**Практическая работа 6**

**Типы данных**

**Ответьте на вопросы:**

1. Что такое запись? Из чего она состоит. Приведите свой пример.

Запись (Record) – это составной тип данных, объединяющий несколько переменных (полей) под одним именем. В отличие от массива, поля записи могут иметь разные типы данных. Записи используются для представления объектов, имеющих несколько характеристик.

Состоит из поля (Fields): Переменные, входящие в состав записи. Каждое поле имеет имя и тип данных.

Пример:

type

TStudent = record

Name: string;

Age: integer;

GPA: real; // Средний балл

end;

1. Как объявляются и создаются записи в Паскале? Как обращаться к полям записи? Приведите примеры.

Записи объявляются с использованием ключевого слова record и end.

type

TPerson = record

FirstName: string;

LastName: string;

BirthYear: integer;

end;

Создание переменной типа запись:

var

MyPerson: TPerson;

Обращение к полям записи: Через имя переменной и имя поля, разделенные точкой (.).

MyPerson.FirstName := 'John';

MyPerson.LastName := 'Doe';

MyPerson.BirthYear := 1990;

writeln(MyPerson.FirstName, ' ', MyPerson.LastName);

1. Зачем используются записи в виде двумерной таблицы? Приведите пример использования.

Записи в виде двумерной таблицы позволяют структурировать данные, где каждая строка представляет собой отдельный объект (запись), а столбцы - характеристики этого объекта (поля записи). Это удобно для хранения и обработки однотипной информации о множестве объектов.

Пример использования: Хранение информации о сотрудниках компании:

type

TEmployee = record

ID: integer;

Name: string;

Position: string;

Salary: real;

end;

var

Employees: array[1..100] of TEmployee;

1. Как использовать конструкции with при работе с записями?

Конструкция with упрощает обращение к полям записи, позволяя не указывать имя переменной записи каждый раз.

with MyPerson do

begin

FirstName := 'John';

LastName := 'Doe';

BirthYear := 1990;

writeln(FirstName, ' ', LastName); // Не нужно писать MyPerson.FirstName

end;

1. Как происходит считывание/запись записей в файл?

Используются процедуры Write и Read для записи и чтения данных из файловой переменной. Важно записывать и считывать поля записи в том же порядке, в котором они объявлены.

var

F: file of TPerson;

Pers: TPerson;

begin

AssignFile(F, 'persons.dat');

Rewrite(F); // для записи

Pers.FirstName := 'John'; Pers.LastName := 'Doe'; Pers.BirthYear := 1990;

Write(F, Pers);

CloseFile(F);

Reset(F); // для чтения

Read(F, Pers);

writeln(Pers.FirstName, ' ', Pers.LastName);

CloseFile(F);

end;

1. Что такое множество в Паскале? Как оно задается?

Множество (Set) - это тип данных, представляющий собой неупорядоченную коллекцию уникальных элементов одного и того же базового типа (обычно перечислимого или ограниченного диапазона). Порядок элементов в множестве не имеет значения.

Задание множества: Используется ключевое слово set of и базовый тип элементов.

type

TDigits = set of 0..9;

TVowels = set of ('a', 'e', 'i', 'o', 'u');

1. Какие действия можно выполнять с множествами? Приведите примеры.

Операции с множествами:

* + + (Объединение): Создает новое множество, содержащее все элементы обоих множеств.
  + \* (Пересечение): Создает новое множество, содержащее только общие элементы обоих множеств.
  + - (Разность): Создает новое множество, содержащее элементы первого множества, не входящие во второе множество.
  + = (Равенство): Проверяет, содержат ли множества одинаковые элементы.
  + <> (Неравенство): Проверяет, отличаются ли множества.
  + <= (Включение): Проверяет, является ли первое множество подмножеством второго.
  + >= (Включение): Проверяет, является ли второе множество подмножеством первого.
  + IN (Принадлежность): Проверяет, принадлежит ли элемент множеству.

Примеры:

var

Set1, Set2, Result: TDigits;

begin

Set1 := [1, 2, 3, 4, 5];

Set2 := [3, 5, 7, 9];

Result := Set1 + Set2; // [1, 2, 3, 4, 5, 7, 9]

Result := Set1 \* Set2; // [3, 5]

Result := Set1 - Set2; // [1, 2, 4]

if 3 in Set1 then writeln('3 is in Set1');

end;

1. Для чего используется операция IN при работе с множествами? Приведите примеры.

Операция IN проверяет, является ли элемент членом множества. Возвращает True, если элемент принадлежит множеству, и False в противном случае.

var

Vowels: TVowels;

Letter: char;

begin

Vowels := ['a', 'e', 'i', 'o', 'u'];

Letter := 'e';

if Letter in Vowels then

writeln(Letter, ' is a vowel')

else

writeln(Letter, ' is not a vowel');

end;

1. Опишите динамические данные, в чем их особенность?

Динамические данные - это структуры данных, размер и расположение которых в памяти определяется во время выполнения программы, а не на этапе компиляции. Они создаются и уничтожаются по мере необходимости, что позволяет эффективно использовать память.

Особенности:

* Размер изменяется во время выполнения: Можно добавлять и удалять элементы без ограничений, пока есть свободная память.
* Используют указатели: Для связывания элементов используются указатели.
* Гибкость: Более гибкие, чем статические массивы, так как не требуют заранее определенного размера.
* Динамическое выделение памяти: Используются процедуры New и Dispose для выделения и освобождения памяти.

1. Что такое указатель? Как он объявляется? Как присваиваются значения? Приведите примеры.

Указатель (Pointer) - это переменная, которая хранит адрес ячейки памяти, где находится другая переменная определенного типа. Указатели используются для работы с динамической памятью и для косвенного доступа к данным.

Объявление указателя:

type

PInteger = ^Integer; // Указатель на целое число

var

Ptr: PInteger;

Присваивание значения:

1. Выделение памяти: New(Ptr); (выделяет память для целого числа и присваивает адрес этой памяти указателю Ptr)
2. Присваивание значения по адресу: Ptr^ := 10; (оператор ^ разыменовывает указатель, то есть обращается к значению, находящемуся по адресу, хранящемуся в указателе).

var

Ptr: PInteger;

begin

New(Ptr); // Выделяем память

Ptr^ := 10; // Присваиваем значение

writeln(Ptr^); // Выводим значение

Dispose(Ptr); // Освобождаем память

end;

1. Что такое список? Какие виды списков бывают?

Список (List) — это линейная структура данных, состоящая из элементов, связанных друг с другом. В отличие от массивов, элементы списка не обязательно хранятся в последовательных ячейках памяти.

Виды списков:

* Односвязный список (Singly Linked List): Каждый элемент содержит ссылку только на следующий элемент.
* Двусвязный список (Doubly Linked List): Каждый элемент содержит ссылки на следующий и предыдущий элементы.
* Кольцевой список (Circular Linked List): Последний элемент указывает на первый, образуя кольцо. Могут быть одно- и двусвязными.

1. Что такое односвязный список? Как он выглядит? Как происходит объявление типа данных?

Односвязный список - это линейная структура данных, где каждый элемент (узел) содержит данные и указатель на следующий элемент в списке. Последний элемент списка указывает на nil (пустой указатель).

Рисунок 1 – Внешнее представление односвязного списка

type

PNode = ^Node; // Предварительное объявление типа PNode

Node = record

Data: DataType; // Тип данных элемента списка

Next: PNode; // Указатель на следующий элемент

end;

1. Опишите алгоритм работы с алфавитно-частотным словарем. Опишите используемые подпрограммы.

Алгоритм работы с алфавитно-частотным словарём (на базе односвязного списка):

1. Чтение текста: читаем текст из файла или вводим с клавиатуры.
2. Разбиение на слова: разбиваем текст на отдельные слова, игнорируя знаки препинания и пробелы.
3. Создание словаря:
   1. Для каждого слова проверяем, есть ли оно уже в словаре (в списке).
      1. Если слово есть в словаре, увеличиваем его частоту на 1.
      2. Если слова нет в словаре, создаем новый элемент списка с этим словом и частотой 1, добавляем его в список.
4. Вывод словаря: выводим список слов и их частот.

Используемые подпрограммы:

* CreateNode(Word: string): PNode: Создает новый элемент списка (узел) с заданным словом и частотой 1.
* InsertWord(var Head: PNode; Word: string): Добавляет слово в словарь. Если слово уже есть, увеличивает частоту, иначе добавляет новый узел в список (может быть в начало, в конец или с сохранением алфавитного порядка).
* FindWord(Head: PNode; Word: string): PNode: Ищет слово в списке. Возвращает указатель на узел, если слово найдено, и nil, если не найдено.
* PrintDictionary(Head: PNode): Выводит содержимое словаря (список слов и их частот).
* FreeDictionary(var Head: PNode): Освобождает память, занятую списком. Важно делать это в конце работы программы, чтобы избежать утечек памяти.

1. Что такое двусвязные списки? В чем особенность работы с двусвязными списками?

Двусвязный список (Doubly Linked List) - это линейная структура данных, где каждый элемент (узел) содержит данные и два указателя: на следующий и на предыдущий элемент в списке.

Особенности работы:

* Движение в обоих направлениях: Можно перемещаться по списку как вперед, так и назад.
* Упрощение некоторых операций: Удаление элемента проще, так как есть доступ к предыдущему элементу.
* Больше памяти: Требуется больше памяти для хранения двух указателей в каждом узле.
* Сложность реализации: Реализация некоторых операций (например, вставки) немного сложнее, чем в односвязном списке.

1. **Измените алгоритмы для работы с алфавитно-частотным словарем, используя двусвязные списки.**

**type**

**PDNode = ^DNode;**

**DNode = record**

**Word: string;**

**Frequency: integer;**

**Next: PDNode;**

**Prev: PDNode; // Добавляем указатель на предыдущий элемент**

**end;**

**// Создание нового узла (двусвязный список)**

**function CreateDNode(Word: string): PDNode;**

**var**

**NewNode: PDNode;**

**begin**

**New(NewNode);**

**NewNode^.Word := Word;**

**NewNode^.Frequency := 1;**

**NewNode^.Next := nil;**

**NewNode^.Prev := nil;**

**CreateDNode := NewNode;**

**end;**

**// Вставка слова в словарь (двусвязный список)**

**procedure InsertWord(var Head: PDNode; Word: string);**

**var**

**Current, NewNode: PDNode;**

**begin**

**Current := Head;**

**while Current <> nil do**

**begin**

**if Current^.Word = Word then**

**begin**

**Current^.Frequency := Current^.Frequency + 1;**

**Exit;**

**end;**

**Current := Current^.Next;**

**end;**

**// Слово не найдено, добавляем в начало списка (для простоты)**

**NewNode := CreateDNode(Word);**

**NewNode^.Next := Head;**

**if Head <> nil then**

**Head^.Prev := NewNode;**

**Head := NewNode;**

**end;**

**// ... (остальные подпрограммы аналогично адаптируются для двусвязного списка)**

1. **Выведите список слов из словаря в порядке убывания частоты, то есть, сначала те слова, которые встречаются чаще всего.**

**// Сортировка списка по убыванию частоты (пузырьковая сортировка)**

**procedure SortDictionary(var Head: PDNode);**

**var**

**Current, i, j: PDNode;**

**TempWord: string;**

**TempFreq: integer;**

**begin**

**if Head = nil then Exit; // Нечего сортировать**

**Current := Head;**

**while Current^.Next <> nil do // Проходим по списку**

**begin**

**j := Head;**

**while j^.Next <> nil do // Сравниваем соседние элементы**

**begin**

**if j^.Frequency < j^.Next^.Frequency then // Если нужно, меняем местами**

**begin**

**TempWord := j^.Word;**

**TempFreq := j^.Frequency;**

**j^.Word := j^.Next^.Word;**

**j^.Frequency := j^.Next^.Frequency;**

**j^.Next^.Word := TempWord;**

**j^.Next^.Frequency := TempFreq;**

**end;**

**j := j^.Next; // Переходим к следующему элементу для сравнения**

**end;**

**Current := Current^.Next;**

**end;**

**end;**

**// Вывод отсортированного словаря**

**procedure PrintSortedDictionary(Head: PDNode);**

**begin**

**SortDictionary(Head);**

**PrintDictionary(Head); // используем существующую функцию PrintDictionary**

**end;**

1. Что такое стек? Приведите пример.

Стек (Stack) - это структура данных, организованная по принципу LIFO (Last-In, First-Out) - "последним пришел, первым ушел". Это означает, что элемент, добавленный в стек последним, будет извлечен первым.

Пример: Стопка тарелок. Вы берете тарелку с вершины стопки (последнюю положенную), а не из середины или снизу.

1. Поясните основной принцип стека.

Основной принцип LIFO (Last-In, First-Out): Доступ к элементам стека осуществляется только через его вершину. Добавление элемента (push) происходит на вершину стека, а удаление элемента (pop) также происходит с вершины стека.

1. Опишите основные операции со стеком. Опишите используемые подпрограммы.

Основные операции:

• Push(Stack, Element): Добавляет элемент Element на вершину стека Stack.

• Pop(Stack): Удаляет и возвращает элемент с вершины стека Stack. Если стек пуст, обычно возвращается значение ошибки (или генерируется исключение).

• Peek(Stack) (иногда Top): Возвращает значение элемента на вершине стека Stack, но не удаляет его.

• IsEmpty(Stack): Проверяет, пуст ли стек Stack. Возвращает True, если стек пуст, и False в противном случае.

• IsFull(Stack) (для стеков ограниченного размера): Проверяет, заполнен ли стек Stack. Возвращает True, если стек заполнен, и False в противном случае.

type

TStack = record

Items: array[1..100] of integer; // Стек на 100 элементов

Top: integer; // Индекс вершины стека

end;

procedure Push(var S: TStack; Value: integer);

begin

if S.Top = 100 then

writeln('Stack Overflow')

else

begin

S.Top := S.Top + 1;

S.Items[S.Top] := Value;

end;

end;

function Pop(var S: TStack): integer;

begin

if S.Top = 0 then

begin

writeln('Stack Underflow');

Pop := -1; // Возвращаем значение ошибки

end

else

begin

Pop := S.Items[S.Top];

S.Top := S.Top - 1;

end;

end;

1. Что такое очередь? Приведите пример.

Очередь (Queue) - это структура данных, организованная по принципу FIFO (First-In, First-Out) - "первым пришел, первым ушел". Это означает, что элемент, добавленный в очередь первым, будет извлечен первым.

Пример: Очередь в магазине. Первый человек, вставший в очередь, первым будет обслужен.

1. Что такое стек? Приведите пример.

Стек (Stack) - это структура данных, организованная по принципу LIFO (Last-In, First-Out) - "последним пришел, первым ушел". Это означает, что элемент, добавленный в стек последним, будет извлечен первым.

1. Поясните основной принцип очереди.

Основной принцип FIFO (First-In, First-Out): Элементы добавляются в конец очереди (enqueue) и удаляются из начала очереди (dequeue).

1. Опишите основные операции с очередями.

Основные операции:

* Enqueue(Queue, Element): Добавляет элемент Element в конец очереди Queue.
* Dequeue(Queue): Удаляет и возвращает элемент из начала очереди Queue. Если очередь пуста, обычно возвращается значение ошибки (или генерируется исключение).
* Peek(Queue) (иногда Front): Возвращает значение элемента в начале очереди Queue, но не удаляет его.
* IsEmpty(Queue): Проверяет, пуста ли очередь Queue. Возвращает True, если очередь пуста, и False в противном случае.
* IsFull(Queue) (для очередей ограниченного размера): Проверяет, заполнена ли очередь Queue. Возвращает True, если очередь заполнена, и False в противном случае.

1. Что такое дек? Приведите пример.

Дек (Deque - Double-Ended Queue) - это обобщение очереди, которое позволяет добавлять и удалять элементы с обоих концов. Дек объединяет свойства стека и очереди.

Пример: Представьте трубу, в которую можно класть и доставать предметы с обоих концов.

1. Опишите подпрограммы из задания с деком.

Для дека необходимо реализовать следующие подпрограммы:

• AddFront(Deque, Element): Добавляет элемент Element в начало дека Deque.

• AddRear(Deque, Element): Добавляет элемент Element в конец дека Deque.

• RemoveFront(Deque): Удаляет и возвращает элемент из начала дека Deque.

• RemoveRear(Deque): Удаляет и возвращает элемент из конца дека Deque.

• GetFront(Deque): Возвращает элемент из начала дека Deque (без удаления).

• GetRear(Deque): Возвращает элемент из конца дека Deque (без удаления).

• IsEmpty(Deque): Проверяет, пуст ли дек.

• IsFull(Deque): Проверяет, полон ли дек (если дек имеет фиксированный размер).

1. Что такое дерево? Перечислите основные элементы дерева.

Дерево (Tree) - это иерархическая структура данных, состоящая из узлов, связанных между собой ребрами. В отличие от линейных структур данных (списков, стеков, очередей), дерево может иметь несколько ветвей, расходящихся от каждого узла.

Основные элементы дерева:

* + Корень (Root): Верхний узел дерева, у которого нет предков.
  + Узел (Node): Основной элемент дерева, содержащий данные и ссылки на другие узлы (потомки).
  + Ребро (Edge): Связь между двумя узлами, указывающая на отношение "родитель-потомок".
  + Лист (Leaf): Узел, у которого нет потомков.
  + Родитель (Parent): Узел, имеющий потомков.
  + Потомок (Child): Узел, связанный с родительским узлом ребром.
  + Поддерево (Subtree): Дерево, состоящее из узла и всех его потомков.
  + Глубина (Depth): Расстояние от корня до узла.
  + Высота (Height): Максимальное расстояние от корня до самого дальнего листа.

1. Что такое двоичное дерево?

Двоичное дерево (Binary Tree) - это дерево, в котором каждый узел имеет не более двух потомков: левого потомка и правого потомка.

1. Опишите алгоритм поиска по дереву.

Алгоритм поиска в общем дереве зависит от того, что именно мы ищем и какую структуру имеет дерево. Обычно используются следующие подходы:

1) Поиск в глубину (Depth-First Search, DFS):

* + - * Начинаем с корня.
      * Переходим к одному из потомков (например, к самому левому).
      * Рекурсивно повторяем шаги 2 и 3 для этого потомка, пока не достигнем листа.
      * Если искомый элемент не найден, возвращаемся к предыдущему узлу (родителю) и переходим к другому его потомку.
      * Продолжаем этот процесс, пока не обойдем все узлы дерева.

2) Поиск в ширину (Breadth-First Search, BFS):

* + - * Начинаем с корня.
      * Просматриваем всех потомков корня.
      * Затем просматриваем всех потомков каждого из этих потомков и так далее, двигаясь по уровням дерева.

1. Опишите алгоритм поиска в бинарном дереве.

Для бинарных деревьев часто используется более эффективный алгоритм, если дерево обладает определенными свойствами (например, является деревом поиска):

* Поиск в бинарном дереве поиска (Binary Search Tree, BST): (Предполагается, что в левом поддереве находятся узлы с ключами меньше, чем в текущем узле, а в правом - больше).

1) Начинаем с корня.

2) Сравниваем искомый ключ со значением в текущем узле.

* + - * Если ключ равен значению в узле, поиск успешен.
      * Если ключ меньше значения в узле, переходим к левому потомку.
      * Если ключ больше значения в узле, переходим к правому потомку.

3) Повторяем шаг 2, пока не найдем искомый ключ или не достигнем nil (конец дерева).

1. Улыбнитесь😊