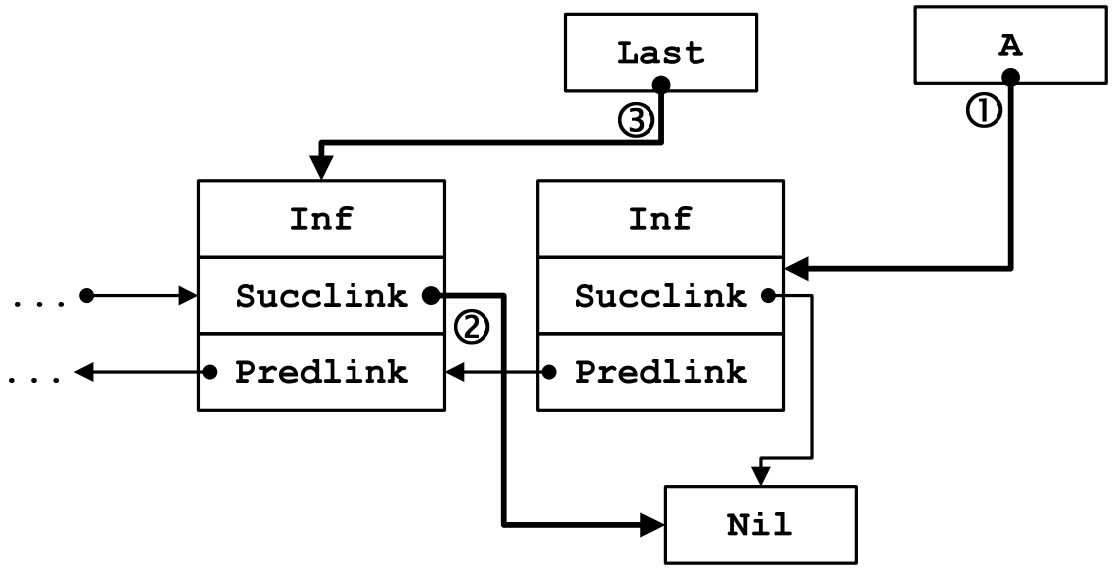


Удалить последний узел

Исходное состояние удаления последнего узла:



Этапы удаления последнего узла списка:



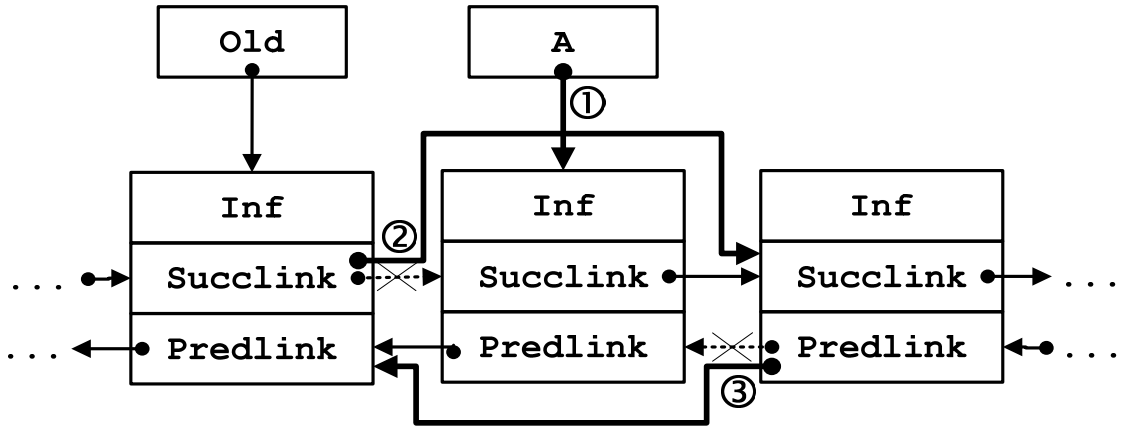
Соответственно получим следующую процедуру удаления последнего узла списка.

```
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).

Удалить узел после указателя на узел (середина списка)

Этапы удаления узла в середине двунаправленного списка после узла Old:

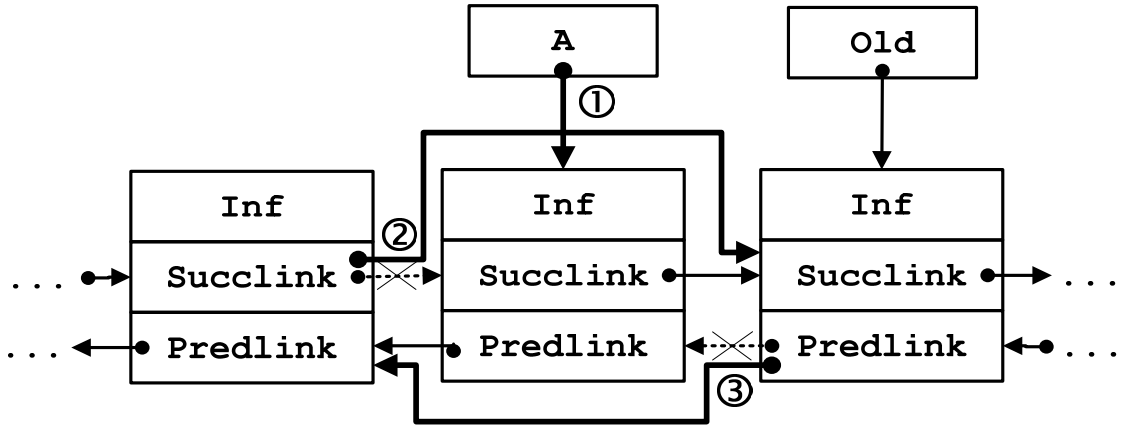


Предложенную схему удаления узла в середине списка после узла Old опишем в процедуры DelAfter_2.

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

Удалить узел перед указателем на элемент (середина списка)

Этапы удаления узла в середине двунаправленного списка перед узлом `Old`:

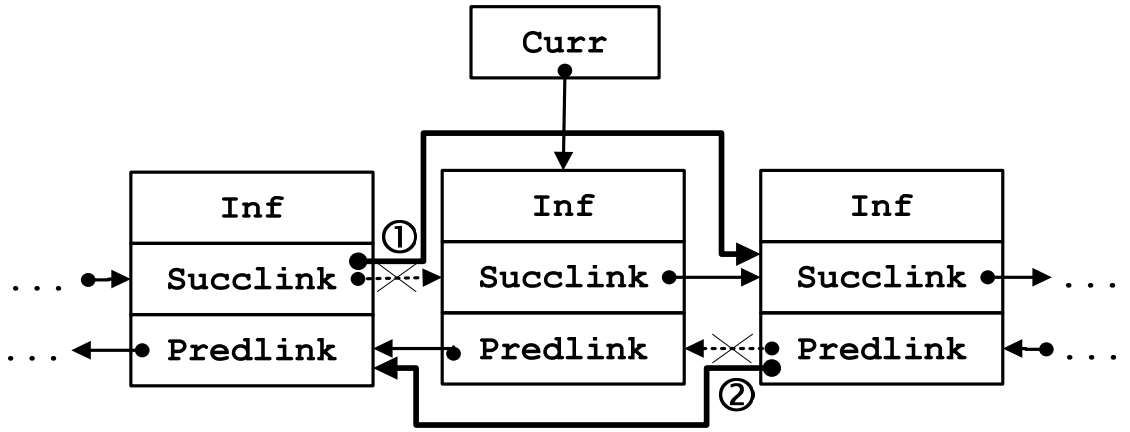


Получим следующую процедуру удаления узла в середине двунаправленного списка перед узлом `Old`.

```
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)

Этапы удаления текущего узла в середине двунаправленного списка:



Соответственно получим следующую процедуру удаления текущего узла в середине двунаправленного 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 нужно усовершенствовать на те случаи, когда в удалении будут задействованы «крайние» узлы – `First` и `Last`.

Удалить 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;
```

Удалить 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;
```

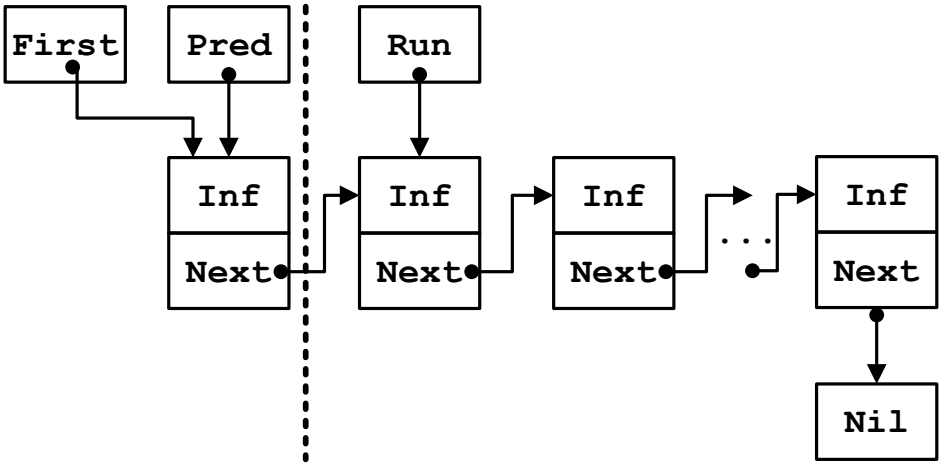
Прохождение связного списка не представляет никаких трудностей. Фактически нужно переходить от узла к узлу по указателю из ссылочной части до достижения указателя, равного `nil`, который свидетельствует об окончании списка.

СОРТИРОВКА И СЛИЯНИЕ СПИСКОВ

Сортировка списков

Ввиду того, что списки представляют собой динамические структуры данных с последовательным, а не прямым доступом, то методы сортировок массивов для списков почти не применяют, за малым исключением. Рассмотрим их.

Первый алгоритм. Рассмотрим неупорядоченный список и применим для его сортировки метод «сортировка включениями»:

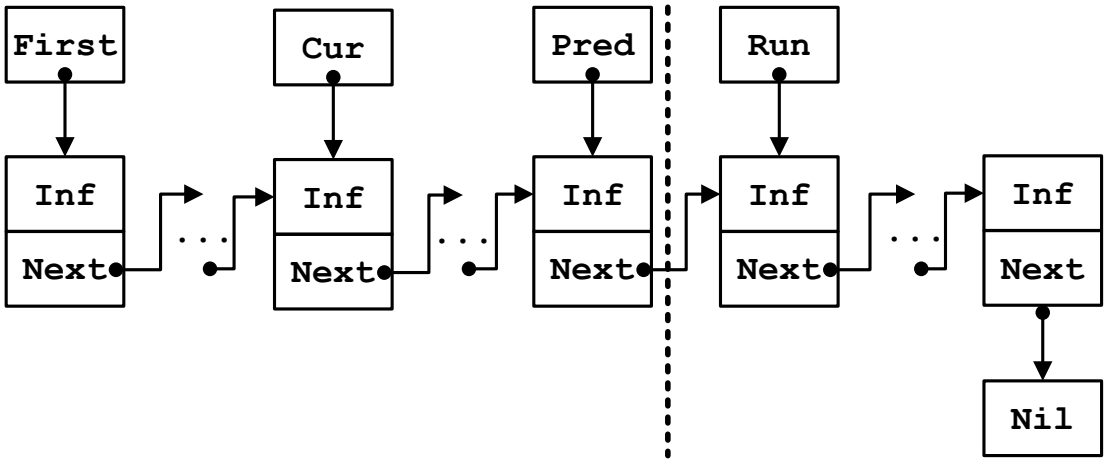


Для каждого узла списка (на него указывает `Run`), начиная со второго узла и до конца списка, ищем этому узлу место от начала списка (с `First`) до конца отсортированной части (на нее показывает `Pred`). Текущий узел

включаем в нужное место или оставляем на старом. Здесь выделяются следующие этапы.

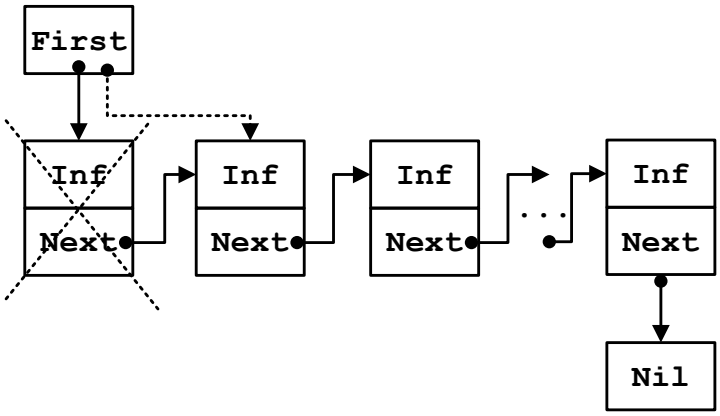
1. Поиск места, куда нужно вставить элемент.

2. Если вставляем после `Pred`, то оставляем узел на месте и сдвигаем `Pred` на `Run`; иначе вставляем в нужное место, а узел, на который указывает `Run`, удаляем из списка:

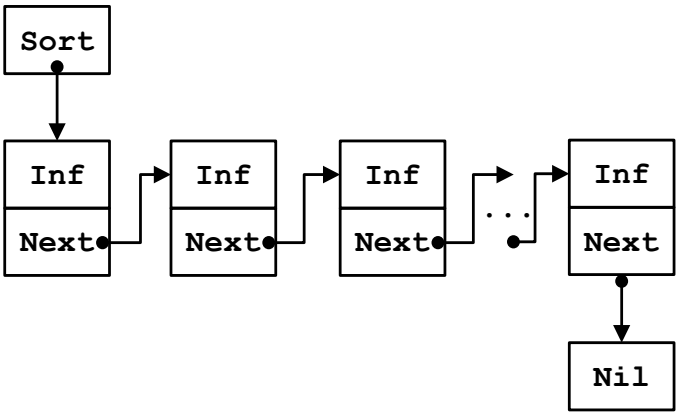


Поскольку здесь возникают сложности с указателями, лучше использовать другой подход.

Второй алгоритм. Проходя по первому списку, каждый раз вычленяем из него первый узел и вставляем на нужное место во второй отсортированный список:



Удаляем первый узел из одного списка и ищем ему место вставки в другой список:



Слияние упорядоченных списков

Первый алгоритм. Пусть заданы два упорядоченных списка. Один из них будем пополнять элементами из второго списка, не нарушая упорядочение.

Эта задача подобна предыдущей задаче сортировки (второй алгоритм), но тут поиск места вставки узла из первого списка во второй список надо начинать с места последней вставки, а не с самого начала второго списка.

Второй алгоритм. Является улучшением первого алгоритма. Он повторяет алгоритм слияния двух отсортированных файлов в третий отсортированный.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 14
СТЕКИ

Содержание темы

- Организация стека последовательным методом хранения.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

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 стека такую операцию называют `Push` – «затолкнуть». Вторая операция – `DelFirst` – «вытолкнуть» элемент из стека, для стека такую операцию называют `Pop`. Указатель на вершину стека называют `Top`.

Поскольку в переполненный стек нельзя затолкать очередной элемент и из пустого стека нельзя вытолкнуть элемент, то при работе со стеками нужны еще следующие дополнительные подпрограммы:

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

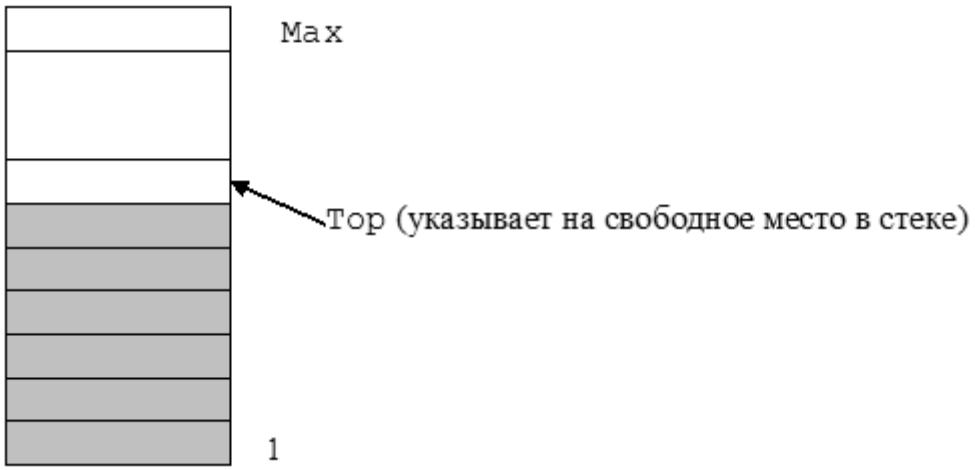
Поскольку для таких видов списка доступ односторонний, то их можно образовывать не только при помощи связных списков, используя разработанные выше процедуры, но и при помощи массивов.

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`. Получим 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 – объявим 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 затолкать очередной элемент, но нельзя из него вытолкнуть элемент. Если указатель `Top = Max + 1`, то стек «полный» и в него нельзя затолкать элемент, но можно вытолкнуть.



Опишем далее 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.
```

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 15
ОЧЕРЕДИ

Содержание темы

- Организация очереди последовательным методом хранения.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

ОЧЕРЕДИ

Очередь – это отдельный случай списка и простая динамическая структура. Добавление элементов в очередь выполняется в конец очереди, а выборка из очереди – 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`. Это 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 еще следующие подпрограммы:

- `function Empty`, которая выдаёт значение `true`, когда очередь пустая, и значение `false` в противном случае;
- `function Full`, которая выдаёт значение `true`, когда очередь переполнилась, и значение `false` в противном случае;
- `procedure Mistake`, которая обрабатывает ошибочную ситуацию, параметр – код ошибки;
- `procedure Clear`, которая нужна, чтобы «почистить» 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 с помощью 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`, при этом, 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, поэтому если элементы очереди достигают правого края массива, тогда новые элементы записываются на свободное место в начало массива. Второй подход более дешевый по времени работы программы.

Когда очередь пуста, индекс `Front` установим в 1 – первый незанятый элемент массива, за которым можно обслуживать, а индекс `Rear` установим 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, в нем будут описаны вышезаявленные ресурсы без использования динамических переменных. Получим следующую АТД «очередь».

Модуль для работы с очередьми

```
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

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 – эта очередь, с каждым элементом которой ассоциирован 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 сообразно его приоритету и обслуживание элемента, который имеет самый высокий приоритет. Для такой очереди при реализации процедуры `Push` надо сначала найти место вставки элемента в очередь сообразно его приоритету. Тогда процедура `Pop` будет правильно обслуживать очередь с приоритетом.

Пример на использование очереди.

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

Алгоритм.

Запомним начальный ход в очереди ходов и укажем, что он нулевого порядка.

Будем повторять пока не попадаем в конечную точку:

1. Извлечем из очереди координаты очередного хода.
2. Организуем на следующем проходе добавление в очередь всех возможных перемещений коня согласно таблице перемещения коня, если ход приемлем и, если 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

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

ТЕМА 16
ДЕРЕВЬЯ

Содержание темы

- Деревья. Основные понятия и определения.
- Типы деревьев.
- Бинарные деревья.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

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 имеют следующую иерархию: если узел x имеет уровень i, то его непосредственный потомок – уровень $i+1$.

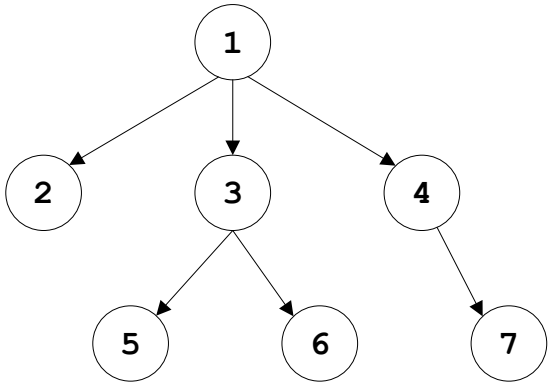
Любой узел дерева с его потомками также образует дерево, называемое *поддеревом* (относительно исходного дерева).

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 $B_1, B_2, ..., B_M$, которые не пересекаются, каждое из них является *поддеревом*.

Эти 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, наз. *однородным*.

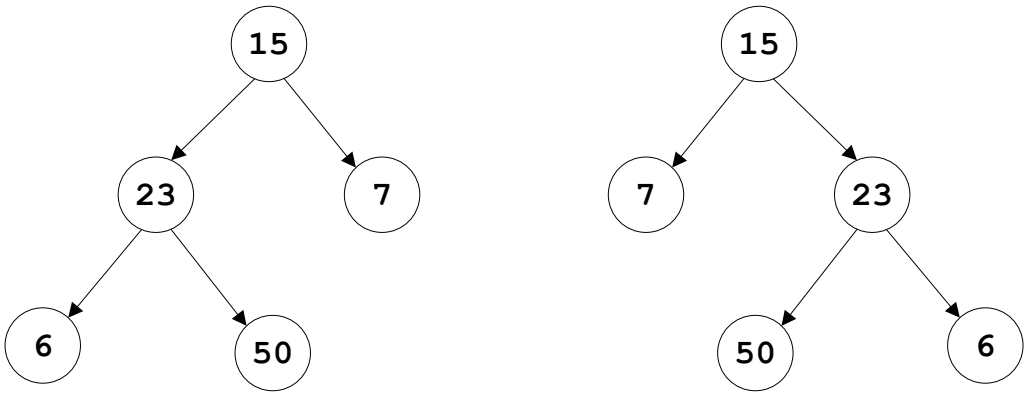
Если узлы дерева представлены разными типами записей, то оно называется *неоднородным*.

Типы деревьев

В зависимости от количества потомков (ветвистость) 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, каким он является: первым, вторым, ..., n-м (левым, средним, ..., правым).

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



Бинарные деревья

Прямой и симметричный обходы деревьев

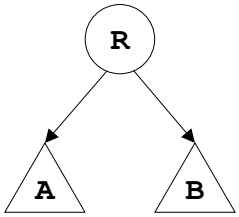
Рассмотрим упорядоченное дерево. Дочерние узлы обычно упорядочиваются слева направо – левостороннее упорядочивание, если не оговорено о правостороннем упорядочивании.

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 в обратном порядке все узлы поддеревьев, 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`);

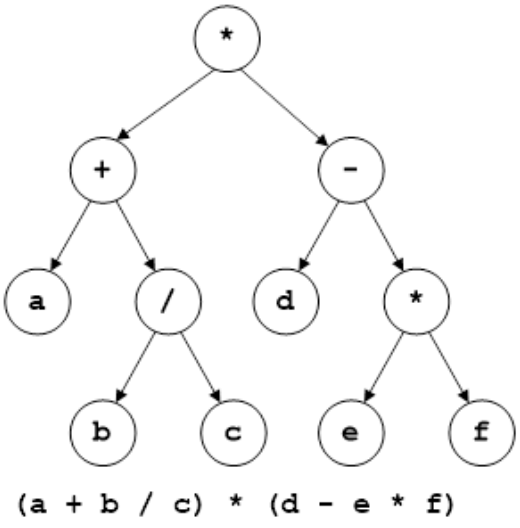
б) при правостороннем упорядочивании:

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

Более очевидными становятся три первых варианта обхода на обходе 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 дальше 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

Представление деревьев можно делать и при помощи массивов, и при помощи ссылок.

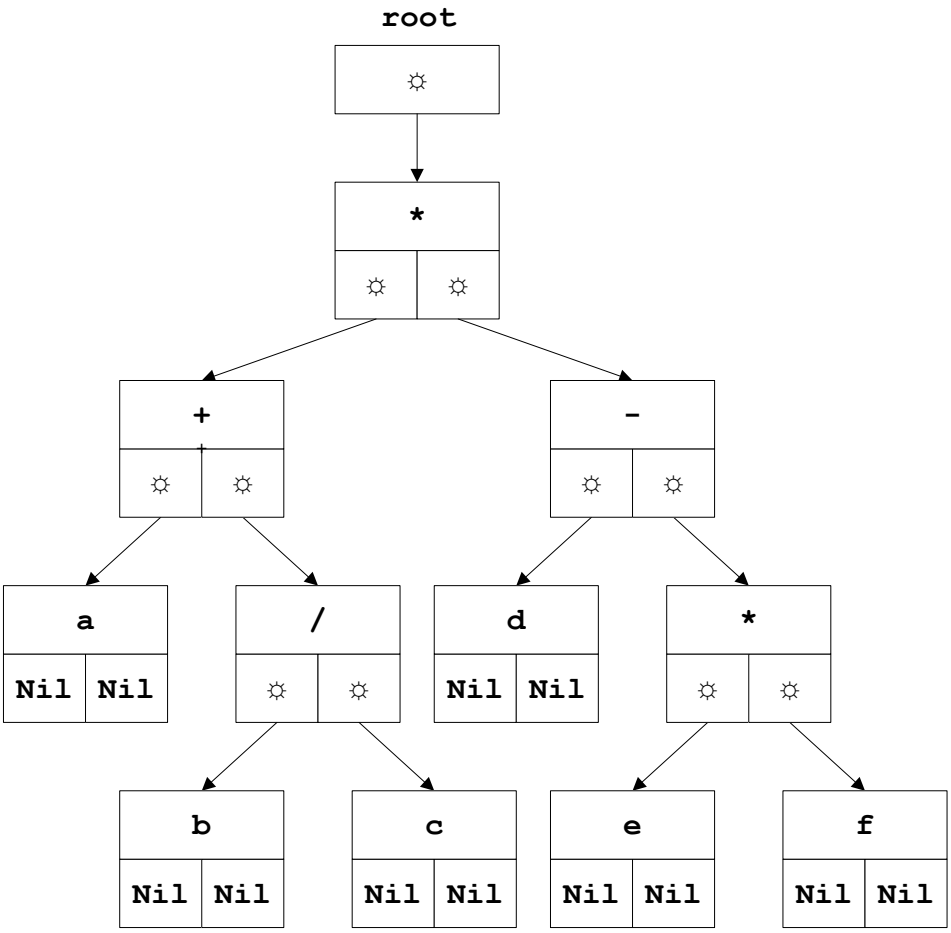
Приведенное выше дерево с помощью массива представляется так: описывается переменная-массив

```
t : array [1..11] of
                    record
                     op    : char;
                     left,
                     rigth : Integer
                     end;
```

значения компонент которой следующие:

Индекс элемента массива	Значение поля записи op	Значение поля записи left	Значение поля записи rigth
1	*	2	3
2	+	6	4
3	–	9	5
4	/	7	8
5	*	10	11
6	a	0	0
7	b	0	0
8	c	0	0
9	d	0	0
10	e	0	0
11	f	0	0

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



Введем следующие определения:

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

Над деревьями обычно выполняются следующие операции:

- добавить в дерево узел;
- 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, значит, и действия над деревьями лучше всего задавать с помощью рекурсии.

Работа с бинарными деревьями

Рассмотрим сначала процедуры левостороннего обхода упорядоченного дерева.

Разные 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 еще 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.

Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?

ТЕМА 17
НЕКОТОРЫЕ ТИПЫ ДЕРЕВЬЕВ

Содержание темы

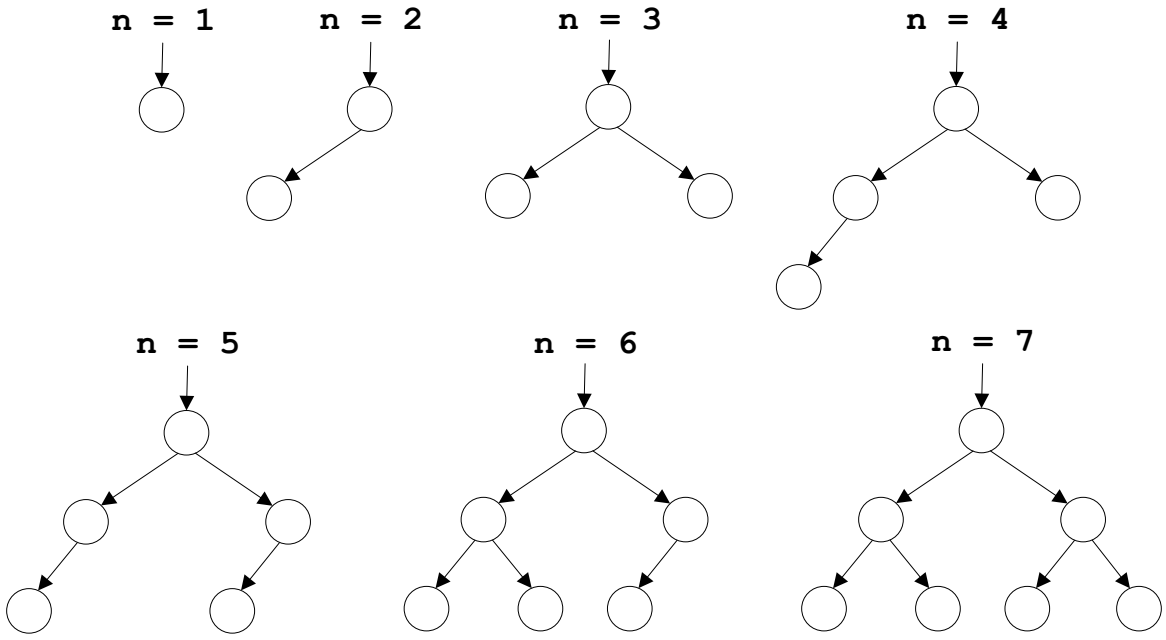
- Идеально сбалансированные деревья.
- Дерево поиска.
- Лексикографическое дерево поиска.

Целеполагание студента

- Какие цели Вы перед собой ставите в рамках данной темы?
- Что интересует?
- Как данный материал повлияет на Ваше умение программировать?

ИДЕАЛЬНО СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ

Мы уже рассматривали деревья-формулы. Существуют еще *идеально сбалансированные деревья*, которые имеют следующий вид:



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

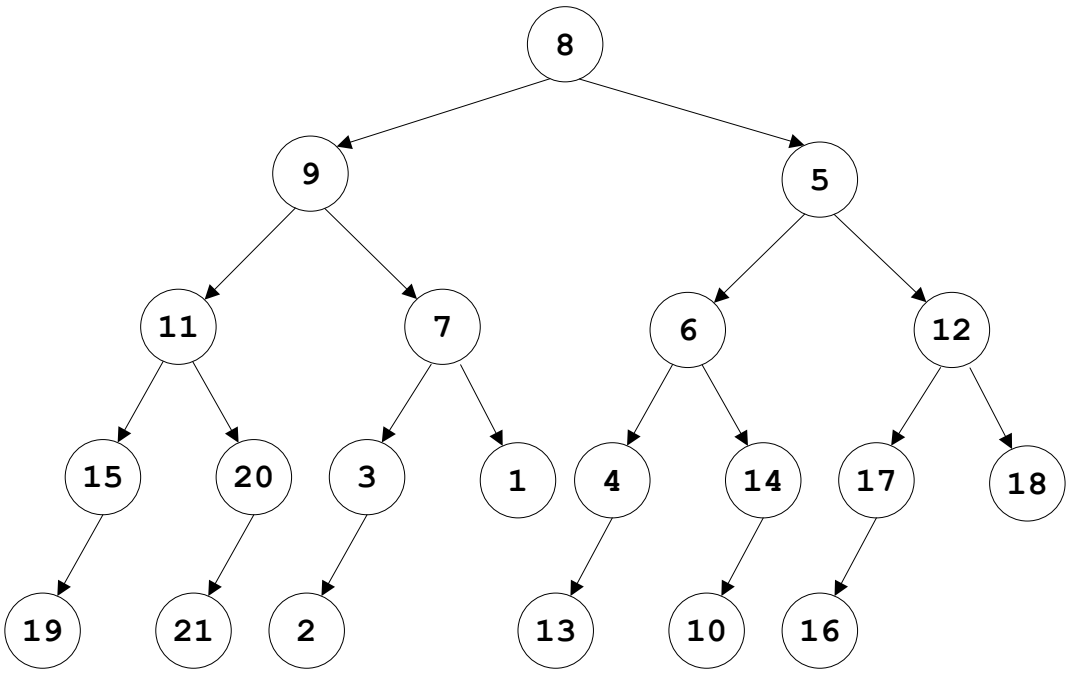
Это правило легко сформулировать при помощи рекурсии:

- 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 следующей 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);
```

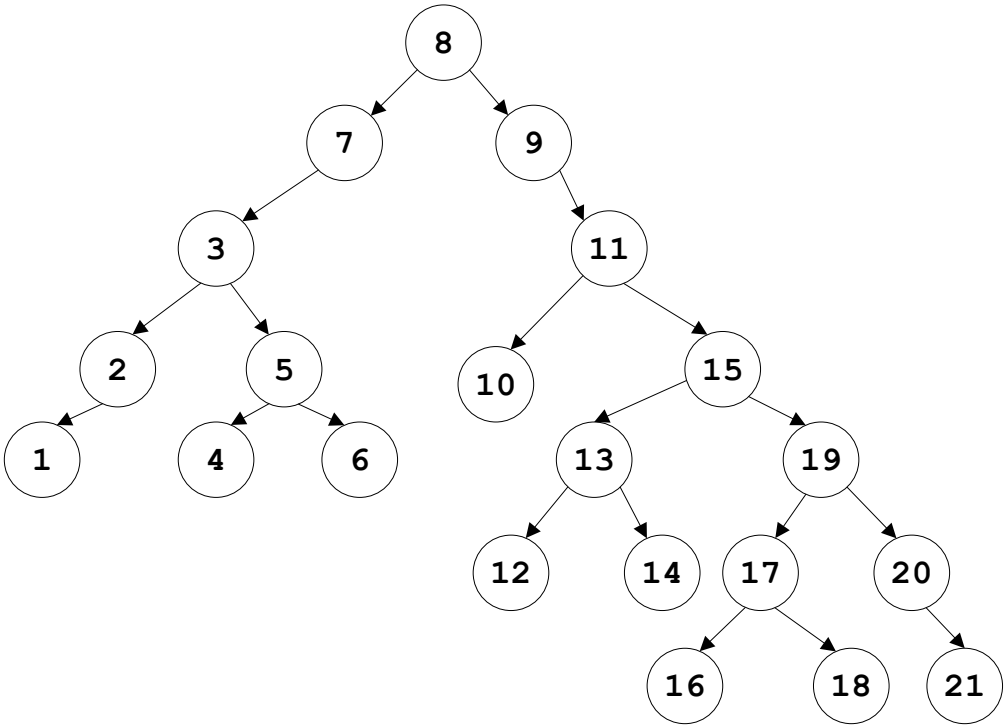
```
end;

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 значения 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). Подсчитать, сколько раз встречается каждое слово (число) в массиве. Распечатать 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, иначе, 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.
```

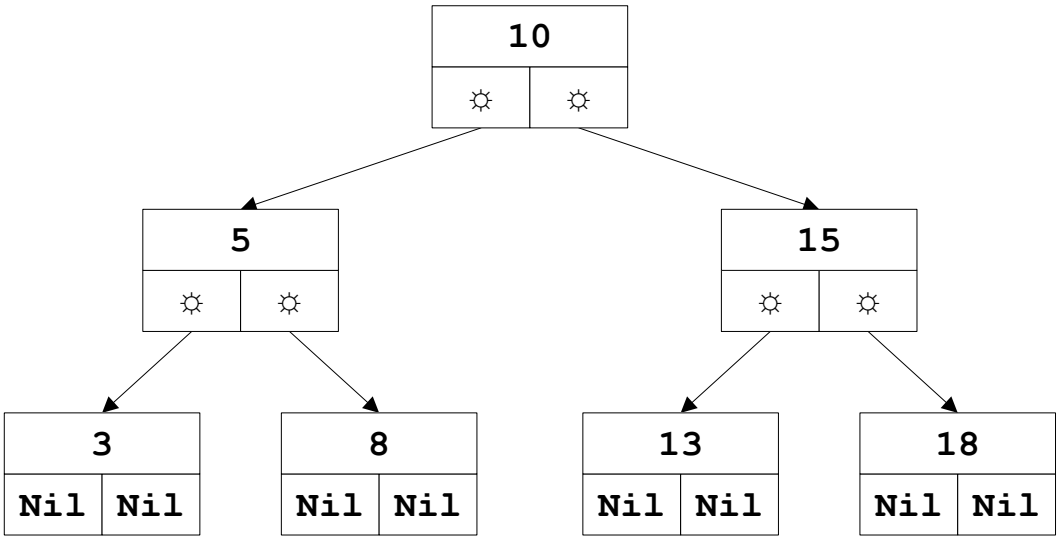
Симметричный обход построенного дерева слева направо даст отсортированный по возрастанию массив чисел, справа налево – по убыванию.

Удаление из бинарного дерева поиска

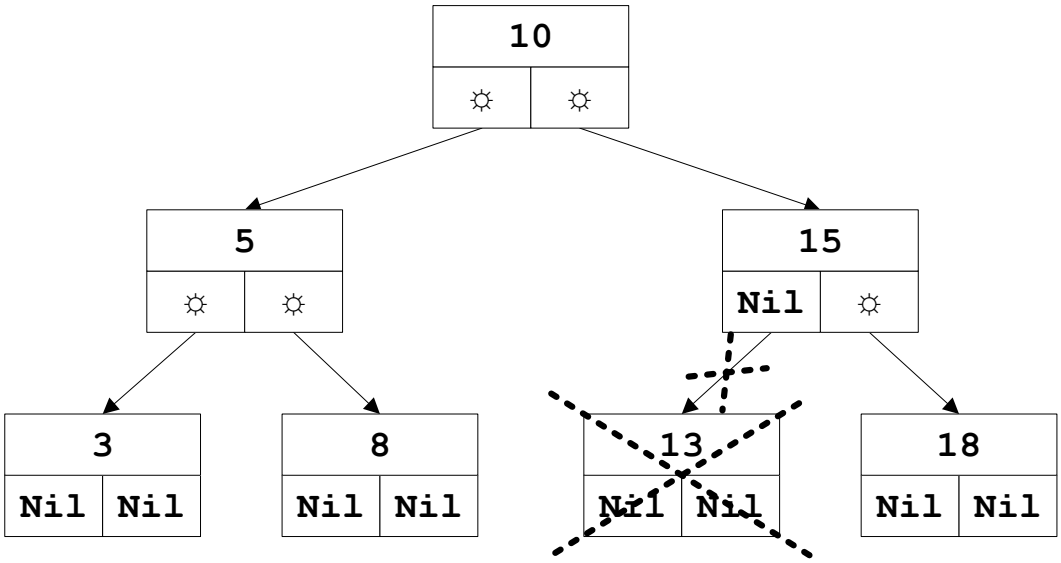
Когда встает задача удаления узла из дерева, то могут возникнуть различные обстоятельства.

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

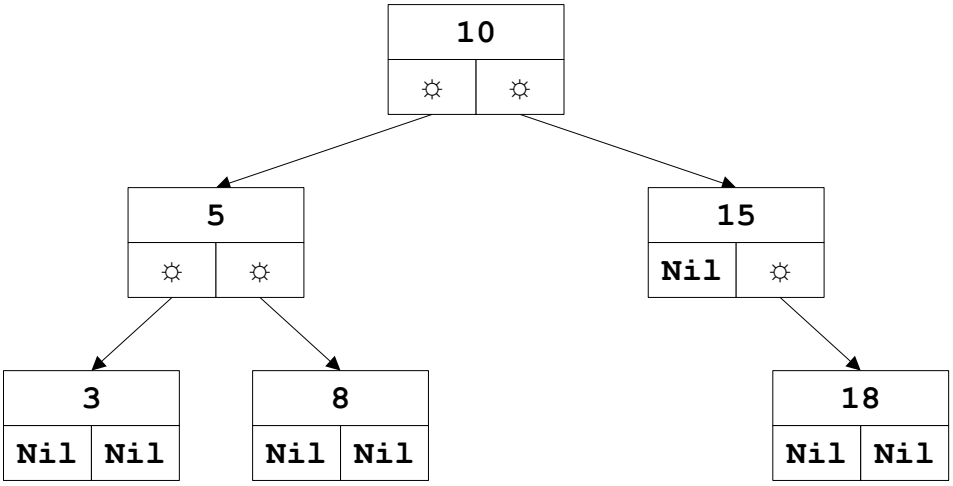
1. Пусть нужно удалить лист следующего дерева (например, элемент «13»):



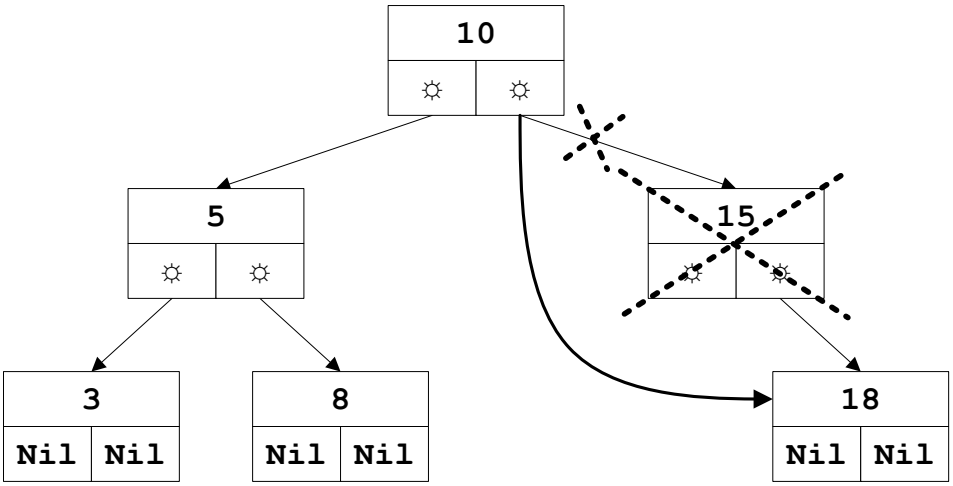
Тогда родительский узел «15» вместо ссылки на дочерний узел должен записать `nil`:



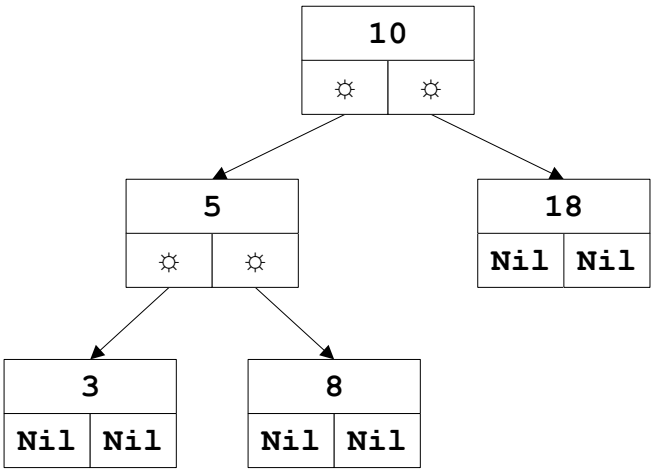
2. Пусть в полученном дереве нужно удалить узел, который имеет одного потомка (например, элемент «15»):



Тогда надо на место родительского узла переместить дочерний узел:

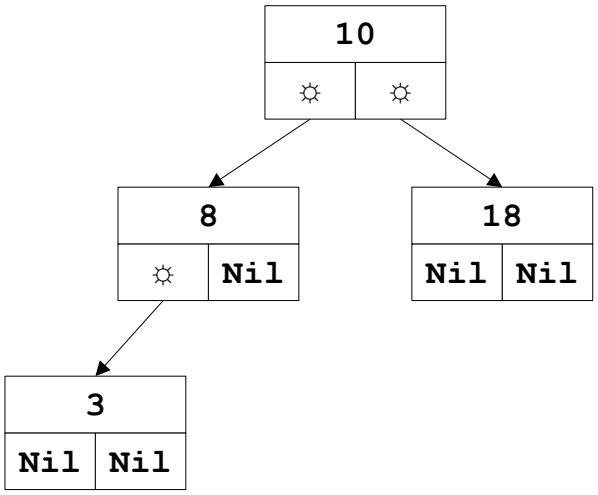


3. Пусть в последнем дереве нужно удалить узел, имеющий двоих потомков (например, элемент «5»):

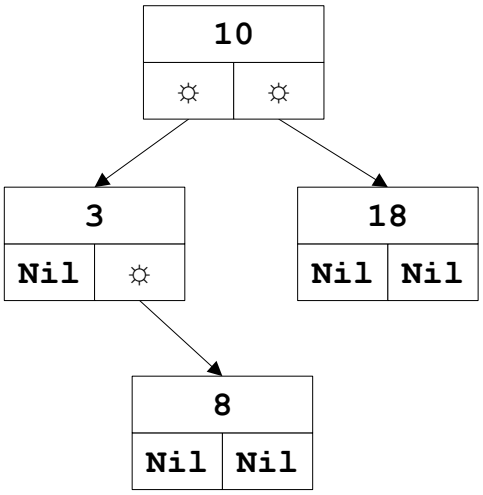


Это можно сделать двумя способами, так как на место родительского узла можно подставить левый (вариант 1) или правый (вариант 2) дочерний узел.

Получим следующие варианты схем удаления:



Вариант 1

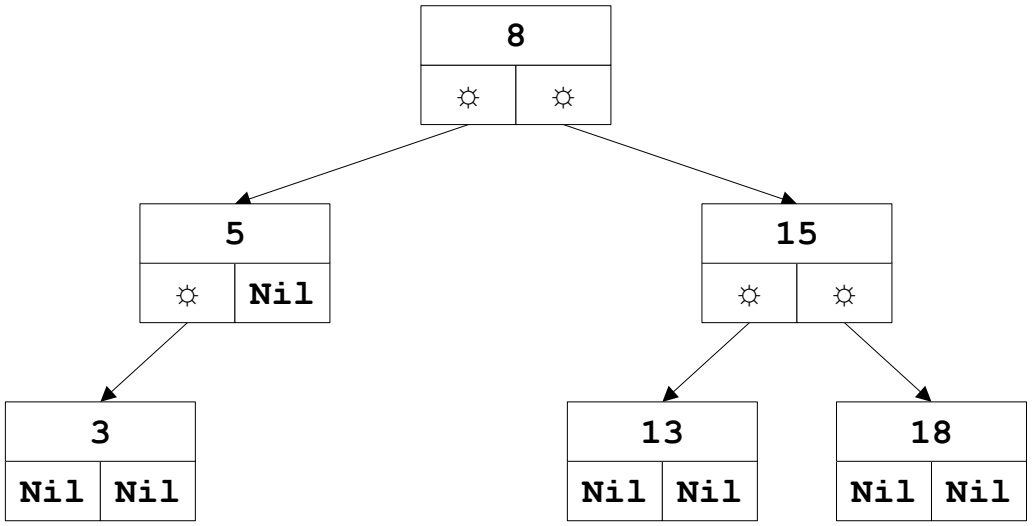


Вариант 2

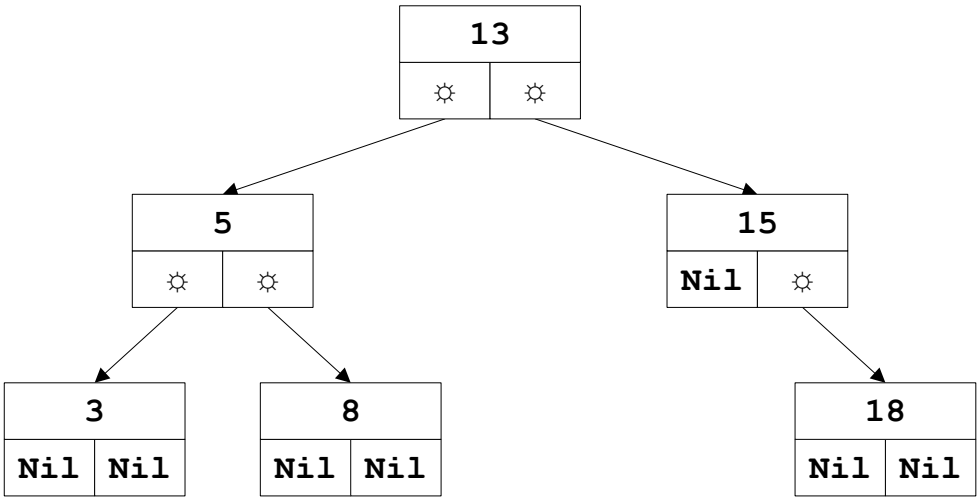
Можно заметить, что ситуация 3 является частным случаем следующей ситуации 4.

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

Р е ш е н и е . Напишем процедуру, которая различает три случая:

1) узел с ключом *x* не нашли;

2) узел с ключом *x* имеет одного наследника;

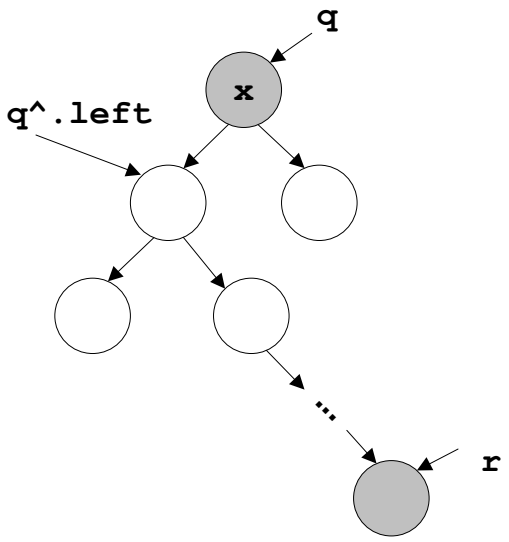
3) узел с ключом *x* имеет двух наследников.

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;
```



Рефлексия

- Каковы были ваши цели перед занятием и насколько их удалось реализовать?
- Перечислите трудности, с которыми вы столкнулись при изучении темы?
- Каков главный результат для вас лично при изучении темы?
- Каким образом вы преодолевали трудности? За счет чего?
- Чему вы научились лучше всего?
- Что вам удалось больше всего при изучении темы и почему?
- Что не получилось и почему?
- Опишите динамику Ваших чувств и настроений при изучении темы (можно в виде графика).

СПИСОК ЛИТЕРАТУРЫ

Расолька, Г. А. Метады праграміравання. Алгарытмы апрацоўкі данных / Г. А. Расолько, Ю. А. Кремень. – Мінск : БДУ, 2008.

Расолько, Г. А. Методы программирования и информатика. Практика использования ООП : учеб. материалы для студентов мех.-мат. фак. специальности 1-31 03 01-02 «Математика (научно-педагогическая деятельность)» / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2014.
http://elib.bsu.by/handle/123456789/94236

Расолько, Г. А. Методы программирования и информатика. Технологии программирования : учеб. материалы для студентов мех.-мат. фак. специальности 1-31 03 01-02 «Математика (научно-педагогическая деятельность)» / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2014.
http://elib.bsu.by/handle/123456789/94232

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

Расолько, Г. А. Электронный учебно-методический комплекс по учебной дисциплине «Методы программирования и информатика» для специальностей 1-31 03 01 «Математика (по направлениям)», 1-31 03 01-02 «Математика (научно-педагогическая деятельность)» / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2014. http://elib.bsu.by/handle/123456789/97905