ФЕДЕРАЛЬНОЕ АГЕНСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА» (СГАУ)

Е.В. Симонова

СТРУКТУРЫ ДАННЫХ. ЧАСТЬ I. ЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

Утверждено редакционно-издательским советом университета в качестве учебного пособия

САМАРА
Издательство СГАУ
2006

ОГЛАВЛЕНИЕ

Аннотация

Учебное пособие предназначено для студентов, обучающихся по направлениям 010500 - “Прикладная математика и информатика”, 010400 - “Информационные технологии”. Рекомендуется использовать учебное пособие при изучении курсов “Языки программирования и методы трансляции”, “Практикум на ЭВМ (языки программирования)” для направления 010500 и “Основы программирования”, “Языки программирования”, “Практикум на ЭВМ” для направления 010400. Включает разделы, которые подробно описывают абстрагирование типов, идентификацию объектов, классы памяти, динамические структуры данных (односвязные, двусвязные списки, мультисписки). Теоретический материал иллюстрируется большим количеством программных фрагментов, реализующих алгоритмы обработки различных структур данных. Учебное пособие содержит контрольные вопросы и упражнения по всем разделам.

Данное учебное пособие будет также полезно и студентам других специальностей, изучающих курсы программирования и информационных технологий, и обучающихся как по очной, так и заочной форме обучения. Учебное пособие разработано на кафедре информационных систем и технологий.

ПРЕДИСЛОВИЕ

В учебном пособии описаны структуры данных и алгоритмы, которые широко используются при решении разнообразных задач в широком спектре предметных областей.

Учебное пособие предназначено для студентов, обучающихся по направлениям 010500 - “Прикладная математика и информатика”, 010400 – “Информационные технологии”, 010200 - “Автоматизированные системы обработки информации и управления”. Рекомендуется использовать учебное пособие при изучении курсов “Языки программирования и методы трансляции”, “Практикум на ЭВМ (языки программирования)” для направления 010500; “Основы программирования”, “Языки программирования”, “Практикум на ЭВМ” для направления 010400; “Программирование на языке высокого уровня” и “Информационные технологии” для направления 010200.

Содержание учебного пособия соответствует разделам рабочих программ по дисциплинам “Языки программирования и методы трансляции», “Практикум на ЭВМ (языки программирования)», «Основы программирования», «Языки программирования», «Практикум на ЭВМ», “Программирование на языке высокого уровня”, “Информационные технологии” федерального компонента ГОС подготовки бакалавров по направлениям 010500 – Прикладная математика и информатика и 010400 – Информационные технологии.

В главе 1 представлены сведения о фундаментальных и абстрактных типах данных, о внутреннем представлении данных различных типов в оперативной памяти.

Глава 2 посвящена рассмотрению двух видов идентификации объектов в программе – именования и указания, рассмотрению понятия адреса объекта и указателя как универсального идентификатора объектов. Приводятся сведения об организации адресного пространства оперативной памяти.

В главе 3 описывается распределение рабочего пространства оперативной памяти компьютера во время исполнения программы, функциональность различных классов оперативной памяти.

В главе 4 подробно рассматриваются особенности организации линейных динамических структур данных (односвязных, двусвязных списков, мультисписков). Особое внимание уделено вопросам управления динамической памятью и эффективного использования ее возможостей при работе с динамическими структурами данных в программах пользователей.

Программы, реализующие алгоритмы обработки структур данных различных типов, соответствуют современному уровню информационных технологий. Разделы, рассмотренные в пособии, имеют большое учебно-методическое значение и необходимы при самостоятельной работе студентов во время выполнения ими домашних заданий, лабораторного практикума и курсовой работы.

В конце каждой главы приведены упражнения и контрольные вопросы, с помощью которых можно проверить усвоение изложенного матриала.

В учебном пособии содержатся сведения, недостаточно освещенные в учебной литературе, а также заложены новые методики преподавания, активизирующие самостоятельную работу студентов.

Учебное пособие может быть полезно широкому кругу читателей, практикующихся в области компьютерных наук.

ВВЕДЕНИЕ

Виды структур данных

Данные, относящиеся к какой-либо проблеме, являются абстрактным, т.е. упрощенным представлением объектов реального мира. Алгоритмы и строение данных неразрывно связаны между собой: представление данных невозможно выбрать, не зная, какие алгоритмы к ним будут применяться, и, наоборот, выбор алгоритма часто очень сильно зависит от строения данных. По определению Н. Вирта, программы представляют собой, в конечном счете, конкретные формулировки абстрактных алгоритмов, основанные на конкретных представлениях и структурах данных.

Понятие структуры всегда соответствует сложному объекту, обладающему свойством целостности, и вместе с тем сконструированному из простых компонентов путем использования определенной системы правил. Можно выделить следующие основные виды структур данных:

- простые (т.е. встроенные в язык) структуры – это основные конструкции, из которых строятся более сложные структуры-агрегаты. Простые структуры и агрегаты образуют ***фундаментальные структуры***. Данные, представленные в виде фундаментальных структур, во время выполнения программы могут изменять значение, но изменить их строение нельзя;
- составные (***динамические***) ***структуры*** в процессе выполнения программы могут изменять как значение, так и строение;
- ***иерархические структуры*** - это динамические структуры, которые содержат данные, распределенные по уровням;
- ***объектно-ориентированные*** структуры воплощают концепцию совместной обработки данных и алгоритмов.

Развитие концепции структуризации в программировании

Развитие концепции структуризации прежде всего нашло отражение в топологии языков программирования. ***Топология*** – это основные элементы языка программирования и их взаимодействие.

1 ПОКОЛЕНИЕ (FORTRAN –1, ALGOL-58)

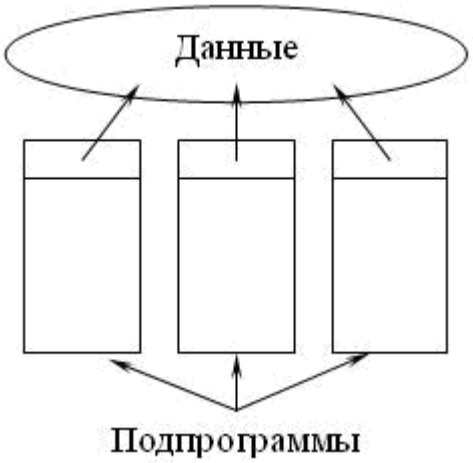


Рис. 1. Топология языков программирования первого поколения

Программы имеют простую структуру, состоящую из области глобальных данных и подпрограмм (рис.1). Ошибка в какой-либо подпрограмме может повлиять на исполнение других подпрограмм, т.к. область данных является общей. Большое количество перекрестных связей между подпрограммами, запутанные схемы управления, неясный смысл данных снижают понимаемость и надежность программ. В языках

появились простейшие управляющие структуры – операторы условного и безусловного перехода, циклы с различными условиями повторения и выхода и т.п.

2 ПОКОЛЕНИЕ (FORTRAN-II, ALGOL-68, LISP, PL/1)

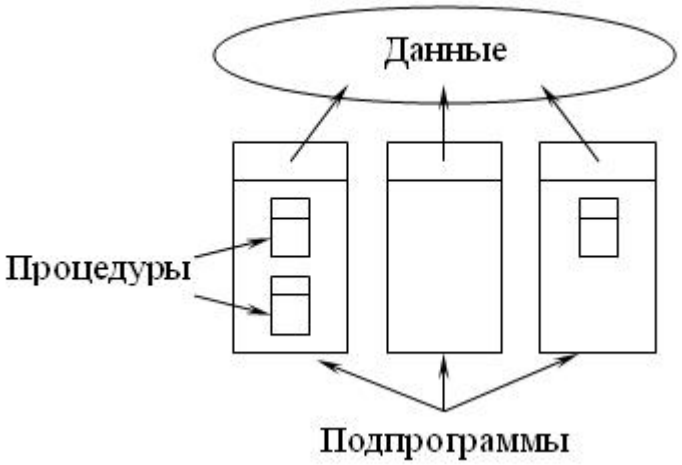


Рис. 2. Топология языков программирования второго поколения

В языках реализованы разнообразные механизмы передачи параметров, а также механизмы управления вложенностью подпрограмм и областями видимости, что послужило основой структурного программирования (рис.2). Возникли методы структурного проектирования, позволяющие создавать большие программные системы, используя подпрограммы как готовые строительные блоки. Были заложены основы структуризации данных, т.е. появились типы данных, конструируемые пользователем, например, записи.

3 ПОКОЛЕНИЕ (PASCAL, MODULA, SIMULA, C, PROLOG)

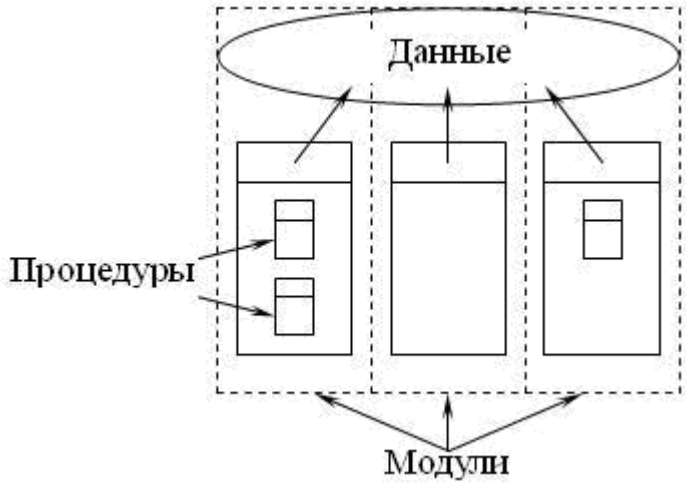


Рис. 3. Топология языков программирования третьего поколения

Для языков этого поколения характерно развитие и широкое использование метода структурного программирования, развитие средств абстрагирования типов (появилась даже теория типов). Получила развитие концепция модуля как программной оболочки, внутри которой можно скрыть данные и процедуры, а взаимодействие между различными модулями организовать с помощью интерфейса (рис.3). Скрытие данных поддерживается за счет раздельной компиляции модулей, а также механизмов управления доступом к

данным и процедурам внутри модуля. В языке SIMULA появились классы – основа объектно-ориентированного программирования.

4 ПОКОЛЕНИЕ (OBJECT PASCAL, SMALLTALK, MODULA WITH CLASSES, C++, JAVA)

В объектно-ориентированных языках реализована идея совместной разработки алгоритмов и данных за счет реализации отношения тип-подтип (класс-подкласс), связанного с использованием наследования свойств и развивающего концепцию абстрактного типа данных. Основным конструктивным элементом программирования служит модуль, составленный из логически связанных классов и объектов, а не подпрограмм (рис. 4). Взаимодействие объектов организуется с помощью механизма “посылки сообщений”.

В настоящее время объектно-ориентированные языки реализуются как системы визуального программирования. Отличительной особенностью таких систем является мощная среда разработки программ из готовых «строительных блоков», позволяющая создать интерфейсную часть программного продукта в диалоговом режиме, практически без кодирования программных операций. К числу объектно-ориентированных систем визуального программирования относятся Visial Basic, Delphi, C++ Builder, Visual C++, J2EE.

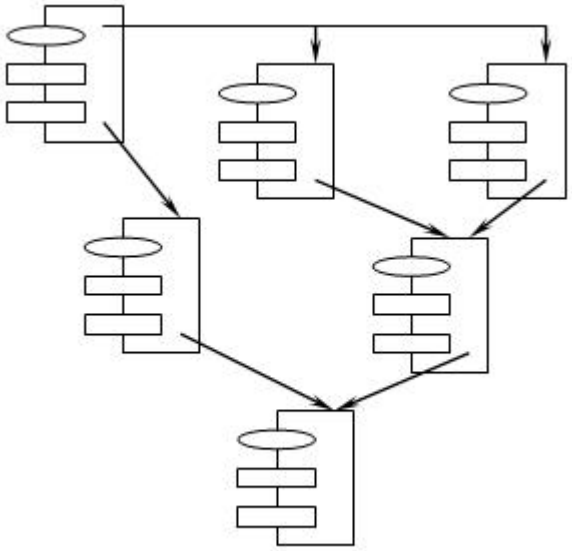


Рис. 4. Топология языков программирования четвертого поколения

1. АБСТРАГИРОВАНИЕ ТИПОВ

Данный раздел содержит обзор встроенных типов языков программирования на примере языка PASCAL и правила конструирования пользовательских типов.

1.1. Понятие типа данных

Тип данных рассматривается как множество данных, т.е. допустимых значений, которые некоторый объект может принимать в программе, совместно с множеством операций по обработке этих данных. Данные задаются с помощью описания их свойств (атрибутов). Совокупность операций, т.е. действий, производимых над данными, определяется с помощью процедур и функций, реализующих эти действия.

Фундаментальные типы данных подразделяются на простые или встроенные в язык программирования, и конструируемые пользователем, т.е. абстрактные. Классификация типов данных в языке PASCAL приведена на рис. 5.

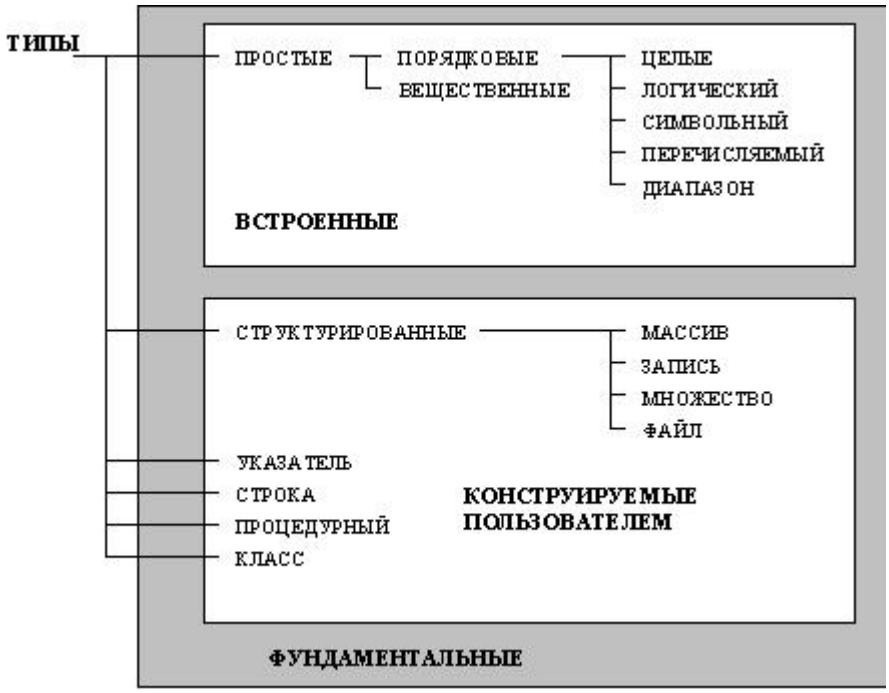


Рис. 5. Типы данных в языке PASCAL

В программе объектом некоторого типа является константа или переменная. Значения, которые может принимать объект данного типа, называются ***константами типа***. Во время выполнения программы для каждого объекта выделяется область оперативной памяти, называемая ***элементом хранения***, в которой размещаются константы типа. Максимальное количество элементов в множестве, включающем все константы типа, называется ***мощностью типа***. Мощность типа определяет ***размер элемента хранения*** объекта данного типа так, чтобы этот размер был достаточен для размещения любой константы типа. Формат внутреннего представления данных в элементе хранения определяется типом объекта.

1.1.1. Простые типы

В языке программирования за каждым простым типом закреплено имя. Любой простой тип можно задать перечислением констант этого типа.

1.1.1.1. Порядковые типы

Простые типы, которые имеют счетное множество констант, называются ПОРЯДКОВЫМИ. Например, в PASCAL множество натуральных чисел 0,1,2,…,65535 образует тип ***WORD***. Множество целочисленных констант в диапазоне -32768…32767 образует тип ***INTEGER***. Соответствие между константами порядкового типа и их номерами в множестве констант типа устанавливается порядком перечисления констант.

Множество операций по обработке данных простых типов встроено в язык программирования. Для порядковых типов это операции сложения, вычитания, умножения, деления, инкремента, декремента, определения предшествующего и следующего элемента в множестве констант. К любому порядковому типу применима функция ***ORD(X),*** которая вычисляет порядковый номер объекта *X*. Для целых типов функция *ORD(X)* возвращает само значение *X*. Результатом *ORD(X)* является положительное целое число в диапазоне 0…1 для логического типа, 0…255 для символьного типа, 0…65535 для перечислимого типа.

Обозначим мощность типа ***POWER(имя_т ипа)*** (это наше собственное обозначение, в языках программирования подобная функция отсутствует). Для определения размера элемента хранения типа (в байтах) в языках программирования используется встроенная функция ***SIZEOF(имя_т ипа)*** или ***SIZEOF(имя_объект а_т ипа)***. Встроенные функции ***LOW(имя_т ипа)*** и ***HIGH(имя_т ипа)*** возвращают соответственно минимальное и максимальное значения констант порядкового типа.

ЦЕЛЫЕ ТИПЫ. Мощность типа *WORD POWER(WORD) = 65536 = 2^{16}*. Размер элемента хранения переменной данного типа - *SIZEOF(WORD) = 2 (байт а),* т.к. 16 двоичных разрядов достаточно для того, чтобы закодировать любое значение из диапазона 0…65535. Например, число **2000** кодируется следующим образом:

2000 = 1024 + 512 + 256 + 128 + 64 + 16 = $2^{10} + 2^9 + 2^8 + 2^7 + 2^6 + 2^4$.

0	0	0	0	0	1	1	1	1	1	0	1	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

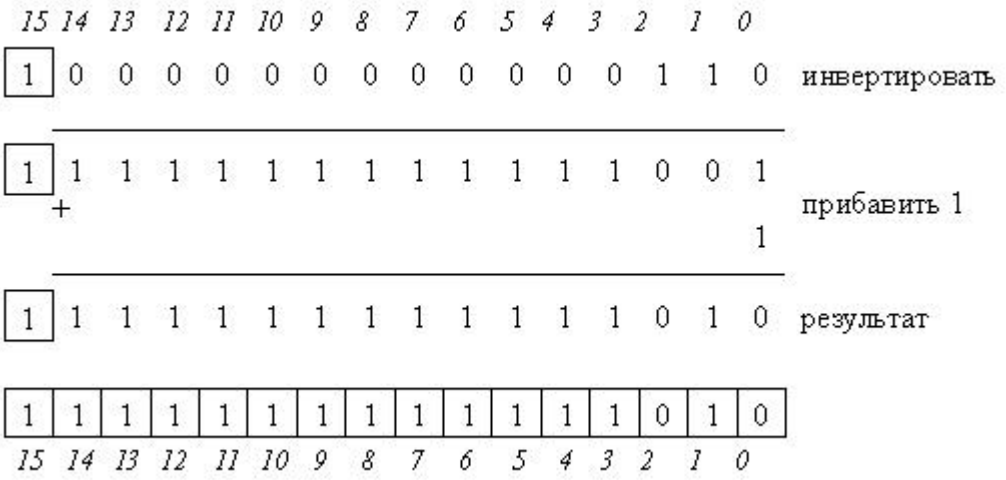
Ord(2000) = 2000.

Power(INTEGER) = 65536 = 2^{16}. Sizeof(INTEGER) = 2 (байт а).

Положительные числа в элементе хранения типа *INTEGER* кодируются так же как и в элементе хранения типа *WORD*. Отрицательные числа в элементе хранения типа *INTEGER* представляются в дополнительном коде. Дополнительный код получается следующим образом:

- представить отрицательное число в виде двоичного значения со знаком,
- инвертировать двоичное представление числа, оставив знаковый разряд без изменения,
- прибавить 1 к инвертированному числу,
- добавить знаковый бит к результату, полученному на предыдущем шаге.

Например, представим число ***-6*** в дополнительном коде.



Ord(-6) = -6.

В таблице 1 приводятся названия целых типов, размер их элемента хранения и диапазон возможных значений, образующих множество констант типа.

Таблица 1.

Целые типы

Название	Размер элемента хранения, байт	Множество констант
Byte	1	0…255
ShortInt	1	-128…127
Word	2	0…65535
Integer	2	-32768…32767
LongInt	4	-2147483648…2147483647

ЛОГИЧЕСКИЙ ТИП. Значениями логического типа может быть одна из предварительно объявленных констант ***FALSE***(ложь) и ***TRUE*** (истина).

Power(BOOLEAN) = 2, Sizeof(BOOLEAN) = 1 (байт),
Ord(False) = 0, Ord(True) = 1.

СИМВОЛЬНЫЙ ТИП. Значениями символьного типа является множество всех символов компьютера. Каждому символу приписывается целое число в диапазоне 0..255. Это число служит кодом внутреннего представления символа. Первая половина символов с кодами 0..127 соответствует стандарту *ASCII*. Вторая половина символов с кодами 128..255 не ограничена жесткими рамками стандарта и может меняться на компьютерах разных типов.

Power(CHAR) = 256, Sizeof(CHAR) = 1 (байт), Ord('F') = 70;

ПОЛЬЗОВАТЕЛЬСКИЙ ПЕРЕЧИСЛИМЫЙ ТИП (обозначим ***Enum_Type***) задает некоторое подмножество целого типа *WORD* и может рассматриваться как компактное объявление группы N именованных целочисленных констант со значениями 0,1,…65535. Например,

Type Note = (do, re, mi, fa, sol, la, si).
0 1 2 3 4 5 6

Так, значение константы с именем ***fa*** и соответственно ее порядковый номер в множестве констант типа Note равны 3, т.е. *Ord(fa) = 3.*

Power (Enum_Type) = N, N <= 65536.
Sizeof(Enum_Type) = 1 + log2(Power (Enum_Type) - 1) div 8.

Power(Note) = 7, Sizeof(Note) = 1 (байт).

Для ограничения полного множества значений некоторого порядкового типа используются отрезки типа. ОТРЕЗОК ТИПА ИЛИ ДИАПАЗОН (обозначим ***Diap_Type***) - упорядоченное подмножество полного множества значений любого порядкового типа, исключая диапазон. Тип-диапазон задается границами своих значений (обозначим их ***MIN*** и ***MAX***) внутри базового типа:

Type Diap_Type = MIN .. MAX;

MIN = Low(Diap_Type), MAX = High(Diap_Type).
Power (Diap_Type) = MAX - MIN + 1 <= 65536.
Sizeof(Diap_Type) = 1 + log2(Power (Diap_Type) - 1) div 8.

Например,

Type Otr = 10..19;
Low(Otr) = 10, High(Otr) = 19, Ord(15) = 15,
Power(Otr) = 10, Sizeof(Otr) = 1 (байт).

1.1.1.2. Вещественные типы

Порядковые типы конечны и счетны, значения их всегда сопоставимы с рядом целых чисел, и, следовательно, кодируются абсолютно точно. Множество констант ВЕЩЕСТВЕННЫХ ТИПОВ бесконечно, значения вещественных типов определяются с некоторой конечной точностью, зависящей от внутреннего формата вещественных чисел.

Таблица 2.

Вещественные типы

Название	Размер элемента хранения, байт	Количество значащих цифр	Диапазон десятичного порядка
single	4	7..8	-45…+38
real	6	11…12	-39…+38
double	8	15…16	-324…+308
Extended	10	19…20	-4951…+4932
comp	8	19…20	$-2*10^{63}+1$…$+2*10^{63}-1$

Элемент хранения вещественного типа имеет следующую структуру:

s	e	m

Здесь ***s*** - знаковый разряд числа; ***e*** - экспоненциальная часть, содержащая порядок; ***m*** - мантисса числа. Мантисса имеет длину от 23 (для *SINGLE*) до 63 (для *EXTENDED*) двоичных разрядов, что и обеспечивает точность 7..8 для *SINGLE* и 19..20 для *EXTENDED* десятичных цифр. Знак мантиссы определяет знак числа и имеет значения: 0 - для положительных чисел, 1 - для отрицательных чисел. Порядок числа запоминается увеличенным на 200_8 (128). Такой способ хранения порядка называется смещенным. Десятичная точка подразумевается перед левым (старшим) разрядом мантиссы, но при действиях с числом ее положение сдвигается влево или вправо в соответствии с двоичным порядком числа, хранящимся в экспоненциальной части, поэтому действия над вещественными числами называются арифметикой с плавающей точкой. На рис. 6 показано внутреннее представление вещественного числа **46,5** в формате *SINGLE*.

Перевод целой части

$$\begin{array}{r|l} 46 & 8 \\ \underline{40} & 5 \\ 6 & \end{array}$$

Перевод дробной части

$$\begin{array}{r|l} 0.5 & 8^{-1}=0.125 \\ \underline{0.5} & 4 \\ 0 & \end{array}$$

$46{,}5_{10} = 56{,}4_8 = 0{,}564_8 * 8^2 = 0{,}564_8 * 2^6$

Смещенное представление порядка (в формате *SINGLE* под порядок отводится 8 разрядов):

$6_{10} = 6_8$ → *00000110*

200_8 → + *10000000*

порядок = *10000110*

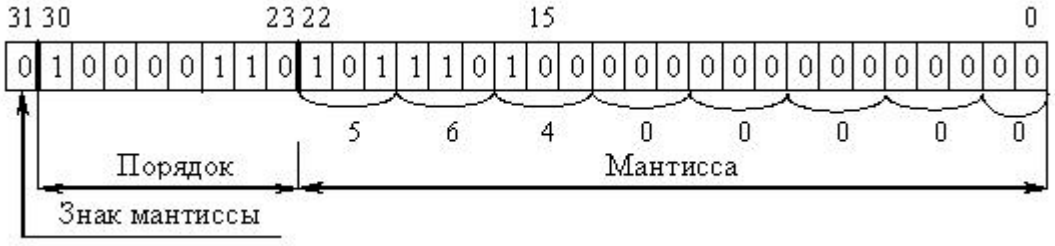


Рис. 6. Внутреннее представление вещественного числа 46,5.

1.1.2. Абстрактные типы

Абстрактные типы данных (АТД) конструируются в программе на основе встроенных и ранее сконструированных абстрактных типов. Если конструирование осуществляется путем ***агрегирования***, т.е. объединения свойств составляющих типов, полученный тип является ***структурированным типом или агрегатом***. Если при конструировании объединяются разнородные свойства, получается новый тип – ЗАПИСЬ (обозначим ***Rec_Type***). Пусть тип ***Rec_Type*** строится на основе типов Type1, Type2,… Type N.

```
Type Rec_Type = record
   < Имя свойства 1 >: Type1;
   < Имя свойства 2 >: Type2;
        . . .
   < Имя свойства N >: Type N;
end;
```

Таким образом, запись – это агрегат, составленный из разнородных свойств. Мощность типа запись представляется произведением мощностей составляющих типов. Размер элемента хранения типа запись равен сумме размеров элементов хранения составляющих подтипов.

*Power(Rec_Type) = Power(Type1) * Power(Type2) *…* Power(TypeN),*
Sizeof(Rec_Type) = Sizeof(Type1) + Sizeof(Type2) + … + Sizeof(TypeN).

Например,

```
Type Rec = record
   X: 0..9; Y: 0..9
end;
```

Множество констант этого типа включает 100 пар значений (0,0), (0,1), …, (0,9), (1,0), (1,1), …, (1,9), (9,0), … ,(9,9).

*Power(Rec) = Power(X) * Power(Y) = 10 * 10 = 100,*
Sizeof(Rec) = Sizeof(X) + Sizeof(Y) = 1 + 1 = 2 (байта).

Если при конструировании типа объединяются однородные свойства, получается новый тип - МАССИВ (обозначим ***Array_Type***). Массив – это агрегат, составленный из однородных свойств. Пусть тип ***Array_Type*** формируется из N элементов типа Type1:

```
Const N = …;                                  { N – количество элементов массива }
Type Array_Type = array [1..N ] of Type1;
```

$Power(Array\_Type) = Power(Type1)^N,$
*Sizeof(Array_Type) = N * Sizeof(Type1).*

На базе типа с ограниченным множеством значений (обозначим ***Base_Type***) можно построить тип с более широким спектром значений – тип МНОЖЕСТВО (обозначим ***Set_Type)***. Множество определяется как набор всевозможных комбинаций однотипных логически связанных объектов некоторого базового типа. Мощность базового типа не дожна превышать 256 констант. Пусть в качестве базового типа для построения АТД КАРТИНА используется перечисление цветов

Type Colour = (white, black, red, green, blue, yellow, gray, magenta, cyan);
Type Picture = Set of Colour;

Примеры множественных констант:
[ ], [red], [green, blue, red], … [green, black], … [white .. cyan].

Power(Set_Type) = 2 $^{Power(Base\_Type)}$,
Sizeof(Set_Type) = 1 + (Power(Base_Type) – 1) div 8 или
Sizeof(Set_Type) = (MAX div 8) – (MIN div 8) + 1, где
MIN = Low(Base_Type), MAX = High(Base_Type).

Power(Colour) = 9, Sizeof(Colour) = 1 (байт),
Power(Picture) = 2^9 *= 512, Sizeof(Picture) = 2 (байта).*

Элемент хранения объекта множественного типа должен допускать размещение $2^{Power(Base\_Type)}$ значений. В качестве представления объекта множественного типа используется ***характеристическая функция***, являющаяся массивом логических значений, i–я компонента которого означает наличие или отсутствие i-й константы базового типа в множестве. Каждая константа базового типа в представлении константы множественного типа имеет сопоставимый номер и занимает 1 бит, соответствующий ее порядковому номеру в базовом типе. Например, представление объекта [white, red, green, gray, cyan] в его элементе хранения выглядит следующим образом (рис.7):

			cyan	magenta	gray	yellow	blue	green	red	black	white
0	…	0	1	0	1	0	0	1	1	0	1
15	…	9	8	7	6	5	4	3	2	1	0

Рис. 7. Внутреннее представление объекта множественного типа

Константа базового типа с порядковым номером K в элементе хранения объекта множественного типа представлена битом с номером
BitNumber = K mod 8
в байте с номером
ByteNumber = (K div 8) – (MIN div 8), где
MIN = Low(Base_Type).

Ord(cyan) = 8, BitNumber(cyan) = 0, ByteNumber(cyan) = 1.

Операции над множествами определяются как теоретико-множественные операции над их характеристическими функциями.

ФАЙЛ – это либо именованная область внешней памяти (жесткого диска, дискеты, компакт-диска, электронного “виртуального” диска), либо логическое устройство – потенциальный источник или приемник информации. С логическими устройствами связаны стандартные аппаратные средства, такие как клавиатура, экран дисплея, принтер. В PASCAL существуют три вида файлов в зависимости от способа хранения информации в них: типизированные (обозначим ***File_Type***), текстовые (тип ***TEXT***) и нетипизированные (тип ***FILE***). С каждым файлом связана специальная структура, называемая дескриптором (описателем), в которой хранится имя файла, дата создания, размер файла, атрибут доступа к файлу и т.п. При описании в программе объекта файлового типа в оперативной памяти создается элемент хранения, в котором размещается дескриптор файла. Размер

дескриптора файла фиксирован для файла любого вида (128 байт) и никак не связан с размером файла во внешней памяти.

Type File_Type = File of <Name_Type>**,**
где <Name_Type> - любой тип, кроме файлового.

Sizeof(File_Type) = 128.

Для того чтобы описать совокупность операций, производимых над объектом, пользователь может сконструировать АТД – ПРОЦЕДУРНЫЙ ТИП (обозначим ***Proc_Type***). Любой процедурный тип определяет:

- множество возможных действий,
- множество объектов, над которыми могут быть произведены эти действия.

Например, процедурный тип

Type DST = PROCEDURE (var P: Point);

определяет возможные действия над объектами типа ТОЧКА. Любая процедура, описание которой совпадает с объявлением типа DST, может рассматриваться как объект типа DST. Например, процедуры

```
Init( var P: Point );          { инициализировать }
Show( var P: Point );          { отобразить }
Hide ( var P: Point );         { стереть }
Move( var P: Point );          {переместить }
```

являются объектами типа DST. В элементах хранения объектов процедурного типа размещаются адреса точек входа в соответствующие процедуры (точки запуска – активации процедур). Поэтому множество констант процедурного типа является подмножеством множества адресов оперативной памяти, т.е. подмножеством типа УКАЗАТЕЛЬ (обозначим ***Pointer_Type***), а размер элемента хранения процедурного типа определяется размером элемента хранения типа указатель (см. 2.2.2).

Sizeof(Proc_Type) = Sizeof(Pointer_Type).

1.1.2.1. Модуль - средство реализации абстрактного типа данных

Объявление и реализация АТД обычно помещаются в ***модуль***. Рассмотрим абстрактный тип данных ВРЕМЯ. Значение каждого момента времени состоит из 3 атрибутов: старших, средних и младших единиц временной шкалы (например, часы:минуты:секунды) (рис. 8).

hour	1
minute	15
second	37

Рис. 8. Элемент хранения АТД “ВРЕМЯ”

```
UNIT Time;
  { Раздел объявлений АТД   ВРЕМЯ  }
Interface
  { Описание  ат рибут ов }
 TTimer =record
    hour, minute, second: word
 end;
  { Описание  дейст вий по обработ ке объект ов т ипа ВРЕМЯ  }
  { Инициализация ат рибут ов объект а т ипа ВРЕМЯ }
Procedure Init (var t: TTimer);
  { Приращение значения времени }
Procedure Add ( var   t: Ttimer; dt: word);
  { Преобразование значения времени во внут реннее предст авление  }
Procedure TtoReal (t: Ttimer;   var r: real);
  { Преобразование значения времени из внут реннего предст авления  }
Procedure RealtoT (r: real; var t: TTimer);

{ Раздел реализации  дейст вий по обработ ке объект ов т ипа ВРЕМЯ  }
Implementation
  Procedure Init (var t: TTimer);
  begin
     . . .
  end;
    . . .
end.
```

Использование модуля Time:

```
UNIT Main;
uses   Time;
  var   t1,t2: Ttimer; r: real;
begin
     Init ( t1 );   Add( t1,10 );    TtoReal( t1,r );   . . .
end.
```

2. ИДЕНТИФИКАЦИЯ ОБЪЕКТОВ

Идентификация – это определение местонахождения элемента хранения объекта в оперативной памяти и получение доступа к представлению объекта, т.е. индивидуальным значениям его свойств. Существует два способа идентификации объектов:

- именование
- указание.

2.1. Именование

Именование заключается в назначении объекту определенного имени (идентификатора). Имена однозначно связываются с объектами на этапе компиляции программы, эту связь в процессе выполнения программы изменить нельзя. Именоваться могут и отдельные свойства объектов-агрегатов. Имена свойств называются квалифицированными идентификаторами (***квалидентами***). Длина пути, ведущего к имени конкретного свойства, называется ***длиной дистанции доступа***. Простой идентификатор можно рассматривать как квалидент с нулевой дистанцией доступа. Например:

```
Type Point = record                                  { тип ТОЧКА }
  X, Y: word
end;
Type Circle = record                                 { тип ОКРУЖНОСТЬ }
   R: word;
   Center: Point
end;
 Var C: Circle;
 Примеры квалидентов:
    C                          { длина дистанции доступа равна 0 }
   C.R                         { длина дистанции доступа равна 1 }
   C.Center.Y                  { длина дистанции доступа равна 2 }
```

Для того чтобы сократить время обращения к атрибутам объектов, используется оператор присоединения

WITH <квалидент> DO begin <присоединяемый фрагмент> end;

Например,
with C do begin R:=10; writeln(Center.Y) end;

При работе с массивами объектов и массивами однородных свойств идентификация осуществляется на основе ***индексирования*** (нумерации). Индекс определяет порядковый номер объекта (или свойства) и является уточненным именем в представлении агрегата.

Доступ к объекту, идентифицируемому именем (в том числе именем, уточненным индексом), реализуется на основе ***вычисления адреса*** элемента хранения объекта (см. 4.1).

2.2. Указание

Идентификация указанием основана на использовании адресов объектов.

2.2.1. Организация адресного пространства оперативной памяти MS DOS

Оперативная память представляет собой совокупность элементарных ячеек для хранения информации – байтов, каждый из которых имеет свой собственный номер, называемый адресом. ***Адрес*** позволяет обращаться к любому байту памяти. Структура адреса MS DOS – это два 16-разрядных слова типа WORD, которые трактуются как ***сегмен*т** и ***смещение*** внутри сегмента. Сегмент – это участок памяти, имеющий длину 65536 байт (64 Кбайт = 2^{16} байт) и начинающийся с физического адреса, кратного 16 (т.е. 0, 16, 32 и т.д.). Смещение указывает, на каком расстоянии от начала сегмента находится нужный байт памяти. Таким образом, любая ячейка адресного пространства определяется парой чисел СЕГМЕНТ : СМЕЩЕНИЕ.

Адресное пространство современных компьютеров гораздо больше 64 Кбайт и организовано оно последовательными непрерывными областями -сегментами. Для адресации в пределах, например, 1 Мбайта необходимо 20 двоичных разрядов (1 Мбайт = 2^{20} байт), которые получаются из двух 16-разрядных слов (сегмента и смещения) следующим образом: содержимое сегмента сдвигается влево на 4 разряда, освободившиеся правые разряды заполняются нулями, результат складывается с содержимым смещения (рис. 9).

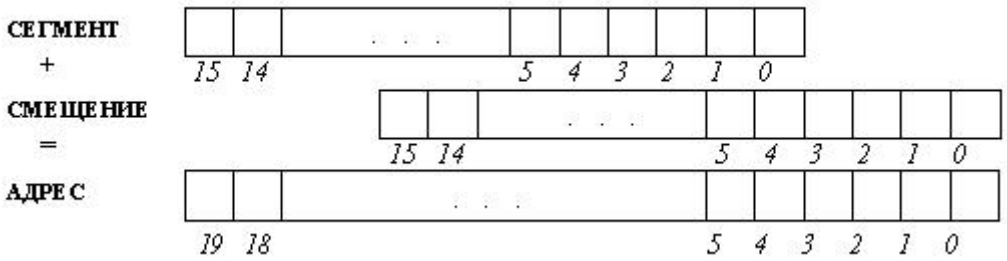


Рис. 9. Схема формирования адреса

Фрагмент памяти размером 16 байт называется параграфом, следовательно, сегмент адресует память с точностью до параграфа, а смещение – с точностью до байта. Адреса принято записывать в 16-ричном формате.

Можно получить адрес ячейки оперативной памяти, отсчитанный от начала памяти, т.е. от адреса $0000 : $0000. Такой адрес называется ***сплошным***, его элементом хранения является число типа *LongInt*.

*Сплошной адрес = СЕГМЕНТ * 16 + СМЕЩЕНИЕ.*

Адрес, у которого смещение находится в диапазоне 0..15 ($0000..$000F), называется ***нормализованным***. Сплошной адрес переводится в нормализованный формат следующим образом:

СЕГМЕНТ = Сплошной адрес div 16,
СМЕЩЕНИЕ = Сплошной адрес mod 16.

Один и тот же сплошной адрес можно представить несколькими способами. Например, 39-й байт памяти можно адресовать с помощью нормализованного адреса, записав $0002 : $0007. Тот же самый адрес может быть записан в другом виде, если изменить значение сегмента, например, $0001 : $0017.

2.2.2. Понятие указателя

Указание связано с использованием ссылочного или указательного типа. ***Указатель*** – это особый объект, в элементе хранения которого могут содержаться адреса любых других объектов. Таким образом, константами ссылочного типа являются адреса ячеек оперативной памяти и особое значение указателя - ***NIL***, которое не указывает ни на один из существующих в программе объектов. Мощность ссылочного типа определяется адресным пространством оперативной памяти. Размер элемента хранения ссылочного типа равен размеру элемента хранения адреса и составляет 4 байта.

Sizeof(Pointer_Type) = 4 (байта).

Объекты ссылочного типа подразделяются на ***типизированные*** (ограниченные) и ***нетипизированные*** (свободные) указатели. Свободный указатель имеет встроенный тип ***POINTER***, может хранить адрес любого объекта (в том числе и объекта ссылочного типа), но не позволяет получить доступ к атрибутам объекта, т.к. ни с каким конкретным типом свободный указатель не связан. Ограниченный указатель всегда задается так, чтобы указывать на объекты определенного типа.

```
Type PPoint = ^ Point;                    { указатели на объекты типа ТОЧКА }
Type PCircle = ^ Circle;                { указатели на объекты типа ОКРУЖНОСТЬ }
```

Определить тип ограниченного указателя можно и до определения того типа, с которым он связан. Пример описания объектов ссылочного типа:

```
Var  pp: PPoint;  pc: PCircle;
```

2.2.3. Действия над указателями

Различают следующие действия над указателями:

- присваивание;
- раскрытие ссылки;
- сравнение указателей.

2.2.3.1. Присваивание

Присваивание для указателей сводится к пересылке значения одного указателя другому. Совместимыми по присваиванию являются:

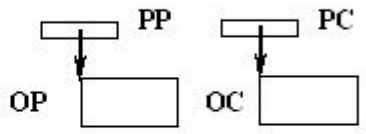
- два указателя одного и того же типа;
- константа *NIL* и свободный указатель;
- константа *NIL* и ограниченный указатель любого типа;
- свободный указатель и ограниченный указатель любого типа.

Ограниченному указателю одного типа можно присвоить значение ограниченного указателя другого типа с помощью функции приведения типов ***POINTER (но без крайней необходимост и делат ь эт о не следует , т .к. подобные дейст вия приводят к нарушению ст рогой т ипизации).***

Каждый указатель перед использованием необходимо ***инициализироват ь***, т.е. установить на соответствующий объект.

Установка указателя на объект, адрес которого неизвестен, производится с помощью встроенной функции взятия адреса - ***Addr()***, аргументом которой является идентификатор объекта. Эта функция возвращает результат типа *Pointer*, в котором содержится адрес аргумента. Аналогичный результат возвращает операция @.

```
Type
  PPoint = ^ Point;                    { тип - указатели на объекты типа ТОЧКА }
  Point = record                                                  { тип ТОЧКА }
    X, Y: word
  end;
  PCircle = ^ Circle;
  Circle = record                  { тип - указатели на объекты типа ОКРУЖНОСТЬ }
    R: word;                                                  { тип ОКРУЖНОСТЬ }
    Center: Point
  end;
Var
  op: Point;                                                    { объект ТОЧКА }
  pp: PPoint;                                     { объект – указатель на объект ТОЧКА }
  oc: Circle;                                                { объект ОКРУЖНОСТЬ }
  pc: PCircle;                                 { объект – указатель на объект ОКРУЖНОСТЬ }
begin
  op.X:=100; op.Y:=200;                          { заполнение атрибутов объекта ТОЧКА}
  pp:=@op;                                       { установка указателя на объект ТОЧКА }
  pc:=addr(oc);                               { установка указателя на объект ОКРУЖНОСТЬ }
```




```
with oc do                                  { заполнение атрибутов объекта ОКРУЖНОСТЬ }
  begin  R:=5;
    with Center do
    begin X:=10; Y:=20 end;
end;
```

Установка указателя на объект, адрес которого известен и хранится в другом указателе, производится с помощью присваивания.

```
Var
    pp, pp1, pp2: PPoint; pc: PCircle; p: Pointer;
begin
     pp1:= pp;   p:= pc;   pp2:= nil;
```

Результат выполнения этих операций иллюстрируется рис. 10.

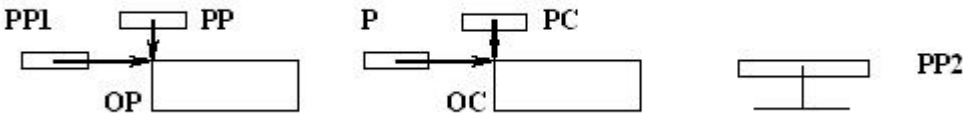


Рис. 10. Присваивание указателей

2.2.3.1. Доступ к объекту через указатель. Раскрытие ссылкии

Доступ к объекту через ограниченный указатель связан с предварительной установкой указателя на объект и последующим раскрытием ссылки. При этом имя указателя используется как идентификатор, за которым следует символ “^”, (т.е. ***квалидент с пост фиксом “^”***). Доступ к объекту и его атрибутам осуществляется в несколько шагов:

- ***имя указат еля*** - указатель используется для получения адреса того объекта, с которым он связан,
- ***имя указат еля^*** - открывает доступ ко всему объекту, адрес которого хранится в указателе,
- ***имя указат еля^ .*** - открывает доступ к атрибутам объекта,
- ***имя указат еля^ . имя ат рибут а*** - открывает доступ к конкретному атрибуту объекта.

Каждый из подобных квалидентов открывает доступ к уникальному объекту или атрибуту объекта.

```
Var p: PCircle;
    p                          { доступ к объекту ОКРУЖНОСТЬ }
    p^.R                { доступ к атрибуту РАДИУС объекта ОКРУЖНОСТЬ }
    p^.Center.X           { доступ к атрибуту X объекта ОКРУЖНОСТЬ }
    p^.Center.Y           { доступ к атрибуту Y объекта ОКРУЖНОСТЬ }
```

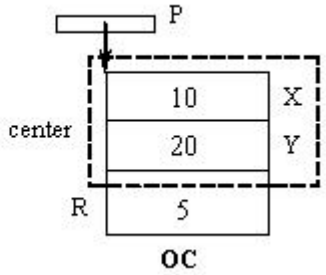
В общем случае размер объекта типа указатель не равен размеру элемента хранения объекта, доступ к которому он открывает, и не равен размеру атрибута этого объекта, т.е.

Sizeof(указат ель) <> Sizeof(указат ель^) <> Sizeof(указат ель^.ат рибут).

Для того чтобы сократить длину дистанции доступа при идентификации указанием, можно использовать оператор присоединения

WITH < указат ель^ > DO begin < присоединяемый фрагмент > end;

```
Var oc: Circle; p: PCircle;
begin
  p:=addr( oc );
  with p^ do begin
    Center.X:=10; Center.Y:=20; R:=5
  end;
```



Любое присоединение, объявленное оператором *WITH*, выполняется после того, как определено значение присоединяющего квалидента (другими словами, адреса того объекта, к атрибутам которого будет осуществляться доступ через указатель), т.е. до “входа” в обрабатываемый фрагмент. Изменение значения присоединяющего указателя внутри присоединяемого фрагмента не изменит уже созданного присоединения. Поэтому переустановка указателя, присоединяющего к обрабатываемому объекту, внутри присоединяемого фрагмента недопустима, а может выполняться только до оператора *WITH*. Правильный пример использования оператора присоединения приведен выше. Ошибочный пример следует далее:

```
Var oc1, oc2: Circle; p: PCircle;
 begin
   p: = @oc1;
   with p^ do begin
     p: = @oc2;                          { ошибка: переустановка указателя внутри

                  присоединяемого фрагмента }
     p^. R: = ……
  end;
```

Если два типизированных указателя указывают на разные объекты одного и того же типа, то одному объекту можно присвоить значения атрибутов другого объекта. Например:

```
Var   p, q: PCircle;
 begin
   p^. R:=5;        p^. Center.X:=10;        p^.Center.Y:=20;          {   1   }
          q^:=p^;                                                     {   2   }
```

Выполнение операторов { 1 } и { 2 } иллюстрирует рис. 11.

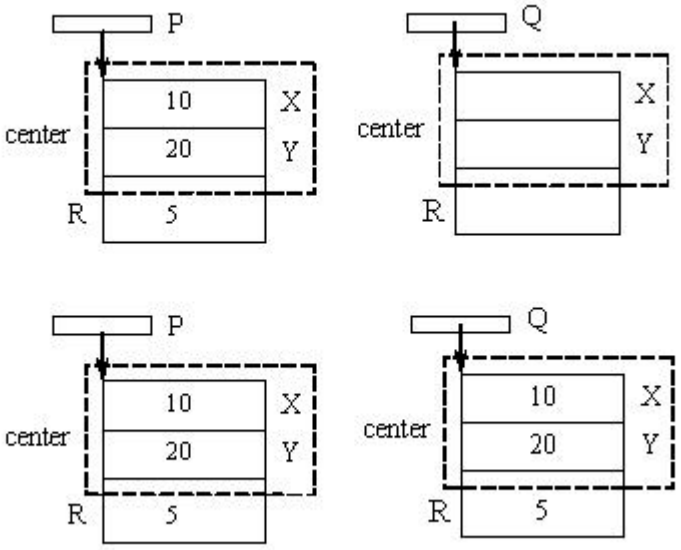


Рис. 11. Выполнение операторов { 1 } и { 2 }

Оператор { 2 } эквивалентен следующей группе операторов:

```
q^.R:=p^.R;   q^.Center.X:= p^.Center.X;     q^.Center.Y:=p^.Center.Y;
```

Заметим, что с использованием присоединения эту группу операторов можно записать следующим образом:

```
with q^ do begin
   R:=p^.R; Center.X:=p^.Center.X; Center.Y:=p^.Center.Y
end;
```

или

```
with p^ do begin
    q^.R:=R; q^.Center.X:=Center.X; q^.Center.Y:=Center.Y
end;
```

После выполнения присваивания значения атрибутов первого и второго объекта становятся равными, а указатели по-прежнему указывают на разные объекты.

2.2.3.1. Сравнение указателей

Указатели можно сравнивать между собой на равенство и неравенство. Два указателя считаются равными, если они указывают на один и тот же объект или оба никуда не указывают (оба равны *NIL*). Неравные указатели указывают на разные объекты или один из них никуда не указывает. Указатель можно сравнивать с константой *NIL*, чтобы узнать, ссылается ли данный указатель на конкретный объект. Порядок вычисления булевских выражений в условных операторах, использующих доступ к атрибутам объектов через указатели, очень важен. Например, оператор:

```
If ( p <> nil )   and   ( p^. R = 10 ) then …
```

будет работать корректно, даже если *p = nil*. В этом случае второе условие проверяться не будет согласно правилу вычисления булевских выражений.

Оператор

```
If ( p^. R   = 10 )   and   ( p <> nil )   then …
```

является некорректным, т.к. если p = nil, выражение p^. R = 10 не имеет смысла, поскольку указатель p ни на какой конкретный объект не указывает.

2.2.4. Связывание идентификатора объекта и его элемента хранения

Связывание – это определение взаимосвязи между идентификатором объекта (именем или указателем на объект) и элементом хранения объекта.

При идентификации именованием существует статическая связь между именем объекта и его элементом хранения (рис. 12).

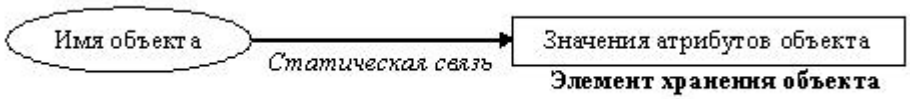


Рис. 12. Связывание при идентификации именованием

При идентификации указанием существует статическая связь между именем указателя и элементом хранения указателя. Между элементом хранения указателя и тем объектом, на который он указывает, устанавливается динамическая связь (рис.13).

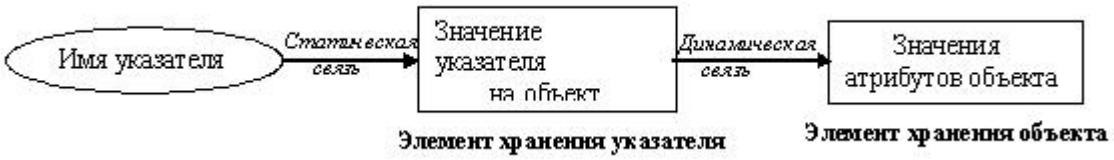
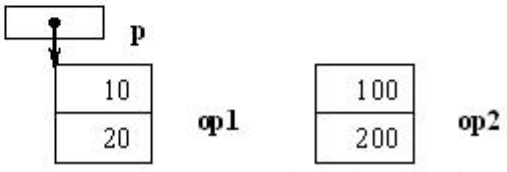


Рис. 13. Связывание при идентификации указанием

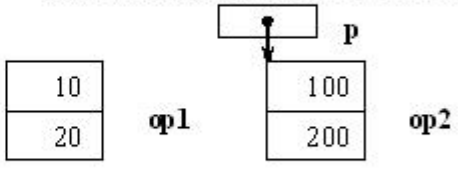
Статическая связь устанавливается на этапе компиляции, динамическая – на этапе выполнения программы.

Т.к. элемент хранения указателя содержит адрес объекта, в процессе выполнения программы один и тот же указатель может открывать ***доступ к различным объектам одного и того же типа*** (и атрибутам этих объектов).

```
Var op1, op2: Point; p: PPoint;
Begin
   op1.X:=10; op1.Y:=20;          { заполнение атрибутов объекта op1 }
   op2.X:=100; op2.Y:=200;        { заполнение атрибутов объекта op2 }
   p:=@op1;                       { установка указателя p на объект op1}
```




```
   writeln( p^.X, p^.Y );         { p^.X=10, p^.Y=20 }
   p:=@op2;                       { установка указателя p на объект op2 }
```




```
   writeln( p^.X, p^.Y );         { p^.X=100, p^.Y=200 }
```

3. ВРЕМЯ ЖИЗНИ ОБЪЕКТА. КЛАССЫ ПАМЯТИ

Во время выполнения программы объекты программы располагаются в оперативной памяти, которая по функциональному назначению подразделяется на ряд областей или классов.

3.1. Понятие “времени жизни” объекта

Все объекты программы подразделяются на ***статические*** и ***динамические***. Эти категории определяются через понятие “времени жизни” объекта. Объекты, продолжительность существования которых равна времени выполнения программы, называют статическими, а объекты, время жизни которых меньше времени выполнения программы, - динамическими. С понятием времени жизни объекта связаны его создание и уничтожение. Объекты, идентифицируемые именем, всегда являются статическими. Статические объекты создаются на этапе компиляции программы, до окончания работы программы для них сохраняется однозначное соответствие между элементом хранения объекта и именем объекта, по окончании работы программы они прекращают свое существование. Динамические объекты идентифицируются только через указатели, создаются и уничтожаются они в процессе выполнения программы. По окончании работы программы все динамические объекты также удаляются.

3.2. Классы памяти

Распределение рабочего пространства оперативной памяти не является жестким, а происходит во время выполнения программы (см. рис. 14).

В нижних адресах располагаются системные программы. Выше располагается код исполняемого файла (файла с расширением .EXE), размер которого может превышать 64 К. Выполняемому файлу придается сегмент данных, размер которого не превышает 64 К. Далее располагается область системного стека, необходимая для работы процедур и функций. Размер стека составляет не более 64 К. Стек заполняется от своей верхней границы по направлению к началу. Размер стека может быть назначен директивой *$M*. Выше стека располагается буфер для работы оверлеев – перекрывающихся частей программы. В верхних адресах оперативной памяти размещается куча (*heap*), необходимая для работы с динамическими объектами программы. Размером кучи пользователь может управлять при помощи директивы *$M*, которая имеет следующий формат:

{$M <stacksize>,<heapmin>,<heapmax>} – установить размеры памяти.

- <*stacksize*> - размер стека, изменяется от 1024 до 65520 байт;
- <*heapmin*> - минимальный размер динамической памяти, изменяется от 0 до 655360 байт;
- <*heapmax*> - максимальный размер динамической памяти, изменяется от <*heapmin*> до 655360 байт.

Размеры памяти по умолчанию - {$M 16384,0,655360}.

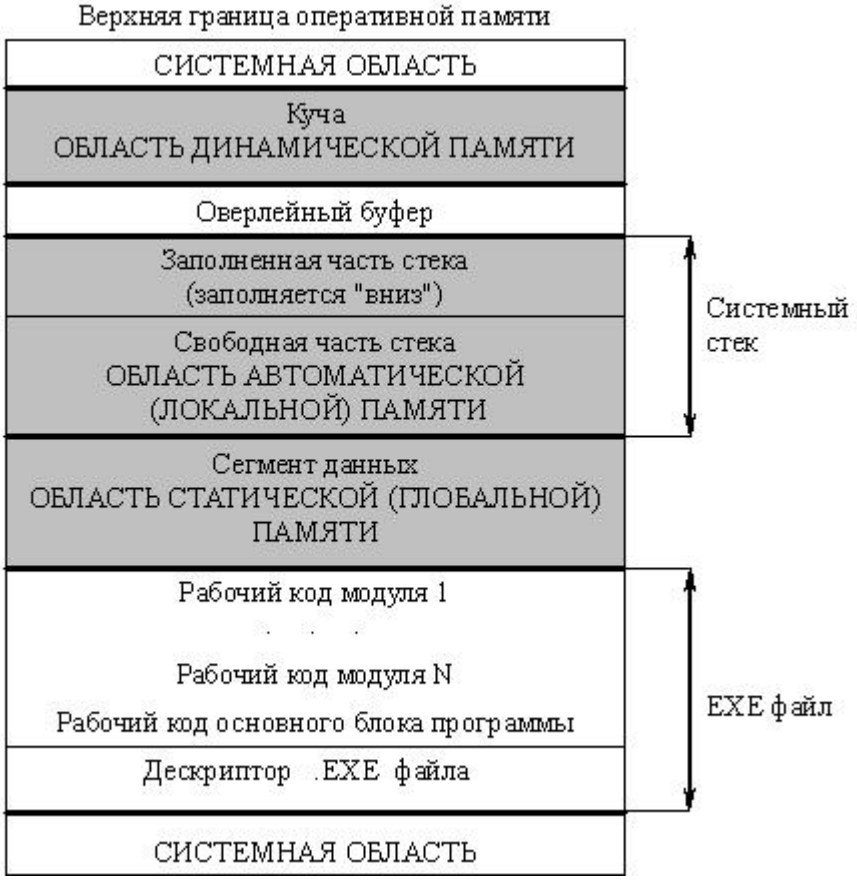


Рис. 14. Распределение рабочего пространства оперативной памяти

Создание объекта следует рассматривать как выделение памяти под его элемент хранения. Согласно такому подходу, рабочее пространство оперативной памяти подразделяется на три класса: статическая память, автоматическая память, динамическая память.

3.2.1. Статическая память

Статической памятью является сегмент данных размером не более 64 Кбайт, который выделяется каждой выполняемой программе на этапе загрузки. Элементы хранения объектов в статической памяти располагаются подряд в порядке их объявления в глобальной области программы.

3.2.2. Автоматическая память

Автоматическая память управляется директивами программы, связанными с вызовами процедур и их окончанием. Каждая процедура для своей работы требует индивидуальной локальной среды, которая называется фреймом активации процедуры. ***Фрейм активации*** включает значения фактических параметров, подставляемых на место формальных параметров, указанных в заголовке процедуры, значения локальных переменных, описываемых внутри процедуры, а также элемент хранения адреса возврата из процедуры. Фрейм активации однозначно характеризует процедуру, т.к. содержит набор объектов, необходимых для ее выполнения. Размещение локальной среды связано с активацией процедуры и происходит автоматически в момент ее вызова, а удаление локальной среды связано с пассивацией процедуры при завершении ее выполнения. Поэтому объекты, размещаемые в локальной среде, следует отнести к категории динамических. В программе может одновременно существовать несколько активных процедур. Последовательность активации и пассивации процедур связана с вложенностью их вызовов (первой должна завершиться процедура, которая была позже всех вызвана). Управление автоматической памятью должно обеспечивать возможность корректного выполнения вызовов процедур в соответствии с дисциплиной "последним пришел – первым вышел" (*LIFO – Last Input First Output*). Для этой цели наиболее подходящей является структура стека, и поэтому область автоматической памяти располагается в области системного стека. При активации каждой новой процедуры верхушка стека "опускается вниз" на величину, определяемую размером локальной среды данной процедуры. При пассивации процедуры верхушка стека "поднимается вверх" на эту же величину.

Ниже для фрагмента программы приведена иллюстрация распределения статической и автоматической памяти (рис. 15). В глобальной области программы описаны две переменные: *x* и *y*. В статической памяти для переменной с именем *x* выделен элемент хранения размером *Sizeof(WORD)=2(байта)*, в который в результате выполнения операции присваивания занесено значение 10, для переменной с именем *y* выделен элемент хранения размером *Sizeof(REAL) = 6(байтов)*, значение которого будет неопределенным до завершения выполнения процедуры *W1*. Активация процедуры *W1* приведет к созданию локальной среды, в которой будут размещены элементы хранения следующих объектов:

- значение фактического параметра *x* (2 байта), равное 10, подставляемого на место формального параметра *x1: word*, т.к. данный параметр передается по значению,
- указатель (4 байта) на фактический параметр *y*, подставляемый вместо формального параметра *var y1: real,* т.к. данный параметр передается по ссылке, значение указателя равно адресу переменной *y* в статической памяти,
- значение локальной переменной *A: integer* (2 байта),
- адрес возврата из процедуры (4 байта).

В процессе выполнения процедуры *W1* происходит вызов процедуры *W2*. Активация процедуры *W2* приведет к созданию локальной среды, в которой будут размещены элементы хранения следующих объектов:

- значение локальной переменной *B: word* (2 байта),
- адрес возврата из процедуры (4 байта).

Пассивация процедур *W1, W2* и, соответственно, освобождение локальной среды каждой из них происходит в обратном порядке.

```
Var x: word; y: real;
```


```
Procedure W1 (x1: word; var y1: real);
Var A: integer;
 begin
   …
   W2;
   …
end;

begin
   x:=10; W1( x,y ); writeln( y );
end.
```

```
Procedure W2;
Var  B: word;
begin
        …
end;
```

Рис. 15. Распределение статической и автоматической памяти

3.2.3. Динамическая память

Динамическая память связана с созданием и уничтожением объектов по явному запросу из программы на выделение и освобождение памяти.

ПРОЦЕДУРЫ И ФУНКЦИИ ДЛЯ РАБОТЫ С ДИНАМИЧЕСКОЙ ПАМЯТЬЮ.

Функция ADDR. Возвращает результат типа POINTER, в котором содержится адрес аргумента. Обращение:

ADDR(X),

где *X* – любой объект программы (имя любой переменной, процедуры, функции). Возвращаемый адрес совместим с указателем любого типа. Аналогичный результат возвращает операция @.

Процедура SIZEOF. Возвращает длину в байтах элемента хранения указанного объекта. Обращение:

SIZEOF (X),

где *X* – имя переменной или типа.

Процедура NEW. Резервирует фрагмент кучи для размещения элемента хранения динамического обьекта программы. Обращение:

NEW (TP),

где *TP* – типизированный указатель. Размер элемента хранения определяется автоматически согласно размеру того типа, с которым связан типизированный указатель *TP*. За одно обращение к процедуре можно зарезервировать не более 65520 байт динамической памяти. В результате выполнения процедуры в куче размещается элемент хранения объекта, адрес первого байта которого занесен в указатель *TP*. Выделенная область памяти не инициализирована, программист должен занести в нее необходимые значения атрибутов объекта.

```
Var p: PCircle;
 begin
    new (p);  p^.R:=5; p^.Center.X:=10; p^.Center.Y:=20; …
```

Динамически можно создавать объекты не только абстрактных, но и встроенных типов.

```
Var pint: ^Integer;
begin
  new( pi ); pint^ := -100;  …
```

Процедура GETMEM. Резервирует фрагмент кучи требуемого размера. Обращение:

GETMEM(P,SIZE),

где *P* – нетипизированный указатель, в который заносится адрес выделенной области памяти, *SIZE* – размер резервируемой памяти в байтах. За одно обращение к процедуре можно зарезервировать не более 65520 байт динамической памяти. Использование этой процедуры, как правило, связано с множественной интерпретацией области памяти, выполняемой при помощи маскирования (см. 5.2.5).

Процедура DISPOSE. Возвращает в кучу фрагмент динамической памяти, ранее связанный с типизированным указателем. Обращение:

DISPOSE (TP),

где *TP* – типизированный указатель, который должен быть установлен на реальный объект, т.е. не может быть равен *NIL*. Размер элемента хранения определяется автоматически согласно размеру того типа, с которым связан типизированный указатель *TP*. В результате выполнения процедуры фрагмент динамической памяти считается свободным, он не инициализируется, значение указателя не изменяется. Повторное применение этой процедуры к тому же самому указателю приведет к ошибке времени исполнения. Чтобы избежать подобных ошибок, можно инициализировать освободившийся указатель значением *NIL*.

```
Var p: PCircle; pint: ^Integer;
begin
  new( p );  new( pint ); … dispose( p ); p:=nil; dispose( pint ); pint:=nil; …
```

Процедура FREEMEM. Возвращает в кучу фрагмент динамической памяти заданного размера. Обращение:

FREEMEM(P,SIZE),

где *P* – нетипизированный указатель, в котором находится адрес освобождаемой области памяти, *SIZE* – размер освобождаемой памяти в байтах. Все замечания относительно работы процедуры *DISPOSE* справедливы и для процедуры *FREEMEM*. Освобождать следует ровно столько памяти, сколько ранее было зарезервировано и именно с того адреса, с которого она была зарезервирована. Использование процедуры *FREEMEM*, как и *GETMEM*, также связано с множественной интерпретацией области памяти, выполняемой при помощи маскирования (см. 5.2.5).

Функция MAXAVAIL. Возвращает размер в байтах наибольшего непрерывного участка кучи. Обращение:

MAXAVAIL.

Результат имеет тип *LongInt*. За один вызов процедуры *NEW* или *GETMEM* нельзя зарезервировать памяти больше, чем значение, возвращаемое этой функцией.

Функция MEMAVAIL. Возвращает размер в байтах общего свободного пространства кучи. Обращение:

MEMAVAIL.

Результат имеет тип *LongInt*.

Функции SEG и OFS. Возвращают значения типа WORD, содержащие соответственно сегмент и смещение адреса указанного объекта. Обращение:

SEG(X)
OFS(X),

где *X* – выражение любого типа или имя процедуры.

```
Var pint: ^ Integer;
begin
  new( pint ); pint^:=5; …

SEG( pint );    { возвращает сегментную часть адреса, по которому
                  расположен указатель pint }
SEG( pint^ );   { возвращает сегментную часть адреса, по которому
                  расположен элемент хранения типа Integer }
```

Функция PTR. Возвращает значение указателя, которое формируется по заданному сегменту и смещению. Обращение:

PTR(SEG,OFS),

где *SEG* – выражение типа *WORD*, содержащее сегмент, *OFS* – выражение типа *WORD*, содержащее смещение. Значение, возвращаемое функцией, совместимо с указателем любого типа.

3.3. Контрольные вопросы к разделам 1 - 3

1. Определите понятие типа данных. Дайте определение константы типа, элемента хранения типа, мощности типа.

2. Перечислите встроенные типы языка Pascal. Какое внутреннее представление имеют элементы хранения этих типов? Какова мощность каждого из этих типов и размер соответствующего элемента хранения?

3. Какие виды идентификации объектов существуют? Чем они отличаются?

4. Дайте определение адреса. Для чего используется адрес? Какова его структура? Какие способы представления адреса Вам известны?

5. Дайте определение указательного типа? Какие виды указателей Вам известны? Каковы их функции? Чем они отличаются?

6. Какие действия над переменными указательного типа Вы знаете?

7. В чем особенности выполнения операции раскрытия ссылки для получения доступа к объектам встроенных и конструируемых типов?

8. Как выполняется связывание идентификатора объекта и его элемента хранения при идентификации именованием? При идентификации указанием?

9. Какие классы памяти Вам известны? Каким образом распределяется рабочее пространство оперативной памяти во время исполнения программы?

10. Каким образом функционирует статическая память? Для чего она предназначена?

11. Каким образом функционирует автоматическая память? Для чего она предназначена?

12. Какие операции для работы с динамической памятью Вам известны?

3.4. Упражнения к разделам 1 - 3

1. Дано:

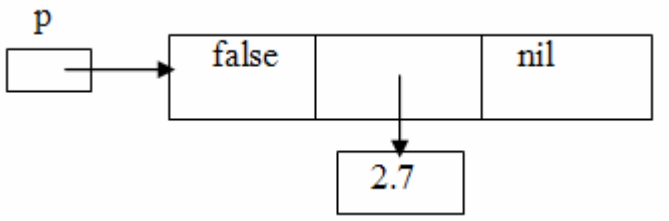
```
type pe = ^ elem;
  elem = record   data: integer;  link: pe  end;
var p,q : pe;
```

Какие значения получат переменные в результате выполнения следующих операторов? Приведите графическую иллюстрацию для каждой из групп операторов a), b), c), d), e) отдельно.

a) new(p); p^.data := 4; p^.link := nil;
b) new(p); p^.data := 7; p^.link := p;
c) new(q); q^.data := 2; q^.link := nil;
   new(p); p^.data := 1; p^.link := q;
d) new(p); p^.data := 5; new(p^.link); p^.link^ := p^;

2. Опишите переменные и их типы, а также операторы, в результате выполнения которых переменные получат значения, указанные на рисунке. Память для элементов хранения объектов, на которые ссылаются указатели, выделите

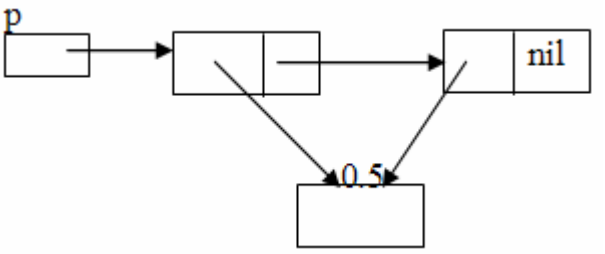
a) статически,
b) динамически.



Чему равны следующие значения:
sizeof(p) = ?; sizeof(p^) = ?

3. Опишите переменные и их типы, а также операторы, в результате выполнения которых переменные получат значения, указанные на рисунке. Память для элементов хранения объектов, на которые ссылаются указатели, выделите

a) статически,
b) динамически.



Чему равны следующие значения:
sizeof(p) = ?; sizeof(p^) = ?

4. Дано:

```
type  pr = ^ real;
```

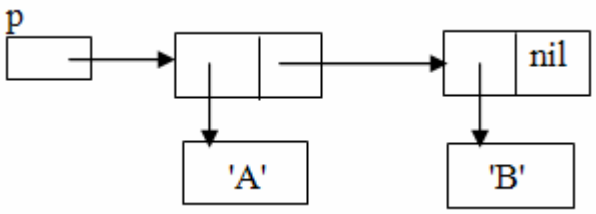
```
var  p,q: pr;  r: real;
begin

a) getmem( p, sizeof(real) ); new( q ); q^ := 3;
    p^ := 2;  q := p;  r := q^;  writeln( r );
```
Чему равно значение переменной r?

```
b) getmem( p, sizeof(real) );  q := p;
   dispose( p ); r := q^;
```
Можно ли вместо оператора getmem использовать оператор new? Почему?

Приведите графическую иллюстрацию выполнения каждого оператора алгоритмов a) и b). Каковы недостатки этих алгоритмов?

5. Опишите переменные и их типы, а также операторы, в результате выполнения которых переменные получат значения, указанные на рисунке. Память для элементов хранения объектов, на которые ссылаются указатели, выделите

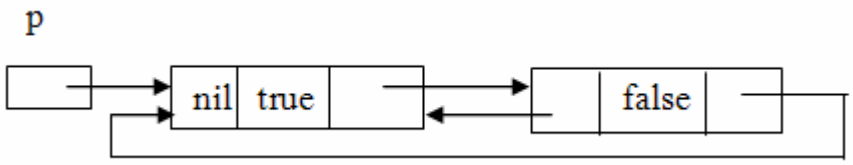
a) статически,
b) динамически.



Чему равны следующие значения:
sizeof(p) = ?; sizeof(p^) = ?

6. Опишите переменные и их типы, а также операторы, в результате выполнения которых переменные получат значения, указанные на рисунке. Память для элементов хранения объектов, на которые ссылаются указатели, выделите

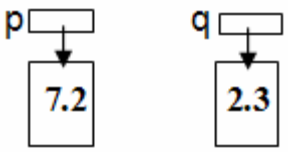
a) статически,
b) динамически.



Чему равны следующие значения:
sizeof(p) = ?; sizeof(p^) = ?

7. Дано:

```
type pw = ^ word;
var p,q: pw;
```



Определите операторы, в результате выполнения которых переменные p и q получат значения, представленные на рисунке. Память для элементов хранения объектов, на которые ссылаются указатели, выделите

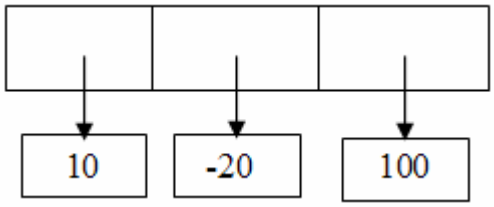
a) статически,
b) динамически;

Какое значение будет распечатано в результате выполнения следующих операторов? Приведите графическую иллюстрацию. Каковы недостатки этого алгоритма?

```
q^ := p^;
if p = q then p:=nil else if p^ = q^ then p:=q;
if p = q then p^:=10; writeln( q^ );
```

8. Опишите переменные и их типы, а также операторы, в результате выполнения которых переменные получат значения, указанные на рисунке. Память для элементов хранения объектов, на которые ссылаются указатели, выделите

a) статически,
b) динамически.



9. Дано:

```
type pw = ^ word;  pr = ^ real;
var p: pw;  y: real;  q: pr;
begin
{A} new( q );  q^ := 10.5; {B}  new( p ); p^ := 5;
{C}  y: =  q^ + real( p^ );
    {D} dispose( p );   {E}  writeln( y );  {F}
```

a) Какие переменные существуют в каждой из точек A, B, C, D, E, F и каковы их значения в эти моменты? Приведите графическую иллюстрацию;
b) Можно ли переменной q присвоить ссылку на переменную y? Почему? Какие объекты данного фрагмента программы можно уничтожить с помощью оператора dispose? Почему?
c) Чему равны значения

```
sizeof(  pr ) =?  sizeof(  q^ ) =?  sizeof(  p^ ) =?
```

10. Дано:

```
type pint = ^ integer;
   B = record f1: integer; f2: pint end;
   pB = ^ B;
```

```
var  p: pB;  q: pint;
begin
    new( q );  q^ :=10;   new( p );
    p^.f1 := q^ + 5;  p^.f2 := q;
```

а) Какие значения получат переменные в результате выполнения указанных выше операторов? Приведите графическую иллюстрацию.

б) можно ли присвоить переменной p значение, хранящееся в переменной q? Почему?

4. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Динамические структуры данных предназначены для описания множеств объектов, для которых определен порядок следования, а их состав и взаимное расположение могут изменяться в процессе выполнения программы.

4.1. Метод вычисляемого и хранимого адреса. Последовательная и связанная организация памяти

В соответствии с двумя методами идентификации объекта существует два метода доступа к объектам: ***метод вычисляемого адреса*** и ***метод хранимого адреса***. Согласно методу ***вычисляемого адреса,*** на этапе компиляции исходного текста программы создается элемент хранения объекта, причем однозначная взаимосвязь между именем объекта и адресом его элемента хранения в сегменте данных фиксируется в специальной таблице, формируемой компилятором, на все время работы программы. На этапе загрузки сегмент данных настраивается на конкретный физический адрес в оперативной памяти. Доступ через имя всегда открывает один и тот же объект. Методу вычисляемого адреса соответствует ***последовательная организация памяти***. При такой организации объекты размещаются в смежных последовательно расположенных ячейках памяти, что характерно, например, для статической памяти (см. 3.2.1). Адреса объектов вычисляются от начального адреса сегмента данных с учетом размеров их элементов хранения. Более сложные методы последовательной организации связаны с индексацией и соответственно вычислением адресов объектов (и их атрибутов) через индексы (рис. 16).

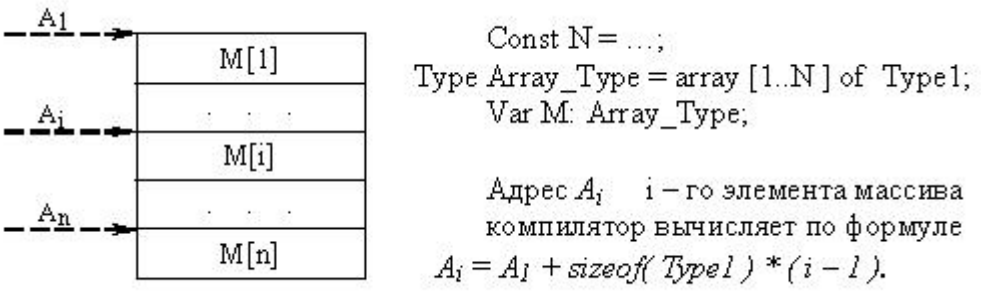


Рис. 16. Последовательная организация памяти: индексирование

Достоинством последовательной организации является простота доступа к объектам по имени, а недостатком - проблемы при модификации структур объектов. Например, нельзя изменить размер массива в процессе выполнения программы. Для того чтобы включить в массив какое-либо новое значение, требуется «раздвинуть» массив за счет копирования всех элементов в элементы с большими индексами, начиная от заданного индекса. Подобная проблема возникает, если какое-либо значение требуется из массива исключить, т.е. “сжать” массив. Необходимо заранее резервировать такое количество памяти, которое потребуется для работы со структурой, содержащей максимальное количество объектов.

Согласно ***методу хранимого адреса,*** адреса объектов не вычисляются, а хранятся в указателях на эти объекты. Для доступа к объекту сначала необходимо получить ссылку на него (т.е. значение указателя), а затем выполнить операцию раскрытия ссылки. Каждый объект имеет возможность хранить в виде ссылок связи с другими объектами, с которыми он взаимодействует в программе. Для реализации этой возможности необходимо ввести в структуру объекта специальные атрибуты, называемые полями связи или ссылочными полями для хранения связей со смежными объектами. Методу хранимого адреса соответствует ***связанная организация памяти***. Графическая иллюстрация структур связанной организации памяти использует прямоугольники для изображения элементов хранения объектов и стрелки для изображения связей (указателей) между объектами. Необходим специальный указатель, который бы определял местонахождение начального объекта связанной структуры. На рис. 17 приведен пример графической иллюстрации связанной организации структуры из трех объектов. В полях “информация объекта” находятся атрибуты данных объекта, а в полях “адрес объекта” – ссылочные поля для хранения связей. Доступ к объекту 1 открывает указатель p. Доступ к объекту 2 возможен

только через объект 1, а к объекту 3 – через объект 2 с использованием соответствующих полей связи.

Рис. 17. Графическое представление связанной организации памяти

На рис. 18 приведено упрощенное графическое представление связанной организации памяти.

Рис. 18. Упрощенное графическое представление связанной организации памяти

Достоинством связанной организации памяти является удобство модификации структур, т.к. в них соседние объекты могут располагаться в физически несмежных областях памяти. Необязательно сразу создавать структуру максимального размера. Включение / исключение объектов можно выполнять в процессе работы программы, что не потребует “раздвигать” или “сжимать” структуру за счет копирования информации. Однако “платой” за использование гибкой и эффективной связанной организации памяти являются дополнительные затраты памяти для хранения адресов соседних объектов и более сложный доступ к атрибутам объектов.

4.2. Понятие динамической структуры данных

Динамической структурой называется упорядоченное множество объектов, состав и взаимное расположение которых в процессе выполнения программы может динамически изменяться. Динамические структуры конструируются программистом с использованием связанной организации памяти и метода хранимого адреса.

Операции по модификации динамических структур:

- создание / разрушение структуры
- включение объектов в структуру / исключение объектов из структуры
- выделение подмножества объектов структуры по определенным признакам
- объединение нескольких подмножеств объектов в определенном порядке в единую структуру.

В зависимости от отношения порядка, определенного на множестве объектов, различают линейные и нелинейные структуры данных.

4.3. Линейные динамические структуры данных (списки)

Линейной динамической структурой (списком) называется множество объектов (элементов, узлов) $S=\{s_i\}$, $i=1,...,n$, на котором определены отношения предшествования / следования, причем для любого объекта s_i, $i=2,...,n-1$ существует единственный "предшественник" s_{i-1} и единственный "последователь" s_{i+1}. Объект s_1 не имеет предшественника и является первым элементом списка, объект s_n не имеет последователя и является "хвостом" списка. Ситуация n=0 определяет особое состояние: "список пуст".

Реализация динамической структуры линейного списка на связанной памяти требует включения в структуру каждого его элемента полей для связи с соседними элементами. В зависимости от того, с каким количеством соседних объектов связан каждый объект в списке, различаются односвязные, двусвязные и многосвязные списки.

4.3.1. Основные виды списков

СТЕК – структура, у которой включение / исключение элементов и доступ к элементам производятся на одном конце структуры, называемом верхушкой стека (рис. 19). Для стека характерна дисциплина обслуживания "последним пришел – первым вышел" (*LIFO – Last Input First Output*).

Рис. 19. Структура стека

ОЧЕРЕДЬ – структура, у которой включение элемента производится в хвост, а исключение элемента и доступ к элементам выполняются в начале списка (рис. 20). Для очереди характерна дисциплина обслуживания "первым пришел – первым вышел" (*FIFO – First Input First Output*).

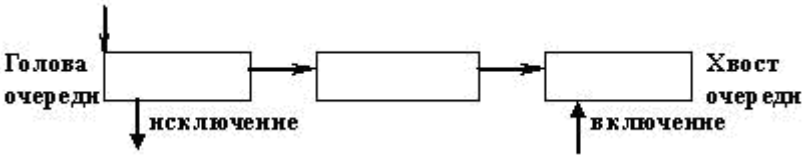


Рис. 20. Структура очереди

ДЕК (двусторонняя очередь) – операции включения / исключения элементов и доступ к элементам выполняются как в начале, так и в хвосте списка.

СПИСКИ ПРОИЗВОЛЬНОГО ВИДА – операции включения / исключения элементов выполняются в любой точке структуры, возможен доступ к произвольному элементу списка.

4.4. Односвязные (однонаправленные) списки

Каждый элемент односвязного списка содержит одно или несколько информационных полей и единственное поле связи. Элемент хранения односвязного списка описывается следующим образом:

```
Type
 PList=^List;                          { указатель на узел списка }
 List= record                          { описание узла списка }
  info: word;                          { информационное поле узла }
  link: plist;                         { поле связи узла }
 end;
Var first: PList;                      { указатель на первый узел списка }
    p: PList;                          { вспомогательный указатель }
    x: word;
```

На рис. 21 представлен пример структуры односвязного списка.

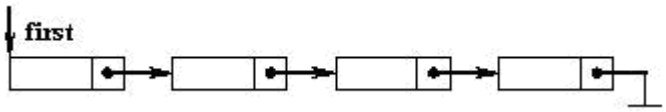


Рис. 21. Структура односвязного списка

На первый элемент списка указывает указатель ***first***. Если список пуст, то ***first = nil***. Если список не пуст, то к атрибуту любого его элемента (например, первого) можно получить доступ через указатель.

```
x:=first ^.info;      { значение информационного поля первого элемента }
p:=first ^.link;      { значение поля связи первого элемента – адрес второго элемента }
```

К атрибутам любого элемента списка, кроме первого, возможен дистанционный доступ.

```
x:=first ^.link ^.info;          { значение информационного поля второго элемента }
```

Дистанционный доступ ко второму узлу списка эквивалентен следующей последовательности операторов:

```
p:=first ^.link;      { установка вспомогательного указателя на второй узел списка }
x:=p^.info;           { значение информационного поля второго узла списка }
```

4.4.1. Включение узла в начало списка

Одна из самых простых операций по модификации списка - включение нового узла в его начало (рис. 22): элемент хранения типа *List* размещается в памяти и указатель на него присваивается некоторой вспомогательной переменной *p*, затем устанавливается связь между вставленным узлом и списком, после чего указатель на первый элемент списка получает новое значение.

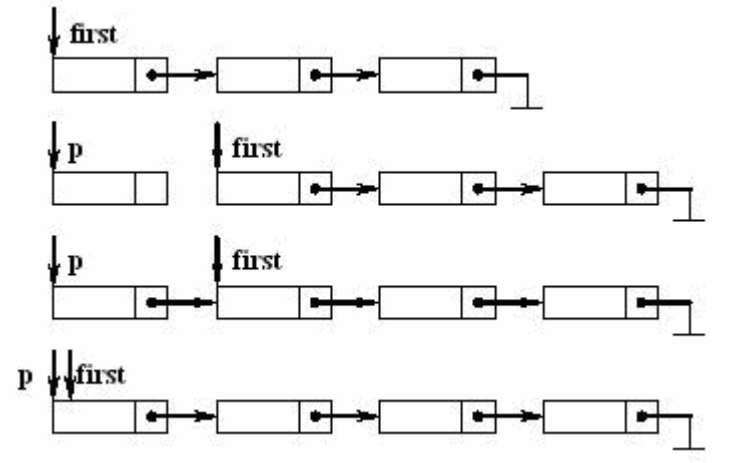


Рис. 22. Включение узла в начало списка

```
Procedure Ins_First( var first: PList );
var p: PList;
begin
    new( p );
    readln( p^.info );    { first - указатель на первый узел списка }
{ создание узла списка }
{ заполнение информационного поля узла }
    p^.link:=first;          { установка связи между вставленным узлом и списком }
    first:=p;                { новое значение указателя на первый узел }
end;
```

4.4.2. Создание списка из N узлов: добавление узлов в начало списка

Используя операцию включения элемента в начало списка, можно сформировать список из n элементов: начиная с пустого списка, следует n раз выделить память для узлов списка и последовательно добавить узлы в начало списка. Эти операции можно реализовать с помощью любого итерационного цикла. Порядок следования узлов получается обратным, т.е. первым в списке оказывается элемент, который был добавлен последним.

```
Procedure Create1( var first: PList;n: byte);
var p: PList; i: byte;
begin
                                        first:=nil;    { first – указатель на первый узел списка,
                                                         n – количество узлов в списке }

{ создание пустого списка }
  for i:=1 to n do begin
    new( p );                           { создание узла списка }
    readln(p^.info);                    { заполнение информационного поля узла }
    p^.link:=first;          { установка связи  между вставленным узлом  и списком }
    first:=p;                           { новое значение указателя на первый узел }
  end; end;
```

Создание первого узла списка и включение его в пустой список (“перед” несуществующим узлом) выполняется точно так же, как включение в непустой список любого другого узла (см. рис. 23).

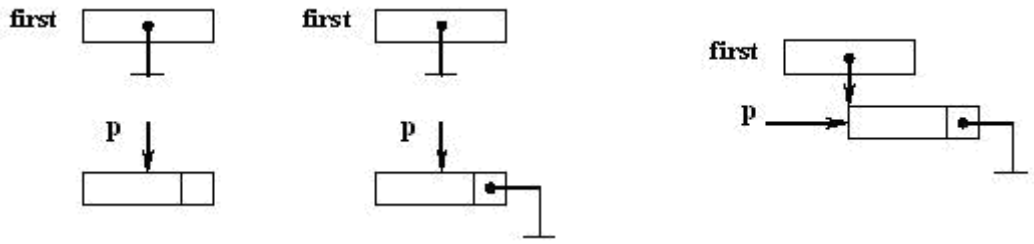


Рис. 23. Включение узла в пустой список

4.4.3. Создание списка из N узлов: добавление узлов в конец списка

Первый узел создается отдельно (т.к. включить узел «за» несуществующим узлом невозможно), а остальные (n-1) узлов создаются и включаются в хвост списка одинаковым образом. При этом удобно использовать вспомогательный указатель на последний добавленный узел. Значение этого указателя изменяется в процессе создания списка, значение указателя на первый узел списка не изменяется после создания первого узла. Порядок следования узлов в списке получается прямым, т.к. в начале списка находится тот узел, который был включен в список первым (рис. 24).

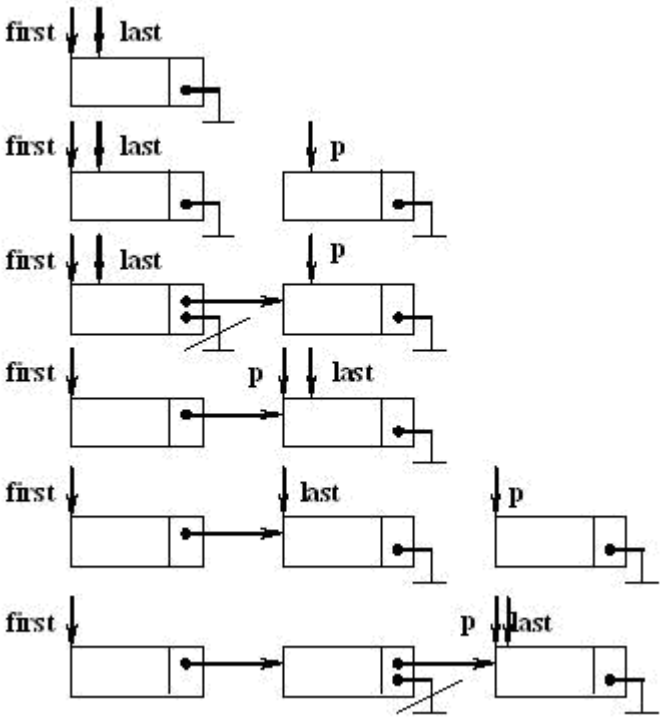


Рис. 24. Включение узла в конец списка

```
Procedure Create2( var first: PList;n: byte);

var p, last: PList; i: byte;              { first – указатель на первый узел списка,
begin                                       n – количество узлов в списке }

  if n=0 then first:=nil                          { создание пустого списка }
  else begin
      new( first);                                 { создание первого узла списка }
      readln(first^.info);          { заполнение информационного поля первого узла }
      first^.info:=nil;                              { первый узел списка }
      last:=first;                  { является в данный момент единственным }
      for i:=2 to n do begin                { цикл включения в список (n-1) элемента }
        new( p );                           { создание узла списка }
        readln(p^.info);                    { заполнение информационного поля узла }
        p^.link:=nil;             { вставленный узел является последним в списке }
        last^.link:=p;        { установка связи между предпоследним узлом и вставленным }
        last:=p;                            { новое значение указателя на последний узел }
      end;
  end;
end;
```

4.4.4. Исключение узла из начала списка

Для того чтобы исключить из списка первый элемент, необходимо установить на него вспомогательный указатель, присвоить указателю на начало списка адрес следующего элемента списка, после чего область памяти, занятую первым элементом списка, вернуть в кучу (рис. 25).

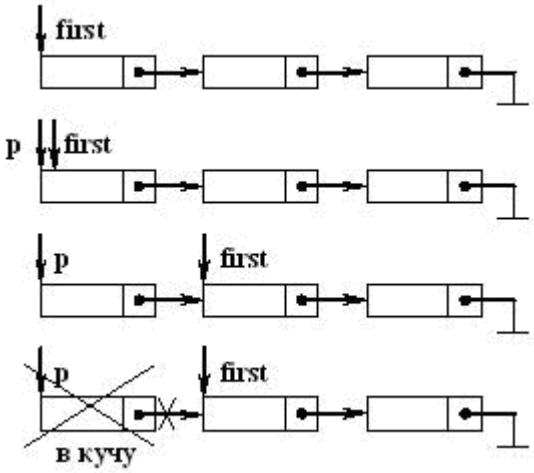


Рис. 25. Исключение узла из начала списка

```
Procedure Del_First( var first: PList );          { first – указатель на первый узел списка }
var p: PList;
begin
   if ( first <> nil ) then begin                 { список не пуст? }
      p:=first;              { установка вспомогательного указателя на первый узел списка }
      first:=p^.link;        { установка указателя first на второй узел списка }
      dispose( p );          { элемент хранения первого узла списка вернуть в кучу }
   end; end;
```

4.4.5. Переустановка указателя

Доступ к объектам динамической структуры может быть получен с помощью единственного вспомогательного указателя, который будет последовательно изменяться, всякий раз принимая значение адреса соседнего объекта, в направлении стрелки, изображающей связь. Адрес соседнего объекта извлекается из поля связи того элемента списка, на который в текущий момент ссылается указатель, затем полученный адрес присваивается этому указателю, который теперь открывает доступ к соседнему элементу списка. Такая операция называется перестановкой указателя (рис. 26). Операция переустановки указателя используется, если необходимо единообразно обработать все или несколько следующих подряд элементов списка (для этого следует организовать цикл, включающий операции обработки элемента и перестановки указателя). В этом случае последовательность операций переустановки указателя обеспечивает проход по списку.

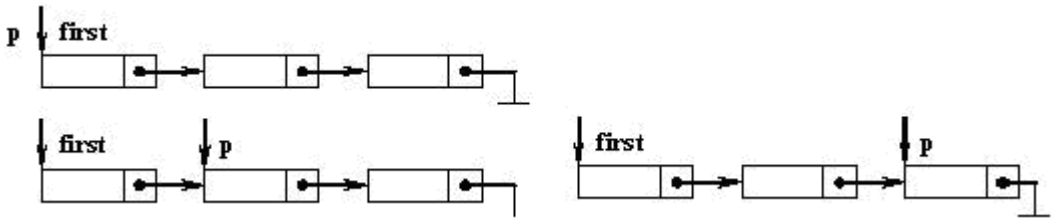


Рис. 26. Переустановка указателя

```
. . .
if first<> nil then begin                          { список не пуст? }
  p:=first;          { установка вспомогательного указателя на первый узел списка }
  p:=p^.link;        { переустановка вспомогательного указателя на второй узел списка }
  writeln( p^.info );          { обработка информационного поля второго узла списка }
end;  . . .
```

4.4.6. Поиск узла в списке по заданному условию

Условием поиска элемента в списке может быть:

- значение информационного поля элемента,
- порядковый номер элемента в списке, начиная от первого узла,
- адрес элемента списка, который хранится в некотором указателе.

Поиск элемента в списке по заданному условию обычно организуется в цикле, включающем операции проверки выполнения условия для текущего элемента, на который ссылается указатель, и переустановки указателя на соседний элемент списка (т.е. поиск осуществляется в процессе прохода по списку). Проверка условия связана с вычислением булевских выражений в условных операторах или операторах цикла, использующих доступ к атрибутам объектов через указатели (см. 2.2.3). Например,

if (p<> nil) and (p^ .info = значение) then < тело условного оператора >
или
while (p<> nil) and (p^.info<> значение) do < тело цикла >.

Поиск заканчивается либо при обнаружении элемента списка, соответствующего заданному условию (результатом поиска в этом случае является значение указателя, установленного на искомый узел), либо при достижении конца списка, если элемент, соответствующий условию поиска, не обнаружен (результатом поиска в этом случае является *NIL*).

4.4.7. Включение нового узла в список за тем узлом, на который предварительно установлен указатель

Для того чтобы включить в список новый элемент за тем элементом, на который предварительно установлен указатель, необходимо разместить элемент хранения в области динамической памяти и выполнить последовательность операций, которая иллюстрируется рис. 27. После выполнения операции вставки значение указателя на первый элемент списка не изменяется. Значение указателя на тот элемент, за которым выполнена вставка, также не изменяется.

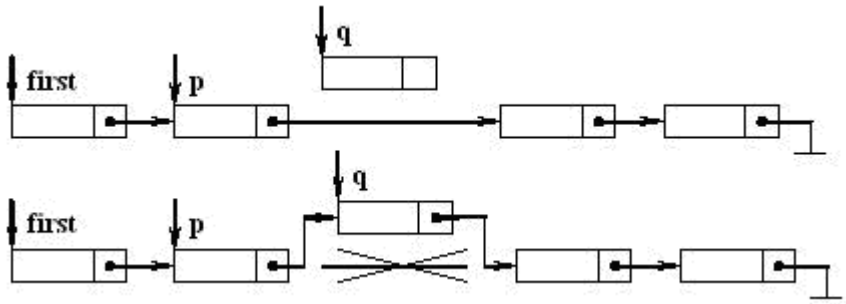


Рис. 27. Включение узла в список за тем узлом, на который предварительно установлен указатель

```
Procedure Ins( first, p: PList );                    { first – указатель на первый узел списка,
                                                       p – предварительно установленный указатель}
var q: PList;                                        { q – указатель на вставляемый узел }
begin
  if  ( first <> nil ) and  ( p <> nil ) then        { указатель p действительно установлен? }
  begin
     new( q  );                                      { создание нового узла списка для вставки }
     readln( q^.info );              { заполнение информационного поля нового узла }
     q^.link:=p^.link;               { заполнение поля связи нового узла }
     p^.link:=q;          { изменение поля связи того узла, за которым вставлен новый узел }
  end;
end;
```

4.4.8. Исключение из списка узла за тем узлом, на который предварительно установлен указатель

Исключение из списка элемента за тем элементом, на который предварительно установлен указатель, выполняется с использованием вспомогательного указателя на исключаемый элемент. Последовательность операций исключения иллюстрируется рис. 28. После выполнения операции исключения значение указателя на первый элемент списка не изменяется. Значение указателя на тот элемент, за которым выполнено исключение, также не изменяется.

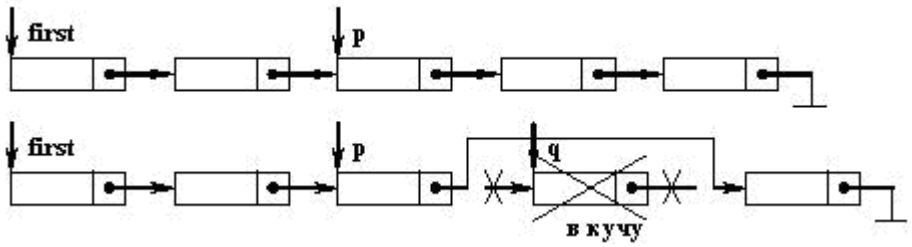


Рис. 28. Исключение узла за тем узлом, на который предварительно установлен указатель

```
Procedure Del( first, p: PList );                    { first – указатель на первый узел списка,
                                                       p – предварительно установленный указатель}
var q: PList;                                        { q – указатель на исключаемый узел }
begin
  if  ( first <> nil ) and  ( p <> nil ) and         { указатель p действительно установлен?}
     ( p^.link <> nil ) then                         { указатель p не указывает на последний узел в списке? }
  begin
     q:=p^.link;                                     { установить указатель q на узел, следующий за элементом p^ }
     p^.link:=q^.link;                               { изменить поле связи узла, за которым выполняется
исключение }
     dispose ( q );                                  { элемент хранения исключаемого узла списка вернуть в кучу }
  end;
end;
```

4.4.9. Исключение из списка узла, на который предварительно установлен указатель

Если требуется включение / исключение перед указанным элементом или исключение самого указанного элемента, то необходима предварительная установка указателя на элемент, предшествующий заданному, т.к. в этом случае с целью сохранения структуры списка у узла-предшественника должна быть изменена связь (см. рис. 29). Для установки указателя на узел, предшествующий данному узлу, необходим проход от начала списка до узла-предшественника, т.к. в элементе односвязного линейного списка нет ссылки на предыдущий узел. После исключения узла из списка и возвращения его элемента хранения в кучу, доступ к этому узлу по предварительно установленному указателю становится невозможным, поэтому данному указателю следует присвоить значение *NIL*.

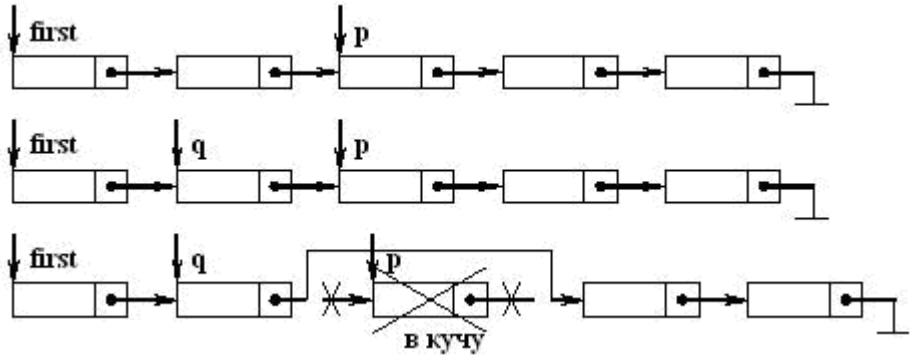


Рис. 29. Исключение узла, на который предварительно установлен указатель

??? *Реализуйте алгоритм исключения узла перед узлом, на который предварительно установлен указатель, самостоятельно.*

4.4.10. Разрушение списка

Операция разрушения списка реализуется в процессе прохода по списку так, что элементы хранения всех узлов списка, начиная с первого, последовательно исключаются из списка и возвращаются в кучу. В результате выполнения операции разрушения список становится пустым, т.е. значение указателя на первый узел равно *NIL*. Для разрушения списка недостаточно просто присвоить *NIL* указателю на его первый узел, т.к. в этом случае память, занятая элементами хранения узлов списка, в действительности не освобождается и в кучу не возвращается.

```
Procedure Destroy( var first: PList );                    { разрушение списка }
Var p: PList;
begin
  while (first <> nil) do begin
    p:=first;              { установить вспомогательный указатель p на первый узел списка}
    first:=first^.link;                    { переустановить указатель  на первый узел списка}
    dispose(p)       { элемент хранения исключаемого узла списка вернуть в кучу }
  end;
end;
```

4.4.11. Программный модуль, реализующий операции создания, обработки, просмотра содержимого списка

```
Uses Crt;
Type PList = ^ List;                    { описание элемента хранения узла списка и }
  List = record                                    { указателя на узел списка }
    info: word; link: plist;
  end;

var f: PList; cod,n: byte; sum: word;              { глобальные переменные }

Procedure Create1(var first: PList;
                          n: byte );
var p: PList; i: byte;
begin
                                        first:=nil;    { first – указатель на первый узел списка }
                                                       { n – количество узлов в списке }



{ создание пустого списка }
   for i:=1 to n do begin
      new( p );                                    { создание узла списка }
      write( 'Значение инф. поля ', i, '-го элемента списка =  ' );
      readln( p^.info );                           { заполнение информационного поля узла }
      p^.link:=first;          { установка связи  между вставленным узлом  и списком }
      first:=p;                                    { новое значение указателя на первый узел }
   end;
end;

Procedure Print( first: PList );          { просмотр информац. полей узлов списка }
var p: PList; i: byte;
begin i:=0;  p:=first;
   while (p <> nil ) do begin
       inc( i );
       writeln( 'Информационное поле ', i, '-го элемента списка =  ', p^.info );
       p:=p^.link;
   end;
end;

Procedure Work( first: PList; var s: word );        { суммирование значений информ. }
var p: PList;                                       { полей узлов списка }
begin s:=0;  p:=first;
   while (p <> nil ) do begin
       s:=s+p^.info; p:=p^.link;
   end;
end;

Procedure Destroy( var first: PList );              { разрушение списка }
Var p: PList;
begin
  while (first <> nil) do begin
```

```
      p:=first; first:=first^.link; dispose(p)
  end;
end;

Procedure Message;                                  { вспомогательная процедура }
begin
   writeln( 'Список пуст' ); write( 'Нажмите любую клавишу' ); readkey
end;

begin
   f:=nil;                                          { первоначально список пуст }
   repeat Clrscr;
    writeln('1-Создание 2-Просмотр 3–Обработка 4–Разрушение 5-Выход');
    write( 'Код действия = ' ); readln( cod  );
    case cod of
    1: begin                                        { создание списка }
         write( 'Количество узлов в списке = ' ); readln( n );
         Create( f,n ); write( 'Нажмите любую клавишу' ); readkey
       end;
    2:                                              { просмотр списка }
         if f=nil then Message
         else  begin
           Print( f  ); write( 'Нажмите любую клавишу' ); readkey
         end;
    3:                                              { обработка списка }
         if f=nil then Message
         else  begin
           Work( f,sum ); writeln( 'Сумма значений инф. полей =  ', sum );
           write( 'Нажмите любую клавишу' ); readkey
         end;
    4:                                              { разрушение списка }
         if f=nil then Message
         else  begin
           Destroy( f ); writeln( 'Список разрушен' );
           write( 'Нажмите любую клавишу' ); readkey
         end;
    5: Destroy( f  )                                { выход }
    end;
   until ( cod = 5 ); Clrscr
end.
```

4.5. Односвязные циклические списки

Циклически связанный список (сокращенно – циклический список) обладает той особенностью, что поле связи его последнего элемента не содержит значения *NIL* а указывает на первый узел списка. В целях удобства обработки в структуру циклического списка включают специальный узел с особым содержанием информационного поля (на рис. 30 это поле заштриховано), называемый ***головой списка или "сторожем"***. Голова списка является дополнительным элементом. Назначение этого элемента состоит в том, чтобы отметить точку входа в циклический список, а также упростить включение узлов в начало списка и исключение узлов из начала списка. На рис. 30 показана структура односвязного циклического списка с головным элементом.

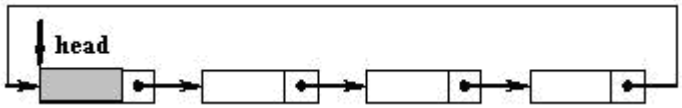


Рис. 30. Структура односвязного циклического списка с головным узлом

Элемент хранения односвязного циклического списка описывается следующим образом:

```
Type
PList=^List;                    { указатель на узел списка }
List= record                    { описание узла списка }
info: word;                     { информационное поле узла }
link: pList;                    { поле связи узла
end;
Var head: PList;                    { указатель на «голову» списка }
    p: Plist;                       { вспомогательный указатель }
```

Выполнение условия ***head = nil*** означает, что односвязный циклический список не существует. Выполнение условия ***head ^. link = head*** означает, что односвязный циклический список существует, но является пустым, т.е. содержит только головной элемент. Пустой циклический список с головным элементом представляется структурой элементарного кольца (рис. 31).

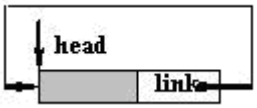


Рис. 31. Структура элементарного односвязного кольца

Односвязные циклические списки можно использовать для реализации линейных структур, таких как очереди, стеки, списки произвольного вида. При создании очереди новый узел включается в "хвост" списка, т.е. "перед" головным элементом (рис. 32). При создании стека новый узел включается в начало списка, т.е. непосредственно "за" головным элементом (рис. 33).

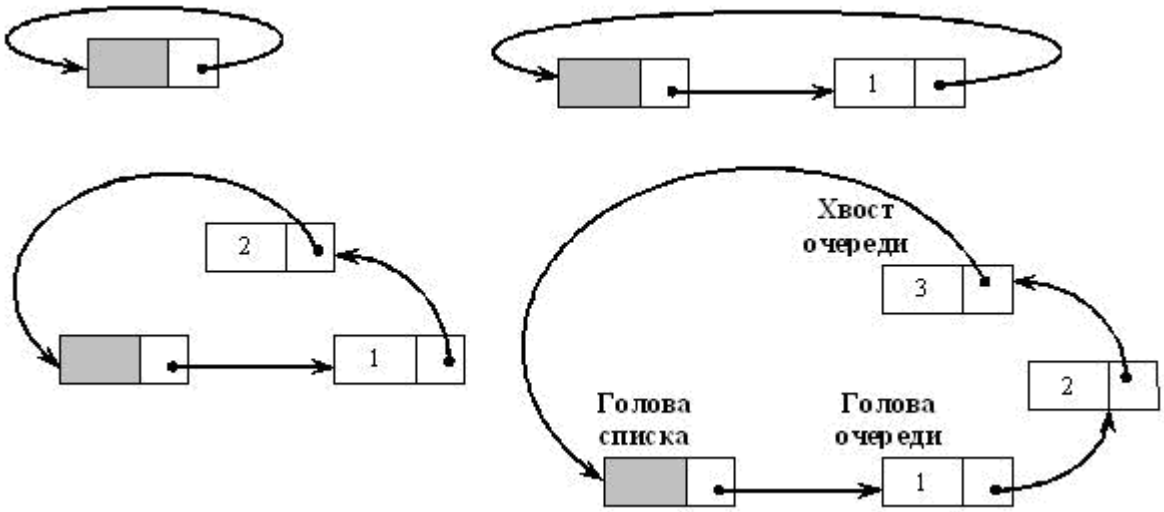


Рис. 32. Создание очереди на структуре односвязного циклического списка

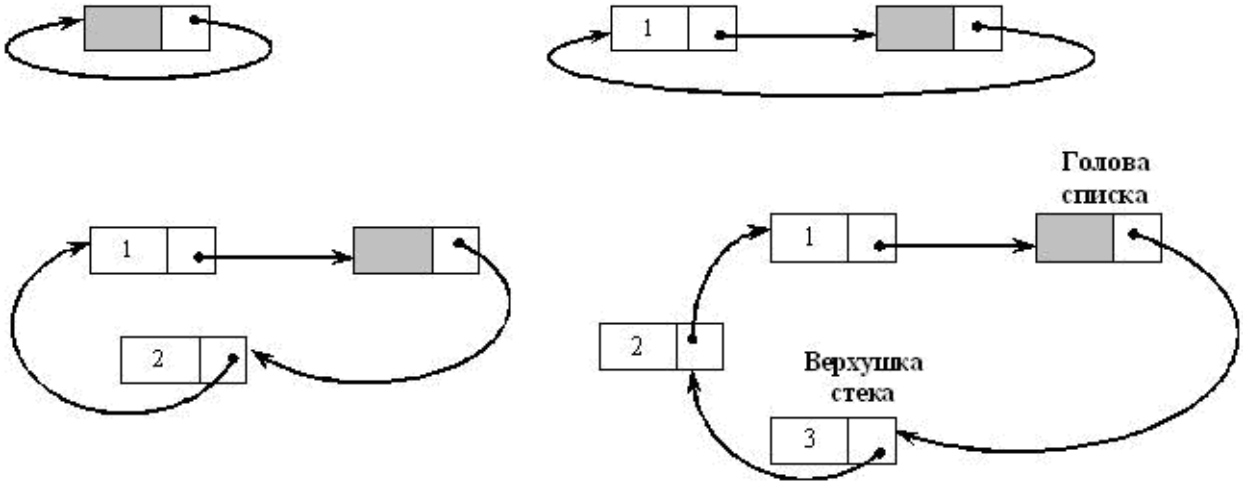


Рис. 33. Создание стека на структуре односвязного циклического списка

Ниже приведена процедура создания односвязного циклического списка из *n* узлов.

```
Procedure Create_Cikl( var head: PList;
                       n: byte );
var p: PList; i: byte;
                                          begin        { head – указатель на голову списка }
                                                       { n – количество узлов в списке }

  new( head ); head^.link:=head;                  { создание элементарного кольца }
  for i:=1 to n do begin                  { создание циклического списка из n узлов }
    new( p );                             { создание узла списка }
    Readln(p^.info);                      { заполнение информационного поля узла }
    P^.link:=head^.link;        { установка связи  между вставленным узлом  и списком}
    head^.link:=p;                        { обновление поля связи головного узла  }
  end;
end;
```

Т.к. в циклическом списке каждый элемент, в том числе первый и последний, имеют предшественника и последователя (“перед” первым элементом и “за” последним

элементом находится голова), все n элементов списка создаются и включаются в список одинаково (см. процедуру Create_Cikl).

??? *Каким образом включаются узлы в список при выполнении процедуры Create_Cikl: "за" головным узлом или "перед" ним?*

Исключение первого или последнего узла циклического списка также не имеет никаких особенностей за счет использования головного элемента. Операции включения / исключения узлов в список произвольного вида, реализованный в виде циклического списка, выполняются так же, как в нециклическом списке.

В циклическом списке можно получить доступ к любому элементу списка, продвигаясь от произвольного элемента по кольцу. Поиск элемента по заданному условию в односвязном циклическом списке организуется в цикле, включающем операции проверки выполнения условия для текущего элемента, на который ссылается указатель, и перестановки указателя на соседний элемент. В процессе поиска используется вспомогательный указатель, который первоначально следует установить на узел, следующий за головным. Например,

```
if head <> nil  then begin                          { список существует? }
  p:=head^.link;                                    { установить вспомогательный указатель }
  while ( p<> head ) and  ( p^.info<> значение ) do < тело цикла >          { поиск }
end;
    . . .
```

Поиск заканчивается либо при обнаружении элемента списка, соответствующего заданному условию (результатом поиска в этом случае является значение указателя, установленного на искомый узел), либо при возвращении к головному узлу после прохождения всего кольца, если элемент, соответствующий условию поиска, не обнаружен (результатом поиска в этом случае является адрес головного узла). Заметим, что в случае пустого списка цикл не выполнится ни разу.

Указатель на головной элемент циклического списка не изменяет своего значения при выполнении любых операций над элементами списка, за исключением разрушения списка. При разрушении следует освободить области памяти, занятые элементами хранения узлов списка и головного элемента, после чего список прекращает свое существование (указатель *head* следует установить равным *NIL*).

4.6. Двусвязные (двунаправленные) списки

Для достижения большей гибкости в работе с линейными списками можно включить в каждый узел два атрибута связи – указатели на следующий узел (т.е. на “правого соседа”) и на предыдущий узел (т.е. на “левого соседа”). Списки с двумя связями занимают больше памяти, чем односвязные, однако в процессе прохода по списку они дают возможность продвигаться как “вперед”, так и “назад”, что повышает эффективность реализации алгоритмов обработки списков.

Элемент хранения узла двусвязного списка описывается следующим образом:

```
Type
PDlist=^ Dlist;          { указатель на узел списка }
Dlist= record;           { описание узла списка }
info: word;              { информационное поле узла }
next: pDlist;            { поле связи со следующим узлом }
prev: pDlist;            { поле связи с предыдущим узлом }

end;
```

В ***двусвязном нециклическом списке*** первый узел не имеет предшественника (поле связи *prev* первого узла равно *NIL*), а последний узел не имеет последователя (поле связи *next* последнего узла равно *NIL*). На первый узел двусвязного списка, имеющего нециклическую структуру, указывает указатель *first: pDlist*. Выполнение условия *first = nil* означает, что двусвязный нециклический список пуст. Структура двусвязного нециклического списка представлена на рис. 34.

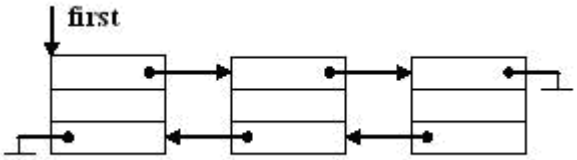


Рис. 34. Структура двусвязного нециклического списка

Нециклическая структура двусвязного списка порождает те же проблемы при включении / исключении первого и последнего узлов, что и структура односвязного нециклического списка.

На практике более широко применяются двусвязные циклические списки. В ***двусвязном циклическом списке*** поле связи *next* его последнего элемента не содержит значения *NIL*, а указывает на первый узел списка и поле связи *prev* его первого элемента также не содержит значения *NIL*, а указывает на последний узел списка. В целях удобства обработки в структуру двусвязного циклического списка включают специальный дополнительный узел с особым содержанием информационного поля (на рис. 35 это поле заштриховано), называемый ***“головой“ списка или “сторожем“***. Заметим, что “левым соседом” первого узла двусвязного циклического списка является его последний узел, а “правым соседом” его последнего узла является первый узел, т.к. информационное поле головного узла имеет особое содержание (а нередко и вовсе не используется). Так что функция “сторожа” в двусвязном циклическом списке оказывается чисто технологической и полностью аналогичной функции “сторожа” в односвязном циклическом списке. Структура двусвязного циклического списка приведена на рис. 35.

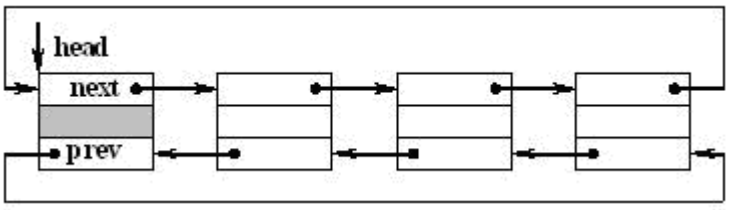


Рис. 35. Структура двусвязного циклического списка

```
var head: pDlist;                    { указатель на голову списка }
    p: pDlist;                       { вспомогательный указатель }
```

Выполнение условия ***head = nil*** означает, что двусвязный циклический список не существует. Выполнение условия ***head ^. next = head ^. prev = head*** означает, что двусвязный циклический список существует, но является пустым, т.е. содержит только головной элемент. Пустой циклический список с головным элементом представляется структурой элементарного кольца (рис. 36).

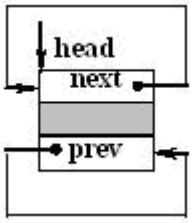


Рис. 36. Структура элементарного двусвязного кольца

Для любого узла двусвязного циклического списка, на который установлен указатель *p* (в том числе, и для головного), справедливо выражение:

p^ .next^ .prev = p^ .prev^ .next = p;

4.6.1. Включение в список нового узла справа или слева от узла, на который предварительно установлен указатель

Чтобы включить в список новый узел, необходимо выделить память для размещения элемента хранения этого узла и выполнить четыре операции установки связей (рис. 37). Ниже приведена процедура включения нового узла “справа” от узла, на который предварительно установлен указатель.

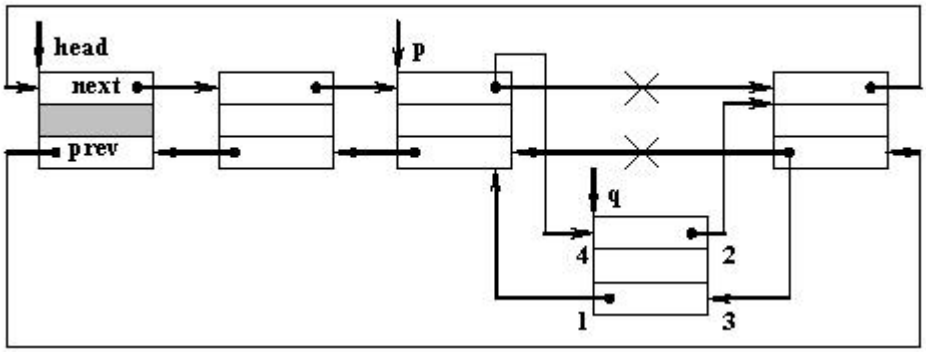


Рис. 37. Включение узла “справа” от узла, на который предварительно установлен указатель

```
Procedure Ins_Right( head,p: pDlist );          { head – указатель на “голову” списка }
                                                { p – предварительно установленный указатель}
var q: pDlist;                                  { q – указатель на новый узел }
begin
   if ( head <> nil ) and ( p <> nil ) then     { указатель p действительно установлен? }
   begin
      new( q );                                 { создание нового узла }
      readln( q^ . info );                      { заполнение информационного поля нового узла }
      q^. prev:=p;                              { 1 - установка связи нового узла с предыдущим }
      q^. next:=p^ . next;                      { 2 - установка связи нового узла со следуюшим }
      p^. next^. prev:=q;                       { 3 - установка связи следующего узла с новым }
      p^. next:=q;                              { 4  - установка связи предыдущего узла с новым }
   end;
end;
```

Процедура включения нового узла “слева” от узла, на который предварительно установлен указатель, выполняется аналогично (рис. 38).

В отличие от односвязного списка, никакого прохода до узла, предшествующего узлу *p*^ не требуется, достаточно обратиться к соответствующему атрибуту связи узла *p*^ .

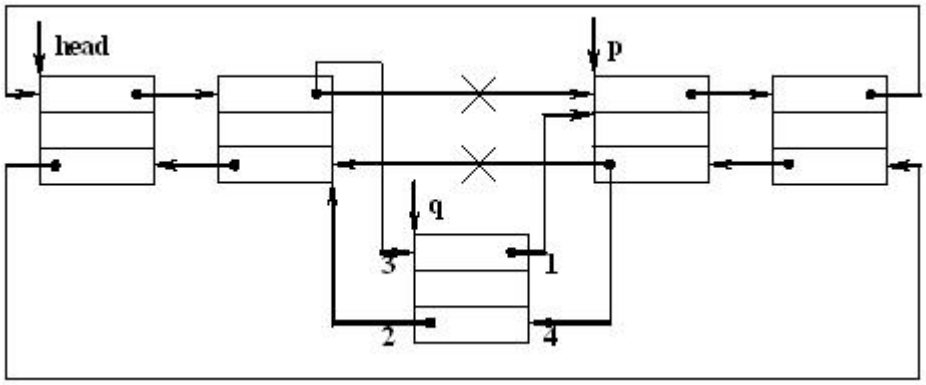


Рис. 38. Включение узла “слева” от узла, на который предварительно установлен указатель

Ниже приведена процедура создания двусвязного циклического списка из n узлов.

```
Procedure Create_Double( var head: PDlist;        { head – указатель на голову списка }
                         n: byte );               { n – количество узлов в списке }
var p: PDlist; i: byte;
begin
   new( head ); head^.next:=head;
   head^.prev:=head;                              { создание элементарного кольца }

   for i:=1 to n do begin                         { создание циклического списка из n узлов }
      new( p );                                   { создание узла списка }
      readln(p^.info);                            { заполнение информационного поля узла }
      p^.next:=head;                              { операции }
      p^.prev:=head^.prev;                        { установки связей }
      head^.prev^.next:=p;                        { нового узла }
      head^.prev:=p;                              { и списка }
   end;
end;
```

??? *Каким образом включаются узлы в список при выполнении процедуры Create_ Double: “за” головным узлом или “перед” ним?*

4.6.2. Исключение из списка узла, на который предварительно установлен указатель

Исключение узла, на который предварительно установлен указатель, не требует поиска предыдущего узла (рис. 39). После исключения узла из списка и возвращения его элемента хранения в кучу, доступ к этому узлу по предварительно установленному указателю более невозможен, поэтому данному указателю следует присвоить значение *NIL*.

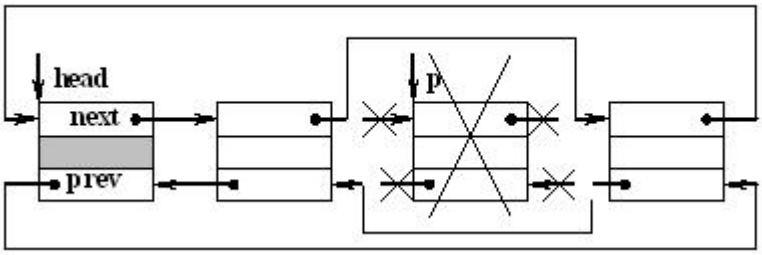


Рис. 39. Исключение узла, на который предварительно установлен указатель

```
Procedure Del_Double( head: PDlist;              { head – указатель на "голову" списка }
                      var p: PDlist);            { p – указатель на исключаемый узел }
begin
  if  ( head <> nil ) and  (head^ .next <> head )          { список не пуст  и указатель p }
      and ( head^.prev <> head ) and ( p <> nil ) then          { установлен? }
  begin
     p^.prev^.next:=p^.next;                   { изменить поле связи предыдущего узла }
     p^.next^.prev:=p^.prev;                   { изменить поле связи следующего узла }
     dispose( p ); p:=nil          { элемент хранения исключаемого узла вернуть в кучу }
  end;
end;
```

Операция поиска узла в двусвязном циклическом списке и операция разрушения выполняются так же, как в односвязном циклическом списке, только проход возможен в любом из двух направлений: с использованием атрибута связи *next* (т.е. "вперед") или с использованием атрибута связи *prev* (т.е. "назад").

Двусвязные циклические списки, как и односвязные, можно использовать для реализации разнообразных линейных структур.

4.7. Ортогональные списки (мультисписки)

Ортогональный список (или мультисписок) – это структура, каждый элемент которой входит более чем в один список одновременно и имеет соответствующее числу списков количество полей связи. Реализация каждого из списков может быть выполнена в виде одно- или двусвязного нециклического или циклического списка. Технология обработки мультисписков ничем не отличается от обработки обычных списков, но, так как мультисписок одновременно содержит несколько списков, каждую операцию следует выполнить отдельно для каждого списка.

Мультисписок позволяет на множестве одних и тех же атрибутов, содержащих информацию, организовать различные списки, упорядоченные по различным признакам. Рассмотрим список студентов, каждый узел которого содержит следующие информационные поля: фамилия_ имя_отчество, средний балл, дата рождения, адрес, номер зачетки и т.п. Пусть необходимо упорядочить список студентов по двум признакам: в алфавитном порядке по фамилии и в соответствии со средним баллом. Этого можно достичь, если построить два отдельных списка, но элементы хранения информационных полей в этом случае продублируются, что приведет к неэффективному использованию оперативной памяти, особенно, если количество информационных полей велико. Более рациональным решением является использование мультисписка, в состав которого входят два списка, каждый из которых организован, например, в виде двусвязного циклического списка. По алфавиту элементы списка упорядочены с помощью атрибутов связи *fnext* и *fprev*, а по среднему баллу те же самые элементы упорядочены с помощью атрибутов связи *mnext* и *mprev*. Для удобства обработки мультисписок содержит головной элемент, на который установлен указатель. Структура мультисписка студентов представлена на рис. 40.

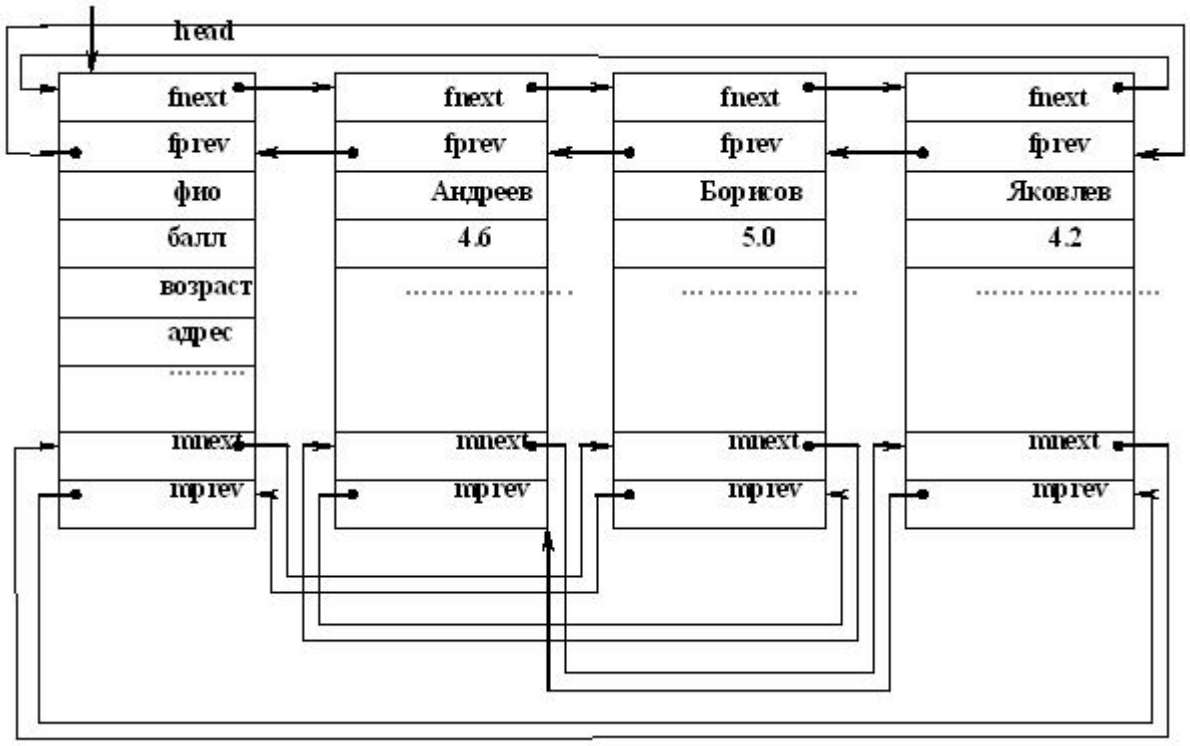


Рис. 40. Структура мультисписка студентов

Описание элемента хранения мультисписка студентов:

```
Type
   Str30 = String[30];          { тип для описания фамилии_имени_отчества }
   PMulty_List: ^ Multy_List;                    { тип – указатель на узел мультисписка }
```

```
  Multy_List = record              { описание элемента хранения узла мультисписка}
      fam: Str30;                          { фамилия_имя_отчество }
      mark: real;                          { средний балл }
      fnext, fprev: PMulty;                { атрибуты связи в списке по фамилии }
      mnext, mprev: PMulty;          { атрибуты связи в списке по среднему баллу }
  end;
Var head: PMulty;                          { указатель на "голову" мультисписка }
```

Пример обработки мультисписка – процедуры, распечатывающие содержимое узлов в виде списка, упорядоченного по алфавиту, и в виде списка, упорядоченного по среднему баллу.

```
Procedure Print_Fam( head: PMulty );       { распечатка мультисписка, }
Var p: PMulty;                             { упорядоченного по алфавиту }
begin
  if  ( head <> nil ) then begin           { список существует? }
     p:=head^.fnext;                       { установить вспомогательный указатель }
     while ( p <> head ) do begin          { весь список пройден? }
        writeln( p^.fam, p^.mark );        { распечатать информационные поля }
        p:=p^.fnext                        { перейти к следующему узлу }
     end;
  end;
end;
```

```
Procedure Print_Ball( head: PMulty );      { распечатка мультисписка, }
Var p: PMulty;                             { упорядоченного по среднему баллу }
begin
  if  ( head <> nil ) then begin           { список существует? }
     p:=head^.mnext;                       { установить вспомогательный указатель }
     while ( p <> head ) do begin          { весь список пройден? }
        writeln( p^.fam, p^.mark );        { распечатать информационные поля }
        p:=p^.mnext                        { перейти к следующему узлу }
     end;
  end;
end;
```

Часто в виде ортогональных списков представляются матрицы очень большой размерности, в которых большинство элементов равны нулю (такие матрицы называются разреженными). Пример разреженной матрицы

$$\begin{bmatrix} 3 & 0 & 5 \\ 0 & 0 & 30 \\ 2 & 10 & 0 \end{bmatrix}$$

Мультисписки обеспечивают эффективное хранение таких структур в памяти, т.к. хранятся только те элементы, которые отличны от нуля (рис. 41). Каждый элемент входит в два списка – в список-строку и в список-столбец. Вся матрица представляется (m + n) списками, где m и n соответственно число строк и число столбцов. Каждый элемент мультисписка хранит значение элемента матрицы, номер строки, номер столбца и две ссылки – на следующий элемент в строке и на следующий элемент в столбце (если используются односвязные списки). Указатели на первые элементы этих списков хранятся в двух массивах (или в списках).

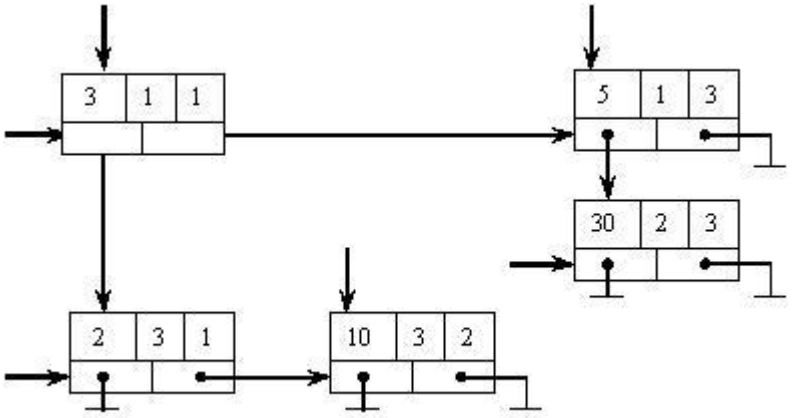


Рис. 41. Разреженная матрица, представленная в виде структуры мультисписка

Описание элемента хранения разреженной матрицы и процедура, распечатывающая значения элементов матрицы по строкам, приведена ниже.

```
Const
   Nrow = 10;                                  { количество строк }
   Ncol = 20;                                  { количество столбцов }
Type
   PMatrix: ^ Matrix;                          { тип – указатель на узел мультисписка }
   Matrix = record                  { описание элемента хранения узла мультисписка}
       val: word;                              { значение элемента }
       row, col: word;                         { номер строки, номер столбца }
       lrow, lcol: PMatrix;             { атрибуты связи в списках по строке и по столбцу }
   end;
   Prow=array[1..Nrow] of  PMatrix;             { тип – массив указателей на первые узлы
списков строк }
   Pcol=array[1..Ncol] of  PMatrix;             { тип – массив указателей на первые узлы
списков столбцов }
Var
  r: Prow;                  { массив указателей на первые узлы списков строк }
  c: Pcol;                { массив указателей на первые узлы списков столбцов }

Procedure Print_Matrix( var fr: Prow );          { fr – массив указателей на списки строк }
Var p: PMatrix; i: byte;          { p – вспомогательный указатель для прохода по строке }
begin
  for i:=1 to Nrow do begin                     { просмотр строк }
    p:=fr[i];       { установка вспогательного указателя на первый элемент списка строки }
    while ( p<> nil ) do begin                  { список строки не пуст? }
        writeln ('Matrix[' , p^.row, ',' , p^.col , ']=' , p^.val );          { вывод  значения}
        p:=p^.lrow;                          { переход к следующему элементу в строке }
    end;
  end;        { массив указателей на списки строк передается по ссылке, чтобы избежать }
end;          { копирования  массива в стек при передаче параметров процедуры }
```

4.8. Разнородные списки

По составу элементов списки подразделяются на однородные и разнородные. Все виды списков, рассматриваемые до сих пор, являются ***однородными***, т.к. они состоят из элементов одного и того же типа. Если список включает элементы различных типов, то он называется ***разнородным***. Особенности обработки разнородных списков состоят в том, что в каждый элемент списка вводится дополнительный атрибут, позволяющий определить тип элемента списка и соответствующим образом интерпретировать содержимое его информационных полей (см. 5.2.1). Примером разнородного списка является список, содержащий информацию о геометрических фигурах (окружность, прямоугольник, треугольник), составляющих некоторый проект (рис. 42).

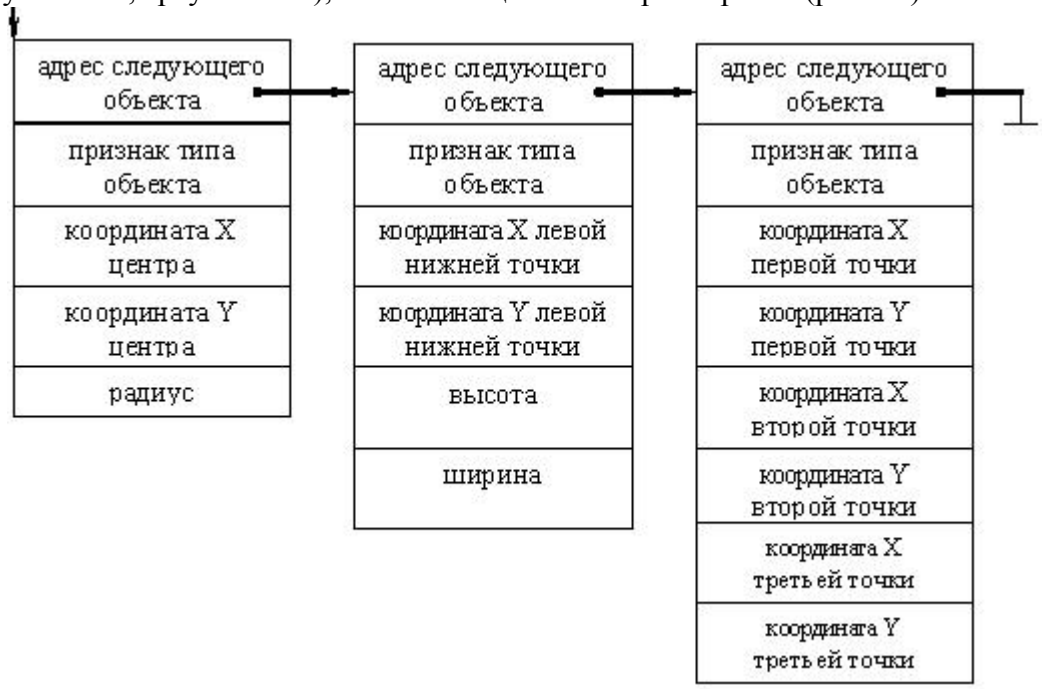


Рис. 42. Структура разнородного списка

Для описания узлов разнородных списков применяются записи с вариантной частью. Элемент хранения узла разнородного списка содержит фиксированные поля, которые располагаются в начале записи: связь в списке, координаты точки, относительно которой строится фигура. Далее располагается поле селектора вариантов и поля вариантов, соответствующие трем типам геометрических фигур.

```
Type
  Figure = (circle, rectangle, triangle);                  { тип фигуры }
  PPolygon = ^ Polygon;          { указатель на эл-т хранения узла разнородного списка }
  Polygon = record               { эл-т хранения узла разнородного списка }
    link: PPolygon;                                        { связь в списке }
    x,y: word;       { координаты точки, относительно которой строится фигура }
    case  kind: Figure  of              { селектор варианта, определяющий тип фигуры }
      circle: ( radius: word );                            { окружность }
      rectangle: ( height, width: word );                  { прямоугольник }
      triangle: ( x1,y1,x2,y2: word )                      { треугольник }
  end;

Var f: PPpolygon;                                          { указатель на первый узел списка }

{ создание элемента хранения узла и занесение его в разнородный список }
Procedure Create_Node( var first: PPolygon; t: Figure );              { t – тип фигуры }
```

```
Var p: PPolygon;            { first – указатель на первый узел списка }
begin
  new( p );
  writeln( ‘введите координаты точки, относительно которой строится фигура’ );
  write( ‘x= ’ ); readln( p^.x );
  write( ‘y= ’ ); readln( p^.y );
  p^.kind:=t;                                          { заполнение поля селектора вариантов }
  case  t  of          { заполнение полей вариантов в зависимости от типа фигуры: }
   circle:                                             { окружность }
    begin write( ‘введите значение радиуса = ’ ); readln( p^.radius ) end;
   rectangle:                                          { прямоугольник }
    begin
     write( ‘введите значение высоты = ’ ); readln( p^.height );
     write( ‘введите значение ширины = ’ ); readln( p^.width )
    end;
   triangle:                                           { треугольник }
    begin
     writeln( ‘введите координаты второй вершины’ );
     write( ‘x= ’ ); readln( p^.x1 );
     write( ‘y= ’ ); readln( p^.y1 );
     writeln( ‘введите координаты третьей вершины’ );
     write( ‘x= ’ ); readln( p^.x2 );
     write( ‘y= ’ ); readln( p^.y2 );
   end;
  end;                                                 { case }
  p^.link:=first; first:=p;          { включение элемента хранения узла в список }
end;

Procedure Print( first: PPolygon );          { просмотр содержимого разнородного списка }
Var p: PPolygon;
begin
 p:=first;
 while p <> nil do begin
 writeln( ‘координаты точки, относительно которой построена фигура’ );
 writeln( ‘x= ‘,  p^.x,  ‘y= ‘,  p^.y );
 case  p^.kind  of                    { проверка значения поля селектора вариантов: }
  circle:                                             { окружность }
   writeln( ‘радиус окружности = ‘, p^.radius );
  rectangle:                                          { прямоуглольник }
   writeln( ‘высота прям-ка = ’ , p^.height, ‘ширина прям-ка = ‘, p^.width );
  triangle:                                           { треугольник }
  begin
   writeln( ‘координаты второй вершины треугольника‘ );
   writeln( ‘x= ‘,  p^.x1,  ‘y= ‘,  p^.y1 );
   writeln( ‘координаты третьей вершины треугольника‘ );
   writeln( ‘x= ‘,  p^.x2,  ‘y= ‘,  p^.y2 );
  end;
 end;
 p:=p^.link
end;


Procedure Destroy( var first: PPolygon );          { разрушение разнородного списка }
var p: Ppolygon;
```

```
begin
   while (first<>nil) then
    begin
      p:=first; first:=first^.link; dispose( p )
    end
end;

begin
  f:=nil;                                          { первоначально список пуст }
  Create_Node( f,rectangle );
  Create_Node( f,triangle );
  Create_Node( f,rectangle );
  Create_Node( f,circle );     . . .
  Print( f );  . . .
  Destroy( f );
end.
```

4.9. Управление динамической памятью

Управление динамической памятью осуществляется с помощью специальной программы – менеджера кучи, которая выполняет функции выделения/освобождения памяти при размещении/удалении динамических объектов программы. При использовании динамической памяти возникают проблемы, связанные с фрагментацией, возникновением “мусора” и “висящих ссылок”.

4.9.1. Администратор кучи

Администратор (менеджер) кучи – служебная программа, которая автоматически пристыковывается к программе пользователя во время компоновки и управляет взаимодействием программы пользователя с кучей. Администратор кучи обрабатывает запросы на выделение и освобождение памяти, определение размера свободной памяти и т.п, используя для этого стандартные указатели, которые определены в модуле *SYSTEM Turbo Pascal* (рис. 43):

HeapOrg – указатель на начало кучи,
HeapEnd – указатель на верхнюю границу кучи,
HeapPtr – указатель на нижнюю границу свободного пространства кучи,
FreeList – указатель на список свободных участков кучи.

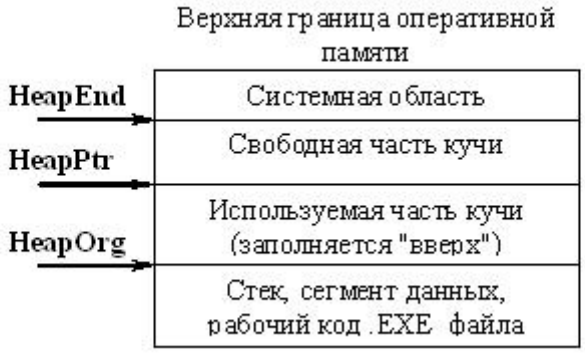


Рис. 43. Распределение области памяти кучи

Каждый свободный блок описывается дескриптором, имеющим следующую структуру:

```
Type PFreeRec = ^ TFreeRec;
       TFreeRec = record
          next: pointer;
          size: pointer
        end;
Var FreeList: PFreeRec;
```

Т.о., указатель *FreeList* указывает на дескриптор первого свободного блока в куче. Данная списковая структура предназначена для описания всех свободных блоков памяти, которые расположены ниже границы *HeapPtr*. Происхождение блоков связано со случайной последовательностью запросов на выделение / освобождение памяти в процессе выполнения программы. Поле *next* в записи *TFreeRec* указывает на дескриптор следующего по списку свободного блока кучи или содержит адрес, совпадающий с *HeapEnd*, если этот участок последний в списке. Поле *size* содержит 0, если ниже адреса, записанного в *HeapPtr*, нет свободных блоков или размер свободного блока, представленный в следующем виде: в старшем слове поля *size* - число свободных параграфов, т.е. участков размером 16 байт, а в младшем слове - число байт в диапазоне 0..15. Значение поля *size* с помощью специальной функции преобразуется в фактическую длину свободного блока (в байтах).

Сразу после загрузки программы указатели *HeapPtr* и *FreeList* содержат один и тот же адрес, который совпадает с началом кучи и находится в *HeapOrg*. При этом в начальных восьми байтах кучи хранится запись, соответствующая типу *TFreeRec* (поле

next содержит адрес, совпадающий со значением *HeapEnd*, а поле *size* – нулевое значение) (рис. 44).

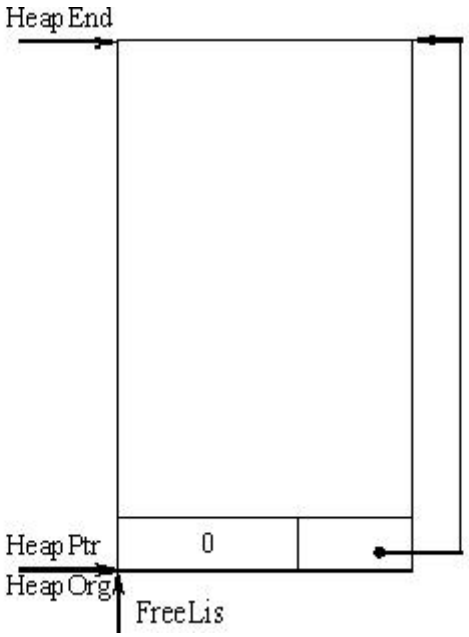


Рис. 44. Распределение области памяти кучи после загрузки программы

После выполнения серии запросов на выделение памяти область кучи будет распределена следующим образом (рис. 45):

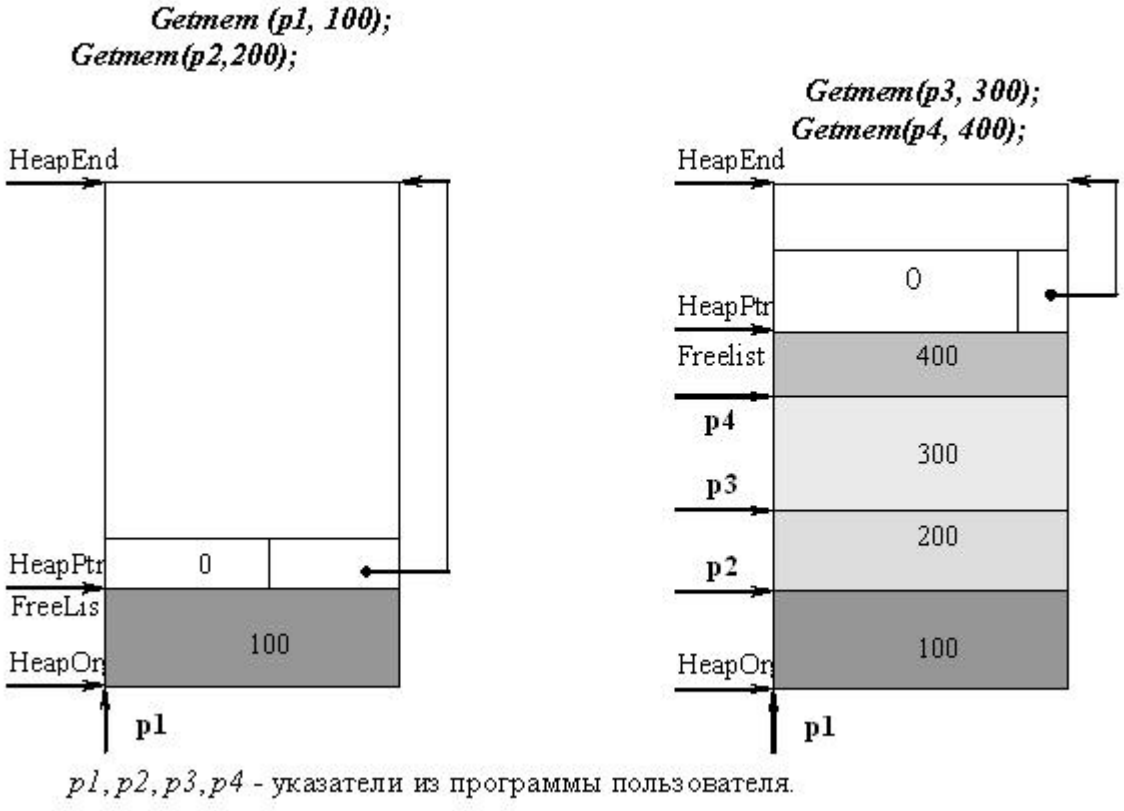


Рис. 45. Распределение области памяти кучи после выполнения запросов на выделение памяти

При работе с кучей указатели *HeapPtr* и *FreeList* будут иметь одинаковые значения до тех пор, пока в куче не образуется хотя бы один свободный блок ниже границы, содержащейся в указателе *HeapPtr*. Как только это произойдет, указатель *FreeList* станет ссылаться на начало этого блока, а в первых восьми байтах освобожденного участка памяти будет размещен дескриптор, т.е. запись *TFreeRec* (рис. 46, 47).

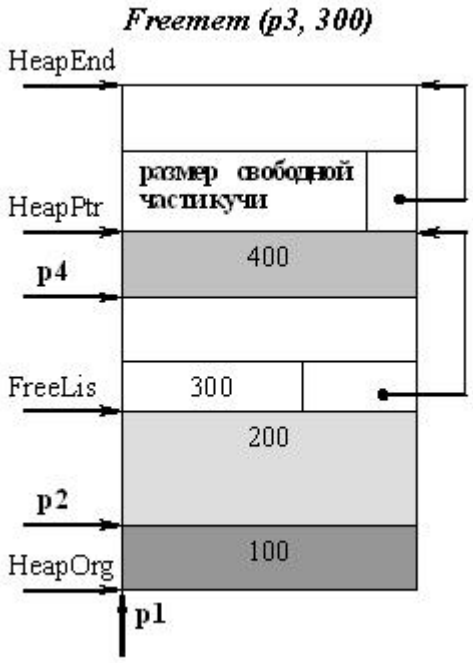


Рис. 46. Список свободных блоков после выполнения запроса на освобождение памяти

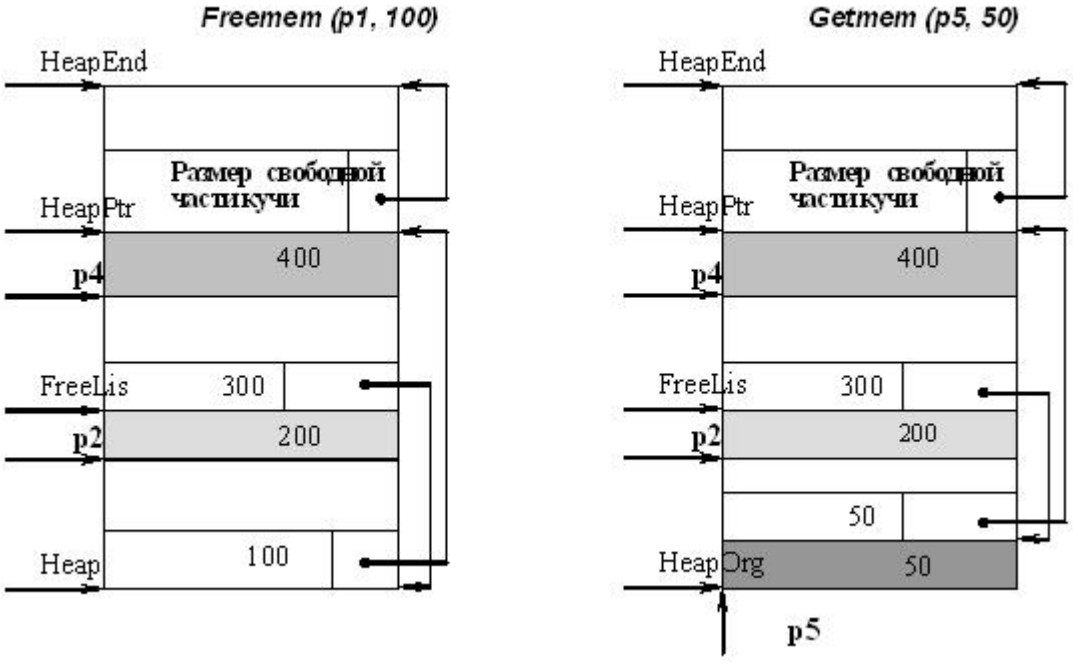


Рис. 47. Распределение области памяти кучи после выполнения запросов на освобождение / выделение памяти

Если освобождаемый участок памяти вплотную прилегает с одной или двух сторон к свободным участкам, происходит слияние и образуется более крупный свободный блок (рис.48).

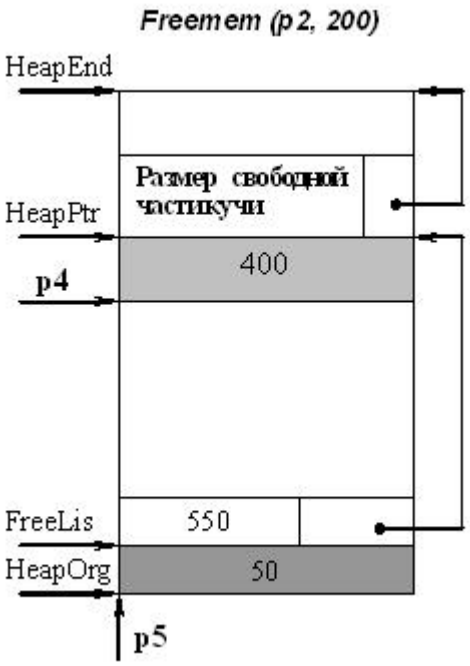


Рис. 48. Слияние свободных блоков в результате освобождения памяти

Организация списка свободных блоков непосредственно в куче порождает следующую проблему. В любой освободившийся блок администратор кучи должен поместить дескриптор этого блока, следовательно, длина блока не может быть меньше восьми байтов. Поэтому администратор кучи всегда выделяет память блоками, размер которых кратен размеру записи *TFreeRec*, т.е. восьми байтам. Даже если программа запросит один байт, администратор выделит ей фактически восемь байтов.

4.9.2. Алгоритмы выделения участков памяти по запросу

Алгоритм “первого подходящего”. При очередном запросе на выделение памяти администратор кучи подбирает в списке свободных блоков первый встретившийся блок, размер которого не меньше требуемого. В среднем необходим просмотр половины всего списка свободных блоков. Недостаток этого алгоритма заключается в том, что он не сохраняет крупные свободные блоки.

Алгоритм “наиболее подходящего”. При очередном запросе на выделение памяти администратор кучи подбирает в списке свободных блоков наименьший блок, размер которого больше или равен запросу. Алгоритм “наиболее подходящего” обеспечивает сохранение более крупных свободных блоков, но может потребовать просмотра всего списка свободных блоков. Кроме того, со временем этот алгоритм имеет тенденцию к созданию большого количества свободных блоков малого размера, которые не смогут удовлетворить ни один запрос на выделение памяти. Администратор кучи *Turbo Pascal* реализует алгоритм “наиболее подходящего”.

Для того чтобы уменьшить количество просмотров списка свободных блоков используются ***модификации этих алгоритмов***. Можно сохранять в специальном указателе адрес свободного блока наибольшего размера и начинать поиск именно с этого блока. Поиск можно начинать с того блока, который был освобожден последним. Свободные блоки можно упорядочить по возрастанию или уменьшению их размеров, но тогда потребуется дополнительно хранить в дескрипторе каждого свободного блока начальный адрес этого блока.

Алгоритм “близнецов”. Идея этого алгоритма состоит в том, что организуются списки свободных блоков отдельно для каждого размера 2^k, $0 <= k <= m$. Вся область памяти кучи состоит из 2^m слов, которые, можно считать, имеют адреса с 0 по $2^m - 1$. Первоначально свободным является весь блок из 2^m слов. Далее, когда требуется блок из 2^k слов, а свободных блоков такого размера нет, расщепляется на две равные части блок большего размера; в результате появится блок размера 2^k (т.е. все блоки имеют длину, кратную 2). Когда один блок расщепляется на два (каждый из которых равен половине первоначального), эти два блока называются ***близнецами***. Позднее, когда оба близнеца освобождаются, они опять объединяются в один блок. Преимуществом этого алгоритма является скорость, но его реализация усложняется за счет необходимости вести систему списков свободных блоков.

4.9.3. Фрагментация

Если блоки переменной длины выделяются или освобождаются в произвольном порядке, то список свободных блоков может стать очень длинным, особенно если блоки при освобождении не укрупняются. Это приводит к ситуации, когда в ответ на очередной запрос о выделении памяти размера *size* байт в куче не окажется непрерывного свободного участка требуемого размера, т.е. *MaxAvail* < *size*, в то время как суммарный объем свободной памяти достаточен для выполнения запроса, т.е. *MemAvail* >= *size*. Разбиение всей доступной памяти области кучи на большое количество блоков относительно малого размера называется ***внешней фрагментацией***.

Внутренняя фрагментация связана с появлением неиспользуемых, но и недоступных участков памяти. Внутренняя фрагментация порождается, в частности, механизмом формирования списка свободных блоков. Так, если запрашиваемый размер памяти не кратен восьми байтам, в куче образуется “дырка” размером 1-7 байт, причем этот участок не может использоваться ни при каком другом запросе динамической памяти до того момента, когда связанный с ним объект не будет удален из кучи. Внутренняя фрагментация может возникнуть также вследствие реализации алгоритма резервирования блоков, имеющих размер, больший запрошенного (т.е. за счет отказа от деления блока на части, одна из которых может оказаться совсем маленькой). Внутреннюю фрагментацию могут вызвать сами пользователи, резервируя память “про запас”, заведомо большего размера, чем может понадобиться в текущий момент выполнения программы. Внутренняя фрагментация приводит к ситуации, когда на некоторый запрос будет получен отказ из-за отсутствия блоков требуемого размера, хотя зарезервированной, но не используемой памяти более чем достаточно для удовлетворения запроса, т.е. *MaxAvail* < *size, MemAvail* < *size*, где *size* – размер запроса.

4.9.4. Накопление мусора

Мусором называются блоки памяти, доступ к которым во время выполнения программы потерян, так что эти блоки больше не используются, но и не могут быть возвращены в кучу. Возникновение мусора иллюстрирует программный фрагмент и рис. 49.

```
var p,q: PList;
begin
  new (p);  new (q);
  …
  p:=q;   . . .
end;
```

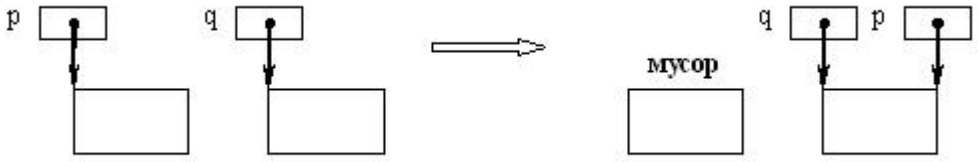


Рис. 49. Возникновение мусора

В результате накопления мусора уменьшается объем доступной свободной памяти и возрастает вероятность отказа на запрос о выделении памяти. Существуют специальные программы, называемые “сборщиками мусора ”. Они могут быть вызваны, когда запасы имеющейся памяти почти исчерпаны или когда невозможно удовлетворить очередной запрос на выделение памяти, или когда размер доступной области памяти стал меньше некоторого заранее заданного числа. На время работы сборщика мусора нормальное выполнение программы приостанавливается. Алгоритм сбора мусора обычно выполняется в два этапа. На первом этапе осуществляется просмотр всех указателей от всех программных объектов ко всем зарезервированным блокам. Каждый блок, к которому есть доступ, маркируется. На втором этапе просматривается вся куча, метки у маркированных блоков стираются, а все блоки, которые не были отмечены, возвращаются в список свободной памяти. Сбор мусора снижает эффективность выполнения программы, особенно если свободная память практически исчерпана, поэтому многие администраторы динамической памяти (в частности, в *Turbo Pascal*) не используют сборщики мусора. Предотвращать возникновение мусора должен сам программист.

4.9.5. Висящие ссылки

Висящая ссылка (указатель) - это существующий в программе указатель, который открывает доступ к уже несуществующему объекту (т.е. к уже освобожденному участку памяти). Возникновение висящей ссылки иллюстрирует программный фрагмент и рис. 50.

```
var p,q: PList;
begin
  new (p);    q:=p;
  . . .
  dispose( p );   p:=nil;                          { q – висящая ссылка }
end;
```

Рис. 50. Возникновение висящей ссылки

Если освобожденный блок будет вновь зарезервирован, а затем к нему применен висящий указатель, то программа вновь обратится к этому блоку, который теперь предназначен совсем для других целей. Ошибочное использование висящего указателя иллюстрирует программный фрагмент и рис. 51.

```
Type
   PT1 = ^ T1;
   T1 = record
     x, y : word
    end;
Var p, q: PT1;  r: PT2;
. . .
new ( p );
readln (p^.x,  p^.y);
q:=p;
. . .
dispose ( p ); p:=nil;
. . .
new ( r );
readln( r^.a );
readln( r^.b );
readln( r^.c );
. . .
writeln( q^.x );      { ? }
writeln( q^.y );      { ? }
```

```
PT2 = ^ T2;
T2 = record
 a: byte; b: char;  c: integer
end;
```

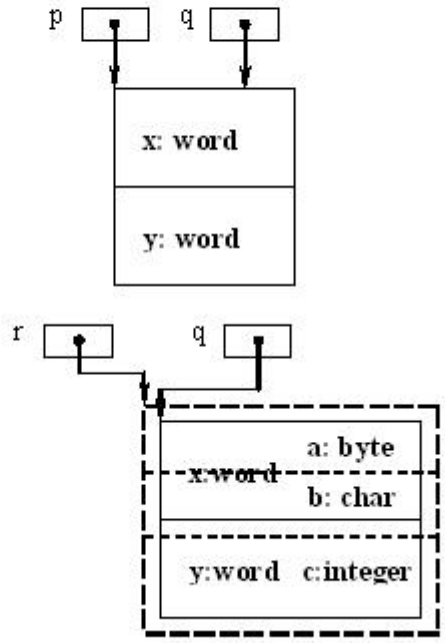


Рис. 51. Ошибочное применение висящего указателя

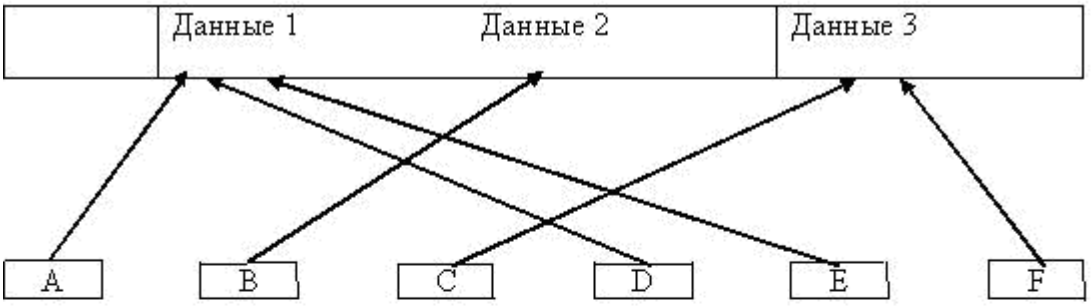
Через указатель *q* по-прежнему видна структура, соответствующая объекту типа *T1* с атрибутами *x* и *y*, ранее размещенному в том блоке памяти, который позже был зарезервирован для объекта типа *T2* с атрибутами *a,b,c*. Результат применения в программе висящей ссылки приводит к ошибкам, которые трудно обнаружить, поэтому висящие ссылки “опаснее” накопления мусора.

Проблему висящих ссылок можно было бы решить, если бы администратор реализовывал метод счетчиков ссылок. В этом методе для каждого зарезервированного блока имеется счетчик, показывающий, сколько объектов программы имеют непосредственный доступ к этому блоку, т.е. указатель на него. Когда некоторый блок резервируется впервые, его счетчик ссылок устанавливается в 1. Каждый раз, когда создается связь между некоторым идентификатором и этим блоком, значение счетчика ссылок увеличивается на 1; каждый раз, когда такая связь уничтожается, значение счетчика ссылок уменьшается на 1. Когда значение счетчика ссылок становится равным нулю, соответствующий блок оказывается недоступным, а, следовательно, неиспользуемым. В этот момент он возвращается в список свободной памяти. Однако ведение счетчиков ссылок может значительно увеличить время исполнения программы, поэтому в администраторах этот метод практически не используется. Контролировать отсутствие висящих ссылок должен сам программист.

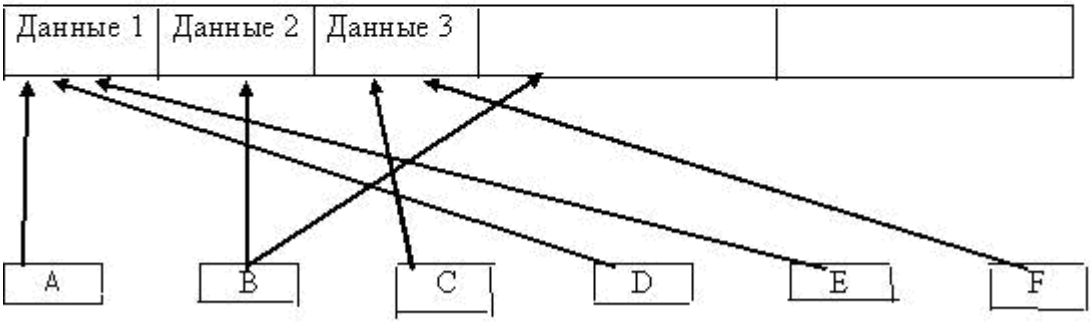
4.9.6. Уплотнение памяти

Существует еще один метод восстановления ранее зарезервированной памяти, называемый уплотнением. ***Уплотнение*** осуществляется путем физического передвижения блоков данных из одних областей памяти в другие с целью сбора всех свободных блоков в один большой блок. Резервирование памяти в этом случае значительно упрощается: просто передвигается указатель этого последовательно укорачиваемого блока. Как только размер этого единственного блока становится слишком мал, включается механизм уплотнения для выявления неиспользуемых блоков памяти, расположенных между зарезервированными блоками. Механизма освобождения памяти здесь совсем нет. Вместо него используется механизм маркировки, который отмечает блоки, используемые в данный момент. Затем, вместо того чтобы освобождать каждый неомеченный блок путем введения в действие механизма освобождения памяти, помещающего этот блок в список свободных блоков, используется уплотнитель, который собирает все неотмеченные блоки в один большой блок в одном конце кучи.

При использовании этого метода возникает проблема переопределения указателей. Она решается путем организации нескольких просмотров кучи. После маркировки блоков просматривается вся куча и для каждого из отмеченных блоков определяется его новый адрес. Этот адрес запоминается в самом блоке. Во время второго просмотра памяти указатели, адресующие отмеченные блоки, заново устанавливаются на те области, где эти блоки будут храниться после уплотнения. После переустановки всех указателей отмеченные блоки перемещаются на новые адреса. Процесс уплотнения памяти показан на рис. 52.



а. Перед уплотнением



б. После уплотнения

Рис. 52. Процесс уплотнения памяти

Простая схема уплотнения заключается в том, что сначала нужно просмотреть все блоки (как заполненные, так и пустые) и вычислить так называемый “адрес передачи” (forwarding address) для каждого заполненного блока. Адрес передачи блока - это его текущая

позиция минус сумма всего пустого пространства ниже его, т.е. позиция, в которую, в конечном счете, необходимо переместить этот блок. Адрес передачи блока вычисляется следующим образом. Когда просматриваются все блоки в направлении от нижних адресов пространства памяти к верхним адресам, нужно суммировать объем встречающихся свободных блоков и вычитать этот объем из адреса каждого встречающегося блока.

После вычисления адресов передачи программа-уплотнитель анализирует все указатели динамической памяти. Каждый указатель на некоторый блок заменяется адресом передачи, найденным для данного блока. Наконец, все заполненные блоки перемещаются по их адресам передачи. Перемещение заполненных блоков занимает время, пропорциональное объему используемой динамической памяти.

4.10. Контрольные вопросы к разделу 4

1. В чем заключаются отличия принципа вычисляемого адреса от принципа хранимого адреса?

2. Сравните последовательное и связанное представление структур данных.

3. Определите понятие линейной динамической структуры данных.

4. С какой дополнительной проверкой связан доступ к объекту через указатель?

5. Какие преимущества дает циклическая организация линейных списков?

6. Каковы особенности организации “проходов” по спискам?

7. Чем отличается организация и обработка двусвязных линейных списков от односвязных?

8. В чем преимущества двусвязной реализации списка перед односвязной? В чем недостатки?

9. Какова функция “сторожа” в циклических списках?

10. Какие связи необходимо установить при включении элемента в односвязный список? В двусвязный?

11. Какие связи необходимо изменить при исключении элемента из односвязного списка? Из двусвязного?

12. В чем заключается особенность удаления элемента, на который предварительно установлен указатель, из односвязного списка?

13. Как представляется стек на структуре односвязного списка? Двусвязного?

14. Как представляется очередь на структуре односвязного списка? Двусвязного?

15. Каково назначение мультисписков? В чем особенности их представления и обработки?

16. Для чего используются разнородные списки?

17. Какие функции выполняет администратор кучи? С какими служебными структурами данных он работает?

18. Как выполняются запросы на выделение/освобождение динамической памяти?

19. Какие алгоритмы выделения участков динамической памяти по запросу Вам известны? Сравните их между собой.

20. Какие виды фрагментации кучи Вам известны? Каковы причины их возникновения?

21. Что такое “накопление мусора“? Как его избежать?

22. Что такое “висящая ссылка“? Как предотвратить возникновение в программе “висящих ссылок“?

23. Для чего используется уплотнение памяти? Как функционирует механизм уплотнения памяти?

4.11. Упражнения к разделу 4

1. Подсчитайте количество положительных и отрицательных элементов в списке.

2. Постройте копию списка.

3. Постройте копию списка, изменив порядок составляющих его элементов на обратный.

4. Переставьте элементы списка в обратном порядке, начиная с номера N до номера K, не меняя их размещения в памяти компьютера.

5. Исключите из списка элементы, начиная с номера N до номера K.

6. Исключите из списка все элементы между двумя элементами с заданными значениями информационных полей.

7. Элементы списка хранят слова, состоящие из 10 символов, включающих только русские и английские прописные и строчные буквы. Исключите из списка слова, начинающиеся с заданной комбинации символов.

8. Элементы списка хранят слова, состоящие из 10 символов, включающих только русские и английские прописные и строчные буквы. Исключите из списка слова, содержащие заданную комбинацию символов.

9. Из исходного списка получите два новых списка, не изменяя размещения элементов в памяти: 1-й список должен содержать элементы, у которых значение информационного поля больше заданного значения N; 2-й - все остальные элементы.

10. Из исходного списка получите два новых списка путем копирования: 1-й список должен содержать слова, начинающиеся с заданной комбинации символов; 2-й список - слова, заканчивающиеся заданной комбинацией символов.

11. Создайте список, вставляя положительные числа непосредственно в начало списка, а отрицательные числа – в хвост списка. Разделите полученный список на два списка, содержащие положительные и отрицательные элементы соответственно, не меняя расположение элементов в памяти.

12. Вставьте в список новый элемент перед каждым элементом с заданным значением информационного поля (информационные поля могут повторяться).

13. Объедините два списка в один путем копирования в следующем порядке: 1-й элемент 1-го списка, 1-й элемент 2-го списка; 2-й элемент 1-го списка, 2-й элемент 2-го списка и т.д.

14. Объедините два списка в один без использования копирования в следующем порядке: 1-й элемент 1-го списка, 1-й элемент 2-го списка; 2-й элемент 1-го списка, 2-й элемент 2-го списка и т.д.

15. Реализуйте представление многочлена

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

с произвольными целыми коэффициентами в виде списка. При этом, если $a_i=0$, то соответствующий элемент-слагаемое должен отсутствовать в списке.

Напишите процедуру, проверяющую два многочлена на равенство, и процедуру вычисления значения многочлена в заданной целочисленной точке.

16. Реализуйте представление многочлена

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

с произвольными целыми коэффициентами в виде списка. При этом, если $a_i=0$, то соответствующий элемент-слагаемое должен отсутствовать в списке.

Напишите процедуру сложения двух многочленов. Результатом такого сложения должен быть новый многочлен-сумма.

17. Напишите процедуру копирования значения информационного поля каждого к - го по счету элемента списка, если его значение не превышает N, в новый список.

18. Напишите процедуру, выполняющую следующие действия: удалить из первого списка элементы, информационные поля которых совпадают с информационными полями элементов второго списка. Информационные поля у элементов каждого из списков могут повторяться.

19. Два списка заданы указателями F1 и F2 на их первые элементы. Определите, содержится ли список F1 в F2, F2 в F1, и подсчитайте число полных вхождений элементов первого списка во второй и элементов второго в первый. Например, первый список состоит из элементов 1, 2, 3 , 4, 5, 1, 2, 3, 6; второй список состоит из элементов 1, 2, 3; второй список входит в первый 2 раза.

20. Три списка заданы указателями F1, F2 и F3 на их первые элементы. Напишите процедуру, которая заменяет в списке F1 каждое полное вхождение списка F2 (если такое есть) на копию списка F3. Например, первый список состоит из элементов 1, 2, 3 , 4, 5, 1, 2, 3, 6; второй список состоит из элементов 1, 2, 3; третий список состоит из элементов 10, 20. В результате замены первый список будет содержать элементы 10, 20, 4, 5, 10, 20, 6. Количество элементов во втором и третьем списках может не совпадать.

21. Реализуйте моделирование очереди элементов ограниченной длины с дисциплиной "первым пришел - первым вышел" на структуре циклического списка. Моделирование связано с постановкой новых элементов в очередь, выводом из очереди и идентификацией ситуаций "Очередь полна" и "Очередь пуста".

22. Реализуйте моделирование дека ограниченной длины на структуре циклического списка. Дек (двусторонняя очередь) - линейный список, в котором включения и исключения элементов выполняются с любого конца структуры. Моделирование связано с постановкой новых элементов в дек, выводом из дека и идентификацией ситуаций "Дек полон" и "Дек пуст".

ЗАКЛЮЧЕНИЕ

Учебное пособие “Структуры данных. Часть I. Линейные динамические структуры” посвящено рассмотрению структур данных и алгоритмов, которые являются фундаментом современной методологии разработки программ.

Учебное пособие включает разделы, которые подробно описывают абстрагирование типов, методы идентификации объектов, классы памяти, линейные динамические структуры данных (односвязные, двусвязные списки, мультисписки), алгоритмы управления динамической памятью. Учебное пособие отличается методикой изложения всех разделов с точки зрения особенностей внутреннего представления структур данных различных видов. Теоретический материал иллюстрируется большим количеством примеров программ, реализующих алгоритмы обработки различных структур данных. Каждый раздел содержит большое количество примеров программ обработки данных различной структуры на языке Pascal. Логическим завершением каждого раздела являются контрольные вопросы и упражнения для самостоятельной работы студентов.

Вопросы, имеющие практическое значение для студентов при выполнении лабораторных работ и курсового проекта, освещены в учебном пособии с необходимой для использования полнотой.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Н. Вирт. Алгоритмы + структуры данных = программы: пер. с англ. / Н.Вирт. -М.: Мир, 1985. – 452 с.
2. Н. Вирт. Алгоритмы и структуры данных: пер. с англ. / Н.Вирт. Изд. 2-е, испр. -СПб.: Невский диалект, 2005. – 352 с.
3. Ахо А. Структуры данных и алгоритмы: пер. с англ. / Ахо А., Хопкрофт Д., Ульман Д. –М.: Вильямс, 2002. – 384 с.
4. Кнут Д. Искусство программирования для ЭВМ: в 3 т. Т 1: пер. с англ. / Кнут Д. – М.: Вильямс, 2000. – 720 с.
5. Иванова Г.С. Основы программирования: учебник для вузов / Иванова Г.С. Изд. 3-е, испр. –М.: Изд-во МГТУ им. Н.Э. Баумана, 2004. – 416с.
6. Фаронов В.В. Турбо Паскаль 7.0. Начальный курс: учебное пособие / Фаронов В.В. – М.: Нолидж, 2006. – 575 с.
7. Фаронов В.В. Delphi. Программирование на языке высокого уровня: учебник для вузов / Фаронов В.В. – СПб.: Питер, 2005. – 640 с.
8. Бобровский С.И. Delphi 7. Учебный курс : учебное пособие для вузов /Бобровский С.И. – СПб.: Питер, 2006. – 736 с.
9. Павловская Т.А. Паскаль. Программирование на языке высокого уровня: учебник для вузов / Павловская Т.А. – Спб.: Питер, 2004. – 393 с.
10. Павловская Т.А. Паскаль. Программирование на языке высокого уровня: Практикум : учебник для вузов / Павловская Т.А. – Спб.: Питер, 2006. – 317 с.
11. Топп У. Структуры данных в С++: пер. с англ. / Топп У., Форд У. –М.: ЗАО “Издательство БИНОМ”, 1999. – 815 с.
12. Кормен Т. Алгоритмы: построение и анализ: пер. с англ. / Кормен Т., Лейзерсон Ч., Ривест Р. –М.: МЦНМО, 2000. – 960 с.

Сведения об авторе

Симонова Елена Витальевна

кандидат технических наук
доцент кафедры информационных систем и технологий
тел. 267-46-72, кафедра ИСТ
e-mail: simonova@magenta-technology.ru