

# РЕАЛИЗАЦИЯ СТРУКТУР ДААННЫХ

*Чтобы рисовать слона, надо  
его сначала увидеть.*

*Вьетнамская пословица*

# Содержание

2

- Понятие структуры данных
- Основные структуры данных
- Реализация структур данных в языке Pascal

# Структуры данных

3

- *Структура данных* – множество элементов данных и множество связей между ними.
- *Физическая структура данных* – способ физического представления данных в памяти компьютера.
- *Логическая структура данных* – абстракция объектов реального мира (данные+операции над ними).
- Программы = Алгоритмы + Структуры данных =  
Алгоритмы +  
Логические структуры данных +  
Отображение логических структур в физические

# Основные структуры данных

4

- *Последовательный список* – список с последовательным распределением элементов в памяти.
- *Связный список* – список, в котором каждый элемент, помимо своего значения, хранит адрес для связи с другим элементом.
- *Дерево* – конечное множество элементов, один из которых называется корнем, а остальные делятся на непересекающиеся подмножества, каждое из которых само является деревом.

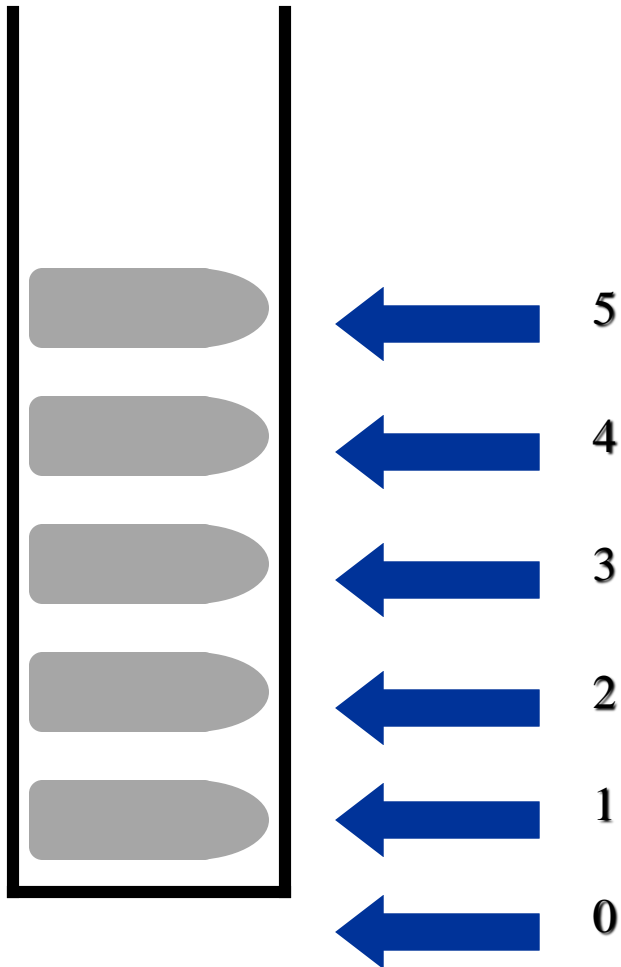
# Последовательные списки

5

- *Стек* – список, организованный по принципу LIFO (Last In, First Out – "последним вошел, первым вышел").
- *Очередь* – список, организованный по принципу FIFO (First In, First Out – "первым вошел, первым вышел").
- *Дек* – двусторонняя очередь.

# Стек

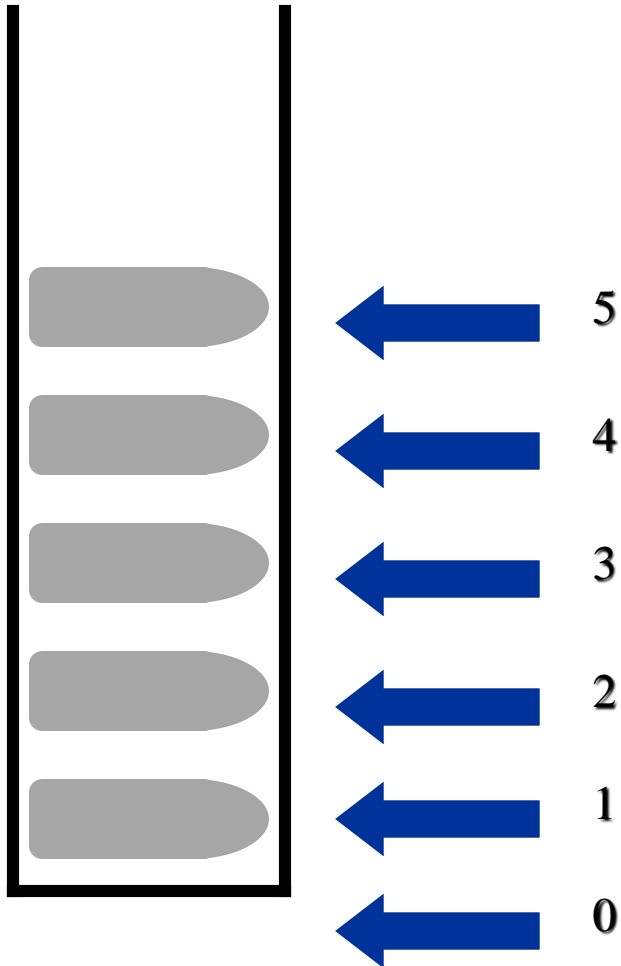
6



- **Push**  
добавить элемент
- **Pop**  
удалить элемент
- **Top**  
просмотреть элемент
- **IsEmpty**  
проверить на пустоту
- **IsFull**  
проверить на заполнение

# Стек

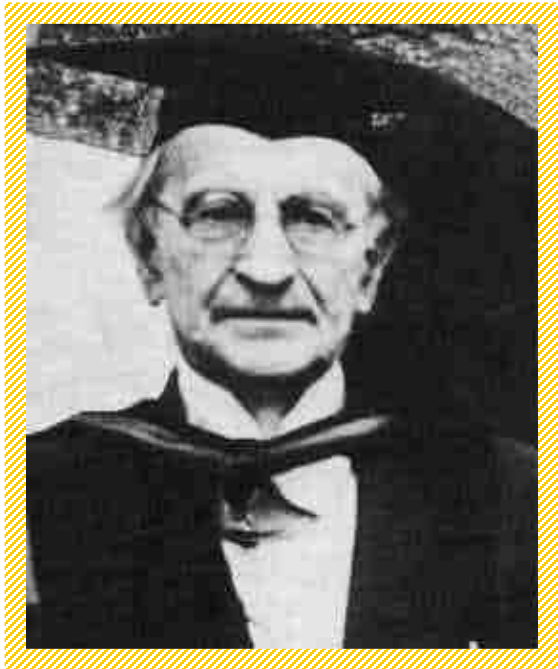
7



- **Push**  
добавить элемент
- **Pop**  
удалить элемент
- **Top**  
просмотреть элемент
- **IsEmpty**  
проверить на пустоту
- **IsFull**  
проверить на заполнение

# Польская запись

8



Ян Лукашевич  
1878 – 1956

- Обычная форма записи арифметических выражений – *инфиксная*.

$$a+b*c$$

$$a*b+c$$

- В *постфиксной* записи знак операции помещается *после* операндов.

$$abc*+$$

$$ab*c+$$



# Польская запись

9

- Польская запись никогда не содержит скобок:

$$(a+b)*c \Rightarrow ab+c*$$

$$a+b*(c+d)*(e+f) \Rightarrow abcd+*ef+*+$$

- Выражения в польской записи легко вычислять.
- В структуру некоторых компиляторов включается *специальный модуль перевода арифметических выражений в постфиксную форму.*

# Вычисление значения выражения в польской записи

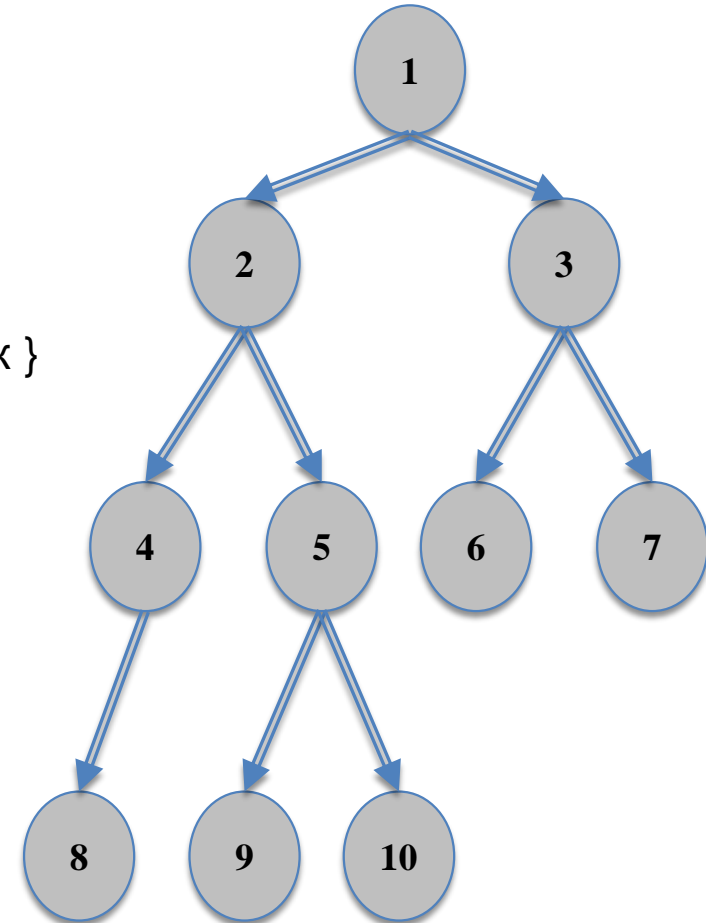
10

```
Stack.Init;
while not (конец_цепочки) do
  case тип_текущего_символа of
    операнд          :
      Stack.Push(значение_операнда);
    знак_операции    :
      begin
        arg1 := Stack.Pop;
        arg2 := Stack.Pop;
        Stack.Push(arg1 знак_операции arg2);
      end;
  end;
end;
Результат := Stack.Top;
```

# Нерекурсивый обход бинарного дерева

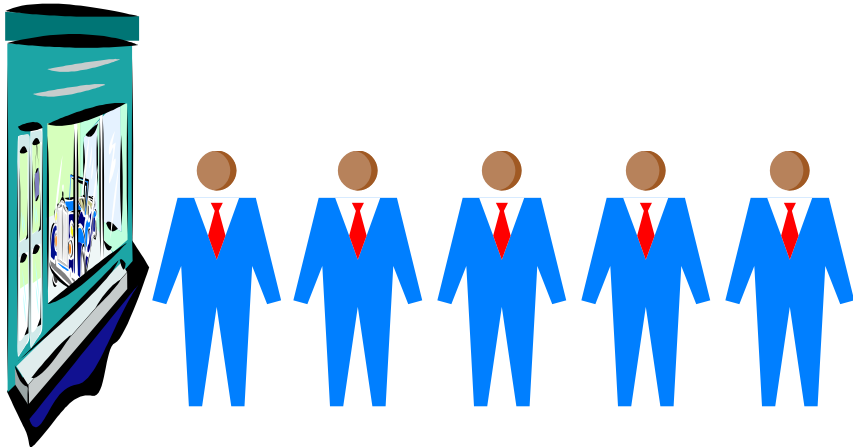
11

```
procedure Traverse(R: Pnode);
begin
  Stack.Init;
  while R<>NIL do begin
    Write(R^.Info, ' ');
    if R^.Left<>NIL then begin { если есть левый сын }
      { в случае наличия помещаем правого сына в стек }
      if R^.Right<>NIL then Stack.Push(R^.Right);
      R:=R^.Left; { и идем к левому сыну }
    end
    else
      { если нет левого сына, но есть правый сын }
      if R^.Right<>NIL then R:=R^.Right
      else begin { если нет обеих сыновей }
        if Stack.IsEmpty then R:=NIL else R:=Stack.Pop;
      end;
    end;
  end;
end;
```



# Очередь

12



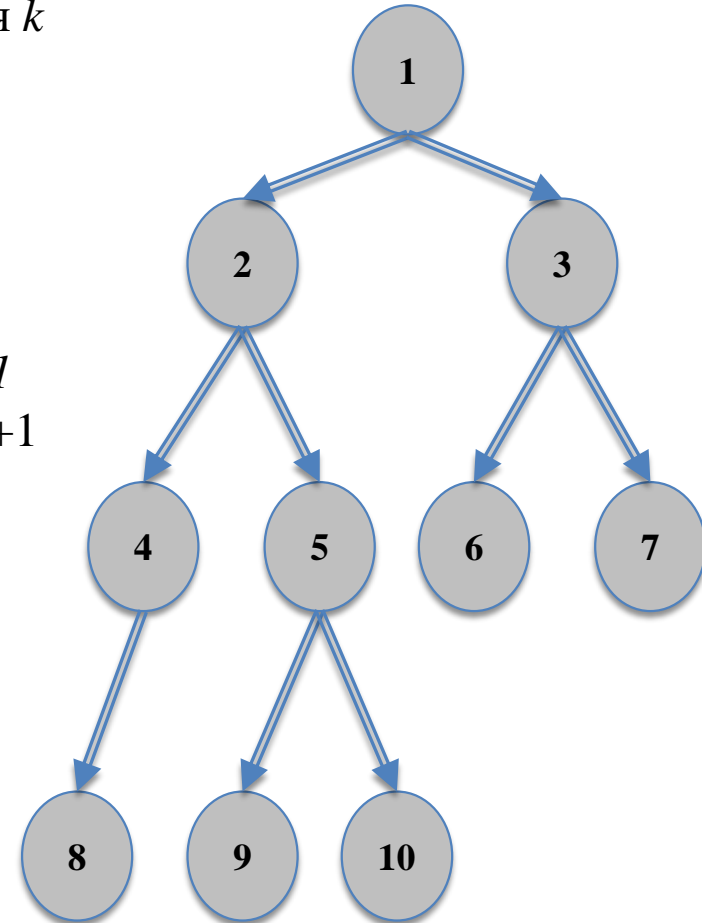
- **Insert**  
добавить элемент
- **Remove**  
удалить элемент
- **Head**  
просмотреть элемент в начале
- **Tail**  
просмотреть элемент в конце
- **IsEmpty**  
проверить на пустоту
- **IsFull**  
проверить на заполнение

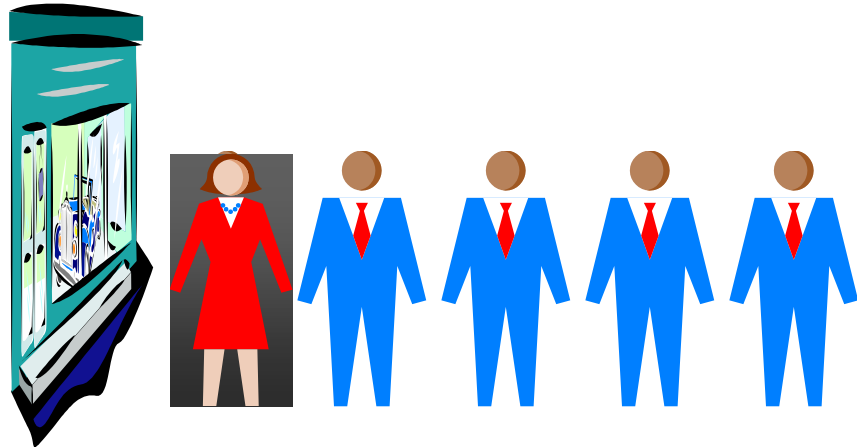
# Печать бинарного дерева в естественном виде

13

```
procedure PrintTree(R: Pnode);
var i, n, k, k1: Integer;
begin
  if R=NIL then break;
  Queue.Init;
  n:=0; k:=1;
  Queue.Insert(R);
  repeat
    k1:=0;
    for i:=1 to k do begin
      Queue.Remove;
      Write(' ', R^.Info);
      if R^.Left<>NIL then begin
        Queue.Insert(R^.Left); k1:=k1+1;
      end;
      if R^.Right<>NIL then begin
        Queue.Insert(R^.Right); k1:=k1+1;
      end;
    end;
    WriteLn;
    n:=n+1; k:=k1;
  until k=0;
end;
```

- В очереди находятся  $k$  вершин  $n$ -го уровня дерева.
- Они считываются и печатаются, и одновременно в очередь заносятся  $k1$  вершин-потомков  $n+1$  уровня.





- **InsertFirst**  
добавить элемент в начало
- **InsertLast**  
добавить элемент в конец
- **RemoveFirst**  
удалить элемент из начала
- **RemoveLast**  
удалить элемент из конца
- **Head**  
просмотреть элемент в начале
- **Tail**  
просмотреть элемент в конце
- **IsEmpty**  
проверить на пустоту
- **IsFull**  
проверить на заполнение

# Связные списки

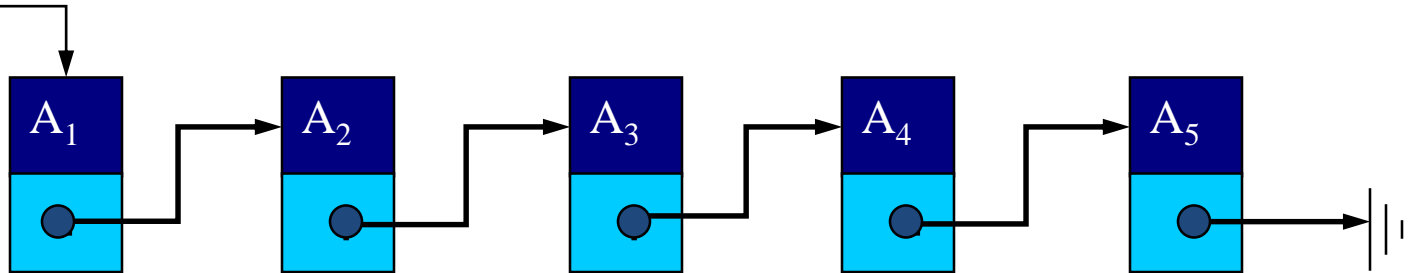
15

- *Односвязный список* – список, в котором каждый элемент содержит указатель с адресом *следующего* элемента. При этом *последний* элемент в качестве адреса хранит **NIL**.
- *Двусвязный список* – список, в котором каждый элемент содержит два указателя: с адресом *предыдущего* и *следующего* элемента. При этом *последний* элемент в качестве адреса *следующего* и *первый* элемент в качестве адреса *предыдущего* хранят **NIL**.
- *Циклический список* – односвязный список, в котором *последний* элемент хранит адрес *первого* элемента.

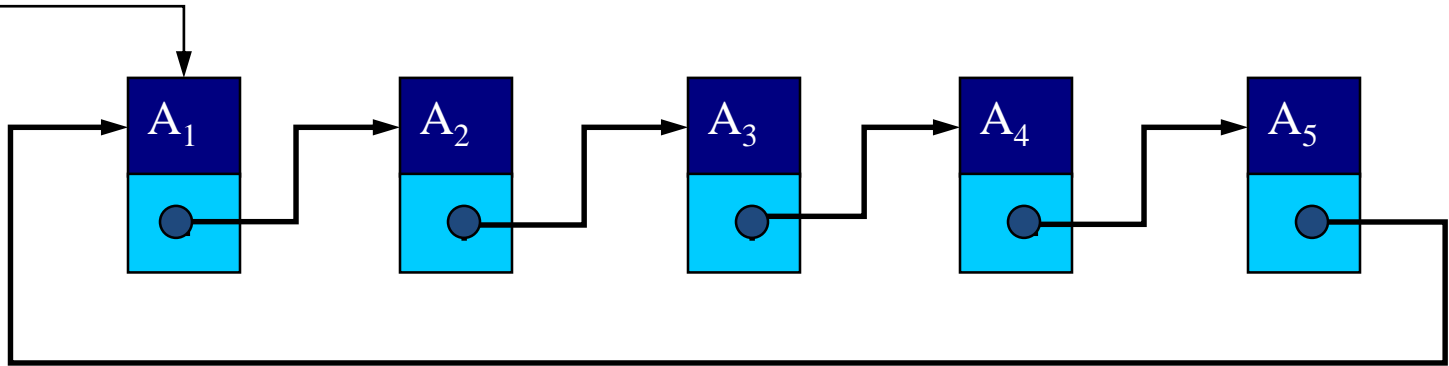
# Связные списки

16

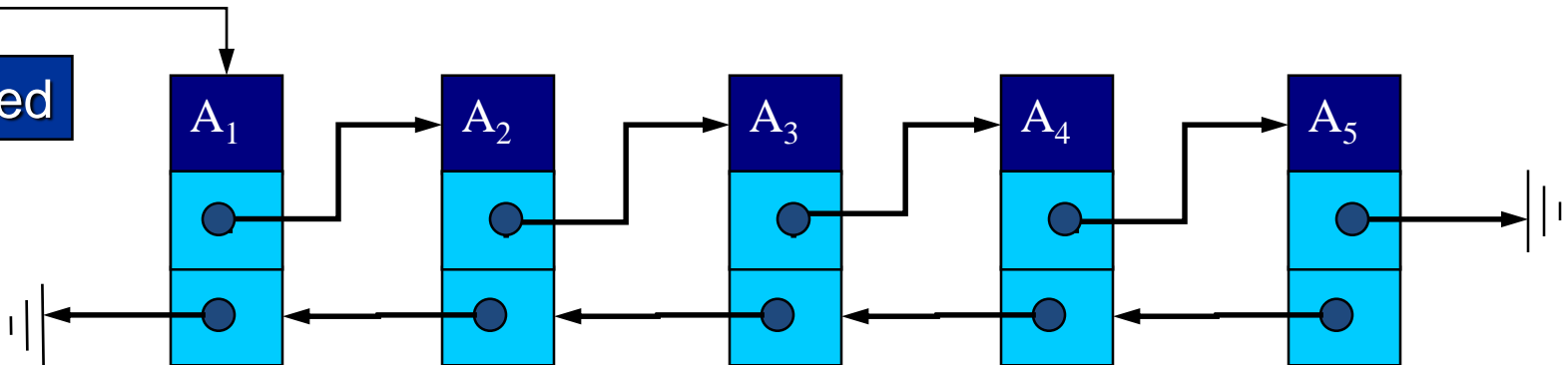
SingleLinked



Cyclic



DoubleLinked





# Реализация стека (последовательное представление)

17

```
unit UStack;
interface
const
    Size=100;
type
    TInfo = Integer;
    Stack = object
    private
        TopPtr: Integer;
        S: array [1..Size] of TInfo;
    public
        procedure Init;
        procedure Push(X: TInfo);
        function Pop: TInfo;
```

```
        function Top: TInfo;
        function IsFull: Boolean;
        function IsEmpty: Boolean;
    end;
implementation
procedure Stack.Init;
begin
    TopPtr := 0;
end;
procedure Stack.Push(X: TInfo);
begin
    TopPtr := TopPtr + 1;
    S[TopPtr] := X;
end;
```

# Реализация стека (последовательное представление)

18

```
function Stack.Pop: TInfo;
begin
    Pop := S[TopPtr];
    TopPtr := TopPtr - 1;
end;
function Stack.Top: TInfo;
begin
    Top := S[TopPtr];
end;
function Stack.IsFull: Boolean;
begin
    IsFull := TopPtr=Size;
end;
```

```
function Stack.IsEmpty: Boolean;
begin
    IsEmpty := TopPtr=0;
end;

end.
```

# Реализация стека (связное представление)

19

```
unit UStack;
```

```
interface
```

```
type
```

```
  TInfo = Integer;  
  PNode = ^TNode;  
  TNode = record  
    Info: TInfo;  
    Next: PNode;  
  end;
```

```
Stack = object
```

```
  private
```

```
    BottomPtr: PNode;
```

```
    TopPtr: PNode;
```

```
  public
```

```
    constructor Init;
```

```
    procedure Push(X: TInfo);
```

```
    function Pop: TInfo;
```

```
    function Top: TInfo;
```

```
    function IsEmpty: Boolean;
```

```
    destructor Done; virtual;
```

```
  end;
```

```
implementation
```

# Реализация стека (связное представление)

20

```
constructor Stack.Init;  
begin  
    BottomPtr := Nil;  
    TopPtr := Nil;  
end;
```

```
procedure Stack.Push(X: TInfo);  
var P: PNode;  
begin  
    New(P);  
    P^.Info := X;  
    P^.Next := Nil;  
    if TopPtr <> Nil then  
        TopPtr^.Next := P  
    else  
        BottomPtr := P;  
        TopPtr := P;  
end;
```

# Реализация стека (связное представление)

21

```
function Stack.Pop: TInfo;  
var P: PNode;  
begin  
    Pop := TopPtr^.Info;  
    P := BottomPtr;  
    while P^.Next <> TopPtr do  
        P := P^.Next;  
    TopPtr := P;  
    Dispose(TopPtr^.Next);  
end;
```

```
function Stack.Top: TInfo;  
begin  
    Top := TopPtr^.Info;  
end;
```

```
function Stack.IsEmpty: Boolean;  
begin  
    IsEmpty := TopPtr = Nil;  
end;
```

```
destructor Stack.Done; virtual;  
var E: TInfo;  
begin  
    while not IsEmpty do  
        E := Pop;  
    end;  
  
end.
```

# Реализация односвязного списка

22

```
unit UList;

interface

type
  TInfo = Integer;
  PNode = ^TNode;
  TNode = record
    Info: TInfo;
    Next: PNode;
  end;

  List = object
  private
    First: PNode;
  public
    constructor Init;
    procedure InsertFirst(X: TInfo);
    procedure InsertAfter(P: PNode; X: TInfo);
    function IsEmpty: Boolean;
    procedure DeleteFirst;
    procedure DeleteAfter (P: PNode);
    function Find(X: TInfo): PNode;
    ...
    destructor Done; virtual;
  end;
```

# Реализация односвязного списка

23

```
constructor List.Init;  
begin  
    First := Nil;  
end;
```

```
procedure InsertFirst(X: TInfo);  
var P: PNode;  
begin  
    New(P);  
    P^.Info := X;  
    P^.Next := First;  
    First := P;  
end;
```

```
procedure List.InsertAfter  
    (P: PNode; X: TInfo);  
var P2: PNode;  
begin  
    New(P2);  
    P2^.Info := X;  
    P2^.Next := Nil;  
    P^.Next := P2;  
end;
```

```
function List.IsEmpty: Boolean;  
begin  
    IsEmpty := First = Nil;  
end;
```

# Реализация односвязного списка

24

```
procedure List.DeleteFirst;
var P: PNode;
begin
    P := First;
    First := First^.Next;
    Dispose(P);
end;
```

```
procedure List.DeleteAfter(P: PNode);
begin
    Dispose(P^.Next);
end;
```

```
function Find(X: TInfo): PNode;
var P: PNode;
begin
    Find := Nil; P := First;
    while P <> Nil do begin
        if P^.Info = X then begin
            Find := P;
            break;
        end;
        P := P^.Next;
    end;
end;
```



# Реализация односвязного списка

25

```
destructor List.Done; virtual;  
begin  
    while not IsEmpty do  
        DeleteFirst;  
end;
```

- Односвязный список не допускает эффективной реализации операций
  - Добавить в конец
  - Добавить перед
  - Удалить последний
  - Удалить текущий