Стек - особый вид списка, обращение к которому идет только через указатель на первый элемент. Если в стек нужно добавить элемент, то он добавляется впереди первого элемента, при этом указатель на начало стека переключается на новый элемент. Алгоритм работы со стеком характеризуется правилом: «последним пришел - первым вышел» (LIFO, last in - first out).

Очередь - это вид списка, имеющего два указателя на первый и последний элемент цепочки. Новые элементы записываются вслед за последним, а выборка элементов идет с первого. Этот алгоритм типа «первым пришел - первым вышел» (FIFO, first in - first out).

Дек или двусторонняя очередь - это линейный список, в котором все операции вставки и удаления (и, как правило, операции доступа к данным) выполняются на обоих концах списка.

Класс — это множество объектов, имеющих общую структуру и общее поведение. Класс — описание (абстракция), которое показывает, как построить существующую во време­ни и пространстве переменную этого класса, называемую объектом. Смысл предложений «описание переменных клас­са» и «описание объектов класса» один и тот же.

1.Инкапсуляция- объединение записей с процедурами и функциями.работающие с полями этих записей,которые формируют новый тип данных- объект.

2.Наследование-определение объекта и дальнейшее использование всех его свойств для построения иерархии порожденных объектов с возможностью для каждого порожденного объекта, относящимся к иерархии, доступа к коду и данным всех порождающих объектов.

3.Полиморфизм-присваивание определенному действию одного имени, которое затем совместно используется по всей иерархии объектов сверху донизу, причем каждый объект иерархии выполняет это действие характерным именно для него способом.

|  |  |  |
| --- | --- | --- |
| **Описание и инициализация динамических объектов.** Честно говоря, я хз, что тут можно написать. По-моему всё очевидно. Описание динамического объекта ничем не отличается от статического, а инициализация возможна 3 способами:  1) через конструктор  2) через внутренний метод (не конструктор)  3) через внешний метод Динамический объект – тот который создается через new в рантайме, а статический – описанный в var.  Динамические объекты обязаны содержать конструктор только в том случае, если они содержат виртуальные методы (т.к. в конструкторе неявно идет связывание с таблицами виртуальных методов, без которых обращение к этим самым методам невозможно).  Конструктор может быть параметрический (содержать аргументы):  Constructor Init(a1, a2 : integer);  Или простой (без параметров):  Constructor init\_t;  Тела конструкторов описываются вместе с процедурами для типа. В одном типе может быть несколько конструкторов, но они должны иметь разные имена.  При построении сложной иерархии объектов с наследованием имеет смысл вызывать конструктор базового класса в конструкторе предка (причем делать это с самого начала, до выполнения других операций!).  Constructor t1.Init(a, b : integer);  Begin  Inherited t0.Init(a); {вызов конструктора предка из потомка с передачей параметра} //другие действия  End;  При описании объектов нужно соблюдать ряд синтаксических правил, чтобы избежать ошибок этапа компиляции:  1)имена полей внутри иерархии объектов недолжны совпадать (т.е. недолжно быть полей с одинаковыми именами ни в рамках одного объекта, ни в рамках иерархии (т.е. имена полей потомка не должны совпадать с именами полей предка)).  2)имена параметров методов не должны совпадать с именами полей (ни в объекте, ни в наследниках)  3)при переопределении методов с ранним связыванием (compile time) эти методы могут иметь разные параметры. Тут должны совпадать только имена (даже возвращаемые типы могут не совпадать O\_o)  4)при переопределении методов с поздним связыванием (run time), сигнатуры этих методов (т.е. имя метода, его параметры и возвращаемое значение) должны совпадать полностью.  5) (просто замечание) при реализации тела функции, принадлежащей объекту для указания возвращаемого значения нельзя использовать имя типа. Только имя функции. Пример:  Function t1.add(x, y : integer) : integer;  Begin  t1.add := x + y; {ошибка. Нужно писать add := x + y;}  add := x + y;  End;  **Созание/удаление динамических объектов:**доступ к динамическим объектам осуществляется через указатель: Type  objPtr = ^t1; Var  Obj : objPtr; {указатель типа objPtr на объект типа t1}  Obj2 : ^t1; {это объявление эквивалентно предыдущему} Обратите внимание, что объявлять objPtr нужно только, если вы хотите передавать указатель на объект типа t1 в методы. Т.е. если у вас нет методов, принимающих указатели на объекты, то objPtr можно вообще не объявлять и обойтись ^t1 (ну по крайней мере у меня это работало =). Объявить указатель на объект не достаточно. Перед использованием объекта для него нужно сначала выделить память в куче. Как в случае структур за это отвечает операция new: New(obj2); Однако если объект содержит виртуальные методы – обязательно нужно вызвать конструктор для него. Вызов конструктора можно делать прямо в операции new, использую её расширенный синтаксис: New(obj2, Init(10,20)); {здесь Init – конструктор с двумя параметрами, объявленный в типе, которым является obj2 (т.е. в нашем случае в t1)} Ну и конечно конструктор может быть вызван отдельно от new. Деструкторы: деструкторы реализуются так же, как и конструкторы. Их задача – освобождать память, занимаемую динамическими полями объекта + при реализации виртуальных методов на них неявно ложится функционал по удалению объекта из хипа. Деструкторы часто рекомендуют делать виртуальными, чтобы всегда вызывать его последнюю версию (это резонно). Для статических объектов нет смысла делать деструкторы. Вызов деструктора должен обозначать конец работы с объектом. (тут важное замечание: для объектов, содержащих в.м. деструкторы всегда должны вызываться с расширенным синтаксисом dispose, как показано ниже. Вызов деструктора в отрыве от dispose не удалит объект из кучи. ) Dispose(obj1, obj1.Done); {вызов dispose с передачей деструктора} Итак: создание объекта – new с вызовом конструктора. Удаление dispose с вызовом деструктора. | | **Раннее и позднее связывание объекта с методом.** Прежде чем переходить к связыванию, нужно сказать пару слов о полиморфизме в целом. Полиморфизм (по Веренинову) – ‘возможность различным образом использовать методы с одинаковыми именами в иерархии типов’. Если перефразировать, то получится что-то вроде: полиморфизм – свойство я.п. , позволяющее одинаково обрабатывать данные разных типов. В Паскале полиморфизм позволяет осуществлять: 1)простое переопределение (с ранним связыванием) 2)переопределением виртуальных методов (с поздним связыванием) 3)вызов метода предка из потомка (я бы не стал включать это сюда, но так говорил В.) **Простое переопределение. Ранее связывание:**  Простое переопределение позволяет создать в классе-потомке метод с таким же именем, как и в базовом классе, но с другой реализацией. Список параметров при этом может отличаться, а может совпадать. Возвращаемое значение также может отличаться. Пример: Typet1 = object function add(x1, x2 : integer) : integer;  end;  t2 = object(t1) function add(x1, x2 : integer) : integer;  end;  function t1.add(x1, x2 : integer) : integer;  begin  add := x1 + x2;  end;  function t2.add(x1, x2 : integer) : integer; {метод в потомке имеет другую реализацию}  begin  add := x2 – x1;  end;  Теперь, если создать два объекта obj1 и obj2 являющиеся t1 и t2 соответственно, то можно проследить отличия при вызове процедуры add для каждого из объектов. Т.к. в t2 реализация отличается, то при вызове результат будет иным, нежели чем у obj1.add. Однако, если бы мы совершили сужающее преобразование ( ^t1 := ^t2 ) (при условии, что obj1 и obj2 будут динамическими, т.е. объявленными через указатели), то при вызове obj1^.add после этого всё равно должен выполняться код из t1.add, несмотря на то, что фактически мы привели к нему t2. Причиной этого является то, что раннее связывание происходит на этапе компиляции программы и таблицы виртуальных методов (ТВМ) не составляются.  **Позднее связывание:** Позднее связывание выполняется для виртуальных методов на этапе компиляции (при вызове конструктора). Суть такова: после вызова конструктора для объекта, реализующего виртуальные методы (при этом неважно – будет это статический, или динамический объект), происходит привязывание ТВМ к нему. Это позволяет вызывать виртуальные методы для инициализированного экземпляра класса. Кратко о ВМ: 1)у виртуальных методов при переопределении в наследниках должны полностью совпадать определения (т.е. название метода, возвращаемый тип, кол-во и типы параметров). 2)класс, реализующий ВМ, обязательно должен иметь конструктор (т.к. при вызове конструктора неявно происходит связывание ТВМ с экземпляром класса). Важная деталь: конструктор необязательно должен быть у всех классов в иерархии. В принципе он может быть только у базового., но вызов конструктора всё равно обязателен. 3)очевидно, что для использования объектов с виртуальными методами необходимо вызывать конструктор. Причем и для статических, и для динамических объектов. (существует расширенная форма записи new : New(obj, init);, сразу включающая вызов конструктора). 4)динамические объекты с ВМ должны реализовывать деструктор, т.к. он выполняет очистку объекта из heap. (очистка происходит только при вызове dispose в расширенной форме с передачей деструктора: dispose(obj1, done); и только для динамики с ВМ). Вызов деструктора отдельно от dispose не будет иметь эффекта.Пример ВМ:  Таким образом, виртуальные методов позволяет создать в классе-потомке метод, дублирующий метод класса-предка, но имеющий другую реализацию. (соответственно все вызовы продублированного метода будут обращаться к последней его реализации в иерархии. Тут много тонкостей, *но истина где-то рядом…*) |
| **Принципы поиска в неупорядоченном массиве.**  Надо сказать, формулировка весьма туманна. Он как-то не заострял на этом внимания.   1. Самым очевидным решением тут будет перебор всех элементов массива. Это дедовский брутфорс, зато реализуется без напряга. Очевидно, что такой подход крайне не оптимален при больших объемах данных (в худшем случае придется обойти все элементы массива). 2. Решение номер 2 – деревья поиска. По сути всё очевидно: создаем дерево поиска на основе данного массива и всё. Создание дерева требует дополнительных расходов памяти, зато увеличивает эффективность поиска в разы. Дело в том, что для дерева количество операций, требуемых для достижения любого элемента <= высоте дерева. Однако тут кроется одна маленькая деталь: эффективность работы дерева напрямую зависит от его балластировки. У сбалансированного дерева максимальная глубина его правого и левого поддеревьев отличается не более чем на 1. В худшем случае одного из поддеревьев может не быть совсем (nil), а все элементы будут содержаться во втором. Тогда дерево вырождается в список (и эффективность падает соответственно). Кол-во операций для достижения элемента будет <= кол-ву элементов. Короче, в худшем случае придется перебирать все элементы. | | **Задача конструктора в объектном типе. Механизм его работы.**  Я уже говорил о конструкторах (см. билет №5 и №3). Повторю основные моменты и уточню формулировки (Для более полного описания рекомендую заглянуть в вышеозначенные билеты):  1)Основная задача конструктора – назначение ТВМ (таблиц виртуальных методов) динамическому объекту. Конструктор делает это неявно для всех объектов содержащих ВМ. Нужно понимать, что сам конструктор не составляет ТВМ. Они делаются на этапе компиляции (по одной для каждого типа). Конструктор только заносит в экземпляр объекта адрес ТВМ.  2)Вторая задача – инициализация полей, однако конструктор может этого и не делать. Т.е. пустой конструктор – тоже конструктор.  Ну в принципе всё. Не знаю, что тут можно добавить. |
|  |  |  |
|  |

в readData использовал стандартные функции работы со строками:

Pos(subS, s) -возвращает индекс первого вхождения подстроки(subS) в строку(s).

возвращает индекс вхождения, или -1, если нет вхождений.

Copy(s, start, len) - возвращает подстроку длиной len, идущюю с позиции strat в строке s

Length(s) - длина строки

**uses**

Crt;

**type**

elPtr = ^element\_t;

element\_t = **record**

number : integer;   
 autor : string;   
name : string;

next, prev : elPtr;

**end**;

**var**

op : text;

inp : text;

pHead : elPtr;

{вставка элемента в список по номеру}

**procedure** insertElement(head : elPtr; number : integer; name, autor : string);

**var**

newEl : elPtr; {создаваемый элемент}

**begin**

{проход по списку, поиск места вставки}

**while** head^.next^.next <> nil **do**

**begin**

{если нашли место вставки - прервать цикл (head указывает на нужное место)}

**if** head^.next^.number > number **then break**;

head := head^.next;

**end**;

{после выполнения предыдущих действий head указывает на место вставки}

**new**(newEl); {создать элемент}

newEl^.next := head^.next; {связхь с следующим элементом}

head^.next^.prev := newEl;

newEl^.prev := head; {и с текущим}

head^.next := newEl;

newEl^.number := number; {заполнение полей (капитан очевидность!)}

newEl^.name := name;

newEl^.autor := autor;

**end**;

{P1}

{чтение данных из файла, составление списка с сортировкой}

**procedure** readData(head : elPtr);

**var**

num : integer;

str : string;

chr : char;

name, autor : string;

commaIndex : integer; {индекс запятой в строке str}

**begin**

**while not** eof(inp) **do**

**begin**

readLn(inp, num, chr, chr, str); {прочитать строку и счисло}

{в середине 2 лишних сммвола -это пробел и запятая}

{т.е. в num - номер, а в str - строка с названием и автором (разделенными одной запятой)}

commaIndex := Pos(',' , str); {получить индекс запятой между именем и названием}

autor := Copy(str, 1, commaIndex - 1); {извлеч подстроку с автором}

name := Copy(str, commaIndex + 1, Length(str) - commaIndex); {извлеч подстроку с именем книги}

insertElement(head, num, name, autor); {вставить элемент в список}

**end**;

**end**;

{P2}

{читает введённую с клавы записть для списка}

**procedure** readRecord(head : elPtr);

**var**

num : integer;

str : string;

chr : char;

name, autor : string;

commaIndex : integer; {индекс запятой в строке str}

**begin**

readLn(num, chr, chr, str); {прочитать записть с клавиатуры}

{анализ данных:}

commaIndex := Pos(',' , str); {получить индекс запятой между именем и названием}

autor := Copy(str, 1, commaIndex - 1); {извлеч подстроку с автором}

name := Copy(str, commaIndex + 1, Length(str) - commaIndex); {извлеч подстроку с именем книги}

insertElement(head, num, name, autor); {вставить элемент в список}

**end**;

{P3}

{вывод списка в текстовый файл}

**procedure** list\_print(head : elPtr; direction : boolean); {direction - задает направление}

**begin**

head := head^.next;

{если требуют проход с конца - прейти в конец}

**if not** (direction) **then**

**while** head^.next^.next <> nil **do**

head := head^.next;

**while** true **do**

**begin**

{проверка условия выхода из цикла в зависимости от направления прохода}

**if** direction **and** (head^.next = nil) **then break**;

**if not** (direction) **and** (head^.prev = nil) **then break**;

writeLn(op, head^.number, ' ,', head^.autor, ' ,', head^.name);

{выбор направления прохода}

**if** direction **then**

**begin**

head := head^.next;

**end**

**else begin**

head := head^.prev;

**end**;

**end**;

writeLn(op);

**end**;

{P4}

{вывод всех книг заданного автора}

**procedure** list\_printAutor(head : elPtr; autor : string);

**begin**

head := head^.next;

**while**(head^.next <> nil) **do**

**begin**

{если у очередной книги автор совпал с требуемым - вывести}

**if** head^.autor = autor **then**

writeLn(op, head^.number, ' ,', head^.autor, ' ,' , head^.name);

head := head^.next;

**end**;

**end**;

**begin**

ClrScr; {очистить экран вывода (это из Crt)}

assign(inp, 'dan1.inp'); assign(op, 'res1.out');

reset(inp); rewrite(op);

{создать головной и хвостовой элементы (не созеержат значения. список между ними)}

**new**(pHead);

**new**(pHead^.next);

pHead^.prev := nil;

pHead^.next^.next := nil;

pHead^.next^.prev := pHead;

readData(pHead); {прочитать данный список книг из файла}

readRecord(pHead); {ввод с клавиатуры записи}

list\_print(pHead, true); {вывести список прямым порядком}

list\_print(pHead, false); {и обратным}

list\_printAutor(pHead, ' aaaa '); {вывести книги автора aaaa (збс имя =)}

close(inp);

close(op);

**end**.

**type**

elPtr = ^element\_t; {тип указателя на элемент списка}

{элемент двусвяхного списка}

element\_t = **record**

number : integer; {номер}

name : string; {имя}

next, prev : elPtr;

**end**;

**var**

pHead : elPtr; {указатель на головной элемент списка}

pNumList, pNameList : elPtr; {указатель на односвязные списки}

{P1}

{чтение данных из файла, составление списка}

{список имеет 1 пустой элемент в начале. в конце дополнительных элементов нет}

**procedure** createList(**var** head : elPtr);

**var**

curr : elPtr;

**begin**

**new**(head);head^.prev := nil;

curr := head;

**while not** eof **do**

**begin**

**new** (curr^.next);

curr^.next^.prev := curr;

curr := curr^.next;

readLn(curr^.number, curr^.name);

**end**;

curr^.next := nil;

**end**;

{P2}

{выводит список в заданный текстовый файл}

{clearFile определяет - перезаписывать файл, или дописывать в конец}

**procedure** printList(head : elPtr; direction, clearFile : boolean; fileName : string);

**var**

OpenFile : text;

**begin**

assign(openFile, fileName);

{выбор - дописать файл, или перезаписать его}

**if** clearFile **then**

**begin**

rewrite(openFile);

**end**

**else begin**

append(openFile);

**end**;

head := head^.next;

{если требуют проход с конца - прейти в конец}

**if not** (direction) **then**

**while** head^.next <> nil **do**

head := head^.next;

**while** true **do**

**begin**

{проверка условия выхода из цикла в зависимости от направления прохода}

**if** direction **and** (head = nil) **then break**;

**if not** (direction) **and** (head^.prev = nil) **then break**;

**if** length(head^.name) > 0 **then** {если есть имя (через эту процедуру выводится и список номеров)}

**begin**

writeLn(openFile, head^.number, ' ', head^.name);

**end**

**else begin** {если нет имяни - вывести только номер}

writeLn(openFile, head^.number);

**end**;

{выбор направления прохода}

**if** direction **then**

**begin**

head := head^.next;

**end**

**else begin**

head := head^.prev;

**end**;

**end**;

writeLn(openFile);

close(openFile);

**end**;

{P3}

{формирует два односвязных списка. один сортирован по алфвиту, другой по номерам}

{элменты обоих список - element\_t, просто указател prev = nil}

**procedure** createSubLists(head : elPtr; **var** numList : elPtr; **var** nameList : elPtr);

**var**

curr : elPtr; {используется для прохода по создаваемым спискам}

currL : elPtr; {используется для прохода по основному списку (доступ к элементам)}

newEl : elPtr; {для создания элементов}

**begin**

**new**(numList);

numList^.prev := nil;

numList^.next := nil;

**new**(nameList);

nameList^.next := nil;

nameList^.prev := nil;

currL := head^.next; {первый значащий элемент}

{проход по основному списку, разбор на список номеров}

**while** currL <> nil **do**

**begin**

curr := numList; {поиск места вставки каждый раз идёт с нуля}

{поиск места вставки в списке номеров}

**while** curr <> nil **do** {можно было while tru}

**begin**

**if** (curr^.next^.number > currL^.number) **or** {если нашли место вставки}

(curr^.next = nil) **then** {или дошли до конца писка} **begin**

{создать элемент}

**new**(newEl);

newEl^.next := curr^.next; {установить связи}

newEl^.prev := nil;

curr^.next := newEl;

newEl^.number := currL^.number;

**break**; {перейти к след. элеемнту из списка head}

**end**;

curr := curr^.next;

**end**;

currL := currL^.next;

**end**;

currL := head^.next; {начать проход сначала (создается второй список)}

**while** currL <> nil **do**

**begin**

curr := nameList; {поиск места вставки каждый раз идёт с нуля}

{поиск места вставки в списке номеров}

**while** curr <> nil **do** {можно было while tru}

**begin**

**if** (curr^.next^.name > currL^.name) **or** {если нашли место вставки}

(curr^.next = nil) **then** {или дошли до конца писка}

**begin**

{создать элемент}

**new**(newEl);

newEl^.next := curr^.next; {установить связи}

newEl^.prev := nil;

curr^.next := newEl;

newEl^.name := currL^.name;

**break**; {перейти к след. элеемнту из списка head}

**end**;

curr := curr^.next;

**end**;

currL := currL^.next;

**end**;

**end**;

**begin**

assign(input, 'dan1.inp'); reset(input);

createList(pHead); {создать список из файла}

printList(pHead, true, true, 'res1.out'); {вывести в прямом направлении}

printList(pHead, false, false, 'res1.out'); {вывести в обратном}

createSubLists(pHead, pNumList, pNameList); {создать односвязные pNumList и pNameList}

printList(pNumList, true, false, 'res1.out'); {вывести номера}

printList(pNameList, true, false, 'res1.out'); {вывести имена}

close(input);

**end**.

**type**

nodePtr = ^node\_t;

node\_t = **record**

name : string[10]; {фамилия}

number : integer;

pRIght, pLeft : nodePtr;

**end**;

{вставляет полученный элеемнт в дерево (рекурсивно)}

**procedure** addToTree(root : nodePtr; node : nodePtr);

**begin**

**if** node^.number > root^.number **then begin**

**if** root^.pRight <> nil **then begin**

addToTree(root^.pRight, node);

**end**

**else begin**

root^.pRight := node;

**end**;

**end**

**else begin**

**if** root^.pLeft <> nil **then begin**

addToTree(root^.pLeft, node);

**end**

**else begin**

root^.pLeft := node;

**end**; **end**;**end**;  
{p1}

{создание деревa из файла}

**procedure** createTree(**var** root : nodePtr);

**var**

node : nodePtr;

**begin**

**new**(root);

root^.pRight := nil;

root^.pLeft := nil;

read(root^.number);

readLn(root^.name);

**while not** eof **do begin**

**new**(node);

node^.pRight := nil;

node^.pLeft := nil;

read(node^.number);

readLn(node^.name);

addToTree(root, node);

**end**;**end**;

{p2}

{вывод дерево по правилу левое-корень-правое}

**procedure** printTree(root : nodePtr);

**begin**

**if** root^.pLeft <> nil **then**

printTree(root^.pLeft);

writeLn(root^.number, ' ', root^.name);

**if** root^.pRight <> nil **then**

printTree(root^.pRight);

**end**;

{p3}

{вывод листьев дерева справа налево}

**procedure** printLeaf(root : nodePtr);

**begin**

**if** root^.pRight <> nil **then**

printLeaf(root^.pRight);

**if** root^.pLeft <> nil **then**

printLeaf(root^.pLeft);

{если оба потомка отсутствуют - это лист. вывести}

**if** (root^.pLeft = nil) **and** (root^.pRight = nil) **then**

writeLn(root^.number, ' \_ ', root^.name);

**end**;

**var**

pRoot : nodePtr;

**begin**

assign(input, 'dan1.inp');

assign(output, 'res1.out');

reset(input);

rewrite(output);

createTree(pRoot); {создать дерево из файла}

printTree(pRoot); {вывести дерево}

writeLn;

printLeaf(pRoot); {вывести листья справа налево}

close(input);

close(output);

**end**.

**program** bil19;

**uses**

New\_crt;

**type**

pbook = ^tbook;

tbook = **record**

next, pred: pbook; nomer: integer; avtor: string[10]; name : string[10]; year : integer;

chena : integer; **end**;  
**var** book, beglist: pbook;

rel: pointer;

fbooks: **file of** tbook;

input, output: text;

**procedure** sort(**var** beglist: pbook);

**var** stop, first: boolean;

book1, book2: pbook;

**begin repeat**

stop := true; book := beglist;

first := true;  
 **while** book^.next <> nil **do**

**begin**

**if** book^.nomer > book^.next^.nomer **then**

**begin**

book1 := book;

book := book^.next;

**if** first **then**

beglist := book **else**

**begin**

book2 := beglist;

**while** book2^.next <> book1 **do**

book2 := book2^.next;

book2^.next := book

**end**;

book1^.next := book^.next;

book^.next := book1;

stop := false;

**end**;

first :=false;

book := book^.next

**end**

**until** stop;**end**;

**procedure** p1(**var** beglist: pbook);

**var** book1: pbook;

**begin** assign(input, 'input.txt');

reset(input); beglist := nil;

**while not** eof(input) **do**

**begin**

book1 := book;

{ sort(beglist);

} **new**(book);

**with** book^ **do**

**begin**

next := nil;

pred := nil;

**if** beglist = nil **then**

beglist := book

**else**

book1^.next := book;

book1^.pred := ;

readln(input, nomer, avtor, name, year, chena);

**end**;

**end**;

close(input)

**end**;  
**procedure** p3(**var** beglist: pbook);

**begin**

assign(output, 'output.txt');

rewrite(output);

writeln(output);

book := beglist;

**while** book <> nil **do begin**

**with** book^ **do**

writeln(output, nomer:2, avtor:10, name:10, year:5, chena:5);

book := book^.next;

**end**;

Writeln(output,'\*\*\*\*\*');

**while** book <> **do begin**

**with** book^ **do**

writeln(output, nomer:2, avtor:10, name:10, year:5, chena:5);

book := book^. ;

**end**;

close(output);

**end**;

**begin**

clrscr; assign(input, 'input.txt');

reset(input); p1(beglist);

p3(beglist);

**end**.

**type**

nodePtr = ^node\_t;

elPtr = ^element\_t;

{узел/лист дерева}

node\_t = **record** age : integer; {возраст}

count : integer; {кол-во сотрудников данного возраста}

pLeft, pRight : nodePtr;**end**;{элемент обходного списка}

element\_t = **record**

pNode : nodePtr; {указатель на узел, которому принадлежит} depth : integer; {глубина этого узла в дереве} pPrev, pNext : elPtr; {список двусвязный}

**end**;{добавление элемента в дерево}

**procedure** addElement(root : nodePtr; age : integer);

**begin**{проход по дереву, поиск места вставки}

**while** true **do begin**если > текущего - переход в правое поддерево}i**f** age > root^.age **then**

**begin**

**if** root^.pRight <> nil **then** {если есть, уда переходить}

**begin**

root := root^.pRight;

**end else begin** {иначе - создать лист}

**new**(root^.pRight);

root := root^.pRight;

root^.pLeft := nil;

root^.pRight := nil;

root^.age := age;

root^.count := 1 **break**; {выход из цикла}

**end**; **end**

**else if** age < root^.age **then** {если < текущего - переход в левое поддерево}

**begin**

**if** root^.pLeft <> nil **then** {если есть, уда переходить}

**begin** root := root^.pLeft;

**end else begin** {иначе - создать лист}

**new**(root^.pLeft);

root := root^.pLeft;

root^.pLeft := nil;

root^.pRight := nil;

root^.age := age;

root^.count := 1; **break**; {выход из цикла}

**end**;

**end**

**else begin** {если = текущему - увеличить счетчик}

root^.count := root^.count + 1;

**break**; {выход из цикла}

**end**; **end**;**end**;

{P1}

{итеративное создание дерева}

**procedure** createTree(**var** root : nodePtr);

**var**

cPos : byte; {позиция запятой в строке}

age, err : integer; {значение возраста, код ошибки для парсера}

str : string;

**begin**

**new**(root); root^.pLeft := nil; root^.pRight := nil;

{читаем первый элемент}

readLn(str); {исходная строка}

cPos := Pos(',', str); {позиция запятой, отделяющий имя от возвраста}

str := copy(str, cPos + 1, Length(str) - cPos); {взять подстроку, содержащую только цифру возраста}

Val(str , root^.age, err); {строка в число}

root^.count := 1;

{читаем остальные}

**while not** eof **do**

**begin**

readLn(str); {исходная строка}

cPos := Pos(',', str); {позиция запятой, отделяющий имя от возвраста}

str := copy(str, cPos + 1, Length(str) - cPos); {взять подстроку, содержащую только цифру возраста}

Val(str , age, err); {перевести в число}

**if** err = 0 **then** {если прочитали число без ошибок}

**begin**

addElement(root, age); {вставка в дерево}

**end**

**else begin**

writeLn('readErr: ' , err, '; str:', str) {если запарсили с ошибкой - вывести}

**end**; **end**;**end**;

**procedure** passList\_print(head :elPtr);

**begin**

head := head^.pNext; {пропустить головной (не значащий)} **while** head^.pNext<> nil **do**

**begin**

writeLn('age: ' , head^.pNode^.age, ' count: ', head^.pNode^.count);

head := head^.pNext;

**end**;

writeLn;

**end**;

{создает головной и замыкающий элементы обходного списка (не значащие)}

**procedure** passList\_create(**var** head : elPtr);

**begin**

**new**(head);

head^.pNode := nil;

head^.pPrev := nil;

**new**(head^.pNext);

head^.pNext^.pNode := nil;

head^.pNext^.pPrev := head;

head^.pNext^.pNext := nil;

**end**;

{вставка в обходной список элемента (слева или справа от текущего)}

**procedure** passList\_insert(curr : elPtr; node : nodePtr; depth : integer; pos : boolean);

**var**

el : elPtr;

**begin**

**if** pos **then** {вставка справа от текущего}

**begin**

**new**(el);

el^.pNode := node;

el^.depth := depth;

el^.pNext := curr^.pNext;

curr^.pNext^.pPrev := el;

el^.pPrev := curr;

curr^.pNext := el;

**end**

**else begin** {слева от текущего}

**new**(el);

el^.pNode := node;

el^.depth := depth;

el^.pPrev := curr^.pPrev;

curr^.pPrev^.pNext := el;

el^.pNext := curr;

curr^.pPrev := el;

**end**;**end**;

{обход дерева интерациями. соствление обходного списка}

**function** getPassList(root : nodePtr) : elPtr;

**var**

desCount : integer; {число потомков на след. уровне (если = 0, цикл прекращается)}

passList : elPtr; {головной элемент (незначащий) списка обхода}

curr\_list : elPtr; {для обхода списка}

curr\_tree : nodePtr; {для обхода дерева}

currDepth : integer; {текущая глубина. используется дли идентификации элементов,

которые будут рассматириватьсяна текущей итерации}

**begin**

desCount := 2; {у корня 2 потомка (может быть и меньше, но сейчас это не важно)}

currDepth := -1;

passList\_create(passList); {создать служебные элементы списка обхода}

passList\_insert(passList, root, 0, true); {вставить головной элемент в список}

{обход дерева в ширину}

**while** desCount <> 0 **do**

**begin** desCount := 0; currDepth := currDepth + 1; {переходим на новый уровень в дереве}

curr\_list := passList^.pNext; {каждй раз проходим весть список с головы}проход по списку, обработка элементов, стоящих на текущем уровне}

**while** curr\_list^.pNext <> nil **do**

**begin**

**if** curr\_list^.depth = currDepth **then** {если элемент стоит на текущем уровне - анализ}

**begin**

curr\_tree := curr\_list^.pNode;

{занести в список всех его потомков}

**if** curr\_tree^.pLeft <> nil **then**

**begin**

passList\_insert(curr\_list, curr\_tree^.pLeft, currDepth + 1, false); {вставить слева}

desCount := desCount + 1; {есть 1 потомок на уровне} **end**;

**if** curr\_tree^.pRight <> nil **then**

**begin** passList\_insert(curr\_list, curr\_tree^.pRight, currDepth + 1, true); {вставить справа} desCount := desCount + 1; {есть 1 потомок на уровне} **end**; **end**;

**program** bilet17;

**type**

p=^zap;

zap=**record**

f:string;

sled,pred:p;

**end**;

**procedure** p1(**var** uk1:p; **var** file1:text);

**var** uk,uk2:p;

**begin**

**new**(uk1);

uk:=uk1;

uk2:=uk1;

readln(file1,uk^.f);

uk^.sled:=nil;

uk^.pred:=nil;

**while not** eof(file1) **do**

**begin**

**new**(uk^.sled);

uk:=uk^.sled;

readln(file1,uk^.f);

uk^.sled:=nil;

uk^.pred:=uk2;

uk2:=uk;

**end**;

**end**;

**procedure** p4(**var** uk1,uk2:p);

**var** walk1,walk11,walk2,uk:p;

**begin**

walk2:=uk2;

**repeat**

walk1:=uk1;

walk11:=uk1^.sled;

**new**(uk);

uk^.sled:=nil;uk^.pred:=nil;

uk^.f:=walk2^.f;

**if** (walk1^.pred=nil)

**and**

(length(walk1^.f)>=length(uk^.f)) **then**

**begin**

uk^.sled:=walk1;walk1^.pred:=uk;

uk1:=uk;

**end**

**else**

**begin**

**while** walk11<>nil **do begin**

**if** (walk11^.sled=nil)**and**

(length(walk11^.f)<=length(uk^.f)) **then**

**begin** walk11^.sled:=uk;

uk^.pred:=walk11; **break**;

**end**

**else begin**

**if** (length(walk1^.f)<=length(uk^.f))

**and**

(length(walk11^.f)>=length(uk^.f)) **then**

**begin**

walk1^.sled:=uk;

uk^.sled:=walk11;

uk^.pred:=walk1;

walk11^.pred:=uk;**break**;

**end**;

**end**;

walk1:=walk1^.sled;

walk11:=walk11^.sled;

**end**;**end**;

walk2:=walk2^.sled;

**until** walk2=nil;

**end**;

**procedure** p5(uk1:p; **var** file2:text);

**begin**

**while** (uk1<>nil) **do**

**begin**

writeln(file2,uk1^.f);

uk1:=uk1^.sled;

**end**;**end**;

**var** ukaz1,ukaz2:p;

f1,f2,f3:text;

**begin**

assign(f1,'A.inp')assign(f2,'B.inp');

reset(f1); reset(f2);

assign(f3,'C.out');

rewrite(f3);

p1(ukaz1,f1);p1(ukaz2,f2);

p4(ukaz1,ukaz2);p5(ukaz1,f3);

close(f1);close(f2);

close(f3);

**end**.

{1) procedure AddElem(var spis1:List;znach1:TInf);

2) procedure Print(spis1:List);

3) Procedure FreeStek(spis1:List);

4) Function SearchElemZnach(spis1:List;znach1:TInf):List;

5) Procedure DelElem(var spis1:List;tmp:List);

6) procedure DelElemZnach(var Spis1:List;znach1:TInf);

7) Procedure DelElemPos(var spis1:List;posi:integer);

8) procedure SortBublInf(nach:list);

9) procedure SortBublLink(nach:List);

10) Procedure Swap(var nach,ends:List;a,b,c:integer); - Берёт часть списка и вставляет в нужную позиицию

11) procedure AddElemVst(var nach,ends:List;znach1:TInf); - Процедура добавления нового элемента в двунаправленный список методом "вставок"}

**Program** Spisok\_dn;

**uses**

crt; {Для использования readkey и clrscr}

**type**

Tinf=integer; {тип данных, который будет храниться в элементе списка}

List=^TList; {Указатель на элемент типа TList}

TList=**record** {А это наименование нашего типа "запись" обычно динамические структуры описываются через запись}

data:TInf; {данные, хранимые в элементе}

next, {указатель на следующий элемент списка}

prev:List; {указатель на предыдущий элемент списка}

**end**;

{Процедура добавления нового элемента в двунаправленный список}

**procedure** AddElem(**var** nach,ends:List;znach1:TInf);

**begin**

**if** nach=nil **then** {не пуст ли список, если пуст, то}

**begin**

Getmem(nach,**SizeOf**(TList)); {создаём элемент, указатель nach уже будет иметь адрес}

nach^.next:=nil; {никогда не забываем "занулять" указатели}

nach^.prev:=nil; {аналогично}

ends:=nach; {изменяем указатель конца списка}

**end**

**else** {если список не пуст}

**begin**

GetMem(ends^.next,**SizeOf**(Tlist)); {создаём новый элемент}

ends^.next^.prev:=ends; {связь нового элемента с последним элементом списка}

ends:=ends^.next;{конец списка изменился и мы указатель "переставляем"}

ends^.next:=nil; {не забываем "занулять" указатели}

**end**;

ends^.data:=znach1; {заносим данные}**end**;

**procedure** AddElemVst(**var** nach:List; **var** ends:List;znach1:TInf);

**var**

tmp,tmpL:List;

flag:boolean;

**begin**

**if** nach=nil **then** {не пуст ли список, если пуст, то}

**begin**

Getmem(nach,**SizeOf**(TList)); {создаём элемент, указатель nach уже будет иметь адрес}

nach^.next:=nil; {никогда не забываем "занулять" указатели}

nach^.prev:=nil; {аналогично}

ends:=nach; {изменяем указатель конца списка}

tmp:=nach;

**end**

**else** {если список не пуст}

**begin**

tmpl:=nach;

flag:=true;

**while** tmpl^.data<znach1 **do**

**begin**

tmpl:=tmpl^.next;

**if** tmpl=nil **then**

**break**;

**end**;

**if** tmpl<>nil **then**

**begin**

tmpl:=tmpl^.prev;

**if** tmpl=nach **then**

flag:=false;

**if** tmpl=nil **then**

tmpl:=nach;

**end**

**else**

**begin**

tmpl:=ends;

flag:=false

**end**;

GetMem(tmp,**SizeOf**(TList));

**if** ends=nach **then**

**if** ends^.data<znach1 **then**

**begin**

ends^.next:=tmp;

tmp^.prev:=ends;

tmp^.next:=nil;

ends:=tmp;

**end**

**else**

**begin**

nach^.prev:=tmp;

tmp^.prev:=nil;

tmp^.next:=nach;

nach:=tmp;

**end**

**else**

**if** (tmpl=nach) **and** (flag) **then**

**begin**

nach^.prev:=tmp;

tmp^.prev:=nil;

tmp^.next:=nach;

nach:=tmp;

**end**

**else**

**if** (tmpl=nach) **and not**(flag) **then**

**begin**

tmp^.next:=nach^.next;

nach^.next^.prev:=tmp;

tmp^.prev:=nach;

nach^.next:=tmp;

**end**

**else**

**if** (tmpL=ends) **and not** (flag) **then**

**begin**

ends^.next:=tmp;

tmp^.prev:=ends;

tmp^.next:=nil;

ends:=tmp;

**end**

**else**

**begin**

tmp^.next:=tmpl^.next;

tmpl^.next^.prev:=tmp;

tmp^.prev:=tmpl;

tmpl^.next:=tmp;

**end**;

**end**;

tmp^.data:=znach1; {заносим данные}

**end**;

{процедура печати списка

полностью расписана при работе со стеком}

**procedure** Print(spis1:List);

**begin**

**if** spis1=nil **then**

**begin**

writeln('‘ЇЁб®Є Їгбв.');

**exit**;

**end**;

**while** spis1<>nil **do**

**begin**

Write(spis1^.data, ' ');

spis1:=spis1^.next

**end**;

**end**;

{процедура удаления списка

полностью расписана при работе со стеком}

**Procedure** FreeStek(spis1:List);

**var**

tmp:List;

**begin**

**while** spis1<>nil **do**

**begin**

tmp:=spis1;

spis1:=spis1^.next;

FreeMem(tmp,**SizeOf**(Tlist));

**end**;

**end**;

{Функция поиска в списке

полностью расписана при работе со стеком}

**Function** SearchElemZnach(spis1:List;znach1:TInf):List;

**begin**

**if** spis1<>nil **then**

**while** (Spis1<>nil) **and** (znach1<>spis1^.data) **do**

spis1:=spis1^.next;

SearchElemZnach:=spis1;

**end**;

{процедура удаления элемента в двунаправленном списке}

**Procedure** DelElem(**var** spis1,spis2:List;tmp:List);

**var**

tmpi:List;

**begin**

**if** (spis1=nil) **or** (tmp=nil) **then**

**exit**;

**if** tmp=spis1 **then** {если удаляемый элемент первый в списке, то}

**begin**

spis1:=tmp^.next; {указатель на первый элемент переставляем на следующий элемент списка}

**if** spis1<>nil **then** {если список оказался не из одного элемента, то}

spis1^.prev:=nil {"зануляем" указатель}

**else** {в случае, если элемент был один, то}

spis2:=nil; {"зануляем" указатель конца списка, а указатель начала уже "занулён"}

FreeMem(tmp,**SizeOf**(TList));

**end**

**else**

**if** tmp=spis2 **then** {если удаляемый элемент оказался последним элементом списка}

**begin**

spis2:=spis2^.prev; {указатель конца списка переставляем на предыдущий элемент}

**if** spis2<>nil **then** {если предыдущий элемент существует,то}

spis2^.next:=nil {"зануляем" указатель}

**else** {в случае, если элемент был один в списке, то}

spis1:=nil; {"зануляем" указатель на начало списка}

FreeMem(tmp,**SizeOf**(TList));

**end**

**else** {если же удаляется список не из начали и не из конца, то}

**begin**

tmpi:=spis1;

**while** tmpi^.next<>tmp **do** {ставим указатель tmpi на элемент перед удаляемым}

tmpi:=tmpi^.next;

tmpi^.next:=tmp^.next; {меняем связи}

**if** tmp^.next<>nil **then**

tmp^.next^.prev:=tmpi; {у элемента до удаляемого и после него}

FreeMem(tmp,**sizeof**(TList));

**end**;

**end**;

{процедура удаления элемента по значению

полностью расписана при работе со стеком}

**procedure** DelElemZnach(**var** Spis1,spis2:List;znach1:TInf);

**var**

tmp:List;

**begin**

**if** Spis1=nil **then**

**begin**

Writeln('‘ЇЁб®Є Їгбв');

**exit**;

**end**;

tmp:=SearchElemZnach(spis1,znach1);

**if** tmp=nil **then**

**begin**

writeln('ќ«Ґ¬Ґ\*в б ЁбЄ®¬л¬ §\*\*зҐ\*ЁҐ¬ ' ,znach1, ' ®вбгвбвўгҐв ў бЇЁбЄҐ.');

**exit**;

**end**;

DelElem(spis1,spis2,tmp);

Writeln('ќ«Ґ¬Ґ\*в г¤\*«с\*.');

**end**;

{процедура удаления элемента по позиции

полностью расписана при работе со стеком}

**Procedure** DelElemPos(**var** spis1,spis2:List;posi:integer);

**var**

i:integer;

tmp:List;

**begin**

**if** posi<1 **then**

**exit**;

**if** spis1=nil **then**

**begin**

Write('‘ЇЁб®Є Їгбв');

**exit**

**end**;

i:=1;

tmp:=spis1;

**while** (tmp<>nil) **and** (i<>posi) **do**

**begin**

tmp:=tmp^.next;

inc(i)

**end**;

**if** tmp=nil **then**

**begin**

Writeln('ќ«Ґ¬Ґ\*в\* б Ї®ап¤Є®ўл¬ \*®¬Ґа®¬ ' ,posi, ' \*Ґв ў бЇЁбЄҐ.');

writeln('‚ бЇЁбЄҐ ўбҐЈ® ' ,i-1, ' н«Ґ¬Ґ\*в®ў.');

**exit**

**end**;

DelElem(spis1,spis2,tmp);

Writeln('ќ«Ґ¬Ґ\*в г¤\*«с\*.');

**end**;

{Процедура сортировки "пузырьком" с изменением только данных

полностью расписана при работе со стеком}

**procedure** SortBublInf(nach:list);

**var**

tmp,rab:List;

tmps:Tinf;

**begin**

GetMem(tmp,**SizeOf**(Tlist));

rab:=nach;

**while** rab<>nil **do**

**begin**

tmp:=rab^.next;

**while** tmp<>nil **do**

**begin**

**if** tmp^.data<rab^.data **then**

**begin**

tmps:=tmp^.data;

tmp^.data:=rab^.data;

rab^.data:=tmps

**end**;

tmp:=tmp^.next

**end**;

rab:=rab^.next

**end**

**end**;

{Процедура сортировки "пузырьком" с изменением только адресов}

{Чтобы разобраться как она работает Вам точно понадобится листок

с рисунком списка и связями между элементами. Внимательно следите

за тем, что происходит в процедуре, описывать всё я не вижу смысла}

**procedure** SortBublLink(**var** nach,ends:List);

**var**

tmp,pocle1,rab:List;

**begin**

rab:=nach;

**while** rab<>nil **do**

**begin**

tmp:=rab^.next;

**while** tmp<>nil **do**

**begin**

**if** tmp^.data<rab^.data **then**

**begin**

pocle1:=tmp^.next;

**if** rab^.next=tmp **then**

**begin**

**if** tmp^.next<>nil **then**

tmp^.next^.prev:=rab;

tmp^.next:=rab;

tmp^.prev:=rab^.prev;

**if** tmp^.prev<>nil **then**

tmp^.prev^.next:=tmp;

rab^.prev:=tmp;

rab^.next:=pocle1;

**end**

**else**

**begin**

**if** rab^.prev<>nil **then**

rab^.prev^.next:=tmp;

tmp^.prev^.next:=rab;

**if** tmp^.next<>nil **then**

tmp^.next^.prev:=rab;

rab^.next^.prev:=tmp;

tmp^.next:=rab^.next;

rab^.next:=pocle1;

pocle1:=rab^.prev;

rab^.prev:=tmp^.prev;

tmp^.prev:=pocle1;

**end**;

**if** rab=nach **then**

**begin**

nach:=tmp;

nach^.prev:=nil;

**end**;

**if** tmp=ends **then**

**begin**

ends:=rab;

ends^.next:=nil

**end**;

pocle1:=rab;

rab:=tmp;

tmp:=pocle1;

**end**;

tmp:=tmp^.next;

**end**;

rab:=rab^.next;

**end**;

**end**;

{процедура берёт часть списка и вставляет в нужную позицию списка}

**Procedure** Swap(**var** nach,ends:List;a,b,c:integer); {а-начало части, b-конец части, с-позиция}

**var**

i:integer;

yk,yk1,yk2,rab:List;

**begin**

rab:=nach;

i:=0;

**while** rab<>nil **do** {цикл определяет количество элементов}

**begin**

inc(i);

rab:=rab^.next

**end**;

{проверка на "нормальность" введённых данных}

**if** (i+1<a) **or** (i+1<b) **or** (i+1<c) **or** ((c>=a) **and** (c<=b)) **then**

**exit**;

**if** a>b **then** {если "ошибочно" спутаны начало и конец части, то}

**begin**

a:=a **xor** b; {произведём замену переменных, т.е.}

b:=b **xor** a; {значение из a поместим в b}

a:=b **xor** a;{из b поместим в a}

**end**;

yk:=nach;

**for** i:=1 **to** a-1 **do**

yk:=yk^.next; {ставим указатель на нужный нам элемент}

yk1:=nach;

**for** i:=1 **to** b-1 **do**

yk1:=yk1^.next;{ставим указатель на нужный нам элемент}

yk2:=nach;

**for** i:=1 **to** c-1 **do**

yk2:=yk2^.next;{ставим указатель на нужный нам элемент}

**if** yk=nach **then** {проверяем следует ли нам изменить начало списка}

**begin**

nach:=yk1^.next; {изменяем начало}

nach^.prev:=nil; {"зануляем" указатель}

**end**

**else** {далее я советую смотреть по листочку}

**begin**

**if** yk1^.next<>nil **then**

yk1^.next^.prev:=yk^.prev

**else**

**begin**

ends:=yk^.prev;

yk^.prev^.next:=nil;

**end**;

yk^.prev^.next:=yk1^.next;

**end**;

**if** yk2^.next=nil **then**

**begin**

yk1^.next:=nil;

ends:=yk1;

**end**

**else**

**begin**

yk2^.next^.prev:=yk1;

yk1^.next:=yk2^.next;;

**end**;

yk2^.next:=yk;

yk^.prev:=yk2;

**end**;

**var**

SpisNach, {указатель на начало списка и}

SpisEnd, {указатель на конец списка. Эти два указателя }

tmpl:List; {неотъемлимая часть в двунаправленном списке}

znach,a,b:integer;

ch:char;

{1)procedure AddToTree (var Tree:PNode;x:integer); - добавление элемента в дерево

2)function Search(Tree:PNode;x:integer):PNode; - функция поиска в дереве

3)procedure Lkp(Tree:PNode); - обход дерева по принципу "Левле поддерево, корень, правое поддерево"

4)procedure DeleteTree(var Tree1:PNode ) - процедура удаления дерева}

**uses** crt;

**type**

PNode=^Node; {Указатель на узел}

Node=**record** {Тип запись в котором будет храниться информация}

data:integer;

left,right:PNode; {Ссылки на левого и правого сыновей}

**end**;

**var**

Tree,p1:PNode; {Tree адрес корня дерева, p1-вспомогательная переменная}

n,x,i:integer;

ch:char; {для работы менюшки}

{Процедура добавления элемента }

**procedure** AddToTree (**var** Tree:PNode;x:integer); {Входные параметры - адрес корня дерева и добавл элемент }

**begin**

**if** Tree=nil **then** {Если дерево пустое то создаём его корень}

**begin**

**New**(Tree); {Выделяем память }

Tree^.data:=x; {Добавляем данные }

Tree^.left:=nil; {Зануляем указатели на левого }

Tree^.right:=nil; {и правого сыновей }

**exit**;

**end**;

**if** x < Tree^.data **then** {Доб к левому или правому поддереву это завсит от вводимого элемента}

AddToTree(Tree^.left,x) {если меньше корня то в левое поддерево }

**else**

AddToTree(Tree^.right,x); {если больше то в правое}

**end**;

{Функция поиска по дереву}

**function** Search(Tree:PNode;x:integer):PNode; {Возвращает адрес искомого элемента, nil если не найден}

**var**

p:PNode; {вспомог переменнная }

**begin**

**if** Tree=nil **then** {если дерево пустое то}

**begin**

Search:=nil; {присваеваем функции результат нил}

**exit**; {и выходим }

**end**;

**if** x=Tree^.data **then** {если искомый элемент равен корню дерева то }

p:=Tree {Пзапоминаем его адрес }

**else** {иначе}

**if** x < Tree^.data **then** {если вводимый элемент менньше корня то }

p:=Search(Tree^.left,x) {то ищем его в левом поддереве}

**else** {иначе }

p:=Search(Tree^.right,x); {ищем в правом поддереве }

Search:=p; {присваеваем переменной с именем фугкции результат работы}

**end**;

{Обход дерева по принципу Левый-корень-правый }

**procedure** Lkp(Tree:PNode);

**begin**

**if** Tree=nil **then** {Если дерево пустое }

**exit**; {то выход }

Lkp(Tree^.left); {иначе начинаем обход с левого подднрева}

write(' ',Tree^.data); {выводим данные }

Lkp(Tree^.right); {обходим правое поддерево}

**end**;

{Процедура удаления дерева}

**procedure** DeleteTree(**var** Tree1:PNode );

**begin**

**if** Tree1 <> nil **then**

**begin**

DeleteTree (Tree1^.LEFT);

DeleteTree (Tree1^.RIGHT);

Dispose(Tree1);

**end**;

**end**;

{основная пограмма}

**begin**

Tree:=nil;

**repeat** {цикл для нашего меню}

Writeln('Выберете действие ');

Textcolor(2);

Writeln('Доступны следующие пункты:');

Writeln('1) Создание дерева поиска');

Writeln('2) Поиск элемента в дереве');

Writeln('3) Вывод дерева');

Writeln('4) ‚Выход');

writeln;

readln(ch); {ожидаем нажатия клавиши}

**case** ch **of** {выбираем клавишу}

'1': **begin**

writeln(' kolv elementov');

readln(n);

**for** i:=1 **to** n **do**

**begin**

writeln('Введите число');

readln(x);

AddToTree(Tree,x);

**end**;

**end**;

'2': **begin**

writeln('Элемент для поиска');

readln(x);

p1:=Search(Tree,x);

**if** p1 <> nil **then**

writeln('Найден')

**else** writeln('Такого элемента нет!');

**end**;

'3': **begin**

writeln('Само дерево');

Lkp(Tree);

writeln;

**end**;

**end**;

**until** ch='4';

DeleteTree(Tree);

**end**.