**20. Связанные структуры.**

В Паскале все типы данных являются статическими. Простые типы данных занимают отдельные ячейки памяти, структурированные типы – группы ячеек. Число ячеек памяти отведенных под структуру определенного типа данных заранее фиксировано, мы не можем управлять размерностью этих структур. Примером динамического типа данных являются файлы, так как мы можем управлять размером файла.

Многие задачи будут решаться более эффективно, а некоторые – единственно возможно при динамическом управлении памятью.

**20.1. Реализация связанных структур данных с использованием массивов.**

Массив – структура данных, которая позволяет произвольный доступ к своим ячейкам. Мы не можем управлять размещением элементов массива, но можем организовать варианты обхода массива в произвольном порядке и использовать эту возможность, например, для задач сортировки.

В задачах сортировки приходится выполнять следующие действия (или/или):

Переставлять записи в памяти

Присваивать записям номера

Организовывать записи связанные в списки

Последний способ является наиболее эффективным в том случае, если число записей достаточно большое, а поля ключей (ключ – то, что определяет порядок записи в сортировке) составляют малую часть от всей длины записи.

**20.1.1 Простой связанный список (List).**

CONST

NameLen = 7;

AddrLen = 25;

Max = 4;

TYPE

Month = (Jan, Feb, Mar, Apr, May, un, Jul, Aug, Sep, Oct, Nov, Dec);

Sex = (Male, Female);

Date = RECORD

Mo: Month;

Day: 1 .. 31;

Year: INTEGER;

END;

Person = RECORD

Name: STRING[NameLen];

Addr: STRING[Addrlen];

Birth: Date;

VSex: Sex;

Next: 0 .. Max;

END;

VAR

PRecs: ARRAY [1 .. Max] OF Person;

First: 0 .. Max;

Переменная First указывает на индекс записи, которая считается первой.

В поле Next каждой записи заносится индекс следующей записи.

В поле Next последней записи заносится значение 0;

| Index | PRecs[Index].Next | PRecs[Index].Name | Номер в списке |
| --- | --- | --- | --- |
| 1 | 3 | Miller | 2 |
| 2 | 0 | Smith | 4 |
| 3 | 2 | Plane | 3 |
| 4 | 1 | Jones | 1 |

Итерация по списку.

Index := First;

WHILE Index <> 0

DO

BEGIN {Обход массива в алфавитном порядке с помощью Index}

...

Index := PRecs[Index].Next;

END;

Порядок обхода массива с помощью поля Next визуализирован на следующем рисунке.



Таким образом мы можем рассматривать массив как упорядоченную структуру, где порядок задается указателями обхода First и Next.



Структуры, в которых каждая запись содержит указатель на следующую называется связанным списком. Последовательность записей в связанном списке может быть изменена с помощью указателей без физического перемещения элементов списка. Элементы могут быть добавлены к списку без изменения связей для большинства из существующих записей.

Добавляется новая запись

| Index | PRecs[Index].Next | PRecs[Index].Name | Номер в списке |
| --- | --- | --- | --- |
| 1 | 3 | Miller | 2 |
| 2 | 0 | Smith | 5 |
| 3 | 5 | Plane | 3 |
| 4 | 1 | Jones | 1 |
| 5 | 2 | Rush | 4 |

PRecs[5].Next = PRecs[3].Next

PRecs[3].Next = 5

Любая запись в связанном списке может быть удалена с помощью одного присвоения.

**8.1.2. Сортировка включением.**

DP1

PROGRAM InsertionSort (INPUT, OUTPUT);

{Сортирует символы из INPUT}

CONST

Max = 16;

ListEnd = 0;

TYPE

RecArray = ARRAY [1 .. Max] OF

RECORD

Key: CHAR;

Next: 0 .. Max;

END;

VAR

Arr: RecArray;

First, Index: 0 .. Max;

Prev, Curr: 0 .. Max;

Extra: CHAR;

Found: BOOLEAN;

BEGIN {InsertionSort}

First := 0;

Index := 0;

WHILE NOT EOLN

DO

BEGIN

{Помещать запись в список, если позволяет пространство,

иначе игнорировать и сообщать об ошибке}

Index := Index + 1;

IF Index > Max

THEN

BEGIN

READ(Extra);

WRITELN('Сообщение содержит: ', Extra, '. Игнорируем.');

END

ELSE

BEGIN

READ(Arr[Index].Key);

{Включение Arr[Index] в связанный список}

END

END; {WHILE}

{Печать списка начиная с Arr[First]}

END. {InsertionSort}

DP 1.1.

{Вставляем запись в связанный список}

Prev := 0;

Curr := First;

{Найти значения Prev и Curr, если существуют такие что

Arr[Prev].Key <= Arr[Index].Key <= Arr[Curr].Key}

Arr[Index].Next = Curr;

IF Prev = 0 {Первый элемент в списке}

THEN

First := Index;

ELSE

Arr[Prev].Next := Index;

DP 1.1.1

{Найти значения Prev и Curr, если существуют такие что

Arr[Prev].Key <= Arr[Index].Key <= Arr[Curr].Key}

Found := FALSE:

WHILE (Curr <> 0) AND NOT Found

DO

IF Arr[Index]. Key > Arr[Curr].Key

THEN

BEGIN

Prev := Curr;

Curr := Arr[Curr].Next

END

ELSE

Found := True;

DP 1.2

{Печать списка начиная с Arr[First]}

Index := First;

WHILE Index <> ListEnd

DO

BEGIN

WRITE(Arr[Index].Key);

Index := Arr[Index].Next

END;

WRITELN;

**20.2. Указатели.**

Pascal имеет специальный встроенный тип данных для поддержки процессов обработки связанных структур, меняющих во время выполнения программы размер занимаемой памяти. Тип указатель не похож на обычный тип данных, он не содержит данные непосредственно, но лишь указывает на переменную где фактически хранятся данные.

Использование указателей эффективно в тех алгоритмах, где важнее размещение значений, чем их обработка.

TYPE

RefInt = ^INTEGER;

VAR

PInt1, PInt2: RefInt;

Операции **=** и **<>** применимы к указателям и позволяют узнать эквивалентны ли значения двух переменных типа указатель.

Для обозначения указателей не связанный с какой-либо переменной, «пустых», используется константа **NIL** – пустой указатель.

Переменной типа указатель может быть присвоено значение другой переменной или значение константы NIL.

| PInt1 := PInt2; |  |
| --- | --- |
| PInt1 := NIL; |  |

Выделение памяти.

| NEW(PInt1); |  |
| --- | --- |
| PInt1^ := 1; |  |
| New(PInt2); |  |
| PInt2^ := PInt1^; |  |

Освобождение памяти.

| DISPOSE(PInt1); |  |
| --- | --- |

После выполнения DISPOSE значение переменной типа указатель становится неопределенным, для последующего использования она должна быть инициализирована.

**8.2.1. Реализация связанных структур с указателями.**

При создании новой ячейки с помощью команды NEW необходимо сохранить ссылку на эту ячейку, иначе может образоваться мусор (garbage).

Очень часто указатель размещается внутри динамически выделяемых структур данных.

TYPE

Node = RECORD

Value: Integer;

Next: ^Node

END;

дугой вариант

TYPE

NodePtr = ^Node;

Node = RECORD

Value: Integer;

Next: NodePtr

END;

VAR

FirstPtr: NodePtr;

BEGIN

NEW(FirstPtr);

FirstPtr^.Value := 1;

NEW(FirstPtr^.Next);

FirstPtr^.Value := 2;

NEW(FirstPtr^.Next^.Next)

FirstPtr^.Value := 3;

END.



Второй способ – ввести новую переменную и контролировать с ее помощью создание новых ячеек.

VAR

NewPtr: NodePtr;

BEGIN

NEW(FirstPtr);

FirstPtr^.Value := 1;

NewPtr := FirstPtr;

FOR Index := 2 TO 20

DO

BEGIN

NEW(NewPtr^.Next);

NewPtr := NewPtr^.Next;

NewPtr^.Value := Index;

END;

NewPtr.Next := NIL

END.

**20.2.3. Сортировка включением.**

Использование указателей снимает ограничения на длину сортируемой последовательности. Использование указателей для организации списка также более наглядно чем в случае со связанными структурами, реализованными на массивах и исключает возможную путаницу индексов.

PROGRAM InsertSort2 (INPUT, OUTPUT);

TYPE

NodePtr = ^Node;

Node = RECORD

Next: NodePtr;

Key: CHAR

END;

VAR

FirstPtr, NewPtr, Curr, Prev: NodePtr;

Found: BOOLEAN;

BEGIN {InsertSort2}

FirstPtr := NIL;

WHILE NOT EOLN

DO

BEGIN

NEW(NewPtr);

READ(NewPtr^.Key);

{1.1. Поместить NewPtr в надлежащее место}

END;

{1.2. Печать значений начиная с FirstPtr^.Key}

END. {InsertSort2}

{1.1. Поместить NewPtr в надлежащее место}

Prev := NIL;

Curr := FirstPtr;

{1.1.1 Найдем значение Prev и Curr, такие что Prev^.Key <= NewPtr^.Key <= Curr^.Key}

NewPtr^.Next := Curr;

IF Prev = NIL

THEN

FirstPtr := NewPtr;

ELSE

Prev^.Next := NewPtr;

{1.1.1 Найдем значение Prev и Curr, такие что Prev^.Key <= NewPtr^.Key <= Curr^.Key}

Found := FALSE;

WHILE (Curr <> NIL) AND NOT Found

DO

IF NewPtr^.Key > Curr^.Key

THEN

BEGIN

Prev := Curr;

Curr := Curr^.Next;

END

ELSE

Found := TRUE;

{1.2. Печать значений начиная с FirstPtr^.Key}

NewPtr := FirstPtr;

WHILE NewPtr <> NIL

DO

BEGIN

WRITE(NewPtr^.Key);

NewPtr := NewPtr^.Next

END

**20.2.3. Сортировка с использованием бинарного дерева.**

Сортировка с использованием дерева выполняется по очень простой процедуре: элемент помещается справа от текущего, если он больше и слева, если он меньше. Обход в сортированном порядке выполняется следующим образом: поддерево слева, вершина, поддерево справа.

INPUT: CBDA



PROGRAM TreeSort(INPUT, OUTPUT);

TYPE

Tree = ^NodeType;

NodeType = RECORD

Ch: CHAR

LLink, RLink: Tree;

END;

VAR

Root: Tree;

Ch: CHAR;

BEGIN {TreeSort}

Root := NIL;

WHILE NOT EOLN

DO

DEGIN

READ(Ch);

Insert(Root, Ch)

END;

PrintTree(Root)

END. {TreeSort}

PROCEDURE Insert(VAR Ptr:Tree, Data: CHAR);

BEGIN {Insert}

IF Ptr = NIL

THEN

BEGIN {Создаем лист со значением Data}

NEW(Ptr);

Ptr^.Key := Data;

Ptr^.LLink := NIL;

Ptr^.RLink := NIL;

END

ELSE

IF Ptr^.Ch > Data

THEN

Insert(Ptr.LLink, Data)

ELSE

Insert(Ptr.RLink. Data)

END; {Insert}

PROCEDURE PrintTree(Ptr: Tree);

BEGIN {PrintTree}

IF Ptr <> NIL

THEN {Печатает поддерево слева, вершину, поддерево справа}

BEGIN

PrintTree(Ptr^.LLink);

WRITE(Ch);

PrintTree(Ptr^.RLink);

END;

WRITELN

END; {PrintTree}