ВВЕДЕНИЕ В НЕЧИСЛОВОЕ ПРОГРАММИРОВАНИЕ (часть 1)

В.Б.ИЛЬИН, А.Д.ПОЛОЖЕНЦЕВ

С.-Петербург 1994

Аннотация

Данное пособие написано на основе курса программирования, читавшегося в течение нескольких лет студентам второго курса математико-механического факультета Ленинградского государственного университета.

Курс представлял собой дополнение к стандартному курсу программирования, где рассматривается какой-либо язык программирования и даются основы методов вычислений.

В пособии рассматриваются некоторые задачи "нечислового" программирования и их решение на языке "Паскаль". Первая глава посвящена типам данных языка "Паскаль". Во второй главе обсуждаются некоторые структуры данных и их реализация при помощи средств языков "Паскаль" и "Фортран" (предполагается, что с основами языка "Фортран" и "Паскаль" читатель знаком). В третьей главе, на примере игры в "орлянку", рассматривается программирование игровых задач (диалоговых). При разборе данной задачи, основное внимание уделено выбору удобных типов и структур данных.

Введение

1.1 Понятие нечислового программирования

Среди встречающихся в программировании задач проведем разделяющую черту (безусловно сильно расплывчатую), и отделим алгоритмы, где основные трудности вызваны значительным числом арифметических (алгебраических) операций от задач, где главным является не это, а организация данных и другие вопросы (например логика в сильно запутанных алгоритмах, представление данных, существенно отличающихся от числовых и т.д.). Решения на ЭВМ задач первого рода будем называть "числовым" программированием, второго рода - соответственно "нечисловым" программированием.

В чем же различие этих двух видов программирования ? В числовом программировании главная трудность заключается в адекватном применении соответствующей методики из области "методов вычислений" и в борьбе с потерей точности, возникающей при большом числе последовательных арифметических операций, вследствие приближенного представления вещественных чисел в ЭВМ.

В нечисловом программировании акцент смещается с этих проблем на организацию данных и другие вопросы, поскольку количественное увеличение объема данных (как, например, в случае задач сортировки), качественное изменение данных (как при работе с аналитическими выражениями) или иное отношение к данным (как в случае сервисных программ) требует применения специальных алгоритмов (т.е. данные и логика становятся определяющими для алгоритма).

1.2 Задачи нечислового программирования

Перечислим основные задачи нечислового программирования, возникающие на границе информатики и астрономии.

Большую часть данных, как известно, астрономия получает в экспериментах. Среди программ, обеспечивающих автоматизацию экспериментов, подавляющее число составляют задачи нечислового программирования: программы управления работой приборов (телескопы, приемники и т.д.), первичной обработки информации (введение поправок за эффекты, вносимые приборами) и организации информации (т.е. кодировка, упаковка, распаковка информации и т.д.). Примером управлящих программ являются программы автоматического гидирования объектов на телескопах, управление многокомпонентными радиотелескопами (например РАТАН-600) и др.; первичной обработки — вычитание инструментального профиля в спектроскопии, вычитание помех вносимых космическими лучами в наблюдениях с ПЗС-матрицами и т.д.; организация информации — упаковка двухмерных изображений астрономических объектов для экономии памяти ЭВМ при их хранении и т.п.

Следующий этап работы с результатами наблюдений в астрономии — обработка данных. Здесь задачи нечислового программирования возникают реже и в основном в тех случаях, когда объем данных велик (например, при обработке изображений) или когда повышаются требования к сервису (для больших программ, предназначенных для длительного и широкого использования).

Некоторая (хотя и не очень большая) часть обработанных данных, предлагаемая для использования широкой астрономической общественностью, передается для включения в банки данных в центры данных (например ЦАД при астрономическом совете АН СССР в Москве и/или международный ЦАД в Страссрбурге). При работе с банками данных основными являются программы нечислового программирования. В простейшем случае каталога данных встречаемость нечисловых программ существенно ниже и зависит обычно от объема каталога и назначения комплексов программ.

Присутствуют задачи нечислового программирования и в теоретических исследованиях. В частности специальные трансляторы, интерпретаторы и т.п. Позволяют автоматизировать кодировку (обычно крупных, однообразных) программ. Например, в физико-химии атмосфер планет и межзвездной химии применяются интерпретаторы, воспринимающие близкую к обычной запись химических реакций и строящие соответствующие программные модули. Относительное число таких программ очень мало, но их объем и растущая нелинейно с ним трудоемкость создания и отладки, приводят к тому, что затраты на их создание (длительность создания и неизбежных модификаций годы !) могут составить значительную долю времени работы программистов (астрономов).

Наивысшим достижением нечислового программирования, являются, по-видимому, системы для работы с аналитическими выражениями. Такие системы в настоящий момент "умеют" производить интегрирование, не говоря о дифференцировании или приведении подобных членов, любых алгебраических выражений, включающих как полиномы, так и все алгебраические функции вплоть до специальных (система MACSIMA). Специальные виды таких систем (SCHOOLSHIP, REDUCE и т.д.) используются в физике элементарных частиц, релятивисткой квантовой механике и небесной механике.

На всех этапах деятельности астрономов иногда возникает необходимость создания сервисных программ, т.е. программ на которые налагаются особо жесткие требования (что конечно приводит к модификации исходного алгоритма). Обычно эти требования таковы: усиленная защита от ошибок, надежность, документированность, улучшенная форма выдачи результатов и т.д. Отличный пример сервисных программ - игровые программы !

Подводя итог, перечислим снова наиболее типичные задачи нечислового программирования: управляющие программы; программы, работающие с очень большими объемами данных (включая программы сортировки); программы, работающие с информацией, значительно отличающейся от числовой и, наконец, сервисные программы любого назначения.

В заключении, стоит отметить вероятно неожиданный для многих факт, что астрономия и программирование связаны не столь слабо, как кажется на первый взгляд. Примерами влияния астрономии на программирование может служить то, что именно астрономические потребности привели к созданию первых вычислительных машин или же широкая распространенность языка "FORTH", созданного для управления радиотелескопом; обратное влияние - некоторые изменения образа жизни астрономов, в частности в ряде стран астрономы не ездят наблюдать в обсерватории, а подают со своего терминала заявку и через некоторое время получают на него результаты наблюдений (трансконтинентальная связь европейская южная обсерватория (ESO) в Чили - Англия).

Типы данных

В данной главе рассматриваются различные группы (типы) данных, которые присутствуют в современных языках программирования (а не только в Паскале или Фортране!)[1].

Сначала обсуждаются обычные, элементарные типы данных (п. 2.2–2.3), встречающиеся во всех языках. Затем описываются составные типы данных (п. 2.4), в которых каждый элемент данных есть некоторая совокупность объектов. Использование данных этих типов оказывается удобным при "нечисловом" программировании. В заключение (п. 2.5) обсуждается существенный для многих аспектов "нечислового" программирования тип данных — ссылочный. Этот тип исключительно важен при построении в программах нелинейных структур данных (например, ветвящихся или с большим числом связей).

Отметим, что пункты, в которых обсуждаются известные читателю типы, могут быть пропущены, исключая п. 2.4.1 и 2.5, где излагаются сведения, необходимые для понимания Главы 3.

2.1 Данные и типы данных

Любой алгоритм (программа) для вычислительной машины состоит из двух основных частей: описание действий, которые предстоит выполнить, и описания объектов, над которыми будут выполняться эти действия. Обычно при обучении программированию основное внимание уделяется первой части, мы рассмотрим не менее важную, вторую часть.

Объекты, которыми оперирует программа, будем называть данными. Важной особенностью данных является то, что они имеют два атрибута: имя и значение. Если перейти на язык машинных команд, то имени будет соответствовать номер ячейки, а значению — содержимое ячейки. В аппаратуре ЭВМ все данные представляют собой последовательность двоичных цифр, т.е. ряды, состоящие из нулей и единиц. Такие данные использовались в прошлом при программировании на языке машинных команд (в машинах первого поколения: Урал-1 и др.). Языки программирования высокого уровня (Алгол, Фортран и др.) позволяют абстрагироваться от неудобных деталей представления данных и происходит это в основном за счет введения концепции типов данных.

[1]Эта глава имеет характер "ликбеза". Основное содержание курса находится в Главе 3 и иллюстрирующей ее Главе 4 (Глава 4 приведена в сокращенном варианте).

Определение: типом данных будем называть класс (группу) объектов, значения которых принадлежат определенному множеству.

Например, целый тип данных, встречающийся почти во всех языках программирования, объединяет объекты, значения которых являются целыми числами.

Обычно введением типов данных объекты, используемые в алгоритмах (программах) в виде переменных или констант, разбиваются на несколько групп по общим областям значений. При этом каждая переменная сопровождается спецификацией ее возможных значений. Это делается по следующим причинам: во-первых, количество элементов в памяти ЭВМ, необходимое для хранения значений переменной зависит от диапазона ее значений. Так для хранения переменной, принимающей N целых значений, необходимо примерно $\log N$ битов. В частности для переменной, значениями которой могут быть 0, 1 и 2 необходимо два бита (в двоичном коде значения есть 00, 01 и 10). Во-вторых, реализация действий, описанных в алгоритме, в командах ЭВМ часто (почти всегда) зависит от количества элементов памяти, которое занимает объект, так как используются различные регистры машинных команд (по крайней мере, в одноадресных серии ЕС ЭВМ и др.)

И так множество значений, которое может принимать переменная играет важнейшую роль для характеристики данных. Поэтому, обычно, все переменные в программах должны быть описаны (исключая правило умолчания некоторых языков программирования) или иными словами отнесены к какому-то типу данных[2].

Типы данных разделяются на простые и составные.

Определение: простые типы данных — классы объектов, каждый из которых представляет собой единое целое, т.е. атом.

Определение: составные типы данных — классы объектов, каждый из которых включает в себя несколько элементов.

Примером простого типа является целый, вещественный тип и т.п., а составного — массивы.

2.2 Простые типы данных

Во многих современных языках программирования разрешается, кроме стандартных (простых) типов: целого, веществено, логического и литерного — вводить и свои типы данных. Для этого в языке Паскаль существует понятие скаляртного типа.

2.2.1 Скалярный тип данных

Определение : скалярный тип данных — класс объектов, которые могут принимать значения только из некоторого заданного упорядоченого множества.

Определение скалярного типа в языке Паскаль выглядит следующим образом:

$$\boxed{\text{TYPE Ts = (w1, w1, ..., wn)}}$$

[2]Описание переменных в языке Паскаль имеет вид: VAR v1, v2, ..., vn: T, где VAR — служебное слово языка Паскаль; v1, v2, ..., vn — имена переменных (индентификаторы); T — имя типа данных.

где TYPE — служебное слово языка Паскаль;
Ts — имя скалярного типа данных;
w1, w2, ..., wn — константые индетификаторы.

Константные индентификаторы должны быть различны; их упорядочиваемость задается порядком следования так, что считается w1 < w2 < ... < wn.

На переменных скалярного типа определены операции:
отношения: $<, >, =$ и $<>$;
и функции получения следующего и предыдущего элемента SUCC и PRED (например, результат SUCC(w2) есть w1 и т.д.).

Пример. Определим скалярный тип, перечисляющий планеты Солнечной системы, и выполним ряд тривиальных операций с перемеными данного типа.

```
TYPE  Planets = (me, v, e, ma, j, s, u, n, p); ...
VAR   planet1, planet2: Planets;  l: BOOLEAN; ...
  ... planet1:= e;  planet2:= V;
      l:= planet1 >= planet2; ...
```

В данном случае переменная l примет значение FALSE.

Отметим, что целью введения скалярных типов данных является абстрагирование от машинно ориентированного представления данных (ряды 0 и 1) так, чтобы работать с объектами в удобной человеку форме, т.е. максимально приближенной к реальной. Следующий шаг в этом направлении — определение специальных операций на новых типах. В некоторых языках (Алгол-68, FORTH и др.) это можно делать явно. В языке Паскаль это делается косвенно, через подпрограммы.

2.2.2 Ограниченный тип данных

На определении скалярного типа базируется определение его части, называемой ограниченным типом.

Определение: ограниченный тип данных — класс объектов, которые могут принимать значения из указанного связного подмножества некоторого множества (задаваемого определением базового скалярного типа).

В языке Паскаль определение данного типа имеет вид:

$$\boxed{\text{TYPE To} = \text{wi .. wl}}$$

где .. (две точки!) — специальный знак языка;
wi и wl – константные индентификаторы, входящие в базовый скалярный тип (заданый выше).

Переменные ограниченного типа могут принимать значения константных идентификаторов скалярного типа Ts в интервале от wi до wl включительно.

На ограниченном типе разрешены те же операции, что и на базовом скалярном типе.

Пример. В массиве индексы элементов ограничены некоторым целым числом. Ограниченный тип, удобный для описания индексов массива в 100 элементов, может быть определен на стандартном скалярном типе (целом типе) так

$$\text{TYPE Index} = 1 \ .. \ 100$$

2.3 Стандартные типы данных

Некоторые скалярные типы используются столь часто (и давно!), что соответствующее определения типа и его стандартное имя заложены в языке. В Паскале стандартными скалярными типами являются: логический, целый, литерный и вещественный[3]. Эти типы присутствуют и в большинстве других языках программирования.

2.3.1 Логический тип данных

Определение: логический (или булевский) тип данных — класс объектов, которые могут принимать лишь два значения: FALSE (ложь) и TRUE (истина).

В языке Паскаль подразумевается следующее описание данного типа, имеющего стандартное фиксированное имя BOOLEAN:

$$\text{TYPE BOOLEAN} = (\text{FALSE, TRUE})$$

и для данного типа определены следующие операции:
логические OR, AND, NOT;
отношения $<, >, =, <>$;
тип результата этих операций всегда BOOLEAN.

Пример. Следущие операторы программы:

```
... VAR  x, y, z, u: BOOLEAN;
         y:= FALSE;  z:= TRUE;
         x:= Y > Z;  u:= y OR z; ...
```

приведут к тому, что переменная x будет иметь значение FALSE, а u — TRUE.

Отметим, что тип BOOLEAN вызван (связан) существованием в алгоритмах структур (действий) типа развилка: IF ... THEN ... ELSE ... .

2.3.2 Целый тип данных

Для каждой вычислительной машины определено некоторое подмножество целых чисел, таких с которыми ее арифметическое устройство может оперировать. В ЕС ЭВМ на целое число отводиться 4 байта = 32 битам и следовательно можно представлять числа в интервале $[-2^{31}, 2^{31}]$.

Определение: целый тип данных — класс объектов, которые могут принимать значения из множества целых машинных чисел.

Подразумевается следующее (очень длинное!) описание в Паскале данного типа (стандартное имя INTEGER):

$$\text{TYPE INTEGER} = (\text{minint, minint+1, minint+2, ..., maxint-1, maxint}),$$

где minint и maxint — соответственно максимальное и минимальное число из множества машинных целых чисел.

[3]Вещественный тип, стоит особняком среди скалярных типов, т.к. на нем нельзя определить ограниченный тип. Накладывание такого ограничения в языке, связано с тем, что определение ограниченного типа на вещественном на разных машинах будет не однозначно (зависит от разрядности машины).

Разрешены следующие операции для целых чисел:
арифметические $+, -, *$, DIV (деление), MOD (взятие остатка) — результат операций типа INTEGER;
отношения $<, >, <=, =>, =, <>$ — результат BOOLEAN.

Существует и ряд встроенных функций.

Подчеркнем, что обычные аксиомы арифметики могут быть не верны для машинной арифметики. Например для машинного сложения двух целых чисел x и y, результат будет не определен, если (x+y) < minint или (x+y) > maxint[4].

2.3.3 Символьный тип данных

Определение: символьный тип данных — класс объектов, которые могут принимать значения литер: 26 латинских букв, 10 арабских цифр и несколько специальных символов, включая пробел (для ЕС ЭВМ).

Подразумевается следующее описание в Паскале:

TYPE Char = ('a','b','c',...,'Z','0',...'9',' ','/','*',...)

т.е. литеры упорядочены некоторым образом.

Разрешены следующие операции: отношения: $<, >, <=, =>, <>$ — результат BOOLEAN.

Отметим, что символьный тип обычно используется при операциях ввода/вывода в программах.

2.3.4 Вещественный тип данных

В вычислительной машине можно представить числа достаточно разреженного конечного подмножества вещественных чисел. В современных цифровых машинах, используется представление вещественного числа с "плавающей точкой", т.е. каждое вещественное число x представляется парой чисел m и i: $x = m \cdot b^i, i = [-K, K], m = [-N, N]$, где b — основание, i — порядок, а m — мантисса числа x. Если $1 > |m| > 0.1$, то форма представления числа называется нормализованной. В машинах серии ЕС ЭВМ для $b = 10$, m — приблизительно семь десятичных знаков, а $i = [-79, 75]$.

Очевидно, что при использовании такого представления плотность "машинных" вещественных чисел уменьшается с их ростом: в интервале 0.1–1.0 содержится столько же вещественных "машинных" чисел, сколько и в отрезке 1000.0–10000.0! Трудно предсказать к чему это может привести в той или иной программе.

Определение: вещественном типом данных будем называть класс объектов, которые могут принимать значения из множества "машинных" вещественных чисел.

Подразумевается очень-очень длинное описание (в языке Паскаль)

TYPE REAL = (minreal, ..., maxreal),

[4]Отметим, что целый тип часто связан с конструкциями программ типа цикла (счетчик цикла) FOR ... DO ...

где должны быть перечислены в порядке возрастания все "машинные" вещественные числа.

Разрешены следующие операции:
арифметические: $+, -, *, /$ — результат типа REAL;
отношения: $<, >, <=, =>, =, <>$ — результат BOOLEAN.

Доступен обычно и ряд широко используемых встроенных функций.

Отметим, что как и в случае целого типа, аксиомы арифметики могут быть не выполнимы для переменных вещественного типа. В частности, нет законов ассоциативности и дистрибутивности, что связано с ограничением длины мантисы у "машинных" вещественных чисел. Из-за этого особую опасность представляет вычитание двух примерно равных чисел. При этом большое количество значащих цифр уничтожается и в полученной разности может оказаться очень мало верных цифр. Например, вычитание из числа 0.2013 числа 0.2008, на машине с длиной мантиссы 4 будет происходит следующим образом:

```
  0.2013    10**0
 -0.2008    10**0
--------------------
      .5xxx 10**(-3)
```

и в мантису результата попадет 5 и еще три цифры (xxx), которые, например, машины серии ЕС ЭВМ "выдумают" сами. Т.о. в результате останется лишь одна верная цифра!

Отметим, что вещественный тип данных необходим в программе для вычислений.

Упражнения

1. Определен скалярный тип имен спектральных классов звезд:

TYPE Spectral = (O, B, A, F, G, K, M)

известно, что в переменных этого типа s1 и s2 находятся имена спектральных классов двух звезд. Написать оператор, проверяющий: являются ли классы звезд соседними.

2. Добавить описание переменных к имеющейся программе:

b:= a=1 OR c=1.0

3. Дано описание переменных:

VAR kh, y, z: REAL; i, j, k: INTEGER

Проверить правильно ли записаны выражения:

а) x + y * i
б) x + k < i + j
в) i DIV j + x

4. Описать типы данных, удобные для работы:
а) с хабловской классификацией галлактик;
б) с оценками учащихся в школе.

2.4 Составные типы данных

Выше мы рассмотрели простые типы элементарных данных, т.е. данных, значения которых являются неделимыми. Однако, часто приходится работать с объектами, представляющими собой некоторую совокупность данных ("классический" пример — массив). При этом может потребоваться совокупность данных разного типа, с которыми необходимо работать как с одним целым. Например, с датой или со сведениями о каком-то объекте: человеке, звезде и т.п. В этих случаях удобно использовать так называемый комбинированный тип данных.

2.4.1 Комбинированный тип данных

Определение: комбинированный тип данных (или записи) — класс объектов, которые состоят из фиксированного числа элементов разного типа.

На записях разрешены только операции присваивания, но нужные нам операции могут быть определены дополнительно (по крайней мере, через подпрограммы, как в языке Паскаль). Для компонентов записи определены все операции, разрешенные для их типов данных.

Описание в языке Паскаль имеет вид:

$$\boxed{\text{TYPE Ts = RECORD p1: T1; p2: T2; ...; pn: TN END}}$$

где Tc — имя комбинированного типа;
RECORD и END — зарезервированные слова Паскаля;
T1, T2, ..., TN — имена типов компонентов записи;
p1, p2, ..., pN — имена полей, необходимые для работы с компонентами.

Для переменной с комбинированого типа Ts (описание: VAR s: Ts) s.p1 — есть компонент типа T1; s.p2 — компонент типа T2 и т.д.

Пример. Определим комбинированный тип с именем Date для представления даты и присвоим некоторые значения компонентам переменной today данного типа.

```
TYPE  Tdate = 1..31;
      Tmonth = (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG,
                SEP, OCT, NOV, DEC);
      Tyear = 0..2005;
      Date = RECORD day: Tdate;
                    mon: Tmonth;
                   year: Year
             END;
VAR   today: Date;
BEGIN
   today.day:= 23; today.mon:= MAR; today.year:= 1995
END.
```

2.4.2 Регулярный тип данных

Часто встречается частный случай комбинированного типа, когда компоненты переменной имеют одинаковый тип. Этот случай выделен особо в так называемый регулярный тип.

Определение: регулярный тип данных (или массивы) — класс объектов, которые состоят из фиксированного числа элементов одного и того же типа (в отличие от записей, у которых тип элементов может быть разным!)

Описание на языке Паскаль данного типа есть:

$$\boxed{\text{TYPE Tr = ARRAY [Ti] OF Tk}}$$

где Tr — имя регулярного типа;
ARRAY и OF — зарезервированные слова в языке Паскаль;
Ti — имя типа данных индексов массива — обязательно какой-либо скалярный тип (кроме вещественного)!
Tk — имя типа компонентов.

Так же, как и для записей, определена лишь одна операция — присваивание (однако можно косвенно вводить и свои специфические), а для компонентов массива разрешенны все операции их типа.

Описание переменной регулярного типа в языке Паскаль и работа с ее компонентами выглядит следующим образом:

```
... VAR  b,a: Tr;  i,j: Ti;  c: Tk;...
... c:= a[i];  a[i]:= a[j];  a[j]:= c;  b:= a...
```

Возможна работа и с многомерными массивами (тоже часто встречается на практике). В языке Паскаль двумерный массив описывается так:

$$\text{TYPE Ta = ARRAY [Ti,Tj] OF Tk}$$

где Ti и Tj — типы индексов (первого и второго).

Пример. Для представления даты и работы с ней можно использовать не только записи (см. комбинированный тип в п. 1.4.1), но и массивы:

```
TYPE   Ti = 1..3; Tk = 0..2005;
       Data = ARRAY [Ti] OF Tk;
VAR    tomorow: Data;
BEGIN
   tomorow[1]:= 24;
   tomorow[2]:= 3;
   tomorow[3]:= 1995
END.
```

Примечание. Ясно, что работа с записями в данном случае нагляднее и, следовательно, их использование предпочтительнее (если постановка задачи не требует использования массивов).

2.4.3 Множественный тип данных

Иногда (не так редко!) возникает необходимость работать с множеством (объектов) и тогда удобно воспользоваться так называемым множественным типом.

Определение: множественный тип — класс объектов, значениями которых являются подмножества некоторого фиксированного множества (последнее задается описанием скалярного типа). Описание типа в языке Паскаль выглядит так:

TYPE Bset = (b1, b2, ..., bn);

$$\boxed{\text{Tm = SET OF Bset}}$$

где Bset — имя базового скалярного(!) типа;
b1, b2, ..., bn — константные идентификаторы типа Bset;
SET OF — зарезервированные слова языка Паскаль.

Определены следующие операции:
а) отношения: $<=, =>$ (включение множеств!) и $=, <>$ (тождество и различие) — тип результата BOOLEAN;
б) над множествами: + (объединение); * (пересечение);
IN (принадлежность элемента множеству) — тип результата BOOLEAN.

Описание переменных множественного типа и работа с ними выглядит следующим образом:

```
VAR  t1, t2: Tm;  L: BOOLEAN;
BEGIN
   t1:= [b2, b5, b8];
   l:= b5 IN t1
END.
```

где t1 — подмножество из трех элементов, а l в результате работы данной программы будет иметь значение TRUE.

Пример. Два объекта наблюдались в различных фотометрических полосах (U,B,V,R). Программа, которая определяет имеются ли наблюдения объектов хотя бы в одной общей полосе может быть следующей:

```
TYPE  Phband = (U, B, V, R);  Obj = SET OF Phband;
VAR   obj1, obj2: Obj;  L: BOOLEAN;
BEGIN
   L:= NOT( (obj1 * obj2) = [])
END.
```

где [] — пустое множество.

2.5 Ссылочный тип данных

Как отмечалось выше, вся информация в памяти ЭВМ представляется в виде множества нулей и единиц, т.к. память состоит из элементов, хранящих по 1 биту информации (элемент имеет только два состояния, одному приписывается

"0", а второму "1"). Однако обмен информацией между памятью и арифметическим устройством ЭВМ происходит не по 1 биту (это очень медленно), а группами бит — словами (и даже группами слов). В ЕС ЭВМ, например, 32 разрядное слово, т.е. содержащее 32 бита (или 4 байта). Для возможности обращения к словам (ячейкам памяти, как говорили раньше), каждое слово имеет свой номер — адрес. Таким образом с любым объектом, с которым оперирует ЭВМ и который располагается в ее памяти, связано два числа: адрес ячейки, в которой хранится информация (об объекте), и содержимое ячейки, т.е. непосредственно сама информация. Язык Паскаль (как и многие другие языки) позволяют хранить в качестве информации адреса ячеек. Объекты значения которых есть адреса ячеек называются ссылками.

Определение: ссылочный тип данных — класс объектов, которые могут иметь значения всех возможных адресов (в памяти машины) объектов некоторого типа.

Описание ссылочного типа в языке Паскаль:

TYPE Tl=@T

где TL — имя ссылочного типа;
T — имя типа, на данные которого определяются ссылки.
Этот тип может быть как простым, так и составным;
@ — специальный символ языка Паскаль, для определения ссылочного типа (символ зависит от версии).

Описание переменных выглядит обычно:

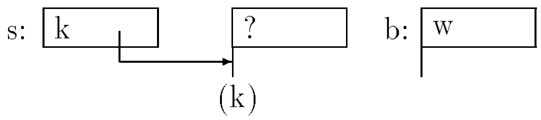
VAR s, p, q: Tl; a, b: T;

Разрешенными операциями на ссылочном типе являются:
а) проверка тождества или неравенства: = и <>;
б) разименование: @ это унарная операция (как логическое отрицание .NOT. для булевского типа), однако ее знак ставится не перед операндом, а после него: S@. Это означает переход от адреса (ссылки) к содержимому ячейки. Т.е., если s — является переменной ссылочного типа, то результат операции разименования s@ — того же типа, на котором определены ссылки (в примере — T).

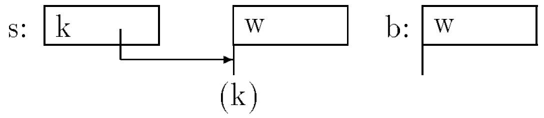
Примеры. Поясним что означают следующие операторы:

а) s@:= b — адрес, содержащийся в s остается прежним, а вот содержимое ячейки по этому адресу, бывшее ранее равным s@, теперь будет равно значению переменной b.

До операции

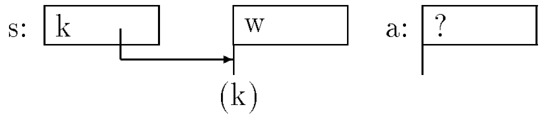


где (k) — адрес ячейки
и после операции s@ := b

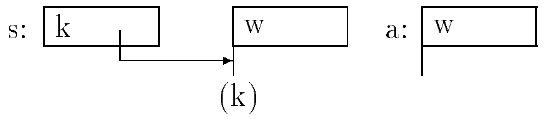


б) a:= s@ — переменной a присваивается значение, хранящееся в ячейке с адресом равным значению находящемуся в переменной s.

До операции

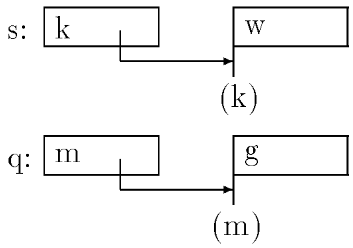


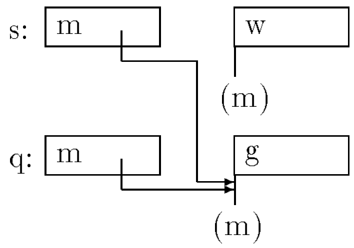
и после операции a:= s@



в) s:= q — в переменной s хранился некий адрес, теперь он будет будет заменен на адрес, храняшийся в q.

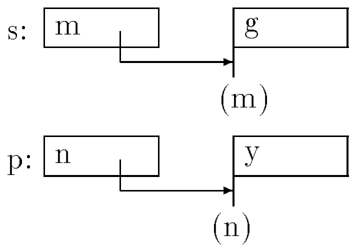
До операции s := q и после операции

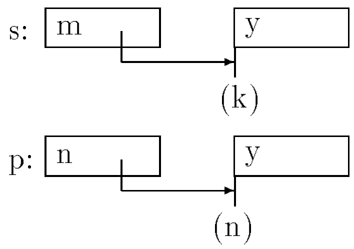




г) s@:= r@ — в переменных s и r хранятся разные адреса, но значения в ячейках по этим адресам теперь будут одинаковыми.

До операции s@ := r@ и после операции





Существует стандартная ссылка в ”никуда”: NIL. Если s:= NIL, то переменной s будет присвоена ссылка на определенный, ”известный” машине, но фиктивный адрес.

Важная стандартная процедура: NEW(p), где NEW — имя процедуры, p — параметр ссылочного типа. В результате работы этой процедуры переменной p будет присвоен адрес (первого слова) участка памяти, отведенного ЭВМ для размещения одного объекта некоторого заданного типа. Тип определяется в соответствии с ссылкой p. Таким образом отводится место достаточное для размещения объекта, на который определена ссылка p. Обратная процедура — DISPOSE (p).

Пример. Отведем в памяти место для размещения целого числа и зашлем в ячейку цифру "5":

```
TYPE  ref = @INTEGER;
VAR   p: ref;
BEGIN
   NEW(p); p@:= 5
END.
```

Упражнения

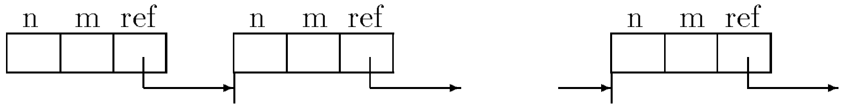
1. Что напечатает программа:

```
TYPE  ref= @REAL;
VAR   s, q: ref
BEGIN
   NEW(s); NEW(q)
   q@:= 4.0;  s:= q;
   WRITE ( s@);
   q@:= 5.0
   WRITE ( s@)
END.
```

2. Создать структуру в виде стека (по принципу — первый пришел и последний ушел) для запоминания заранее неизвестного количества пар целых чисел, используя следующие типы данных:

```
TYPE  Stack = @Elem
      Elem = RECORD n: T1;
                    m: T2;
                    ref: Stack
             END;
        T1 = 1..200;
        T2 = -200..200;
```

Стек на рисунке для данной задачи будет выглядить так:



3. Аналогично упр. 2 написать программу, компактно хранящую разреженную матрицу, т.е. матрицу, содержащую очень много нулевых элементов.

Структуры данных

Определение: Под структурой данных мы будем понимать не только множество каких-то объектов, но и множество связей между этими объектами.

Примерами структурированных данных являются: генеалогическое дерево человека, содержащее информацию о его родственниках и связи между ними; географическая транспортная карта, где даны сведения о пунктах и указаны существующие между ними (м.б. не все) дороги; модели сложной молекулы и т.д. Более простыми структурами являются списки объектов (в частности людей) с указанием их характеристик, каталоги и т.п.

Структуры данных и их представление — один из наиболее важных вопросов нечислового программирования. Понимание некоторых аспектов этого вопроса требует обычно значительных усилий и усложняется отсутствием их ясного описания в существующей литературе.

Мы остановимся на определении основных понятий, описании наиболее существенных технических деталей представления различных структур данных (списков, деревьев и т.д.) и иллюстрации работы с ними.

3.1 Динамические структуры данных

Отметим, что как число объектов в структурах данных, так и связи между этими объектами могут изменяться со временем.

Например, строяться новые населенные пункты или прокладываются новые дороги; в списки добавляются или из списков исключаются некоторые члены. Иногда (и достаточно часто) такие изменения происходят во время работы программы. Такие "нестационарные" структуры данных выделены в отдельный класс.

Определение: Динамическими структурами данных будем называть структуры, которые могут создаваться, меняться и уничтожаться в процессе выполнения программы.

Очевидно, такие структуры наиболее часто встречаются при работе программ в реальном времени.

Из всего многообразия возможных структур данных, возникающих в основном из-за различной сложности связей между объектами, мы рассмотрим более или менее подробно лишь два наиболее часто встречающихся в программировании вида: линейные списки (и их частные случаи — стек и очередь) и деревья (как характерный пример, нелинейных структур).

3.2 Линейные списки

3.2.1 Определения и основные операции

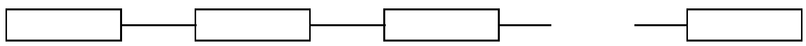
Определение: Линейный список — множество, состоящее из n элементов ($n \geq 0$), со следующими свойствами:

1) существует первый элемент (при $n > 0$);

2) для любых не крайних элементов существует предыдущий и последующий;

3) существует последний элемент (при $n > 0$).

Иными словами, объекты линейного списка связаны лишь с двумя соседними. Данная структура поэтому хорошо представляется спрямленной последовательностью связей (отсюда и ее название):



Примеры групп объектов, подпадающих под данное определение, многочислены и многообразны, в частности: стопка любых предметов (например, подносов), книги, тетради (как список листов) и т.д.

Подробнее рассмотрим очередь к врачу. Очевидно, что здесь имеем множество пациентов, причем множество может быть и пустым. Также ясно, что есть первый и последний элемент, и для любых не крайних существует предыдущий и последующий.

Операции, которые чаще всего производят над линейными списками, являются:

1) просмотр списка и поиск элемента;

2) включение нового элемента;

3) исключение элемента;

4) объединение двух списков в один;

5) копирование списка и другие операции.

В нашем примере очереди к врачу эти операции означают:

1) Кто-то хочет узнать, как далеко он находится от начала и опрашивает людей (с начала или от себя к началу). Фактически просматривая очередь.

2) Пришел еще один больной (и встал в конец очереди или, если "по номеркам", то в середину!).

3) Кто-то отчаялся ждать и ушел домой (или пошел к врачу).

4) Было два врача и две очереди, но один врач ушел ...

5) Оставшийся после окончания приема врача народ скопировал свой порядок в список, чтобы в следующий раз восстановить порядок или медицинская сестра всех переписала перед началом приема врача в порядке очереди.

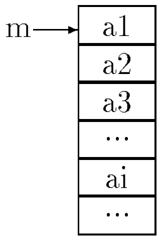
Возратимся к программированию. Отметим, что, по-видимому невозможно найти представление (структурированных) данных, при которых все возможные операции выполнялись бы одинаково эффективно. Однако, к счастью, в редких случаях требуется выполнение всех операций, чаще необходимо обеспечить эффективность работы лишь нескольких операций. В зависимости от этого выбирают как способ представления структуры данных (в частности, линейного списка), так и конкретный подвид структур (например, для линейных списков — стеки, очереди, деки и т.п.).

3.2.2 Способы представления списков

Существует два основных способа представления списков: последовательный и связный. При последовательном — элементы списка располагаются в памяти ЭВМ последовательно друг за другом; при связном — информация группируется в пары (например, записи), где один компонент — очередной элемент списка, а второй — ссылка (адрес) на пару со следующим (или предыдущим) элементом. Т.е. расположение в памяти ЭВМ элементов линейного списка может быть представленно следующим образом:

а) при последовательном представлении (таблично и рисунком):

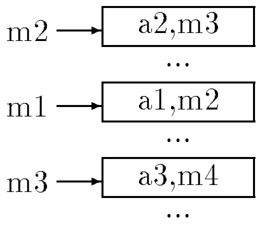
address	info.
m	a1
m+c	a2
m+2c	a3
...	...
m+i*c	ai
...	...



a1, a2, a3, ... — элементы списка;
m — адрес первого эелемента (a1);
c — объем памяти, отводимый для размещения одного элемента.

б) при связном представлении (таблично и рисунком):

address	inform.
m1	a1,m2
m2	a2,m3
m3	a3,m4
...	...
mi-1	ai,mi
...	...



где m1, m2, m3, ... — адреса памяти в ЭВМ соответствующих пар, в которых находится информация об элементе и адресе следующего элемента. При этом адреса m1, m2, m3, ... могут быть не упорядочены.

В подробно рассматриваемом нами примере — очереди к врачу, последовательное представление означает, что люди в очереди сидят на линейно расположенных стульях в правильном порядке (причем подряд!). При связном представлении этого нет, т.е. народ сидит как попало, но у каждого имеется ссылка на предыдущуего (иначе анархия, приводящая обычно к скандалу); эта ссылка в данном случае существует в памяти человека в виде образа впереди стоящего пациента.

Отметим, что в поликлиниках, если очередь длинная, то стихийно (но далеко не случайно!) реализуется связной представление, а, например, в магазине — обычно последовательное. Связано это скорее всего с тем, что перемещаться стоя легко, а сидя сложно. Также и в программировании при некоторых условиях оказывается предпочтительнее один вид представления, а при других — другой.

Упражнение. Привести примеры, когда удобнее пользоваться связным представлением линейного списка, а когда последовательным.

Рассмотрим несколько подробнее последовательный способ представления (линейных) списков. Стандартный способ его реализации — использование массива, длина которого больше предполагаемой длины списка (хорошо задать эту длину иногда бывает трудно!).

Напишем программу, которая будет вводить последовательно объекты (даты[1]) и строить из них линейный список, пока они не кончатся в файле datafile.

```
TYPE Typeofdata = ARRAY [1..3] OF 0..2005;
     Seqreprs = ARRAY [1..m] OF Typeofdata;
VAR  a: Typeofdata;  list: Seqrepres;  n: 1..m;
BEGIN
    n:= 0;
    WHILE  NOT eof(datafile) DO
    BEGIN
       n:= n + 1;  READ (datafile, a);
       list[n]:= a
    END;
END.
```

После этих операций линейный список будет расположен в первых n–элементах массива list.

Рассмотрим как программно можно реализовать перечисленные выше операции.

1. Просмотр — поиск элемента. Пусть нужно узнать, есть ли (и какой от начала) элемент, значение которого равно значению переменной индата (типа Typeofdata, т.е. найти заданную дату). План работы следующий: если элементов в списке нет, то и делать ничего не надо, иначе необходимо просмотреть линейный список с начала, пока не найдем нужный элемент, или пока не дойдем до последнего элемента. Оформим поиск элемента в виде процедуры так:

```
PROCEDURE  Find (n: INTEGER; list: Seqrepres; indata: Typeofdata);
  VAR  yes: BOOLEAN;  k: INTEGER
  BEGIN
    yes:= FALSE;
    IF n=0
      THEN WRITE ('list is empty')
      ELSE
        BEGIN
          k:= 0;
          WHILE  (NOT yes) AND k<=N  DO
            BEGIN
              k:= k + 1;
```

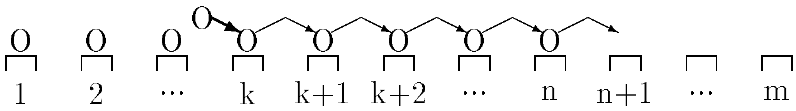
[1]Определение данных типа 'дата' разобрано в предыдущей главе в разделе "Составные типы данных".

```
            yes:= (list [k] = indata)
          END; (* while *)
      END (* else *)
  IF  NOT yes  THEN k:=0;
  WRITE  ( yes, 'k=', k);
END; (* Find *)
```

(list is empty = список пуст)

В результате, если в списке элементов нет, то подпрограмма напечатает "список пуст", в противном случае, если элемент будет найден, то напечатает "TRUE k = "порядковый номер элемента"", если элемент не будет найден, то — "FALSE k = 0".

2. Включение нового элемента. Пусть нужно вставить между k-ым и $(k+1)$-ым элементами еще один ($k < n$), при этом следует проверить не будет ли список переполнен, т.е. выходить за границы массива. В иллюстративном примере очереди алгоритм будет выглядеть следующим образом: проверить, если новое число человек $(n+1)$ станет больше числа стульев, то лучше ничего не делать, иначе попросим людей с конца очереди до $(k+1)$-ого человека пересаживаться на соседний освобождающийся стул (в обратном направлении не выйдет). Затем на освободившийся в середине стул посадим новенького.



Оформим алгоритм в виде процедуры Putin:

```
PROCEDURE  Putin (VAR k,n, m: INTEGER;  list: Seqrepres;
                                        indata: Typeofdata);
  VAR I: INTEGER;
  BEGIN
    IF (n + 1) > m
       THEN WRITE ('list is full')
       ELSE
         BEGIN
           FOR  I:= n  DOWNTO  k+1  DO
             list [I+1]:= list [I];
           list [k+1]:= indata;
           n:= n + 1
         END (* else *)
  END;
```

(list is full = список заполнен)

3. Исключение элемента. Пусть необходимо исключить k-ый с начала очереди элемент ($k \le n$).

В нашей иллюстрации это будет означать, что кто-то ушел и освободился стул. Чтобы его не заняли, "народ" сдвигается к началу на одно место.

Оформим в виде процедуры Exlcude:

```
PROCEDURE  Exlcude (VAR k, n: INTEGER;  list: Seqrepres;
                                        out: Typeofdata);
  VAR i: INTEGER;
  BEGIN
     out:= list [k];
     FOR  i:= k+1 TO  n  DO
       list [i-1]:= list [i];
     n:= n - 1
  END;
```

4. Объединение двух списков. Пусть необходимо объединить два списка list1 и list2 (два массива размерности $m1$ и $m2$). Число элементов в первом списке — $n1$, а во втором — $n2$. Объединение произведем в один массив list1.

Примером слияния очередей к врачу может быть присоединение одной к другой, если хватит стульев, т.е. $n1 + n2$ меньше или равно $m1$. Если стульев не хватит, то тогда ...! В программе же, если этот случай не предусмотреть, то произойдет выход за рамки массива, что приведет к ее аварийному завершению[2].

Оформим данную операцию в виде процедуры Unit:

```
PROCEDURE  Unit (VAR n1, n2, m1: INTEGER;
                     list1, list2: Seqrepres);
  VAR i: INTEGER
  BEGIN
    IF  n1+n2 > m1
      THEN  WRITE ('merging is impossible')
      ELSE
        BEGIN
          FOR  i:= n1+1  TO  n1+n2  DO
            list1 [i]:= list2 [i-n1];
          n1:= n1 + n2
        END (* else *)
  END;
```

(merging is impossible = объединение не возможно)

5. Копирование списков — печать. В примере, если очереди нет, то врач свободен, в противном случае медицинская сестра перепишет людей в порядке очереди.

Оформим это в виде подпрограммы Copylist:

```
PROCEDURE  Copylist (n: INTEGER;  list: Seqrepres);
  VAR i: INTEGER
  BEGIN
    IF  n = 0
      THEN  WRITE ('list is empty')
      ELSE
```

[2] Если программа будет написана на Фортране, то выход за границы массива может привести к потере информации соседних с массивом полей и продолжением работы вашей программы. Аварийный останов программы возможен совсем в другой части программы и поиск этой ошибки довольно сложен.

```
        FOR  i:= 1  TO  n  DO
          WRITE (list [i])
  END;
```

(list is empty = список пуст)

Заметим, что копирование (печать) массива (для последовательного представления линейного списка) возможно лишь указанным образом, т.е. по элементам.

Подведем итоги рассмотрения последовательного представления списков. Основные трудности возникают при неопределенной длине списка, т.к. в этом случае не ясно какой размерности брать массив. Также следует отметить, что основная операция — пересылка. При длинных списках эта операция может потребовать много процессорного времени ЭВМ.

Теперь рассмотрим связный способ представления списков. Как отмечалось в начале п. 3.2.2, нужно создать пары: элемент + ссылка на следующий (или предыдущий). Для этого обычно используют записи и ссылки на них следующим образом:

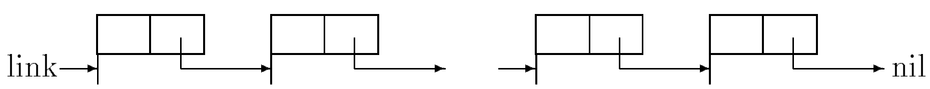
```
TYPE  Typeofdata = ARRAY [1..3] OF 0..2005;
      Relrepes = @Element;
      Element = RECORD  data: Typeofdata;
                        link: Relrepes
                END;
VAR  list, ref: Relrepes;  a: Typeofdata;
```

В переменной list будем хранить ссылку на последний элемент линейного списка (аналог переменной n при последовательном представлении).

Создание списка, аналогичное выполненному выше для последовательного представления, будет иметь вид:

```
BEGIN
   list:= NIL;
   WHILE  eof (file)  DO
    BEGIN
       NEW (ref);  READ (a);
       ref@.data:= a;  ref@.link:= list;
       list:= ref
    END (* while *)
END.
```

Теперь у нас есть линейный связный список, который, чтобы его не потерять, "зацеплен за конец" переменной link.



Рассмотрим программную реализацию перечисленных ранее операций. Как и при последовательном представлении оформим их в виде процедур.

1. Поиск элемента — просмотр. Делается также, как и при последовательном представлении.

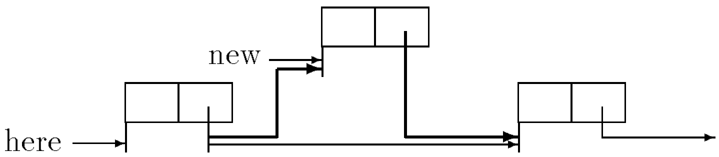
```
PROCEDURE  Findconn (list: Relrepes;  indata: Typeofdata);
  VAR  ref: Relrepes;
  BEGIN
    ref= list;
    IF  ref=NIL
      THEN WRITE ('list is empty')
      ELSE
        WHILE (ref@.data <> indata)  OR  (ref <> NIL)  DO
          ref:= ref@.link;
```

(* ref:= ref@.link — очень важная операция – аналог i:= i+1, означающая переход к следующему элементу *)

```
      IF  ref@.data <> NIL
        THEN  WRITE ('element is found')
        ELSE  WRITE ('no such element in the list')
  END;
```

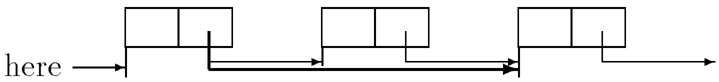
(list is empty = список пуст;
element is found = элемент найден;
no such element in the list = элемента в списке нет)

Вставить новый элемент за элементом, на который указывает ссылка here. Сделаем это так, если использовать наш пример очереди к врачу: спросим у отмеченного, кто находится перед ним и новенькому скажем, что он за этим человеком, а отмеченному, что он за новеньким.



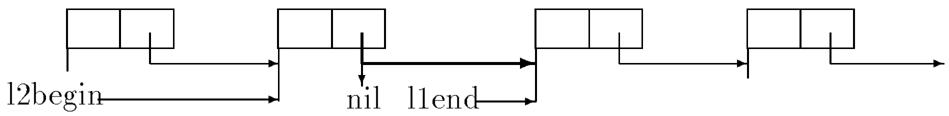

```
PROCEDURE  Putinconn (VAR here: Relrepes;  indata: Typeofdata);
  VAR  new, next: Relrepes;
  BEGIN
    NEW (elnew);  elnew@.data:= indata;
    next:= here@.link;
    elnew@.link:= next;
    here@.link:= elnew
  END;
```

3. Исключение элемента. Исключим элемент, следующий за элементом указанном в ссылке here. Как это сделать, очевидно из следующего рисунка:



```
PROCEDURE  Exclconn (VAR here: Relrepes);
  VAR  next: Relrepes;
  BEGIN
    next:= here@.link;
    here@.link:= next@.link;
    DISPOSE (next)
  END;
```

4. Объединение двух списков. Пусть в переменных l1begin, l1end и l2begin, l2end — ссылки на начало и конец двух списков. Следует выполнить следующее соединение:




```
PROCEDURE  Unitconn (VAR l2begin, l1end: Relrepes);
  BEGIN
    l2begin@.link:= l1end
  END;
```

После выполнения этой процедуры списки будут связаны в один, причем в l1begin будет ссылка на начало, а в l2end — на конец объединенного списка.

5. Копирование списка.

```
PROCEDURE  Copyconn (list: Relrepes);
  VAR  next: Relrepes;
  BEGIN
     next:= list;
      IF  next = NIL
        THEN  WRITE ('list is empty')
        ELSE
          REPEAT
             WRITE (next@.data);  next:= next@.link
          UNTIL  next = NIL
  END;
```

(list is empty = список пуст')

Резюмируем сравнительное рассмотрение реализаций операций при различных представлениях списков:

— просмотр и копирование списков реализуется примерно одинаково;

— объединение списков существенно эффективнее можно сделать при связном их представлении (всего одна операция!)[3];

— включение и исключение элементов в некотором смысле эффективнее при связном представлении (т.к. нет многочисленных пересылок дачнных, но выгода здесь, вероятно, незначительно превосходит отрицательную сторону: непривычность и неочевидность связного представления);

[3] В случае, если объединение происходит по какому-то правилу (например в порядке возрастания элементов), то эффективность примерно одинаковая.

Можно сделать следующие выводы:

1) Если в основном предстоит выполнять операции типа просмотра, или включения и исключения, то возможно использование любого представления.

2) Если операции объединения списков или некоторые модификации включения или исключения играют главную роль, то использование связного представления может способствовать созданию существенно более эффективных программ.

Отметим, что в некоторых языках программирования (Фортран, Алгол-60 и др.) нет прямой возможности работы со ссылками, но связное представление, если немного потрудиться, смоделировать можно.

3.2.3 Частные случаи линейных списков: стек, очередь, дек

В программировании наиболее часто используются несколько частных случаев линейных списков, когда дополнительно к свойствам определения на структуры данных накладываются новые ограничения.

Определение: Стеком будем называть линейный список, в котором все включения и исключения элементов разрешено производить только на одном конце (считается, что включение и исключения самые частые (основные) операции!).

Примеры — магазин автомата с патронами, железнодорожный состав вагонов в тупике и т.д.

Определение: Очередь — линейный список, в котором все включения производятся на одном конце, а исключения — на другом.

Примером является любая очередь, если не разрешено новые элементы вставлять и исключать из середины.

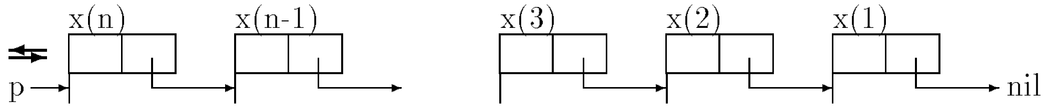
Определение: Дек — линейный список, где исключения и включения производятся с обоих концов.

Пример простой и не искусственный привести трудно, некоторую аналогию имеет вход и выход из вагона электрички.

Отметим, что над стеком, очередью и деком могут производиться разнообразные операции, включая перечисленные выше для линейных списков. Однако, из данных определений ясно, что данные частные случаи линейных списков ориентированны на ситуации, когда в основном производятся операции включения и исключения элементов. Специфичность этих операций (место выполнения) делает более удобным связное представление (см. выводы в п. 3.2.2), хотя последовательное представление не исключено.

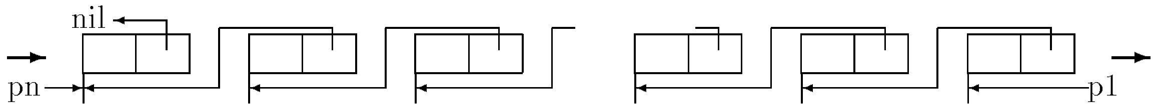
Мы рекомендуем использовать следующий порядок связи элементов.

Для стека:



где p — переменная ссылочного типа, в которой хранится адрес верхнего элемента стека, x — данные в списке. Включение и исключение элементов производится со стороны верхнего элемента, при помощи операторов:
(включение) NEW(s); s@.link:= p; p:= s
(исключение) s:= p; p:= p@.link; DISPOSE (s)

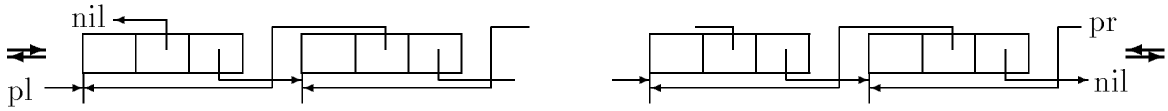
Для очереди[4]:



где в переменных ссылочного типа p1 и pn будем хранить адреса соответственно первого и последнего элемента очереди.

Включение (со стороны последнего элемента) и исключение (со стороны первого) производится следующим образом:
(включение) NEW (s); s@.link:= pn; pn:= s
(исключение) s:= p1; p1:= p1@.link; DISPOSE (s)

Для дека:



где pl и pr — ссылки на левый и правый конец дека, и т.к. исключение и включение элементов возможно с обоих концов, то приходится некоторым образом совместить стек и очередь, создавая так называемый 'двухсвязный' линейный список.

Теперь о том, где и как используются рассмотренные выше частные случаи линейных списков. Стеки и очереди применяются в тех случаях, когда происходит выборка объектов из некоторого их множества для дальнейшей их обработки. Например, из огромных звездных каталогов обычно выбирается некоторая группа звезд по каким-то признакам (например, один спектральный класс, расположение в заданной площадке неба и т.п.) для дальнейшей их статистической или другой обработки. При этом часто неважно, что использовать стек или очередь (хотя и принято строить стек).

Иная ситуация, когда программа работает в режиме реального времени. Часто здесь возникает необходимость образовывать из объектов линейный список, причем так, что прибывший первым элемент уходил бы (обрабатывался) тоже первым. Этот принцип реализуется в очереди (в стеке правило — первым пришел и последним ушел).

Т.о. при работе с данными большого объема, а также в трансляторах, текстовых редакторах и во многих других программных средствах используются стеки.

[4] Отметим обратное направление связи элементов в очереди по сравнению со стеком, что определяется изменением направления "стока" элементов.

Для функционирующей в реальном времени системной программы, управляющей процессом обработки программ пользователей ЭВМ (и называемой операционной системой), создаются очереди: очередь заданий для выполнения их центральным процессором ЭВМ, выходные очереди данных и т.п.

Дек используется значительно реже, чем очередь и стеки.

3.3 Рекурсивный способ работы со списками

При работе со списками (с линейными в меньшей степени, с нелинейными — в большей) оказывается эффективным использование рекурсии. Поясним, что это означает. Отметим сначала, что в математике довольно часто встречаются рекуррентные соотношения последовательностей и функций. Например, функция факториал числа может быть определена так:

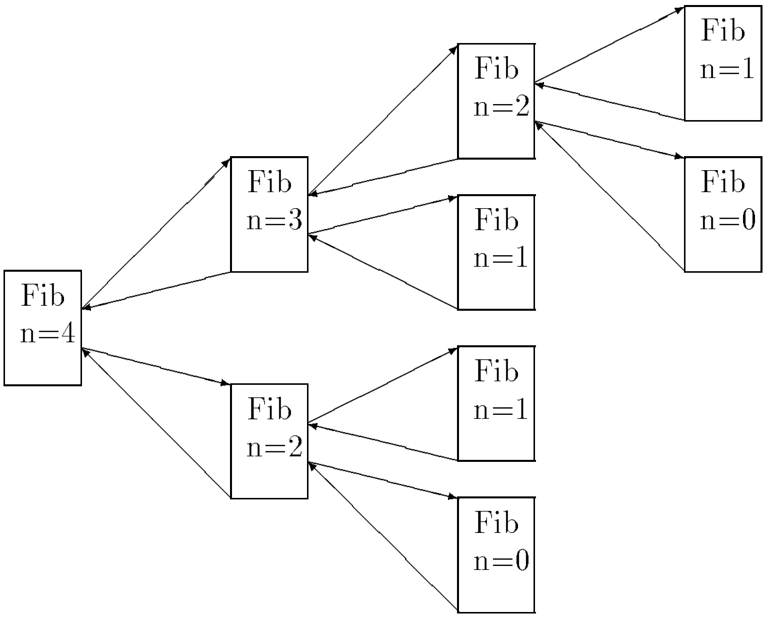
$$n! = \begin{cases} 1 & \text{при} \quad n = 0 \\ n(n-1)! & \text{при} \quad n > 0 \end{cases} \tag{3.1}$$

В некоторых языках программирования (в Паскале — да, в Фортране — нет!) допускается аналогичное действие, называемое рекурсией и заключающееся в обращении подпрограммы самой к себе. Таким образом, подпрограмма — функция вычисления n-ого числа Фибоначчи на языке Паскаль может иметь следующий вид:

```
FUNCTION  Fib (n: INTEGER) : INTEGER;
  BEGIN
   IF  n=0
     THEN  Fib:= 0
     ELSE  IF  n=1
             THEN  Fib:= 1
             ELSE  Fib:= Fib(n-1) + Fib(n-2)
  END;
```

Отметим последний оператор присваивания, где слева стоит имя функции, вычисляемой с использованием этой же процедуры-функции(!).

Рассмотрим, как будет работать данная подпрограмма при таком обращении к ней: m:= Fib (4). Начиная, входим в подрограмму Fib со значением формального параметра n равным 4. Встрчаем обращение Fib (n-1), которое снова вызывает нашу подпрограмму. При этом ЭВМ запомнит адрес точки возрата в подрограмму Fib, значения локальных для Fib переменных и еще кое-что и переходит снова на начало подпрограммы Fib, но уже для нового значения n $= 4 - 1 = 3$ и



и т.д. до n=1. К этому моменту в памяти машине будет записано три точки возврата, три варианта значений локальных переменных, при которых произошло следующее обращение к Fib, и еще кое-что тоже три раза (отметим, кстати, что операционная система при орбаботке рекурсивной программы будет использовать стеки!).

При n=1 подпрограмма-функция вычислит свое значение Fib без нового обращения к самой себе (т.к. Fib(1) = 1) возращается в третью точку возрата (n=2), после чего, продолжив оператор сложения (Fib(n-1) + Fib(n-2)), произойдет снова обращение к Fib (на третьем уровне вложения) с n=0. Выйдя из этого обращения (т.к. Fib(0) = 1), мы произведем сложение Fib(1) + Fib(0) и получим значение функции Fib при n=2 и выйдем из Fib для n=2 и т.д., как это показано на рисунке.

Из рассмотренного примера становятся ясными преимущества и недостатки рекурсивного способа программирования. Положительный момент состоит в том, что программа адекватно отражает алгоритм решения задачи при существовании рекуррентных формул. И поэтому она проста и, как следствие, менее подвержена возможному привнесению программистом ошибок. Отрицательный момент в том, что при большом числе обращений подпрограммы самой к себе (большой так называемая 'глубине рекурсии') может происходить неоправданное и очень серьезное (экспоненциальное!) увеличение затрат памяти ЭВМ и времени счета.

Альтернативой рекурсии является итерация. Понятие итерации встречается достаточно часто и определять его, по-видимому, нет необходимости. Поясним лишь на примере вычисления чисел Фибоначчи:

```
FUNCTION  Fib2 (m: INTEGER) : INTEGER;
  VAR  i, i1, i2, io: INTEGER;
  BEGIN
    IF  m=0
      THEN  Fib:= 0
      ELSE
```

```
                  IF  m=1
                    THEN  Fib2:= 1
                    ELSE
                      BEGIN
                        i:= 1;  i1:= 1;  i2:= 0;
                        WHILE  i<m  DO
                          BEGIN
                            io:= i1 + i2;
                            Fib2:= io;
                            i2:= i1;  i1:= io;
                            i:= i + 1
                          END  (* while *)
                      END  (* else *)
            END;
```

Здесь при $m \geq 2$ в цикле WHILE ровно $m-2$ раза будет применятся рекуррентная формула. В отличие от рекурсивной подпрограммы, вычисления в Fib2 будут начинаться с вычисления первых членов последовательности чисел Фибоначчи, т.е. для $i = 1, 2, 3$ и т.д. до m. При этом в переменных io, i1 и i2 будут запоминаться значения чисел Fib(i), Fib(i-1) и Fib(i-2).

Отметим, что итерационная версия (Fib2) несколько сложнее и существенно менее наглядна, чем рекурсивная Fib.

Однако ответ на общий вопрос — какой из методов вычислений: рекурсию или итерацию использовать в том или ином случае? — оказывается несколько неожиданным: рекурсии следует избегать в тех случаях, когда есть очевидное итеративное решение задачи.

Этот неблагоприятный для рекурсии вывод, однако смягчим замечанием о том, что достаточно часто дать итеративное решение очень не просто. Примерами таких случаев является работа с нелинейными (в частности ветвящимися) структурами (см. п. 3.4) или задачи типа рассмотренной ниже "задачи о ханойской башне".

Для линейных списков использование рекурсии не так эффективно. Рассмотрим, например, операцию поиска элемента в линейном списке. Итеративное решение дано выше. Рекурсивное для последовательного и связного представления выглядит следующим образом[5]:

последовательное   связное

```
-----------------------------I-------------------------------
                             I
  PROCEDURE                  I   PROCEDURE
    Search (n: INTEGER;      I     Search1 (x: Typeofdata;
            list: Seqrepres  I              ref: Relrepres;
            x: Typeofdata);  I          VAR here: Relrepres);
    VAR  k: INTEGER;         I
  BEGIN                      I   BEGIN
     IF list [k] = x         I     IF ref@.Data = x
       THEN  WRITE (k)       I       THEN s:= ref
```

[5]Имена переменных использованы те же что и ранее.

```
      ELSE                   I       ELSE
        IF  k <= N           I         IF  ref@.link <> NIL
          THEN  Search (N,   I           THEN  Search1 (x,
             list, x, k-1)   I               ref@.link, here)
          ELSE WRITE ('no')  I           ELSE WRITE ('no')
  END;                       I     END;
                             I
```

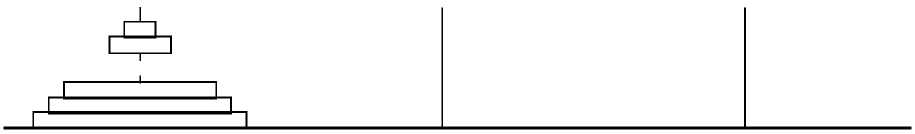
Обращение :

```
k:=1; Search (n, list, x, k);I     Search1 (ref, x, here);
-----------------------------I--------------------------------
```

Поскольку итеративное решение задачи достаточно просто (причем даже проще рекурсивного!), то согласно нашей рекомендации предпочтительнее для аналогичных задач использовать итеративные алгоритмы. Отметим, однако, что и для линейных списков в некоторых случаях может быть более удобным применение рекурсии.

Теперь рассмотрим случай, где без рекурсии обойтись чрезвычайно затруднительно — задачу о "ханойской башне". Она формулируется так:

Согласно древней легенде, в некотором храме находится бронзовая плита с тремя стержнями. На один из стержней Бог при сотворении мира нанизал 64 диска разного диаметра из чистого золота так, что наибольший диск находится на бронзовой плите, а остальные образуют пирамиду, сужающуюся к верху.



Работая день и ночь жрецы переносят диски с одного стержня на другой по следующим правилам: диски можно перемещать со стержня на стержень только по одному и нельзя класть больший диск на меньший. Известно, что когда жрецы перенесут все 64 диска с одного стержня на другой, используя третий стержень, наступит конец света.

Вопрос: как (в какой последовательности) жрецы должны переносить диски (любопытно, также, оценить, когда можно ждать окончания их работы!).

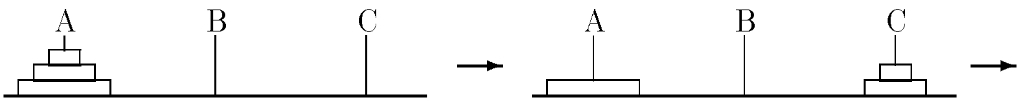
Решение задачи итерациями отыскать совсем непросто, в то время как рекурсия дает чрезвычайно короткое (и элегантное!) решение. Оно фактически состоит в необходимости переноса пирамиды из $m-1$ диска (кстати, для $m=2$ перенос пирамиды из $m-1$ диска есть просто перенос диска!).

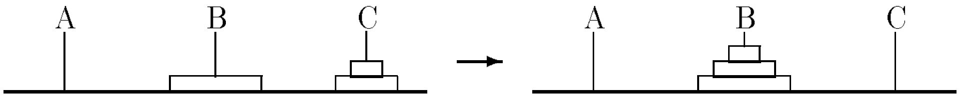
Решение этой подзадачи таково:

1) перенести пирамиду из $m-1$ диска с иглы A на иглу C, используя иглу B.

2) перенести оставшийся на игле A нижний диск на иглу B.

3) перенести пирамиду из $m-1$ диска с иглы C на иглу B.





Нетрудно написать рекурсивную подпрограмму, реализующую этот алгоритм[6]:

```
PROGRAM  Han;
  VAR  f: TEXT;  i: INTEGER;
```

(* процедура переноса башни с n-кольцами с c1 на c2, пользуясь c3 *)

```
    PROCEDURE  Hanoi (n: INTEGER;  c1, c2, c3: CHAR);
      BEGIN
        IF  n <> 0  THEN
          BEGIN
```

(* переносим башню из $n-1$ кольца с c1 на c3, пользуясь c2 *)

```
            Hanoi (n-1, c1, c3, c2);
```

(* переносим n-ое кольцо на c2, т.е. на заданное место *)

```
            WRITE (f, 'ring n=', n:2, ' from ', c1, ' to ', c2);
```

(* переносим башню из $n-1$ кольца с c3 на c2, пользуясь c1 *)

```
            Hanoi (n-1, c3, c2, c1)
          END (* if *)
       END; (* Hanoi *)

     BEGIN
       REWRITE (f, 'Lr:');
       i:= 2;
       Hanoi (i, 'A', 'B', 'C')
     END.
```

Работа данной программы приведет к тому, что будет напечатано очевидное решение: кольцо n1 1 перенести с иглы A на иглу C кольцо n1 2 перенести с иглы A на иглу B кольцо n1 1 перенести с иглы C на иглу B

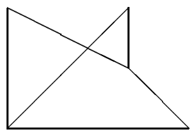
Напомним, что кольца мы пронумеровали сверху вниз, т.е. номер 1 имеет наименьшее кольцо.

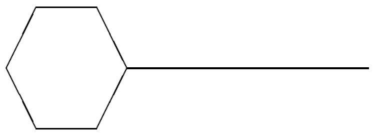
Стоит отметить, что при $n = 2$ потребовалось 3 перемещения, при $n = 3$ потребуется уже 7 и т.д. Т.о., число перемещений равно $2^n - 1$. Следовательно, при $n = 64$ число перемещений будет равно порядка 2^{64} или 10^{19} и если одно кольцо переносить за 1 секунду, то работа потребует около $3\,10^{11}$ лет, т.е. время, большее возраста Галактики и сравнимое с космологическим. Поэтому пока можно жить спокойно, у жрецов впереди еще много работы.

[6] Данная программа проверена на СМ-4.

3.4 Нелинейные структуры данных: деревья и графы

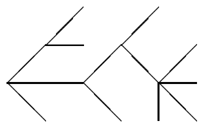
Не вызывает сомнения тот факт, что возможные структуры данных не ограничиваются линейными, рассмотренными выше. Реальные объекты могут потребовать создание достаточно сложных структур данных, например, при представлении транспортных схем или сложных молекул. В этих случаях могут иметь место и такие нетривиальные структуры:





Определение: Структуры данных, элементы которых связаны произвольным образом, будем называть графами.

Линейные списки — частный случай графа. Ниже мы рассмотрим и другой частный случай графов — деревья, являющиеся нелинейными структурами. Несмотря на то, что принцип связи между элементами структуры данных типа дерева легко понять из следующего рисунка,



мы дадим строгое, формальное определение.

Определение: Деревом будем называть структуру данных, объединяющую конечное множество элементов T, где T не пусто, так что:

1) имеется выделенный элемент, называемый корнем;

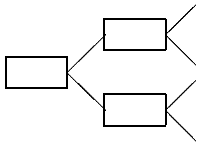
2) остальные элементы содержатся в $m \geq 0$ попарно не пересекающихся множествах T1, T2, T3, ..., Tm, каждое из которых является деревом (точнее поддеревом).

Конечные элементы ветвей деревьев будем называть листьями, а остальные — узлами.

Отметим важную деталь определения — использование рекурсии (дерево определяется через понятие дерева, причем достаточно элегантно!). Поэтому неудивительно, что многие операции над деревьями эффективнее реализуются при использовании рекурсивных алгоритмов.

Примеры деревьев:

— генеалогические деревья, строящиеся от субъекта (корень) с указанием двоих родителей, родителей родителей и т.д.;



— варианты возможных ходов в шахматах и других играх;

— схема вычисления чисел Фибоначчи с использованием рекурсивных подпрограмм.

В первом и третьем примере используется дерево, имеющее специальное название, а именно — бинарное, т.е. такое дерево, у которого из каждого узла выходит не более двух ветвей. Ниже мы в основном будем рассматривать именно этот вид деревьев.

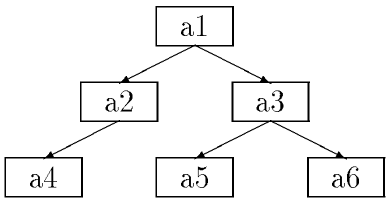
Как и линейные списки деревья могут быть реализованы в программировании путем использования как последовательного, так и связного представления. Отметим, что последовательное представление будет использовать более одного массива. Так например бинарное дерево можно реализовать при помощи трех массивов:

1) массив, каждый элемент которого содержит информацию об одном узле дереве;

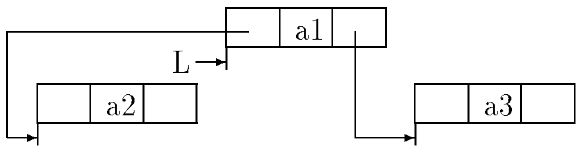
2) массив, k-ый элемент которого содержит информцию (индексы первого массива) о нахождение левого узла k-ого элемента первого массива;

3) массив, k-ый элемент которого содержит аналогичную информацию, что и второй, но о правом узле k-ого элемента.

knot	left	right
a1	2	3
a2	4	0
a3	5	6
a4	0	0
a5	0	0
a6	0	0



Связное представление позволяет существенно более адекватно представить дерево и в дальнейшем мы будем использовать лишь этот вид представления. Бинарное дерево можно представить например так:



где каждый узел содержит следующую информацию: об элементах дерева (a1, a2, ...) и ссылки на соответствующие узлы левого поддерева и правого поддерева. L — ссылка на корень двоичного дерева.

Для описания информации, по-видимому, удобно воспользоваться комбинированным типом (записями), где два поля ссылочного типа (на эти записи), а одно или несколько полей отводятся для информации об элементах дерева. Тип этих полей зависит от данных.

Операции над деревьями выполняются те же, что и над линейными списками, но наиболее важной из них на этот раз становиться частный случай просмотра — обход дерева[7].

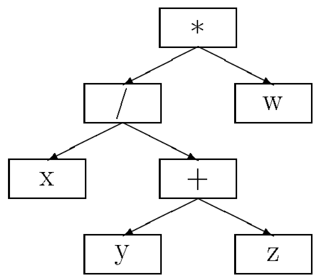
Определение: Обход дерева — такой его просмотр, когда каждый узел дерева анализируется один (и только один) раз.

Основную часть алгоритма обхода дерева удобно сформулировать в рекурсивном виде:

обойти дерево =
обойти левое поддерево
проанализировать корень
обойти правое поддерево

Это так называемый "обратный" порядок обхода, но могут быть и другие.

Покажем работу алгоритма на примере следующего дерева, представляющего некоторое алгебраическое выражение:



Если под анализом узла понимать печать узла, то применение алгоритма обхода даст выражение: $(X/(Y+Z))*W$, где скобки означают уровень глубины рекурсии.

Соответствующая программа на языке паскаль будет иметь вид:

(* процедура печати двоичного дерева *)

```
PROCEDURE  Pr (p: ref);
  BEGIN
    IF  p <> NIL  THEN
      BEGIN
        Pr (p@.left);
        WRITE (f, p@.key: 4);
        Pr (p@.right)
      END (* if *)
  END;
```

Порядок работы этой процедуры аналогичен подробно рассмотренному нами для рекурсивной процедуры Fib, вычисляющей значения чисел Фибоначчи, и Hanoi, решающей задачу о ханойской башне.

О других операциях заметим следующее:

1) просмотр — поиск элемента аналогичен обходу;

[7]В дальнейшем, все операции будут произволится с бинарными деревьями и в целях краткости термин бинарные будет опущен.

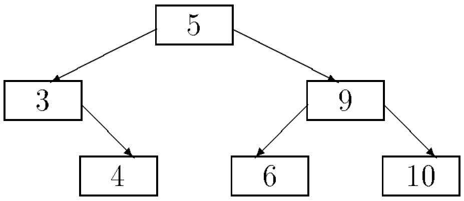
2) включение нового элемента — замену листа поддерева сделать просто, а добавление нового узла — несколько сложнее;

3) исключение элемента–листа — просто, а узла из середины — сложно (причем нужно указать специальные правила как это делать!);

4) объединение деревьев в одно, если включение корня одного из деревьев в качестве листа другого возможно, сделать легко;

5) копирование дерева — эквивалентно обходу.

Процесс включение новых листьев проиллюстрируем на примере следующей программы построения дерева. Пусть имеется некоторый стек с целыми числами (грубо говоря файл). Напишем программу, строящую такое дерево, что в левых поддеревьях будут числа меньшие, чем в корне, а в правом — большие[8]. Например для последовательности 5, 3, 9, 6, 4, 10 должно быть построено следующее дерево:



Соответствующая программа может быть следующей:
(* программа создания двоичного дерева типа поиск *)

```
PROGRAM  Buildtree;
  TYPE  Rootref = @Ntree;
  Typeofdata = ARRAY [1..3] OF 0..4005;
  Ntree = RECORD  key: INTEGER;
                  data: Typeofdata;
                  right, left: Rootref
          END;
  VAR  newkey: INTEGER;  newdata: Typeofdata;
       newroot: Rootref;  f: Text;
```

(* процедура построения дерева *)

```
  PROCEDURE  Build (inkey: INTEGER;  indata: Typeofdata;
                  VAR root: Rootref);
    BEGIN
      IF root = NIL
        THEN
          BEGIN
            NEW (root);
            root@.left:= NIL;  root@.right:= NIL;
            root@.key:= inkey;  root@.data:= indata
          END
```

[8] Подобное дерево имеет специальное название — дерево поиска.

```
        ELSE
         IF  inkey < root.key
           THEN  Build (inkey, indata, root@.left)
           ELSE  Build (inkey, indata, root@.right)
    END; (* Build *)

  BEGIN
    newroot:=NIL; WRITELN ('Input data')
    WHILE  eof (f)  DO
      BEGIN
        READ (newkey, newdata);
        Build (newkey, newdata, newroot)
      END; (* while *)
   (*  Printtree (newroot)  *)
   END.
```

(input data = вводите данные)

Напомним, что в результате работы программы Buildtree мы получим дерево, в котором в левых поддеревьях будут узлы (записи) со значениями в поле key меньшими, чем в поле key подкорня, а в правых — больше. Отметим, что если теперь включить в эту программу подпрограмму Printtree и обратиться к ней, как показано в комментарии, то будут напечатаны введенные числа, но уже упорядоченные в порядке возрастания (т.е. 3, 4, 5, 6, 9, 10). Методы сортировки, основанные на подобном алгоритме, оказываются достаточно эффективными, хотя существует и более быстрый способ сортировки.

Дерево поиска является незаменимым средством в задачах, где кроме сортировки, требуется еще и поиск соответствующего элемента. Продолжительность поиска будет оптимальной (минимальной), если правильно выбрать корень дерева (очевидно, что дерево поиска должно иметь ветви по возможности примерно одинаковой длины). Задачи такого сорта часто встречаются при создании трансляторов и банков данных.

Структуры и типы данных в работе

4.1 Игровые задачи

Игровые задачи являются типичными представителями задач нечислового программирования. Отметим, что обычно программы, моделирующие игры, являются диалоговыми системами. Причем в отличие от других диалоговых программ, игровые имеют часто простую формулировку условий при широком диапазоне сложности.

Например, при создании программы игры в шахматы возникает вопрос о создании искусственного интелекта, а игра в "теннис" требует усилий лишь при программировании графики.

Это, а также занимательность игровых программ делает их удобной иллюстрацией при изучении задач нечислового программирования.

В качестве примера игровой задачи мы рассмотрим простую программу, моделирующую игру в "орлянку".

4.2 Моделирование игры в "орлянку"

4.2.1 Условие задачи

В игре участвует банкир и неопределенное число игроков, которые делают ставки на "орел" ("о") или "решку" ("р"). Банкир бросает монету. В случае, если игрок угадал, на какую сторону упадет монета, ему выплачивается сумма равная его удвоенной ставке. В противном случае игрок теряет ставку.

4.2.2 Постановка задачи для программиста

Программист переформулирует задачу примерно следующим образом:

1) задать некоторое число (сумму денег в банке);

2) ввести: имя игрока, размер ставки и признак, на что сделана ставка;

3) повторить ввод, если играют несколько игроков;

4) получить одно случайное число (т.е. смоделировать бросок монеты);

5) определить число выигравших игроков и произвести соответствующий расчет;

6) если нужно, то вернуться ко второму пункту (произвести новый ввод), в противном случае - конец работы.

4.2.3 Реализация алгоритма

Рассмотрим один из возможных способов реализации данного алгоритма.

1) Задать сумму денег в банке можно при помощи константы:

CONST bank = 1000

2) Для хранения вводимой информации можно использовать линеный список, представленный связанным образом. Каждый элемент которого — запись с четырьмя полями: имя игрока, ставка, указание на что она сделана и ссылка на следующий элемент линейного списка.

Для организации такого линейного списка на Паскале можно определить следующие типы данных (предполагаем, что имя игрока не может быть более 20 символов):

```
TYPE  References = @Records;
      Records = Record name: Arr;
                       stake: REAL;
                       what: CHAR
                       next: References
                END;
      Arr = ARRAY [1..20] OF CHAR
```

3) ввод информации будем производить следующим образом:

```
VAR  ref, last: References;
     yes, answer: CHAR;
BEGIN
  last:=NIL;
  REPEAT
    NEW (ref);
    WRITELN ('Give your name, count, and point');
    READLN (ref@.name, ref@.stake, ref@.what);
    ref@.next:= last;  last:= ref;
    WRITELN ('more players? (Y/N)');
    READLN (answer);
  UNTIL  answer = 'Y'
```

(Give your name, count, and point = сообщите имя, ставку и на что ставите; more players? = еще игроки?).

4) Моделирование броска монеты может быть произведено вызовом функции получения случайного числа в диапозоне [0–1]. В случае, если случайное число <0.5, то считаем, что выпал "орел". В противном случае — "решка".

5) Подсчет числа выигравших и расчет со всеми игроками производиться при помощи просмотра созданого в п. 3 связного линейного списка (операция просмотра линейного списка подробно была рассмотрена в предыдущей главе).

6) Для продолжения игры необходимо организовать внешний цикл при помощи операторов REPEAT .... UNTIL <условие>, или WHILE <условие> DO ... Условие окончания цикла должно быть организовано, аналогично условию окончания в п. 3.

Подчеркнем, что использование связного линейного списка для хранения ставок в данной задаче несколько удобнее использования массива, т.к. наперед неизвестно число играющих.

В случае же, например, игры в рулетку, где число возможных ставок одним игроком велико и число играющих обычно тоже не мало, преимущества использования связного представления становятся ощутимыми.

Отметим, что данные случаи подпадают под отмеченные в резюме к п. 3.2.3 и мы очень рекомендуем снова прочитать данное резюме.

Заключение

Поздравляем читателей, дочитавших до этого пункта. Вы теперь можете относительно квалифицированно подойти к решению даже таких задач как моделирование искусственного интелекта, и мы желаем вам всяческих удач.

Что касается не дочитавших до конца, то можно начать читать снова, а можно и бросить. Но в любом случае мы думаем, что у вас осталось какое-то представление о том, из чего состоят и что лежит в основе задач "нечислового программирования", перечисленных во Введении.

Литература

1. К.Йенсен, Н.Вирт "Паскаль." М.: Финансы и статистика. 1982
(второе русское издание по третьему английскому К.Йенсен, Н.Вирт "Паскаль. Руководство для пользователя." М.: Финансы и статистика, 1989 — на наш взгляд несколько хуже первого).

Стандартное и наиболее удачное описание языка паскаль, которое мы брали за основу при написании Главы 2 может быть заменено любым из имеющихся в настоящий момент большого числа учебников по паскалю. Среди них может быть рекомендована (небольшая по обьему, но изданная относительно большим тиражем (300 000 экз.)) брошюра: Абрамов С.А., Зима Е.В. Начала программирования на языке паскаль" - М.:, Наука, 1987.

2. Н.Вирт "Алгоритмы Ъ̆ структура данных ı программы." М.: Мир., 1985 (второе русское издание: Н.Вирт "Алгоритмы и структуры данных." М.: Мир, 1989 написано для "паскалеобразного" языка Модула-2).

Содержит несколько подробных иллюстраций применения различных структур данных при программировании, включающих задачи сортировки и оканчивающихся подробным обсуждением создания транслятора для языка PL/0. Рекомендуется для дальнейшего чтения.

3. В.Б.Титов нечисловое программирование в задачах небесной механики. Труды АО ЛГУ, Л., 1987.

Дано краткое описание большого числа структур данных и конкретные примеры их использования в небесной механике. Идеология именно этой статьи лежит в основе Главы 3.

4. Р.Форсайт "Паскаль для всех". М.: Машиностроение, 1986.

Содержит описание языка Паскаль и подробно разобрано несколько задач нечисловго программирования (среди них имеются игровые задачи). Рекомендуется для дальнейшего чтения.