Хотя термины *тип данных* (или просто *тип*)*, структура данных* и *абстрактный тип данных* звучат похоже, но имеют они различный смысл. В языках программирования *тип данных* переменной обозначает множество значений, которые может принимать эта переменная. Например, переменная булевого (логического) типа может принимать только два значения: значение true (истина) и значение false (ложь) и никакие другие. Набор базовых типов данных отличается в разных языках: в языке Pascal это типы целых (integer) и действительных (real) чисел, булев (boolean) тип и символьный (char) тип. Правила конструирования составных типов данных (на основе базовых типов) также различаются в разных языках программирования: Pascal легко и быстро строит такие типы.

Абстрактный тип данных (АТД) – это математическая модель плюс различные операторы, определённые в рамках этой модели. Мы можем разрабатывать алгоритм в терминах АТД, но для реализации алгоритма в конкретном языке программирования необходимо найти способ представления АТД в терминах типов данных и операторов, поддерживаемых данным языком программирования. Для представления АТД используются *структуры данных*, которые представляют собой набор переменных, возможно, различных типов данных, объединённых определённым образом.

Базовым строительным блоком структуры данных является *ячейка*, которая предназначена для хранения значения определённого базового или составного типа данных. Структуры данных создаются путём задания имён совокупностям (агрегатам) ячеек и (необязательно) интерпретации значения некоторых ячеек как представителей (т.е. указателей) других ячеек.

В качестве простейшего механизма агрегирования ячеек в Pascal и большинстве других языков программирования можно применять (одномерный) массив, т.е. последовательность ячеек определённого типа. Массив также можно рассматривать как отображение множества индексов (таких как целые числа 1, 2, …, n) в множество ячеек. Ссылка на ячейку обычно состоит из имени массива и значения из множества индексов данного массива. В Pascal множество индексов может быть нечислового типа, например (север, восток, юг, запад), или интервального типа как (1..10).Значения всех ячеек массива должны иметь одинаковый тип данных. Объявление

*Имя*: **array**[ТипИндекса] **of** ТипЯчеек;

задаёт *имя* для последовательности ячеек, тип для элементов множества индексов и тип содержимого ячеек.

Кстати, Pascal необычайно богат на типы индексов. Многие языки программирования позволяют использовать в качестве индексов только множества последовательных целых чисел. Например, чтобы в языке Fortran в качестве индексов массива можно было использовать буквы, надо всё равно использовать целые индексы, заменяя «А» на 1, «В» на 2, и т.д.

Другим общим механизмом агрегирования ячеек в языке программирования является *структура записи*. *Запись* (record) можно рассматривать как ячейку, состоящую из нескольких других ячеек (называемых *полями*), значения в которых могут быть разных типов. Записи часто группируются в массивы; тип данных определяется совокупностью типов полей записи. Например, в Pascal объявление

v**ar**

*reclist*: **array**[1..4] **of record**

*data*: real;

*next*: integer;

**end**

задаёт имя *reclist* (список записей) 4-элементного массива, значениями которого являются записи с двумя полями: *data* (данные) и *next* (следующий).

Третий метод агрегирования ячеек, который можно найти в Pascal и некоторых других языках программирования, - это *файл*. Файл, как и одномерный массив, является последовательностью значений определённого типа. Однако файл не имеет индексов: его элементы доступны только в том порядке, в каком они были записаны в файл. В отличие от файла, массивы и записи являются структурами с «произвольным доступом», подразумевая под этим, что время доступа к компонентам массива и записи не зависит от значения индекса массива или указателя поля записи. Достоинство агрегирования с помощью файла (частично компенсирующее описанный недостаток) заключается в том, что файл не имеет ограничения на количество составляющих его элементов и это количество может изменяться во время выполнения программы.

# Указатели и курсоры

В дополнение к средствам агрегирования ячеек в языках программирования можно использовать указатели и курсоры. *Указатель* (pointer) – это ячейка, чьё значение указывает на другую ячейку. При графическом представлении структуры данных в виде схемы тот факт, что ячейка А является указателем на ячейку В, показывается с помощью стрелки от ячейки А к ячейке В.

В языке Pascal с помощью следующего объявления можно создать переменную – указатель *prt*, указывающую на ячейку определённого типа, например ТипЯчейки:

**var**

*prt*: ТипЯчейки

## Постфикс в виде стрелки, направленной вверх, в Pascal используется как оператор разыменования, т.е. выражение *prt* обозначает значение (типа ТипЯчейки) в ячейке, указанной *prt*.

## *Курсор* (cursor) – это ячейка с целочисленным значением, используемая для указания на массив. В качестве способа указания курсор работает так же, как и указатель, но курсор можно использовать и в языках (подобных Fortran), которые не имеют явного типа указателя. Интерпретировав целочисленную ячейку как индексное значение для массива, можно эффективно реализовать указания на ячейки массива. К сожалению, этот приём подходит только для ячеек массива и не позволяет организовать указания на ячейки, не являющиеся частью массива.

## В схемах структур данных будет рисоваться стрелка из ячейки курсора к ячейке, на которую указывает курсор. Иногда также будет показываться целое число в ячейке курсора, напоминая тем самым, что это не настоящий указатель. Можно заметить, что механизм указателя Pascal разрешает ячейкам массива только «быть указанными» с помощью курсора, но не быть истинным указателем. Другие языки программирования, подобные PL/1 или С, позволяют компонентам массивов быть истинными указателями и, конечно, «быть указанным» с помощью курсора. В отличие от этих языков, в языках Fortran и Algol, где нет типа указателя, можно использовать только курсоры.

## **Пример1.** На рис.1 показана структура данных, состоящая из двух частей. Она имеет цепочку ячеек, содержащих курсоры для массива *reclist* (список записей), определённого выше. Назначение поля *next* (следующий) заключается в указании на следующую запись в массиве *reclist*. Например, *reclist[4].next* равно 1, поэтому запись 4 предшествует записи 1. полагая первой запись 4, в соответствии со значениями поля *next* получим следующий порядок записей: 4, 1, 3, 2. Отметим, что значение поля *next* в записи 2, равное 0, указывает на то, что нет следующей записи. Целесообразно принять соглашение, что число 0 будет обозначать *нуль-указатель* при использовании курсоров и указателей. Но, чтобы не возникали проблемы при реализации этого соглашения, необходимо также условиться, что массивы, на которые указывают курсоры, индексируются начиная с 1, а не с 0.

## header

2

4

* 1. 3

7.8 1

5.6 2

3.4 0

## 1

## 2

## 3

## 4

## *data next*

## *Рис.1Пример структуры данных.*

## Ячейки в цепочке 1 имеют тип

## **type**

## recordtype = **record**

## *cursor*: integer;

## *prt* recordtype

## **end**

## На цепочку можно ссылаться с помощью переменной *header* (заголовок), имеющей тип recordtype, *header* указывает на анонимную запись типа recordtype 1 .Эта запись имеет 4 значения в поле *cursor.* Рассматривается 4 как индекс массива *reclist*. Эта запись имеет истинный указатель в поле *prt* на другую анонимную запись, которая, в свою очередь, указывает на индекс 4 массива *reclist* и имеет нуль-указатель в поле *prt*.

## **Абстрактный тип данных «Список».**

## Списки являются чрезвычайно гибкой структурой, так как их легко сделать большими или меньшими, и их элементы доступны для вставки или удаления в любой позиции списка. Списки также можно объединять, или разбивать на меньшие списки. Списки регулярно используются в приложениях, например в программах информационного поиска, трансляторах программных языков или при моделировании различных процессов. Методы управления памятью широко используют технику обработки списков. Далее будут описаны основные операции, выполняемые над списками, и представлены структуры данных для списков, которые эффективно поддерживают различные подмножества таких операций.

## В математике *список* представляет собой последовательность элементов определённого типа, который в общем случае будет обозначаться как elementtype (тип элемента). Список будет часто представляться в виде последовательности элементов, разделённых запятыми: a1, a2, …, an,гдеn > 0 и все ai имеют тип elementtype. Количество элементов nбудет называться *длиной списка*. Если n > 1 ,то а1 называется *первым элементом* списка. В случае n = 0 список будет называться *пустым*, т.е. не содержащий элементов.

## Важное свойство списка заключается в том, что его элементы можно линейно упорядочить в соответствии с их позицией в списке. Говорится, что элемент аi *предшествует* аi+1 для i = 1, 2, …, n-1 и аi *следует за* ai-1 для i = 2, 3, …, n. Говорится также, что элемент аi *имеет позицию* i. Кроме того, постулируется существование позиции, следующей за последним элементом списка. Функция END(L) будет возвращать позицию, следующую за позицией n в n-элементном списке L. Следует отметить, что позиция END(L), рассматриваемая как расстояние от начала списка, может изменяться при уменьшении или увеличении списка, в то время как другие позиции имеют фиксированное (неизменное) расстояние от начала списка.

## Для формирования абстрактного типа данных на основе математического определения списка нужно задать множество операторов, выполняемых над объектами типа LIST2 (список). Однако не существует одного множества операторов, выполняемых над списками, удовлетворяющего сразу все приложения.

## Чтобы показать некоторые общие операторы, выполняемые над списками, предположим, что имеем приложение, содержащее список почтовой рассылки, который мы хотим очистить от повторяющихся адресов. Концептуально эта задача решается очень просто: для каждого элемента списка удаляются все последующие элементы, совпадающие с данным. Однако для записи такого алгоритма необходимо определить операторы, которые должны найти первый элемент списка, перейти к следующему элементу, осуществить поиск и удаление элементов.

## Теперь перейдём к непосредственному определению множества операторов списка. Примем обозначения: L – список объектов типа elementtype, x – объект этого типа, p – позиция элемента в списке. Следует отметить, что «позиция» имеет другой тип данных, чья реализация может быть различной для разных реализаций списков. Обычно позиция понимается как множество целых положительных чисел, но на практике могут встретиться другие представления.

## INSERT(x, p, L). Этот оператор вставляет объект х в позицию р и далее в следующую, более высокую позицию. Таким образом, если список L состоит из элементов а1, a2, …,an, то после выполнения этого оператора он будет иметь вид а1, a2, …, ap-1, x, ap, …, an. Если р принимает значение END(L), то будем иметь a1, a2, …, an, x. Если в списке L нет позиции р, то результат выполнения этого оператора не определён.

## LOCATE(x, L). Эта функция возвращает позицию объекта х в списке L.если в списке объект х встречается несколько раз, то возвращается позиция первого от начала списка объекта х. Если объекта х нет в списке L, то возвращается END(L).

## RETRIEVE(p, L). Эта функция возвращает элемент, который стоит в позиции р в списке L. Результат не определён, если p = END(L) или в списке L нет позиции p. Элементы должны быть одного типа, который в принципе может возвращать функция. Однако на практике мы всегда можем изменить эту функцию так, что она будет возвращать указатель на объект типа elementtype.

## DELETE(p, L). этот оператор удаляет элемент в позиции p списка L. Так, если список L состоит из элементов a1, a2, …, an, то после выполнения этого оператора он будет иметь вид a1, a2, …, ap-1, ap+1, …, an. Результат не определён, если в списке L нет позиции p или p = END(L).

## NEXT(p, L) и PREVIOUS(p, L). Эти функции возвращают соответственно следующую и предыдущую позиции от позиции p в списке L.если р – последняя позиция в списке L, то NEXT(p, L) = END(L). Функция NEXT не определена, когда p = END(L). Функция PREVIOUS не определена, если p = 1. Обе функции не определены, если в списке L нет позиции p.

## MAKENULL(L). Эта функция делает список L пустым и возвращает позицию END(L).

## FIRST(L). Эта функция возвращает первую позицию в списке L. Если список пустой, то возвращается позиция END(L).

## 8. PRINTLIST(L). Печатает элементы списка L в порядке их расположения.

# Стеки.

## *Стек* – это специальный тип списка, в котором все вставки и удаления выполняются только на одном конце, называемом *вершиной* (top). Стеки также иногда называют «магазинами», а в английской литературе для обозначения стеков ещё используется аббревиатура LIFO (last-in-first-out – последний вошел – первый вышел). Интуитивными моделями стека могут служить колода карт на столе при игре в покер, книги, сложенные в стопку, или стопка тарелок на полке буфета; во всех этих моделях взять можно только верхний предмет, а добавить новый объект можно, только положив его на верхний. Абстрактные типы данных семейства STAK (Стек) обычно используют следующие пять операторов.

## MAKENULL(S). Делает стек S пустым.

## TOP(S). Возвращает элемент из вершины стека S. Обычно вершина стека идентифицируется позицией 1, тогда TOP(S) можно записать в терминах общих операторов списка как RETRIEVE(FIRST(S), S).

## POP(S). Удаляет из вершины стека (выталкивает из стека), в терминах операторов списка этот оператор можно записать как DELETE(FIRST(S), S). Иногда этот оператор реализуется в виде функции, возвращающей удаляемый элемент.

## PUSH(x, S). Вставляет элемент x в вершину стека S (заталкивает элемент в стек). Элемент, ранее находившийся в вершине стека, становится элементом, следующим за вершиной, и т.д. В терминах общих операторов списка данный оператор можно записать как INSERT(x, FIRST(S), S).

## EMPTY(S). Эта функция возвращает значение true (истина), если стек S пустой, и значение false (ложь) в противном случае.

## **Очереди.**

## Другой специальный тип списка – *очередь* (queue), где элементы вставляются с одного конца, называемого *задним* (rear), а удаляются с другого, *переднего* (front). Очереди также называют «списками типа FIFO» (аббревиатура FIFO расшифровывается как first-in-first-out: первым вошел – первым вышел). Операторы, выполняемые над очередями, аналогичны операторам стеков. Существенное отличие между ними состоит в том, что вставка новых элементов осуществляется в конец списка, а не в начало, как в стеках. Кроме того, различна устоявшаяся терминология для стеков и очередей. Для работы с очередями будут использоваться следующие операторы.

## MAKENULL(Q) очищает очередь Q, делая её пустой.

## FRONT(Q) – функция, возвращающая первый элемент очереди Q. Можно реализовать эту функцию с помощью операторов списка как RETRIEVE(FIRST(Q), Q).

## ENQUEUE(x, Q) вставляет элемент x в конец очереди Q. С помощью операторов списка этот оператор можно выполнить следующим образом: INSERT(x, END(Q), Q).

## DEQUEUE(x, Q) удаляет первый элемент очереди Q. Также реализуем с помощью операторов списка как DELETE(FIRST(Q), Q).

## EMPTY(Q) возвращает значение true тогда и только тогда, когда Q является пустой очередью.

## **Отображения.**

## *Отображение* – это функция, определённая на множестве элементов (области определения) одного типа (будет обозначаться domaintype – тип области определения функции) и принимающая значения из множества элементов (области значений) другого типа, этот тип будет обозначаться rangetype – тип области значений (конечно, типы domaintype и rangetype могут совпадать). Тот факт, что отображение М ставит в соответствие элемент d типа domaintype из области определения элементу r типа rangetype из области значений, будет записываться как M(d) = r.

## Некоторые отображения, подобные square(i) = i2, легко реализовать с помощью функций и арифметических выражений языка Pascal. Но для многих отображений нет очевидных способов реализации, кроме хранения для каждого d значения M(d). Например, для реализации функции, ставящей в соответствие работникам их недельную зарплату, требуется хранить текущий заработок каждого работника.

## Рассмотрим операторы, которые можно выполнить над отображением М. Например, по заданному элементу d типа domaintype мы хотим получить M(d) или узнать, определено ли M(d) (т.е. узнать, принадлежит ли элемент d области определения М). Или хотим добавить новые элементы в текущую область определения М и поставить им в соответствие элементы из области значений. Очевидна также необходимость иметь оператор, который изменял бы значение M(d). Кроме того, нужно средство «обнуления» отображения, переводящее любое отображение в *пустое отображение*, т.е. такое, у которого область определения пуста. Всё это можно обобщить в следующие три команды.

## MAKENULL(M). Делает отображение М пустым.

## ASSIGN(M, d, r). Делает М(d) равным r независимо от того, как M(d) определено ранее.

## COMPUTE(M, d, r). Возвращает значение true и присваивает переменной r значение M(d), если последнее определено, и возвращает false в противном случае.

## **Структуры данных и алгоритмы для внешней памяти.**

## В алгоритмах, которые обсуждались до сих пор, предполагалось, что объем входных данных позволяет обходиться исключительно основной (оперативной) памятью. Но как быть, если нужно, например, отсортировать всех государственных служащих по продолжительности их рабочего стажа или хранить информацию из налоговых деклараций всех граждан страны? Когда возникает необходимость решать подобные задачи, объём обрабатываемых данных намного превышает возможности основной памяти. В большинстве компьютерных систем предусмотрены устройства внешней памяти, такие как жесткие диски, или запоминающие устройства большой ёмкости, на которых можно хранить огромные объёмы данных. Однако характеристики доступа к таким устройствам внешней памяти существенно отличаются от характеристик доступа к основной памяти. Чтобы повысить эффективность использования этих устройств был разработан ряд структур данных и алгоритмов.

## В Pascal и некоторых других языках предусмотрен файловый тип данных, предназначенный для представления данных, хранящихся во вторичной памяти. Даже если в языке, которым вы пользуетесь, файловый тип данных не предусмотрен, в операционной системе понятие «внешних» файлов, несомненно, поддерживается. О каких бы файлах ни говорилось (файлах, предусмотренных в Pascal, или файлах, поддерживаемых непосредственно операционной системой), в любом случае придётся действовать в рамках ограничений, касающихся способов доступа к файлам. Операционная система делит вторичную память на *блоки* одинакового размера. Размер блока зависит от конкретного типа операционной систем и обычно находится в пределах от 521 до 4096 байт.

## Файл можно рассматривать как связный список блоков, хотя чаще всего операционная система использует древовидную организацию блоков, при которой блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатель на множество блоков файла. Если, например, 4 байт достаточно, чтобы хранить адрес блока, а длинна блока составляет 4096 байт, тогда корневой каталог может содержать указатели максимум на 1024 блока. Таким образом, файлы, состоящие максимум из 1024 блоков (т.е. примерно четырёх миллионов байт), можно представить одним корневым блоком и блоками, содержащими сам файл. Файлы, состоящие из максимум 220 блоков, или 232 байт, можно представить одним корневым блоком, указывающим на 1024 блока промежуточного уровня, каждый из которых указывает на 1024 блока-листа, содержащих определённую часть файла и т.д.

## Базовой операцией, выполняемой по отношению к файлам, является перенос одного блока в *буфер*, находящийся в основной памяти. Буфер представляет собой зарезервированную область в основной памяти, размер которой соответствует размеру блока. Типичная операционная система обеспечивает чтение блоков в том порядке, в котором они появляются в списке блоков, который содержит соответствующий файл, т.е. сначала мы читаем в буфер первый блок файла, затем заменяем его на второй блок, который записывается в тот же буфер, и т.д.

## Теперь нетрудно понять концепцию, которая лежит в основе правил чтения файлов в языке Pascal. Каждый файл хранится в виде определённости блоков; каждый такой блок содержит целое число записей. (Память будет использоваться нерационально, если хранить части одной и той же записи в разных блоках.) Указатель считываний всегда указывает на одну из записей в блоке, который в данный момент находится в буфере. Когда этот указатель должен переместиться на запись, отсутствующую в буфере, настало время прочитать следующий блок файла.

## Аналогично, процесс записи файла в языке Pascal можно рассматривать как процесс создания файла в буфере. Когда записи «записываются» в файл, фактически они перемещаются в буфер для этого файла – непосредственно вслед за записями, которые уже находятся там. Если очередная запись не помещается в буфер целиком, содержимое буфера копируется в свободный блок вторичной памяти, который присоединяется к концу списка блоков для данного файла. После этого можно считать, что буфер свободен для помещения в него очередной порции записей.

## **Стоимость операций со внешней памятью.**

## Природа устройств вторичной памяти (например, дисководов) такова, что время, необходимое для поиска блока и чтения его в основную память, достаточно велико, в сравнении со временем, которое требуется для относительно простой обработки данных, содержащихся в этом блоке. Допустим, например, что имеется блок из 1000 целых чисел на диске, вращающемся со скоростью 1000 об/мин. Время, которое требуется для позиционирования считывающей головки над дорожкой, содержащей этот блок (так называемое *время установки головок*), плюс время, затрачиваемое на ожидание, пока требуемый блок сделает оборот и окажется под головкой (*время ожидания*), может в среднем составлять 100 миллисекунд. Процесс записи блока в определённое место во вторичной памяти занимает примерно столько же времени. Однако за те же 100 миллисекунд машина, как правило, успевает выполнить 100 000 команд. Этого времени более чем достаточно, чтобы выполнить простую обработку тысячи целых чисел, когда они находятся в основной памяти (например, их суммирование или нахождение среди них наибольшего числа). Этого времени может даже хватить для быстрой сортировки целых чисел.

## Оценивая время работы алгоритмов, в которых используются данные, хранящиеся в виде файлов, придётся в первую очередь учитывать количество обращений к блокам, т.е. сколько раз блок считывается в основную или записывается во вторичную память. Такая операция называется *доступом (или обращением) к блоку*. Предполагается, что размер блока фиксирован в операционной системе, поэтому невозможно ускорить работу алгоритма, увеличив размер блока и сократив тем самым количество обращений к блокам. Таким образом, мерой качества алгоритма, работающего с внешней памятью, является количество обращений к блокам.

## **Хранение данных в файлах.**

## В этом разделе будут рассмотрены структуры данных и алгоритмы для хранения и поиска информации в файлах, находящихся во внешней памяти. Файл будет рассматриваться как последовательность записей, причём каждая запись состоит из одной и той же совокупности полей. Поля могут иметь либо *фиксированную длину* (заранее определённое количество байт), либо *переменную*. Файлы с записями фиксированной длины широко используются в системах управления базами данных для хранения данных со сложной структурой. Файлы с записями переменной длины, как правило, используются для хранения текстовой информации; в языке Pascal такие файлы не предусмотрены. В этом разделе будем иметь дело с полями фиксированной длины; рассмотренные методы после определённой (несложной) модификации могут использоваться для работы с записями переменной длины.

## Будут рассмотрены следующие операторы для работы с файлами.

## INSERT вставляет определённую запись в определённый файл.

## DELETE удаляет из определённого файла все записи, содержащие указанные значения в указанных полях.

## MODIFY изменяет все записи в определённом файле, задав указанные значения определённым полям в тех записях, которые содержат указанные значениях в других полях.

## RETRIEVE отыскивает все записи, содержащие указанные значения в указанных полях.

## **Простая организация данных.**

## Простейшим (и наименее эффективным) способом реализации перечисленных выше операторов работы с файлами является использование таких примитивов чтения и записи файлов, которые встречаются, например, в Pascal. В случае использования подобной «организации» (которая на самом деле является дезорганизацией) записи могут храниться в любом порядке. Поиск записи с указанными значениями в определённых полях осуществляется путём полного просмотра файла и проверки каждой его записи на наличие в ней заданных значений. Вставку в файл можно выполнять путём присоединения соответствующей записи к концу файла.

## В случае изменения записей необходимо просмотреть файл, проверить каждую запись и выяснить, соответствует ли она заданным условиям (значения в указанных полях). Если соответствует, в запись вносятся требуемые изменения. Принцип действия операции удаления почти тот же, но когда мы поля которой соответствуют значениям, заданным в операции удаления, мы должны найти способ удалить её. Один из вариантов – сдвинуть все последовательные записи в своих блоках на одну позицию вперёд, а первую запись в каждом последующем блоке переместить на последнюю позицию предыдущего блока данного файла. Однако такой подход не годится, если записи являются закрепленными, поскольку указатель на i-ю запись в файле после выполнения этой операции будет указывать на (i+1)-ю запись.

## Если записи являются закреплёнными, следует воспользоваться каким-то другим подходом. Мы должны как-то помечать удалённые записи, но не должны смещать оставшиеся на место удалённых (и не должны вставлять на их место новые записи). Таким образом выполняется логическое удаление записи из файла, но её место в файле остаётся незанятым. Это нужно для того, чтобы в случае появления указателя на удалённую запись мы могли, во-первых, понять, что указываемая запись уже удалена, и, во-вторых, предпринять соответствующие меры (например, присвоить этому указателю значение NIL, чтобы в следующий раз не тратить время на его анализ). Существуют два способа помечать удалённые записи.

## Заменить запись на какое-то значение, которое никогда не может стать значением «настоящей» записи, и, встретив указатель на какую-либо запись, считать её удалённой, если она содержит это значение.

## Предусмотреть для каждой записи специальный *бит удаления*; этот бит содержит 1 в удалённых записях и 0 – в «настоящих» записях.

## **Ускорение операций с файлами.**

## Очевидным недостатком последовательного файла является то, что операторы с такими файлами выполняются медленно. Выполнение каждой операции требует, чтобы прочитали весь файл, а после этого ещё и выполнили перезапись некоторых блоков. К счастью, существуют такие способы организации файлов, которые позволяют обращаться к записи, считывая в основную память лишь небольшую часть файла.

## Такие способы организации файлов предусматривают наличие у каждой записи файла так называемого *ключа*, т.е. совокупности полей, которая уникальным образом идентифицирует каждую запись. Например, в файле с полями *фамилия, адрес, телефон* поле *фамилия* само по себе может считаться ключом, т.е. можно предположить, что в таком файле не может одновременно быть двух записей с одинаковым значением поля *фамилия*. Поиск записи, когда заданы значения её ключевых полей, является типичной операцией, на обеспечение максимальной эффективности которой ориентированы многие широко распространённые способы организации файлов.

## Ещё одним непременным атрибутом быстрого выполнения операций с файлами является возможность непосредственного доступа к блокам (в отличие от последовательного перебора всех блоков, содержащих файл). Многие структуры данных, которые используются для быстрого выполнения операций с файлами, используют указатели на сами блоки, которые представляют собой физические адреса этих блоков. К сожалению, на языке Pascal (и многих других языках программирования) невозможно писать программы, работающие с данными на уровне физических блоков и их адресов, – такие операции, как правило, выполняются с помощью команд файловой системы. Однако будет приведено краткое неформальное описание принципа действия операторов, в которых используется прямой доступ к блокам.

## **Хешированные файлы.**

## Хеширование – широко распространённый метод обеспечения быстрого доступа к информации, хранящейся во вторичной памяти. Основная идея этого метода подобна открытому хешированию. Записи файла распределяются между так называемыми *сегментами*, каждый из которых состоит из связного списка одного или нескольких блоков внешней памяти. Имеется таблица сегментов, содержащая В указателей, – по одному на каждый сегмент. Каждый указатель в таблице сегментов представляет собой физический адрес первого блока связного списка блоков для соответствующего сегмента.

## Сегменты пронумерованы от 0 до В-1. Хеш-функция h отображает каждое значение ключа в одно из целых чисел от 0 до В-1. Если x – ключ, то h(x) является номером сегмента, который содержит запись с ключом х (если такая запись вообще существует). Блоки, составляющие каждый сегмент, образуют связный список. Таким образом, заголовок i-го блока содержит указатель на физический адрес (i+1)-го блока. Последний блок сегмента содержит в своём заголовке NIL-указатель.

## Такой способ организации показан на рис.2.

r7 r8

r4 r5 r6

r1  r2 r3

## x

## Таблица

## Сегментов

## Рис.2. Сегменты, состоящие из связных блоков.

## Если размер таблицы сегментов невелик, её можно хранить в основной памяти. В противном случае её можно хранить последовательным способом в отдельных блоках. Если нужно найти запись с ключом х, вычисляется h(x) и находится блок таблицы сегментов, содержащий указатель на первый блок сегмента h(x), пока не обнаружится блок, который содержит запись с ключом х. Если исчерпаны все блоки в связном списке для сегмента h(x), приходим к выводу, что х не является ключом ни одной из записей.

## Такая структура оказывается вполне эффективной, если в выполняемом операторе указываются значения ключевых полей. Среднее количество обращения к блокам, требующееся для выполнения оператора, в котором указан ключ записи, приблизительно равняется среднему количеству блоков в сегменте, которое равно n/bk, если n – количество записей, блок содержит b записей, а k соответствует количеству сегментов. Таким образом, при такой организации данных операторы, использующие значения ключей, выполняются в среднем в k раз быстрее, чем в случае неорганизованного файла. К сожалению, ускорения операций, не основанных на использовании ключей, добиться не удаётся, поскольку при выполнении подобных операций приходится анализировать практически всё содержимое сегментов. Единственным универсальным способом ускорения операций, не основанных на использовании ключей, по-видимому, является применение вторичных индексов.

## Чтобы вставить запись с ключом, значение которого равняется х, нужно сначала проверить, нет ли в файле записи с таким же значением ключа. Если такая запись есть, выдаётся сообщение об ошибке, поскольку мы предполагаем, что ключ уникальным образом идентифицирует каждую запись. Если записи с ключом х нет, мы вставляем новую запись в первый же блок цепочки для сегмента h(x), в которыё эту запись удаётся вставить. Если запись не удаётся вставить ни в какой из существующих блоков сегмента h(x), файловой системе выдаётся команда найти новый блок, в который будет помещена эта запись. Этот новый блок затем добавляется в конец цепочки блоков сегмента h(x).

## Чтобы удалить запись с ключом х, нужно сначала найти эту запись, а затем установить её бит удаления. Ещё одной возможной стратегией удаления (которой, впрочем, нельзя пользоваться, если мы имеем дело с закреплёнными записями) является замена удалённой записи на последнюю запись в цепочке блоков сегмента h(x). Если такое изъятие последней записи приводит к опустошению последнего блока в сегменте h(x), этот пустой блок можно затем вернуть файловой системе для повторного использования.

## Хорошо продуманная организация файлов с хешированным доступом требует лишь незначительного числа обращений к блокам при выпол6нении каждой операции с файлами. Если мы имеем дело с хорошей функцией хеширования, а количество сегментов приблизительно равно количеству записей в файле, делённому на количество записей, которые могут уместиться в одном блоке, тогда средний сегмент состоит из одного блока. Если не учитывать обращения к блокам, которые требуются для просмотра таблицы сегментов, типичная операция поиска данных, основанного на ключах, потребует лишь одного обращения к блоку, а операции вставки, удаления или изменения потребуют двух обращений к блокам. Если среднее количество записей в сегменте намного превосходит количество записей, которые могут уместиться в одном блоке, можно периодически реорганизовывать таблицу сегментов, удваивая количество сегментов и деля каждый сегмент на две части.