

Динамические структуры данных (язык Паскаль)

1. Указатели
2. Динамические массивы
3. Структуры
4. Списки
5. Стеки, очереди, деки
6. Деревья
7. Графы

Динамические структуры данных (язык Паскаль)

Тема 1. Указатели

Статические данные

```
var x, y: integer;  
    z: real;  
    A: array[1..10] of real;  
    str: string;
```

- переменная (массив) имеет **ИМЯ**, по которому к ней можно обращаться
- **размер** заранее известен (задается при написании программы)
- память выделяется **при объявлении**
- размер **нельзя увеличить** во время работы программы

Динамические данные

- размер заранее неизвестен, определяется во время работы программы
- память выделяется во время работы программы
- нет имени?

Проблема:

как обращаться к данным, если нет имени?

Решение:

использовать адрес в памяти

Следующая проблема:

в каких переменных могут храниться адреса?
как работать с адресами?

Указатели

Указатель – это переменная, в которую можно записывать адрес другой переменной (или блока памяти).

Объявление: указатель

```
var pC: ^char;      // адрес символа
    pI: ^integer;    // адрес целой переменной
    pR: ^real;       // адрес вещ. переменной
```

Как записать адрес:

```
var m: integer;      // целая переменная
    pI: ^integer;    // указатель
    A: array[адрес ячейки] of integer; // массив
...
pI := @m;            // адрес переменной m
pI := @A[1];         // адрес элемента массива A[1]
pI := ni;            // нулевой адрес
                1
```

Обращение к данным через указатель

```
program qq;  
var m, n: integer;  
    pI: ^integer;  
begin  
    m := 4;  
    pI := @m;  
    writeln('Адрес m = ', pI); // вывод адреса  
    writeln('m = ', pI^); // вывод значения  
    n := 4*(7 - pI^); // n = 4*(7 - 4) = 12  
    pI^ := 4*(n - m); // m = 4*(12 - 4) = 32  
end.
```

«ВЫТАЩИТЬ»
значение по адресу

Обращение к данным (массивы)

```
program qq;  
var i: integer;  
    A: array[1..4] of integer;  
    pI: ^integer;  
begin  
    for i:=1 to 4 do A[i] := i;  
    pI := @A[1]; // адрес A[1]  
    while ( pI^ <= 4 ) // while( A[i] <= 4 )  
        do begin  
            pI^ := pI^ * 2; // A[i] := A[i]*2;  
            pI := pI + 1; // к следующему элементу  
        end;  
end.
```

переместиться к следующему
элементу = изменить адрес
на **sizeof(integer)**



Не работает в
PascalABC.NET!

Что надо знать об указателях

- указатель – это переменная, в которой можно хранить адрес другой переменной;
- при объявлении указателя надо указать тип переменных, на которых он будет указывать, а перед типом поставить знак `^`;
- знак `@` перед именем переменной обозначает ее адрес;
- запись `p^` обозначает *значение* ячейки, на которую указывает указатель `p`;
- `nil` – это *нулевой указатель*, он никуда не указывает
- при изменении значения указателя на `n` он в самом деле сдвигается к `n`-ому следующему числу данного типа (для указателей на целые числа – на `n*sizeof(integer)` байт).



Нельзя использовать указатель, который указывает неизвестно куда (будет сбой или зависание)!

Динамические структуры данных (язык Паскаль)

Тема 2. Динамические массивы

Где нужны динамические массивы?

Задача. Ввести размер массива, затем – элементы массива. Отсортировать массив и вывести на экран.

Проблема:

размер массива заранее неизвестен.

Пути решения:

- 1) выделить память «с запасом»;
- 2) выделять память тогда, когда размер стал известен.

Алгоритм:

- 3) ввести размер массива;
- 4) **выделить память**
- 5) ввести элементы массива;
- 6) отсортировать и вывести на экран;
- 7) **удалить массив**

Использование указателей (*Delphi*)

какой-то массив целых чисел

```
program qq;  
type intArray = array[1..1] of integer;  
var A: ^intArray;  
    i, N: integer;  
begin  
    writeln('Размер массива>');  
    readln(N);  
    GetMem(pointer(A), N*sizeof(integer));  
    for i := 1 to N do  
        readln(A[i]);  
    ... { сортировка }  
    for i := 1 to N do  
        writeln(A[i]);  
    FreeMem(pointer(A));  
end.
```

выделить память

работаем так же,
как с обычным
массивом!

освободить память

Использование указателей

- для выделения памяти используют процедуру **GetMem**

GetMem (*указатель, размер в байтах*) ;

- указатель должен быть приведен к типу *pointer* – указатель без типа, просто адрес какого-то байта в памяти;
- с динамическим массивом можно работать так же, как и с обычным (статическим);
- для освобождения блока памяти нужно применить процедуру **FreeMem**:

FreeMem (*указатель*) ;

Ошибки при работе с памятью

Запись в «чужую» область памяти:

память не была выделена, а массив используется.

Что делать: так не делать.

Выход за границы массива:

обращение к элементу массива с неправильным номером, при записи портятся данные в «чужой» памяти.

Что делать: если позволяет транслятор, включать проверку выхода за границы массива.

Указатель удаляется второй раз:

структура памяти нарушена, может быть все, что угодно.

Что делать : в удаленный указатель лучше записывать `nil`, ошибка выявится быстрее.

Утечка памяти:

ненужная память не освобождается.

Что делать : убирайте «мусор»
(в среде .NET есть сборщик мусора!)

Динамические массивы(*Delphi*)

```
program qq;  
var A: array of integer;  
    i, N: integer;  
begin  
    writeln('Размер массива>');  
    readln(N);  
    SetLength ( A, N );  
    for i := 0 to N-1 do  
        readln(A[i]);  
    ... { сортировка }  
    for i := 0 to N-1 do  
        writeln(A[i]);  
    SetLength( A, 0 );  
end.
```

какой-то массив
целых чисел

выделить память

нумерация с **НУЛЯ!**

освободить память

Динамические массивы (*Delphi*)

- при объявлении массива указывают только его тип, память не выделяется:

```
var A: array of integer;
```

- для выделения памяти используют процедуру **SetLength** (*установить длину*)

```
SetLength ( массив, размер );
```

- номера элементов начинаются с **НУЛЯ!**
- для освобождения блока памяти нужно установить нулевую длину через процедуру **SetLength**:

```
SetLength ( массив, 0 );
```

Динамические матрицы (*Delphi*)

Задача. Ввести размеры матрицы и выделить для нее место в памяти во время работы программы.

Проблема:

размеры матрицы заранее неизвестны

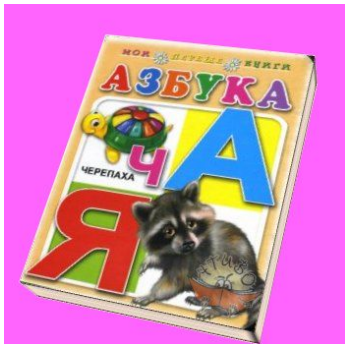
Решение:

```
var A: array of array of integer;  
    N, M: integer;  
begin  
    writeln('Число строк и столбцов>');  
    readln(N, M);  
    SetLength ( A, N, M );  
    ... // работаем, как с обычной матрицей  
    SetLength( A, 0, 0 );  
end.
```


Динамические структуры данных (язык Паскаль)

Тема 3. Структуры (записи)

Структуры (в Паскале – *записи*)



Свойства:

- автор (*строка*)
- название (*строка*)
- год издания (*целое число*)
- количество страниц (*целое число*)

Задача: объединить эти данные в единое целое

Структура (запись) – это тип данных, который может включать в себя несколько *полей* – элементов разных типов (в том числе и другие структуры).

Размещение в памяти

автор	название	год издания	количество страниц
40 символов	80 символов	целое	целое

Одна запись

Объявление (выделение памяти):

название

запись

поля

```
var Book: record
    author: string[40]; // автор, строка
    title:  string[80]; // название, строка
    year:   integer;    // год издания, целое
    pages:  integer;    // кол-во страниц, целое
end;
```

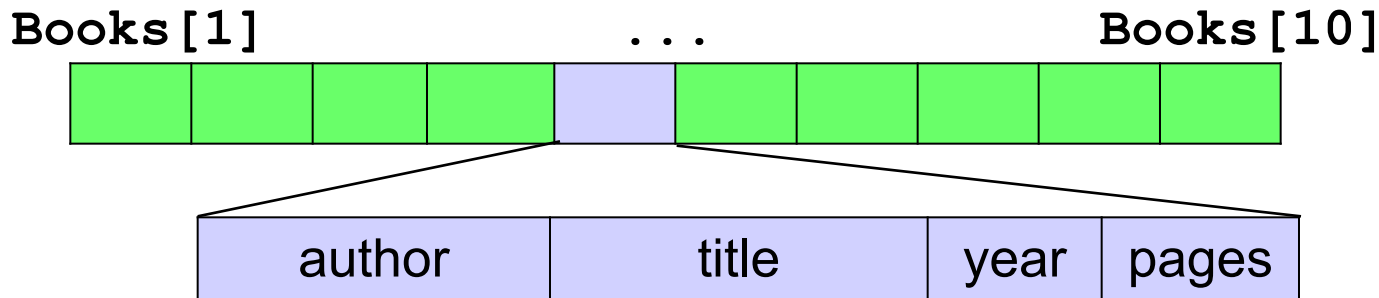
Обращение к полям:

```
readln(Book.author); // ввод
readln(Book.title);
Book.year := 1998;    // присваивание
if Book.pages > 200 then // сравнение
    writeln(Book.author, '.', Book.title); // вывод
```



Для обращения
к полю записи
используется точка!

Массив записей



Объявление (выделение памяти):

```
const N = 10;  
var aBooks: array[1..N] of record  
    author: string[40];  
    title:  string[80];  
    year:   integer;  
    pages:  integer;  
end;
```

Массив записей

Обращение к полям:

```
for i:=1 to N do begin
    readln(aBooks[i].author);
    readln(aBooks[i].title);
    ...
end;
for i:=1 to N do
    if aBooks[i].pages > 200 then
        writeln(aBooks[i].author, '.',
                aBooks[i].title);
```



aBooks[i].author – обращение к полю **author** записи **aBooks[i]**

Новый тип данных – запись

Объявление типа:



Память не выделяется!

```
type TBook = record
  author: string[40]; // автор, строка
  title:  string[80]; // название, строка
  year: integer; // год издания, целое
  pages : integer; // кол-во страниц, целое
end;
```

TBook – *Type Book* («тип книга») – удобно!

Объявление переменных и массивов:

```
const N = 10;
var Book: TBook; // одна запись
    aBooks: array[1..N] of TBook; // массив
```

Записи в процедурах и функциях

Процедура:

```
procedure ShowAuthor ( b: TBook );  
begin  
    writeln ( b.author );  
end;
```

Функция:

```
function IsOld( b: TBook ): boolean;  
begin  
    IsOld := b.year < 1900;  
end;
```

Основная программа:

```
Book.author := 'А.С. Пушкин';  
ShowAuthor ( Book );  
Book.year := 1800;  
writeln( IsOld(Book) );
```

Файлы записей

Объявление указателя на файл:

```
var F: file of TBook;
```

Запись в файл:

```
Assign(F, 'books.dat'); { связать с указателем }  
Rewrite(F);             { открыть файл для запись }  
writeln(F, Book);       { запись }  
for i:=1 to 5 do  
    writeln(aBook[i]); { запись }  
Close(F);               { закрыть файл }
```


Чтение из файла

Известное число записей:

```
Assign(F, 'books.dat'); { связать с указателем }
Reset(F);                { открыть для чтения }
Read(F, Book);           { чтение }
for i:=1 to 5 do
    Read(F, aBook[i]); { чтение }
Close(F);                { закрыть файл }
```

«Пока не кончатся»:

```
count := 0;
while not eof(F) do begin
    count := count + 1;    { счетчик }
    Read(F, aBook[count]); { чтение }
end;
```

пока не дошли до конца файла F
EOF = *end of file*



В чем может быть проблема!

Пример программы

Задача: в файле `books.dat` записаны данные о книгах в виде массива структур типа `TBook` (не более 100). Установить для всех 2008 год издания и записать обратно в тот же файл.

```
type TBook ... ;
```

полное описание
структуры

```
const MAX = 100;
```

```
var aBooks: array[1..MAX] of TBook;
```

```
    i, N: integer;
```

```
    F: file of TBook;
```

```
begin
```

```
    { прочитать записи из файла, N - количество }
```

```
    for i:=1 to N do
```

```
        aBooks[i].year := 2008;
```

```
    { сохранить в файле }
```

```
end.
```

Пример программы

Чтение «пока не кончатся»:

```
Assign(f, 'books.dat');  
Reset(f);  
N := 0;  
while not eof(F) and (N < MAX) do begin  
    N := N + 1;  
    read(F, aBooks[N]);  
end;  
Close(f);
```

чтобы не выйти за
пределы массива

Сохранение:

```
Assign(f, 'books.dat'); { можно без этого }  
Rewrite(f);  
for i:=1 to N do write(F, aBooks[i]);  
Close(f);
```

Выделение памяти под запись

```
var pB: ^TBook;
```

переменная-
указатель на TBook

```
begin
```

```
  New(pB) ;
```

выделить память под запись,
записать адрес в pB

```
  pB^.author := 'А.С. Пушкин' ;
```

```
  pB^.title := 'Полтава' ;
```

```
  pB^.year := 1990 ;
```

```
  pB^.pages := 129 ;
```

```
  Dispose(pB) ;
```

```
end.
```



Для обращения
к полю записи по
адресу
используется
знак ^

освободить
память

Сортировка массива записей

Ключ (ключевое поле) – это поле записи (или комбинация полей), по которому выполняется сортировка.

```
const N = 100;  
var aBooks: array[1..N] of TBook;  
    i, j, N: integer;  
    temp: TBook; { для обмена }  
begin  
    { заполнить массив aBooks }  
    { отсортировать = переставить }  
    for i:=1 to N do  
        writeln(aBooks[i].title,  
                aBooks[i].year:5) ;  
end.
```

Сортировка массива записей

```
for i:=1 to N-1 do
  for j:=N-1 downto i do
    if aBooks[j].year > aBooks[j+1].year
    then begin
      temp := aBooks[j];
      aBooks[j] := aBooks[j+1];
      aBooks[j+1] := temp;
    end;
```



Какой ключ сортировки?



Какой метод сортировки?



Что плохо?

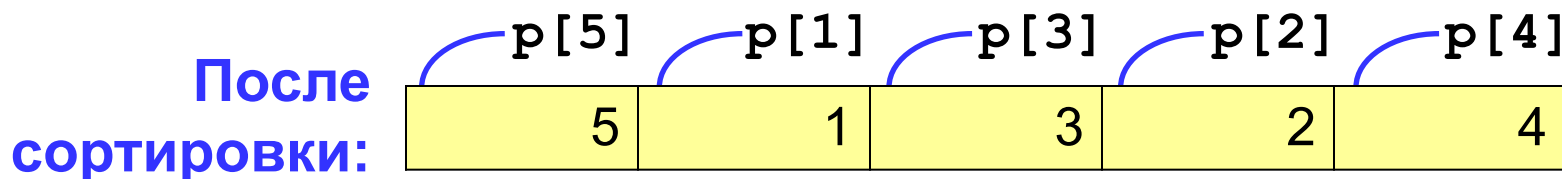
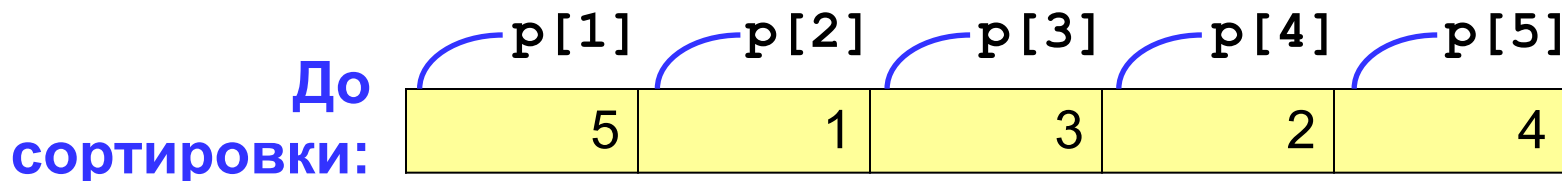
Сортировка массива записей

Проблема:

как избежать копирования записи при сортировке?

Решение:

использовать вспомогательный массив указателей, при сортировке переставлять указатели.



Вывод результата:

```
for i:=1 to N do  
  writeln(p[i]^ .title, p[i]^ .year:5) ;
```

Реализация в программе

```
type PBook = ^TBook; { новый тип данных }
```

```
var p: array[1..N] of PBook;
```

```
begin
```

```
  { заполнение массива записей}
```

```
  for i:=1 to N do
```

```
    p[i] := @aBooks[i];
```

вспомогательные
указатели

начальная
расстановка

```
  for i:=1 to N-1 do
```

```
    for j:=N-1 downto i do
```

```
      if p[j]^year > p[j+1]^year then begin
```

```
        temp := p[j];
```

```
        p[j] := p[j+1];
```

```
        p[j+1] := temp;
```

```
      end;
```

меняем только
указатели, записи
остаются на местах

```
  for i:=1 to N do
```

```
    writeln(p[i]^title, p[i]^year:5);
```

```
end.
```

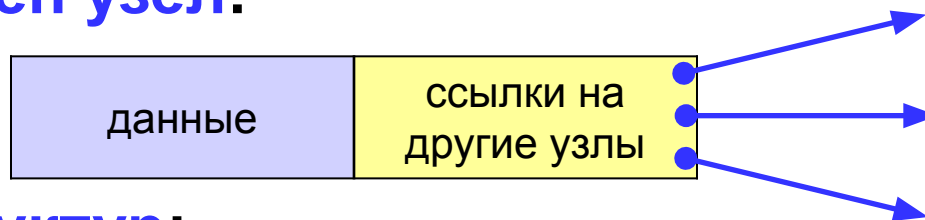

Динамические структуры данных (язык Паскаль)

Тема 4. Списки

Динамические структуры данных

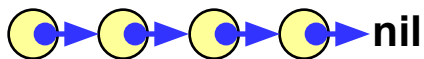
Строение: набор узлов, объединенных с помощью **ССЫЛОК**.

Как устроен узел:



Типы структур:

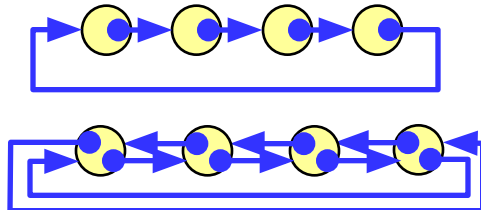
СПИСКИ
односвязный



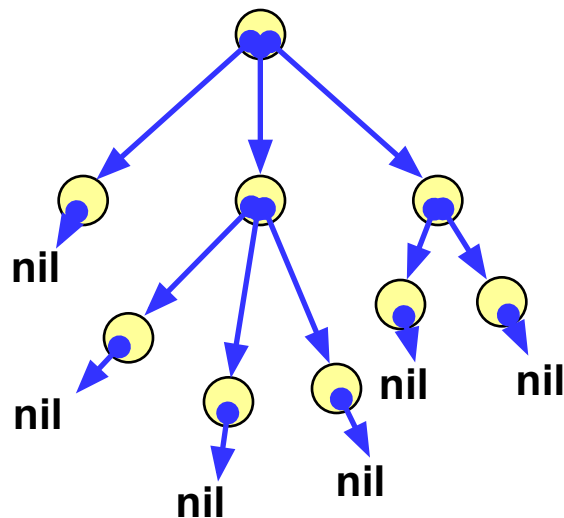
двунаправленный (двусвязный)



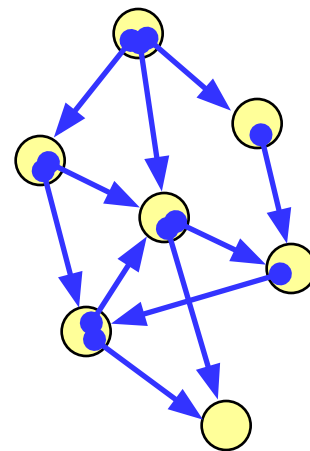
циклические списки (кольца)



деревья



графы



Когда нужны списки?

Задача (алфавитно-частотный словарь). В файле записан текст. Нужно записать в другой файл в столбик все слова, встречающиеся в тексте, в алфавитном порядке, и количество повторений для каждого слова.

Проблемы:

- 1) количество слов заранее неизвестно (~~статический массив~~);
- 2) количество слов определяется только в конце работы (~~динамический массив~~).

Решение – список.

Алгоритм:

- 3) создать список;
- 4) если слова в файле закончились, то стоп.
- 5) прочитать слово и искать его в списке;
- 6) если слово найдено – увеличить счетчик повторений, иначе добавить слово в список;
- 7) перейти к шагу 2.

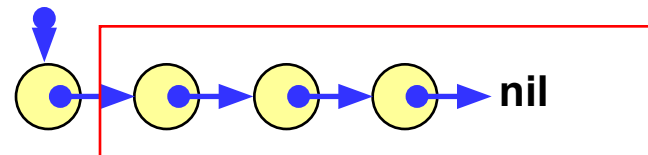
Списки: новые типы данных

Что такое список:

- 1) пустая структура – это список;
- 2) список – это начальный узел (голова) и связанный с ним список.



Рекурсивное определение!



Новые типы данных:

```

type PNode = ^Node;      { указатель на узел }
   Node = record         { структура узла }
       word: string[40]; { слово }
       count: integer;   { счетчик повторений }
       next: PNode;      { ссылка на следующий }
   end;
  
```

Адрес начала списка:

```

var Head: PNode;
...
Head := nil;
  
```



Для доступа к списку достаточно знать адрес его головы!

Что нужно уметь делать со списком?

1. **Создать** новый узел.
2. **Добавить** узел:
 - а) в начало списка;
 - б) в конец списка;
 - в) после заданного узла;
 - г) до заданного узла.
3. **Искать** нужный узел в списке.
4. **Удалить** узел.

Создание узла

Функция `CreateNode` (создать узел):

ВХОД: новое слово, прочитанное из файла;

ВЫХОД: адрес нового узла, созданного в памяти.

НОВОЕ СЛОВО

возвращает адрес
созданного узла

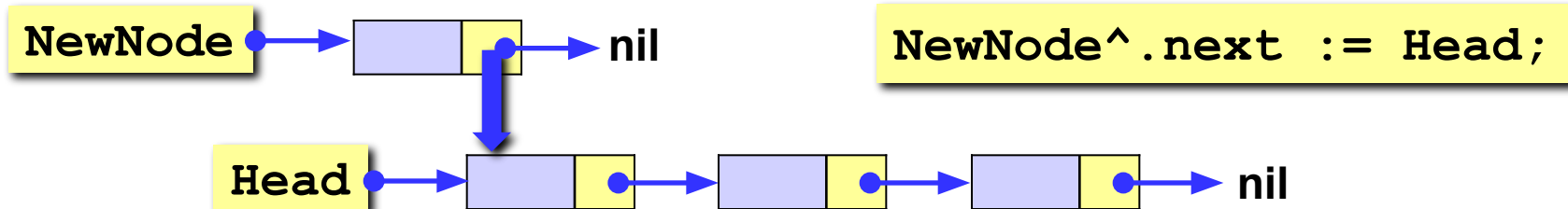
```
function CreateNode(NewWord: string): PNode;  
var NewNode: PNode;  
begin  
    New(NewNode) ;  
    NewNode^.word := NewWord;  
    NewNode^.count := 1;  
    NewNode^.next := nil;  
    Result := NewNode;  
end;
```



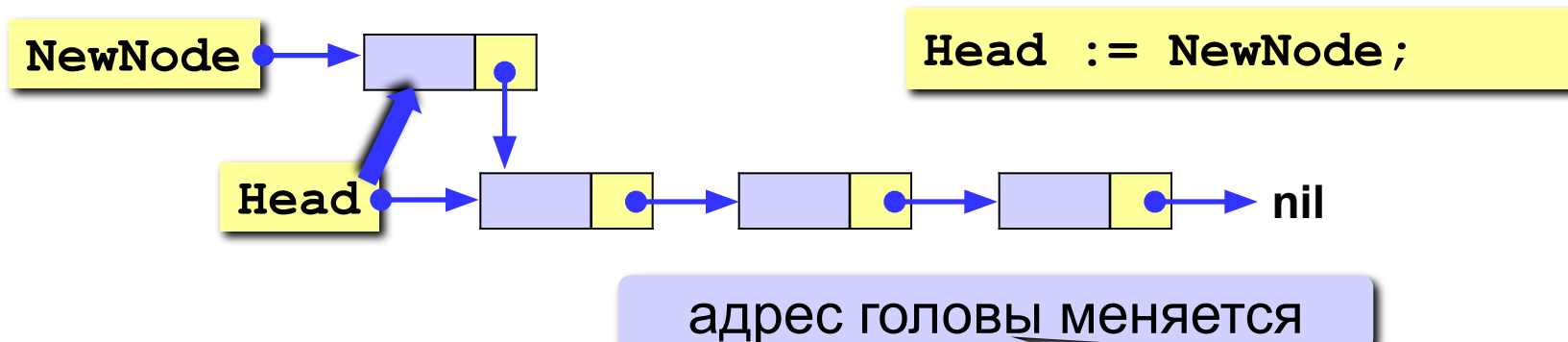
Если память
выделить не
удалось?

Добавление узла в начало списка

1) Установить ссылку нового узла на голову списка:



2) Установить новый узел как голову списка:

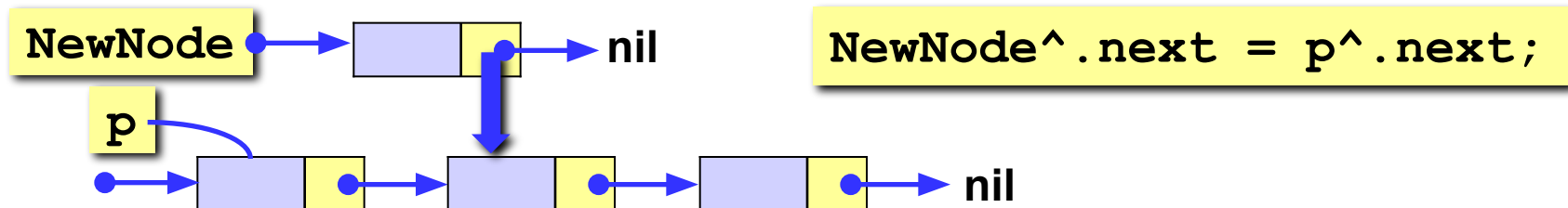


```

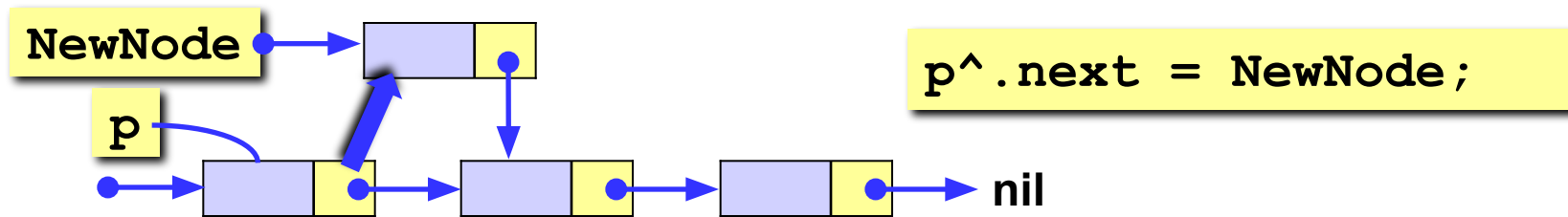
procedure AddFirst ( var Head: PNode; NewNode: PNode );
begin
    NewNode^.next := Head;
    Head := NewNode;
end;
  
```

Добавление узла после заданного

1) Установить ссылку нового узла на узел, следующий за p:



2) Установить ссылку узла p на новый узел:



```

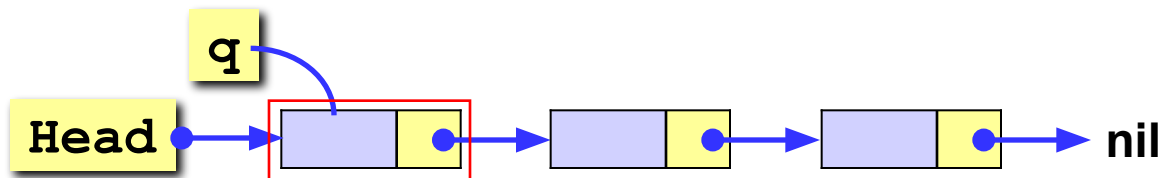
procedure AddAfter ( p, NewNode: PNode );
begin
    NewNode^next := p^next;
    p^next := NewNode;
end;

```


Проход по списку

Задача:

сделать что-нибудь хорошее с каждым элементом списка.



Алгоритм:

- 1) установить вспомогательный указатель **q** на голову списка;
- 2) если указатель **q** равен **nil** (дошли до конца списка), то стоп;
- 3) выполнить действие над узлом с адресом **q**;
- 4) перейти к следующему узлу, **q[^].next**.

```
var q: PNode;
...
q := Head;  // начали с головы
while q <> nil do begin // пока не дошли до конца
    ...           // делаем что-то хорошее с q
    q := q^.next; // переходим к следующему
end;
```

Добавление узла в конец списка

Задача: добавить новый узел в конец списка.

Алгоритм:

- 1) найти последний узел q , такой что q^{next} равен nil ;
- 2) добавить узел после узла с адресом q (процедура **AddAfter**).

Особый случай: добавление в пустой список.

```

procedure AddLast ( var Head: PNode; NewNode: PNode );
var q: PNode;
begin
    if Head = nil then
        AddFirst ( Head, NewNode )
    else begin
        q := Head;
        while q^.next <> nil do
            q := q^.next;
        AddAfter ( q, NewNode );
    end;
end;

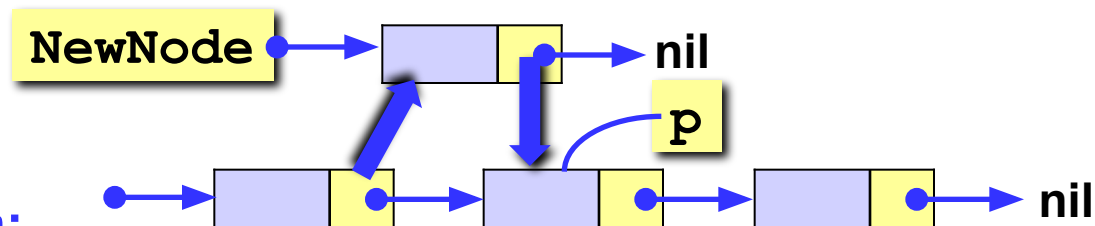
```

особый случай – добавление в пустой список

ищем последний узел

добавить узел
после узла q

Добавление узла перед заданным



Проблема:

нужно знать адрес **предыдущего** узла, а идти назад нельзя!

Решение: найти предыдущий узел **q** (проход с начала списка).

```

procedure AddBefore(var Head: PNode; p, NewNode: PNode);
var q: PNode;
begin
    q := Head;
    if p = Head then
        AddFirst ( Head, NewNode )
    else begin
        while (q <> nil) and (q^.next <> p) do
            q := q^.next;
        if q <> nil then AddAfter ( q, NewNode );
    end;
end;

```

В начало списка

ищем узел, следующий
за которым – узел p

добавить узел
после узла q



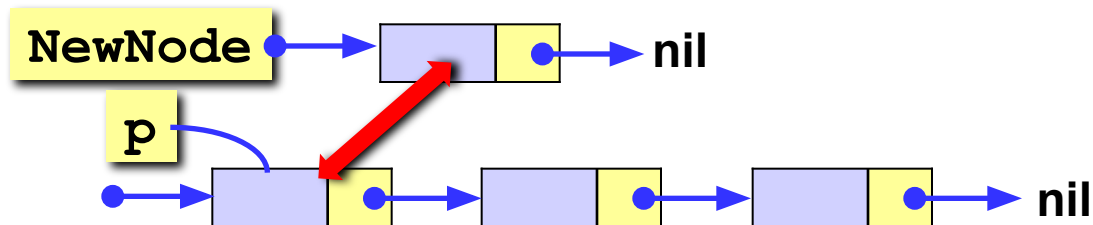
Что плохо?

Добавление узла перед заданным (II)

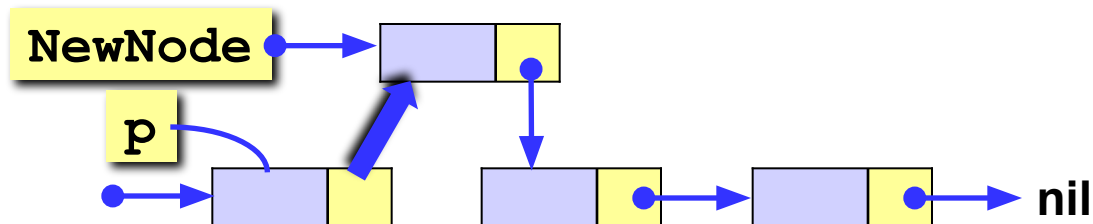
Задача: вставить узел перед заданным без поиска предыдущего.

Алгоритм:

- 1) поменять местами данные нового узла и узла **p**;



- 2) установить ссылку узла **p** на **NewNode**.



```

procedure AddBefore2 ( p, NewNode: PNode );
var temp: Node;
begin
    temp := p^; p^ := NewNode^;
    NewNode^ := temp;
    p^.next := NewNode;
end;

```



Так нельзя, если $p = \text{nil}$ или адреса узлов где-то еще запоминаются!

Поиск слова в списке

Задача:

найти в списке заданное слово или определить, что его нет.

Функция Find:

ВХОД: слово (символьная строка);

ВЫХОД: адрес узла, содержащего это слово или **nil**.

Алгоритм: проход по списку.

ищем это слово

результат – адрес узла
или **nil** (нет такого)

```
function Find(Head: PNode; NewWord: string): PNode;  
var q: PNode;  
begin  
    q := Head;  
    while (q <> nil) and (NewWord <> q^.word) do  
        q := q^.next;  
    Result := q;  
end;
```

пока не дошли до конца списка
и слово не равно заданному

Куда вставить новое слово?

Задача:

найти узел, перед которым нужно вставить, заданное слово, так чтобы в списке сохранился алфавитный порядок слов.

Функция `FindPlace`:

ВХОД: слово (символьная строка);

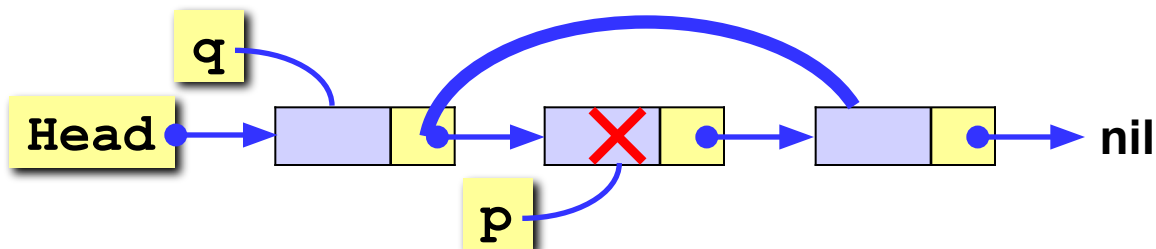
ВЫХОД: адрес узла, перед которым нужно вставить это слово или `nil`, если слово нужно вставить в конец списка.

```
function FindPlace(Head: PNode; NewWord: string): PNode;  
var q: PNode;  
begin  
    q := Head;  
    while (q <> nil) and (NewWord > q^.word) do  
        q := q^.next;  
    Result := q;  
end;
```

СЛОВО `NewWord` СТОИТ ПО
алфавиту *перед* `q^.word`

Удаление узла

Проблема: нужно знать адрес предыдущего узла q .



```

procedure DeleteNode ( var Head: PNode; p: PNode );
var q: PNode;
begin
  if Head = p then
    Head := p^.next
  else begin
    q := Head;
    while (q <> nil) and (q^.next <> p) do
      q := q^.next;
    if q <> nil then q^.next := p^.next;
  end;
  Dispose(p);
end;

```

особый случай:
удаляем первый узел

ищем узел, такой что
 $q^.next = p$

освобождение памяти

Алфавитно-частотный словарь

Алгоритм:

- 1) открыть файл на чтение;

```
var F: Text;  
...  
Assign(F, 'input.dat');  
Reset ( F );
```

- 2) прочитать очередное слово (как?)
- 3) если файл закончился, то перейти к шагу 7;
- 4) если слово найдено, увеличить счетчик (поле **count**);
- 5) если слова нет в списке, то
 - создать новый узел, заполнить поля (**CreateNode**);
 - найти узел, перед которым нужно вставить слово (**FindPlace**);
 - добавить узел (**AddBefore**);
- 6) перейти к шагу 2;
- 7) закрыть файл **Close (F);**
- 8) вывести список слов, используя проход по списку.

Как прочитать одно слово из файла?

Алгоритм:

- 1) пропускаем все спецсимволы и пробелы (с кодами ≤ 32)
- 2) читаем все символы до первого пробела или спецсимвола

```
function GetWord ( F: Text ) : string;  
var c: char;  
begin  
    Result := ''; { пустая строка }  
    c := ' ';      { пробел - чтобы войти в цикл }  
    { пропускаем спецсимволы и пробелы }  
    while not eof(f) and (c <= ' ') do  
        read(F, c);  
    { читаем слово }  
    while not eof(f) and (c > ' ') do begin  
        Result := Result + c;  
        read(F, c);  
    end;  
end;
```



**Можно поменять местами
строчки в цикле?**

Полная программа

```
type PNode = ^Node;
      Node = record ... end; { новые типы данных }
var Head: PNode;             { адрес головы списка }
    NewNode, q: PNode; { вспомогательные указатели }
    w: string;              { слово из файла }
    F: text;                { файловая переменная }
    count: integer;         { счетчик разных слов }
{ процедуры и функции }
begin
  Head := nil;
  Assign ( F, 'input.txt' );
  Reset ( F );
  { читаем слова из файла, строим список }
  Close ( F );
  { выводим список в другой файл }
end.
```

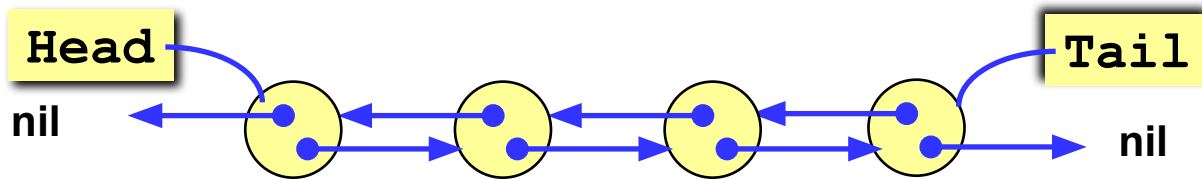
Полная программа (II)

```
{ читаем слова из файла, строим список }
while True do begin           { бесконечный цикл }
  w := GetWord ( F );        { читаем слово }
  if w = '' then break;      { слова закончились, выход }
  q := Find ( Head, w );     { ищем слово в списке }
  if q <> nil then            { нашли, увеличить счетчик }
    q^.count := q^.count + 1
  else begin                  { не нашли, добавить в список }
    NewNode := CreateNode ( w );
    q := FindPlace ( Head, w );
    AddBefore ( Head, q, NewNode );
  end;
end;
```

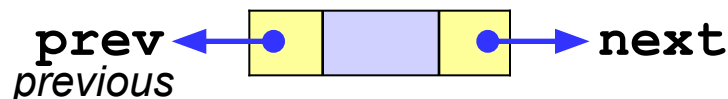
Полная программа (III)

```
{ выводим список в другой файл }  
q := Head;           { проход с начала списка }  
count := 0;          { обнулили счетчик слов }  
Assign(F, 'output.txt');  
Rewrite(F);  
while q <> nil do begin { пока не конец списка }  
    count := count + 1; { еще одно слово }  
    writeln ( F, q^.word, ': ', q^.count );  
    q := q^.next;       { перейти к следующему }  
end;  
writeln ( F, 'Найдено ',  
          count, ' разных слов.' );  
Close(F);
```

Двусвязные списки



Структура узла:



```

type PNode = ^Node;      { указатель на узел }
Node = record            { структура узла }
    word: string[40];    { слово }
    count: integer;      { счетчик повторений }
    next: PNode;         { ссылка на следующий }
    prev: PNode;         { ссылка на предыдущий }
end;
  
```

Адреса «головы» и «хвоста»:

```

var Head, Tail: PNode;
...
Head := nil; Tail := nil;
  
```



можно двигаться в
обе стороны



нужно работать с
двумя указателями
вместо одного

Задания

- «4»:** «Собрать» из этих функций программу для построения алфавитно-частотного словаря. В конце файла вывести общее количество разных слов (количество элементов списка).
- «5»:** То же самое, но использовать двусвязные списки.
- «6»:** То же самое, что и на «5», но вывести список слов в порядке убывания частоты, то есть, сначала те слова, которые встречаются чаще всего.

Динамические структуры данных (язык Паскаль)

Тема 5. Стеки, очереди, деки

Стек



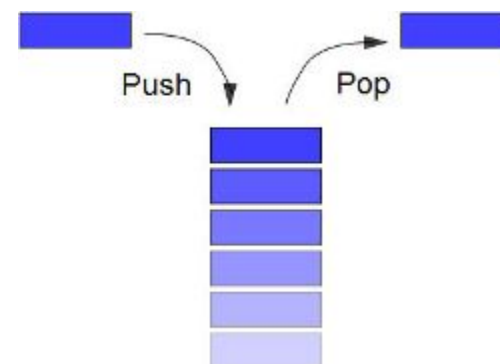
Стек – это линейная структура данных, в которой добавление и удаление элементов возможно только с одного конца (**вершины стека**). *Stack* = кipa, куча, стопка (англ.)

LIFO = Last In – First Out

«Кто последним вошел, тот первым вышел».

Операции со стеком:

- 1) добавить элемент на вершину (*Push* = втолкнуть);
- 2) снять элемент с вершины (*Pop* = вылететь со звуком).



Пример задачи

Задача: вводится символьная строка, в которой записано выражение со скобками трех типов: `[]`, `{ }` и `()`. Определить, верно ли расставлены скобки (не обращая внимания на остальные символы). Примеры:

`[()] { }` `] [` `[({)] }`

Упрощенная задача: то же самое, но с одним видом скобок.

Решение: счетчик вложенности скобок. Последовательность правильная, если в конце счетчик равен нулю и при проходе не разу не становился отрицательным.

(())	()
1	2	1	0	1	0

(()))	(
1	2	1	0	-1	0

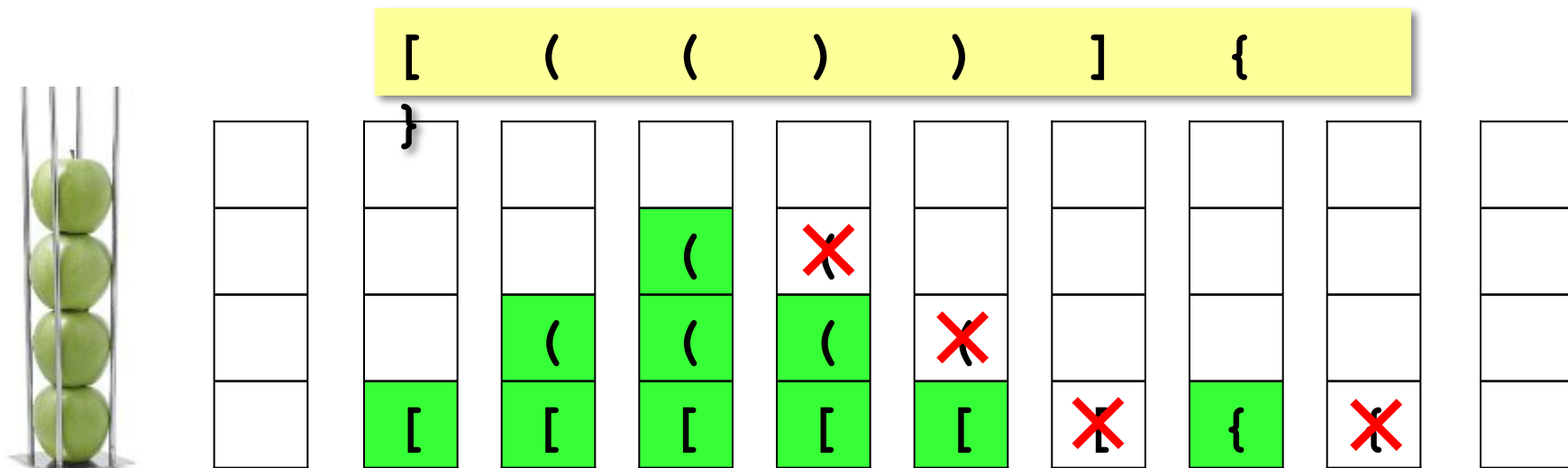
(())	(
1	2	1	0	1



Можно ли решить исходную задачу так же, но с тремя счетчиками?

	[({)]	}
(:	0	1		0		
[:	0	1			0	
{ :	0		1			0

Решение задачи со скобками



Алгоритм:

- 1) в начале стек пуст;
- 2) в цикле просматриваем все символы строки по порядку;
- 3) если очередной символ – открывающая скобка, заносим ее на вершину стека;
- 4) если символ – закрывающая скобка, проверяем вершину стека: там должна быть **соответствующая** открывающая скобка (если это не так, то ошибка);
- 5) если в конце стек не пуст, выражение неправильное.

Реализация стека (массив)

Структура-стек:

```
const MAXSIZE = 100;  
type Stack = record { стек на 100 символов }  
    data: array[1..MAXSIZE] of char;  
    size: integer; { число элементов }  
end;
```

Добавление элемента:

```
procedure Push( var S: Stack; x: char );  
begin  
    if S.size = MAXSIZE then Exit;  
    S.size := S.size + 1;  
    S.data[S.size] := x;  
end;
```

ошибка:
переполнение
стека

добавить элемент



Что плохо?

Реализация стека (массив)

Снятие элемента с вершины:

```
function Pop ( var S:Stack ): char;  
begin  
    if S.size = 0 then begin  
        Result := char(255);  
        Exit;  
    end;  
    Result := S.data[S.size];  
    S.size := S.size - 1;  
end;
```

ошибка:
стек пуст

Пустой или нет?

```
function isEmpty ( S: Stack ): Boolean;  
begin  
    Result := (S.size = 0);  
end;
```

Программа

```
var br1, br2, expr: string;
    i, k: integer;
    upper: char;      { то, что сняли со стека }
    error: Boolean;   { признак ошибки }
    S: Stack;
begin
    br1 := '([{' ; br2 := ')]}' ;
    S.size := 0;
    error := False;
    writeln('Введите выражение со скобками');
    readln(expr);
    ... { здесь будет основной цикл обработки }
    if not error and isEmpty(S) then
        writeln('Выражение правильное.')
    else writeln('Выражение неправильное.')
end.
```

открывающие скобки

закрывающие скобки

Обработка строки (основной цикл)

```
for i:=1 to length(expr)
```

```
do begin
```

```
  for k:=1 to 3 do begin
```

```
    if expr[i] = br1[k] then begin { откр. скобка }
```

```
      Push(S, expr[i]); { втолкнуть в стек }
```

```
      break;
```

```
    end;
```

```
    if expr[i] = br2[k] then begin { закр. скобка }
```

```
      upper := Pop(S); { снять символ со стека }
```

```
      error := upper <> br1[k];
```

```
      break;
```

```
    end;
```

```
  end;
```

```
  if error then break;
```

```
end;
```

ЦИКЛ ПО ВСЕМ СИМВОЛАМ
строки **expr**

ЦИКЛ ПО ВСЕМ ВИДАМ СКОБОК

ошибка: стек пуст
или не та скобка

была ошибка: дальше нет
смысла проверять

Реализация стека (список)

Структура узла:

```
type PNode = ^Node; { указатель на узел }  
    Node = record  
        data: char; { данные }  
        next: PNode; { указатель на след. элемент }  
    end;
```

Добавление элемента:

```
procedure Push( var Head: PNode; x: char);  
var NewNode: PNode;  
begin  
    New(NewNode); { выделить память }  
    NewNode^.data := x; { записать символ }  
    NewNode^.next := Head; { сделать первым узлом }  
    Head := NewNode;  
end;
```

Реализация стека (список)

Снятие элемента с вершины:

```
function Pop ( var Head: PNode ): char;  
var q: PNode;  
begin  
    if Head = nil then begin { стек пуст }  
        Result := char(255); { неиспользуемый символ }  
        Exit;  
    end;  
    Result := Head^.data; { взять верхний символ }  
    q := Head; { запомнить вершину }  
    Head := Head^.next; { удалить вершину из стека }  
    Dispose(q); { удалить из памяти }  
end;
```



Можно ли переставлять операторы?

Реализация стека (список)

Пустой или нет?

```
function isEmpty ( S: Stack ): Boolean;  
begin  
    Result := (S = nil);  
end;
```

Изменения в основной программе:

var S: Stack;	var S: PNode;
...	
S.size = 0;	S := nil;



Больше ничего не меняется!

Вычисление арифметических выражений

Как вычислять автоматически:

$(a + b) / (c + d - 1)$

Инфиксная запись

(знак операции **между** операндами)



необходимы скобки!

Префиксная запись (знак операции **до** операндов)

$/$ $a + b$ $c + d - 1$

1

польская нотация,
[Jan Łukasiewicz](#) (1920)



скобки не нужны, можно однозначно
вычислить!

Постфиксная запись (знак операции **после** операндов)

$a + b$ $c + d - 1$

/

обратная польская нотация,
[F. L. Bauer](#) F. L. Bauer and [E. W. Dijkstra](#)

Запишите в постфиксной форме

$$(32 * 6 - 5) * (2 * 3 + 4) / (3 + 7 * 2)$$

$$(2 * 4 + 3 * 5) * (2 * 3 + 18 / 3 * 2) * (12 - 3)$$

$$(4 - 2 * 3) * (3 - 12 / 3 / 4) * (24 - 3 * 12)$$

Вычисление выражений

Постфиксная форма:

X = a b + c d + 1 - /

					d		1		
		b		c	c	c+d	c+d	c+d-1	
	a	a	a+b	a+b	a+b	a+b	a+b	a+b	x

Алгоритм:

- 1) взять очередной элемент;
- 2) если это не знак операции, добавить его в стек;
- 3) если это знак операции, то
 - взять из стека два операнда;
 - выполнить операцию и записать результат в стек;
- 4) перейти к шагу 1.

Системный стек (*Windows – 1 Мб*)

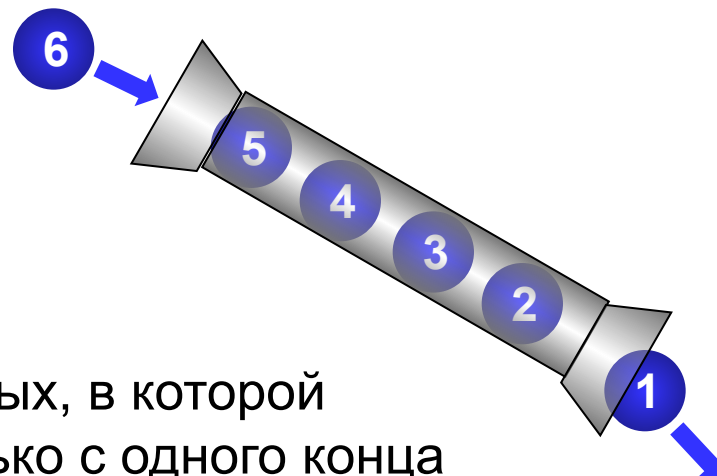
Используется для

- 1) размещения **локальных переменных**;
- 2) хранения **адресов возврата** (по которым переходит программа после выполнения функции или процедуры);
- 3) передачи **параметров** в функции и процедуры;
- 4) временного хранения данных (в программах на языке *Ассемблер*).

Переполнение стека (*stack overflow*):

- 1) слишком много локальных переменных
(**выход** – использовать динамические массивы);
- 2) очень много рекурсивных вызовов функций и процедур
(**выход** – переделать алгоритм так, чтобы уменьшить глубину рекурсии или отказаться от нее вообще).

Очередь



Очередь – это линейная структура данных, в которой добавление элементов возможно только с одного конца (**конца очереди**), а удаление элементов – только с другого конца (**начала очереди**).

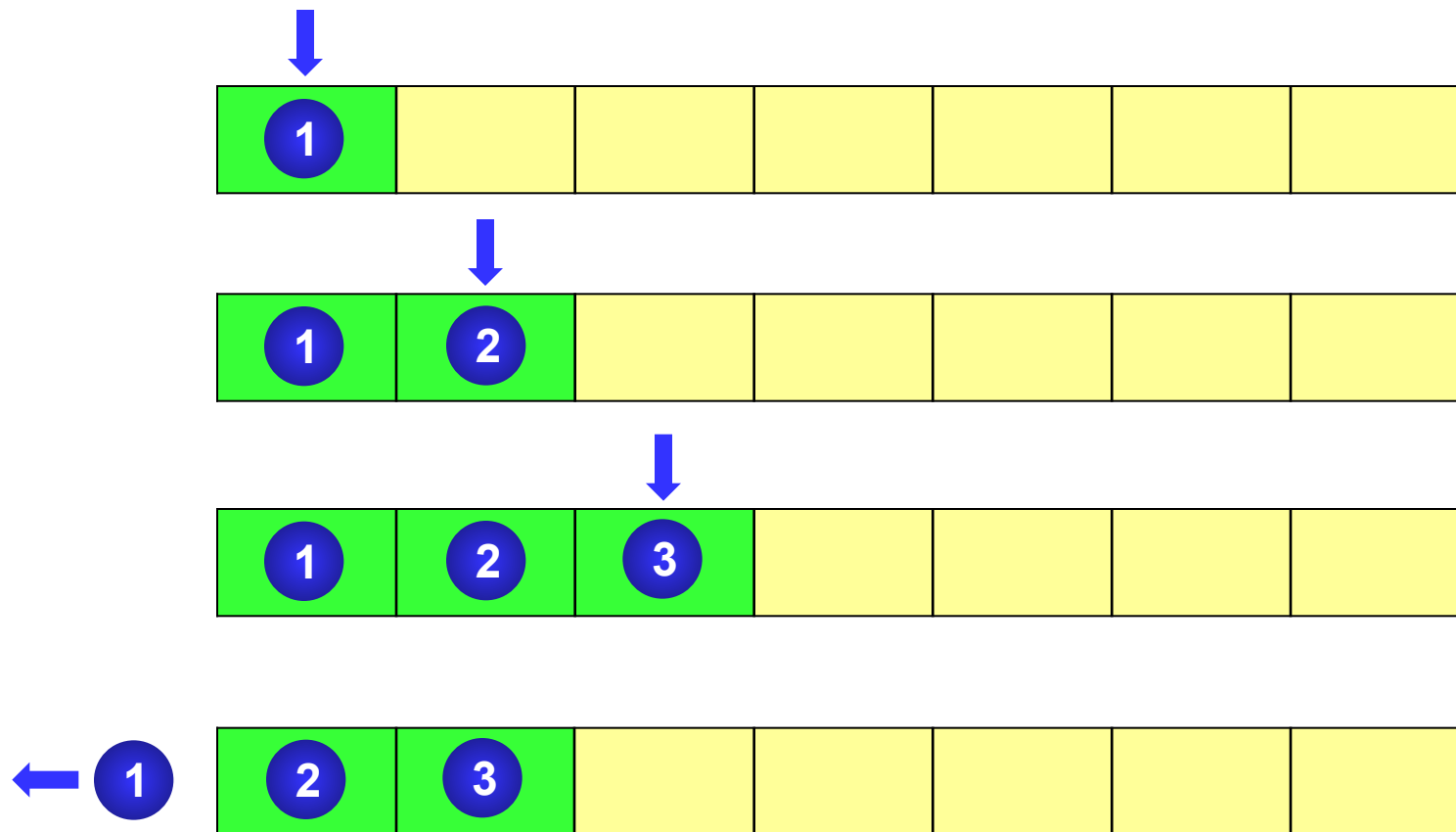
FIFO = *First In – First Out*

«Кто первым вошел, тот первым вышел».

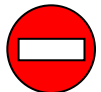
Операции с очередью:

- 1) добавить элемент в конец очереди (*PushTail* = втолкнуть в конец);
- 2) удалить элемент с начала очереди (*Pop*).

Реализация очереди (массив)

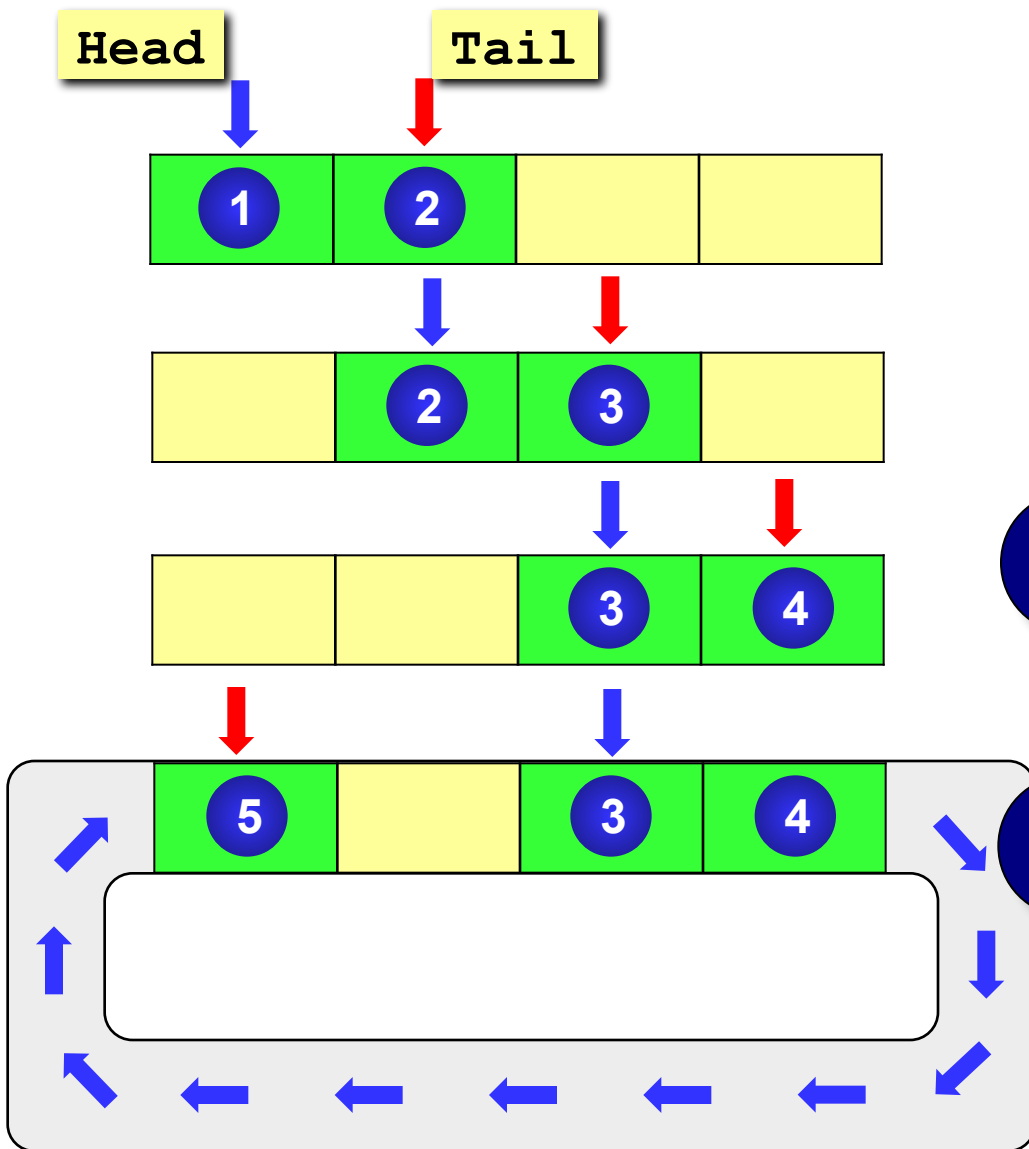


самый простой способ



- 1) нужно заранее выделить массив;
- 2) при выборке из очереди нужно сдвигать все элементы.

Реализация очереди (кольцевой массив)



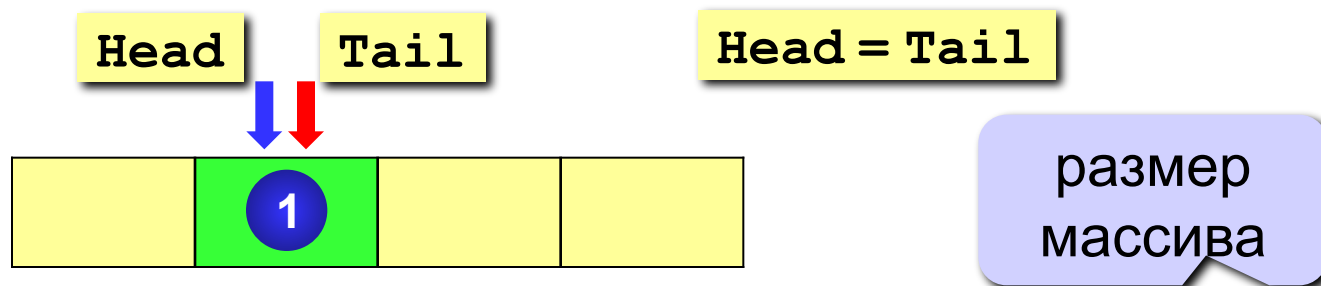
Сколько элементов можно хранить в такой очереди?



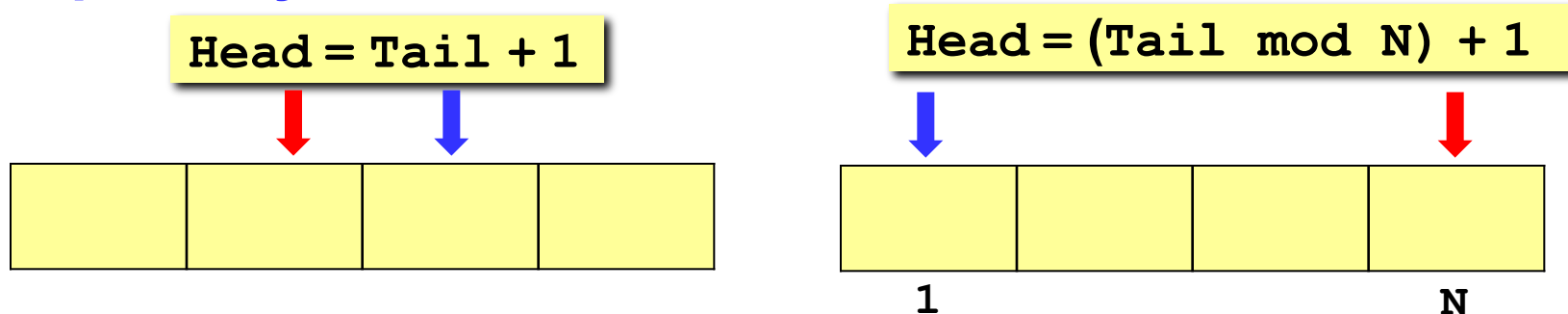
Как различить состояния «очередь пуста» и «очередь полна»?

Реализация очереди (кольцевой массив)

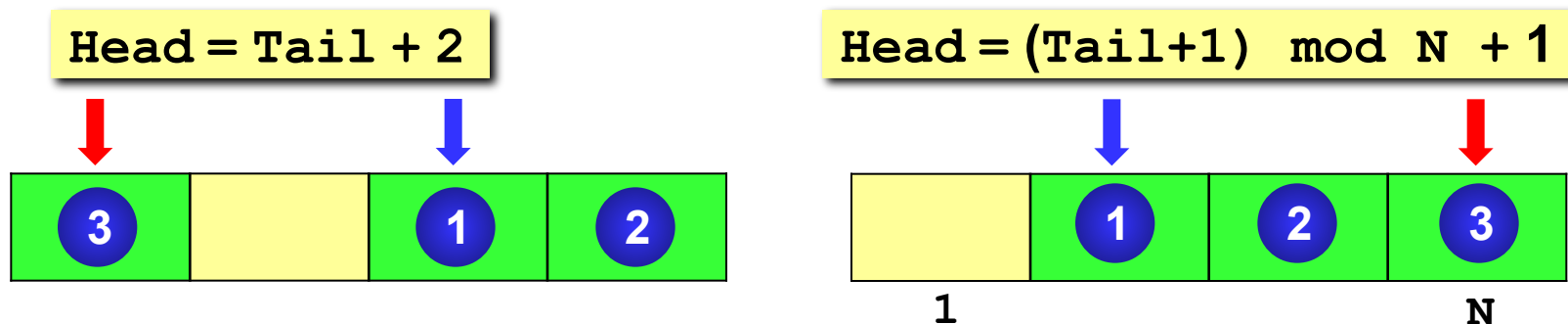
В очереди 1 элемент:



Очередь пуста:



Очередь полна:



Реализация очереди (кольцевой массив)

Структура данных:

```
type Queue = record
    data: array[1..MAXSIZE] of integer;
    head, tail: integer;
end;
```

Добавление в очередь:

```
procedure PushTail( var Q: Queue; x: integer);
begin
    if Q.head = (Q.tail+1) mod MAXSIZE + 1
    then Exit; { очередь полна, не добавить }
    Q.tail := Q.tail mod MAXSIZE + 1;
    Q.data[Q.tail] := x;
end;
```

замкнуть
в кольцо

Реализация очереди (кольцевой массив)

Выборка из очереди:

```
function Pop ( var S: Queue ): integer;  
begin  
    if Q.head = Q.tail mod MAXSIZE + 1 then begin  
        Result := MaxInt;  
        Exit;  
    end;  
    Result := Q.data[Q.head];  
    Q.head := Q.head mod MAXSIZE + 1;  
end;
```

очередь
пуста

максимальное
целое число

взять первый
элемент

удалить его из
очереди

Реализация очереди (списки)

Структура узла:

```
type PNode = ^Node;  
  Node = record  
    data: integer;  
    next: PNode;  
  end;
```

Тип данных «очередь»:

```
type Queue = record  
  head, tail: PNode;  
end;
```

Реализация очереди (списки)

Добавление элемента:

```
procedure PushTail( var Q: Queue; x: integer );  
var NewNode: PNode;  
begin  
    New(NewNode);  
    NewNode^.data := x;  
    NewNode^.next := nil;  
    if Q.tail <> nil then  
        Q.tail^.next := NewNode;  
    Q.tail := NewNode; { новый узел - в конец }  
    if Q.head = nil then Q.head := Q.tail;  
end;
```

создаем
новый узел

если в списке уже
что-то было,
добавляем в конец

если в списке
ничего не было, ...

Реализация очереди (списки)

Выборка элемента:

```
function Pop ( var S: Queue ) : integer;  
var top: PNode;  
begin  
    if Q.head = nil then begin  
        Result := MaxInt;  
        Exit;  
    end;  
    top := Q.head;  
    Result := top^.data;  
    Q.head := top^.next;  
    if Q.head = nil then Q.tail := nil;  
    Dispose(top);  
end;
```

если список
пуст, ...

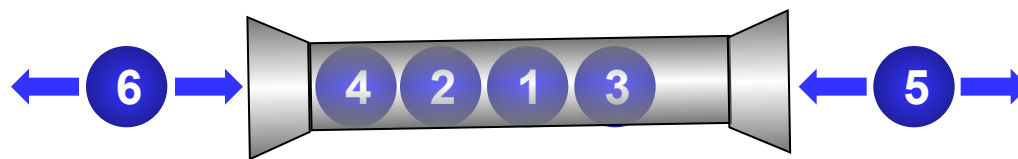
запомнили
первый элемент

если в списке
ничего не
осталось, ...

освободить
память

Дек

Дек (*deque* = ***double ended queue***, очередь с двумя концами) – это линейная структура данных, в которой добавление и удаление элементов возможно с обоих концов.



Операции с деком:

- 1) добавление элемента в начало (*Push*);
- 2) удаление элемента с начала (*Pop*);
- 3) добавление элемента в конец (*PushTail*);
- 4) удаление элемента с конца (*PopTail*).

Реализация:

- 1) кольцевой массив;
- 2) двусвязный список.

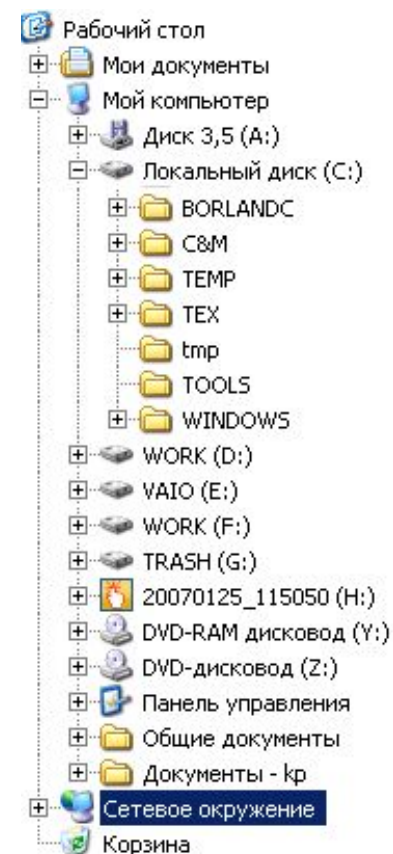
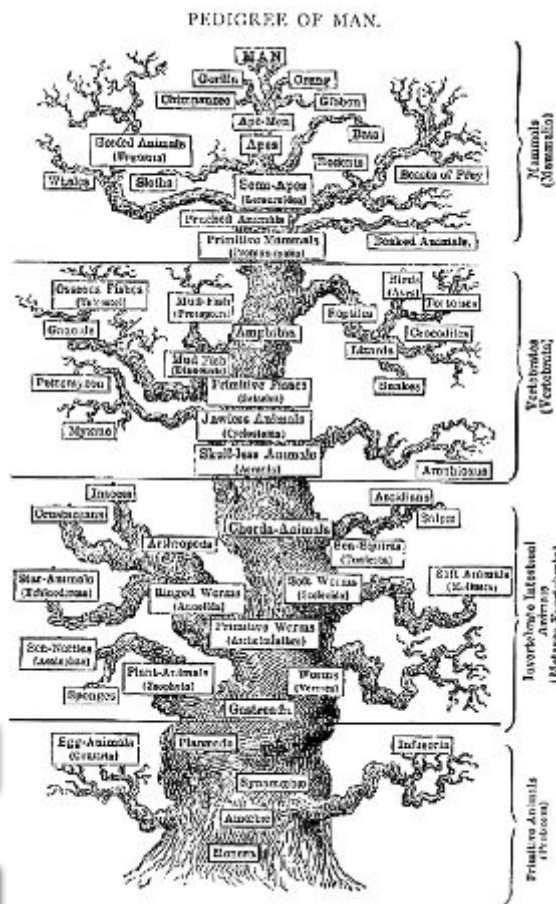
Задания

- «4»:** В файле `input.dat` находится список чисел (или слов). Переписать его в файл `output.dat` в обратном порядке.
- «5»:** Составить программу, которая вычисляет значение арифметического выражения, записанного в постфиксной форме, с помощью стека. Выражение правильное, допускаются только однозначные числа и знаки `+`, `-`, `*`, `/`.
- «6»:** То же самое, что и на «5», но допускаются многозначные числа.

Динамические структуры данных (язык Паскаль)

Тема 6. Деревья

Деревья



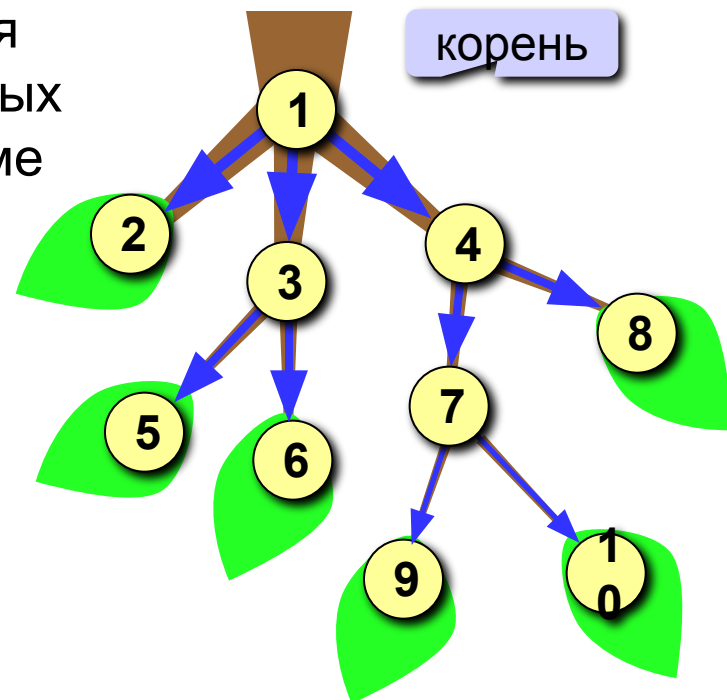
Что общего во всех примерах?

Деревья

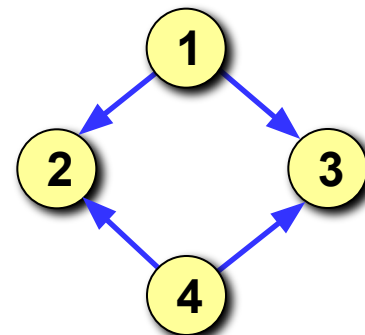
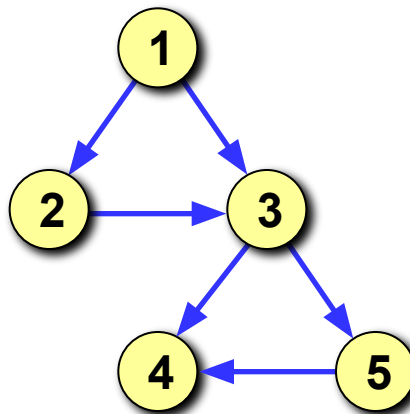
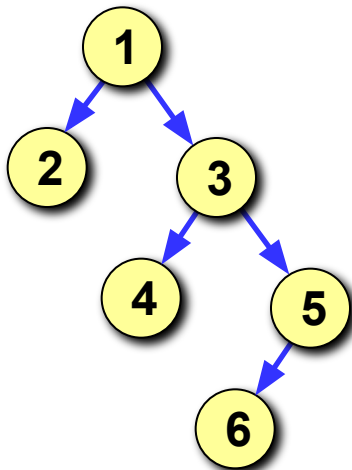
Дерево – это структура данных, состоящая из узлов и соединяющих их направленных ребер (дуг), причем в каждый узел (кроме корневого) ведет ровно одна дуга.

Корень – это начальный узел дерева.

Лист – это узел, из которого не выходит ни одной дуги.



Какие структуры – не деревья?



Деревья



С помощью деревьев изображаются отношения подчиненности (иерархия, «старший – младший», «родитель – ребенок»).

Предок узла x – это узел, из которого существует путь по стрелкам в узел x .

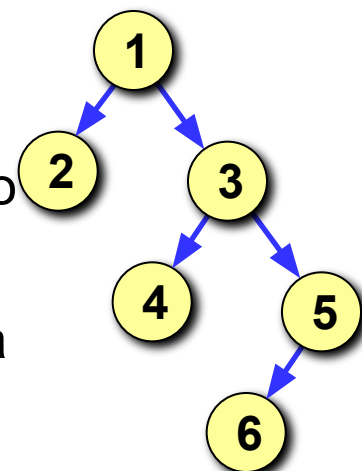
Потомок узла x – это узел, в который существует путь по стрелкам из узла x .

Родитель узла x – это узел, из которого существует дуга непосредственно в узел x .

Сын узла x – это узел, в который существует дуга непосредственно из узла x .

Брат узла x (*sibling*) – это узел, у которого тот же родитель, что и у узла x .

Высота дерева – это наибольшее расстояние от корня до листа (количество дуг).



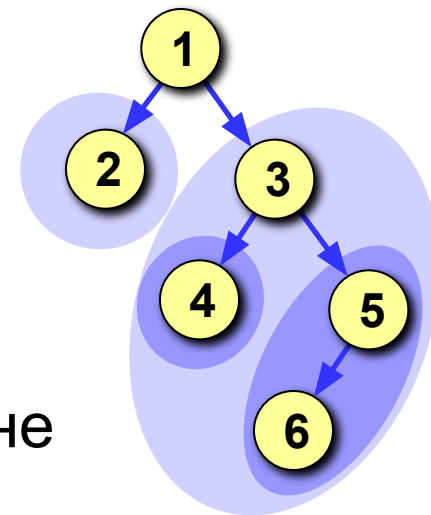
Дерево – рекурсивная структура данных

Рекурсивное определение:

1. Пустая структура – это дерево.
2. Дерево – это корень и несколько связанных с ним деревьев.

Двоичное (бинарное) дерево – это дерево, в котором каждый узел имеет не более двух сыновей.

1. Пустая структура – это двоичное дерево.
2. Двоичное дерево – это корень и два связанных с ним двоичных дерева (левое и правое поддеревья).

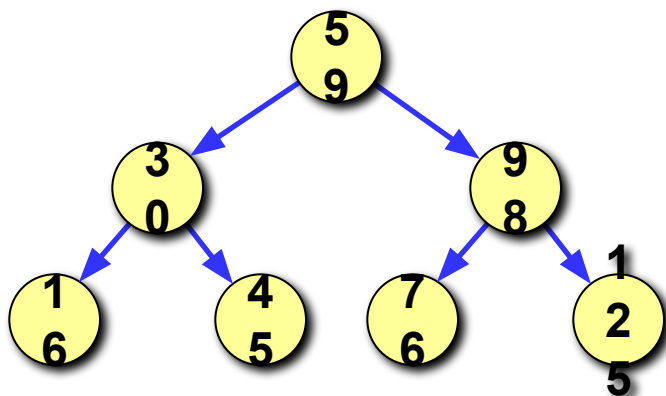


- 1) поиск данных в специально построенных деревьях (базы данных);
- 2) сортировка данных;
- 3) вычисление арифметических выражений;
- 4) кодирование (метод Хаффмана).

```
type PNode = ^Node;           { указатель на узел }
Node = record
    data: integer;            { полезные данные }
    left, right: PNode;      { ссылки на левого и
                              правого сыновей }
end;
```

Двоичные деревья поиска

Ключ – это характеристика узла, по которой выполняется поиск (чаще всего – одно из полей структуры).



Какая закономерность?

Слева от каждого узла находятся узлы с меньшими ключами, а справа – с бóльшими.

Как искать ключ, равный x :

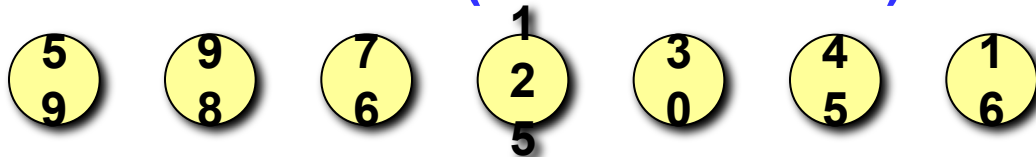
- 1) если дерево пустое, ключ не найден;
- 2) если ключ узла равен x , то стоп.
- 3) если ключ узла меньше x , то искать x в левом поддереве;
- 4) если ключ узла больше x , то искать x в правом поддереве.



Сведение задачи к такой же задаче меньшей размерности – это ...?

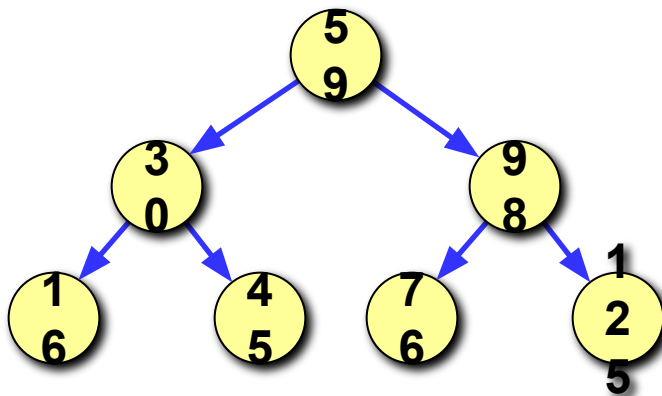
Двоичные деревья поиска

Поиск в массиве (N элементов):



При каждом сравнении отбрасывается 1 элемент.
Число сравнений – N .

Поиск по дереву (N элементов):



При каждом сравнении отбрасывается половина оставшихся элементов.
Число сравнений $\sim \log_2 N$.



быстрый поиск



- 1) нужно заранее построить дерево;
- 2) желательно, чтобы дерево было минимальной высоты.

Реализация алгоритма поиска

{ Функция Search – поиск по дереву

Вход: Tree – адрес корня,
x – что ищем

Выход: адрес узла или nil (не нашли) }

```
function Search(Tree: PNode; x: integer): PNode;  
begin
```

```
  if Tree = nil then begin  
    Result := nil;  
    Exit;
```

дерево пустое:
ключ не нашли...

```
  end;
```

```
  if x = Tree^.data then  
    Result := Tree
```

нашли,
возвращаем
адрес корня

```
  else
```

```
    if x < Tree^.data then
```

```
      Result := Search(Tree^.left, x)
```

```
    else Result := Search(Tree^.right, x);
```

искать в
левом
поддереве

```
  end;
```

искать в правом поддереве

Как построить дерево поиска?

{ Процедура AddToTree – добавить элемент
Вход: Tree – адрес корня,
 x – что добавляем }

```
procedure AddToTree( var Tree: PNode; x: integer );  
begin  
  if Tree = nil then begin  
    New(Tree);  
    Tree^.data := x;  
    Tree^.left := nil;  
    Tree^.right := nil;  
    Exit;  
  end;  
  if x < Tree^.data then  
    AddToTree(Tree^.left, x)  
  else AddToTree(Tree^.right, x);  
end;
```

адрес корня может
измениться

дерево пустое: создаем
новый узел (корень)

добавляем к левому
или правому
поддереву

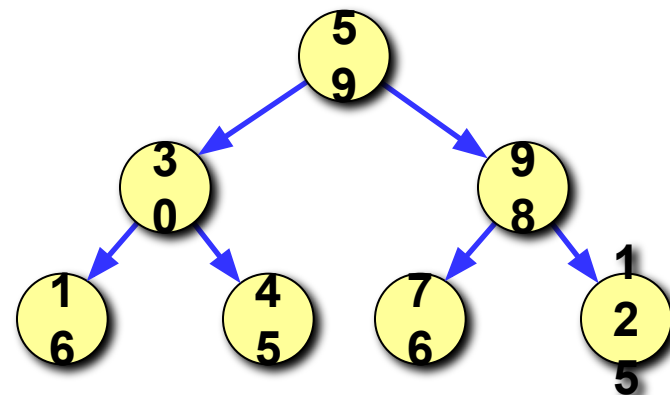


Минимальная высота не гарантируется!

Обход дерева

Обход дерева – это перечисление всех узлов в определенном порядке.

Обход ЛКП («левый – корень – правый»):



Обход ПКЛ («правый – корень – левый»):



Обход КЛП («корень – левый – правый»):



Обход ЛПК («левый – правый – корень»):



Обход дерева – реализация

{ Процедура LKP – обход дерева в порядке ЛКП
(левый – корень – правый)

Вход: Tree – адрес корня }

procedure LKP(Tree: PNode);

begin

if Tree = nil then Exit;

LKP(Tree^.left);

write(' ', Tree^.data);

LKP(Tree^.right);

end;

обход этой ветки
закончен

обход левого
поддерева

вывод данных корня

обход правого
поддерева

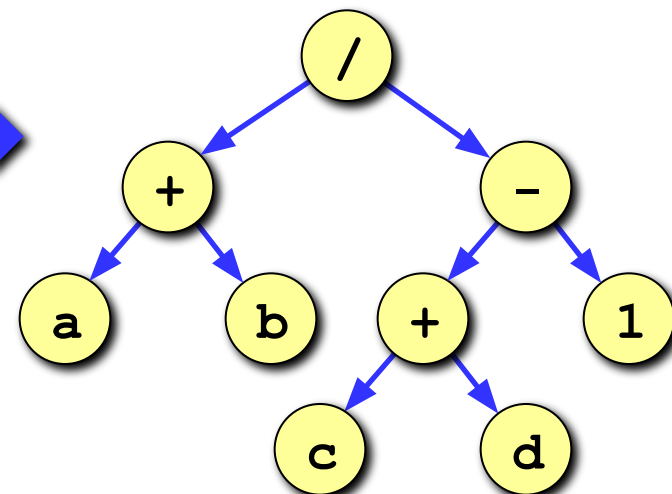


Для рекурсивной структуры удобно
применять рекурсивную обработку!

Разбор арифметических выражений

Как вычислять автоматически:

$(a + b) / (c + d - 1)$



Инфиксная запись, обход ЛКП

(знак операции **между** операндами)

$a + b / c + d -$



¹ необходимы скобки!

Префиксная запись, КЛП (знак операции **до** операндов)

$/ + a b - + c d$

польская нотация,
[Jan Łukasiewicz](#) (1920)



¹ скобки не нужны, можно однозначно вычислить!

Постфиксная запись, ЛПК (знак операции **после** операндов)

$a b + c d + 1 -$

обратная польская нотация,
[F. L. Bauer](#) F. L. Bauer and [E. W. Dijkstra](#)

Вычисление выражений

Постфиксная форма:

X = a b + c d + 1 - /

					d		1		
		b		c	c	c+d	c+d	c+d-1	
	a	a	a+b	a+b	a+b	a+b	a+b	a+b	x

Алгоритм:

- 1) взять очередной элемент;
- 2) если это не знак операции, добавить его в стек;
- 3) если это знак операции, то
 - взять из стека два операнда;
 - выполнить операцию и записать результат в стек;
- 4) перейти к шагу 1.

Вычисление выражений

Задача: в символьной строке записано правильное арифметическое выражение, которое может содержать только однозначные числа и знаки операций $+-*\backslash$. Вычислить это выражение.

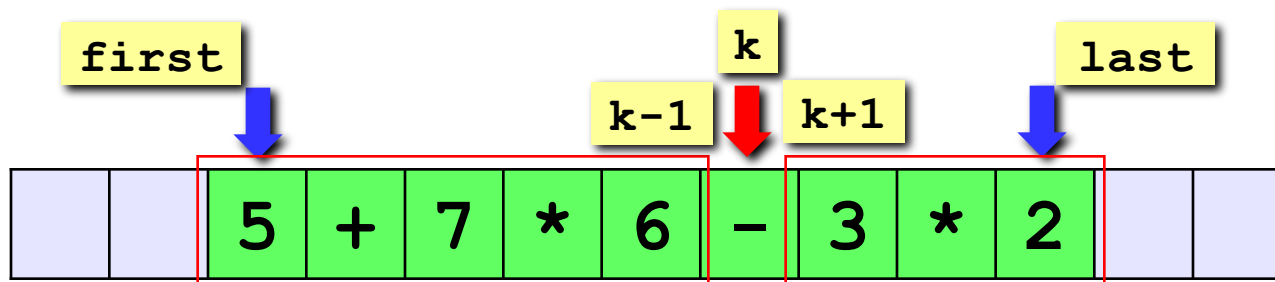
Алгоритм:

- 1) ввести строку;
- 2) построить дерево;
- 3) вычислить выражение по дереву.

Ограничения:

- 1) ошибки не обрабатываем;
- 2) многозначные числа не разрешены;
- 3) дробные числа не разрешены;
- 4) скобки не разрешены.

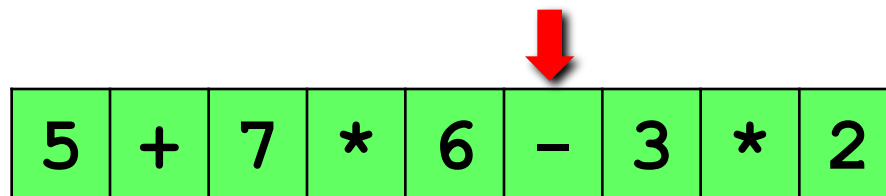
Построение дерева



Алгоритм:

- 1) если `first=last` (остался один символ – число), то создать новый узел и записать в него этот элемент; иначе...
- 2) среди элементов от `first` до `last` включительно найти **последнюю** операцию (элемент с номером k);
- 3) создать новый узел (корень) и записать в него **знак операции**;
- 4) рекурсивно применить этот алгоритм два раза:
 - построить **левое** поддерево, разобрав выражение из элементов массива с номерами от `first` до $k-1$;
 - построить **правое** поддерево, разобрав выражение из элементов массива с номерами от $k+1$ до `last`.

Как найти последнюю операцию?



Порядок выполнения операций

- умножение и деление;
- сложение и вычитание.

Приоритет (старшинство) – число, определяющее последовательность выполнения операций: раньше выполняются операции с большим приоритетом:

- умножение и деление (приоритет **2**);
- сложение и вычитание (приоритет **1**).



Нужно искать последнюю операцию с наименьшим приоритетом!

Приоритет операции

{ Функция Priority – приоритет операции
Вход: символ операции
Выход: приоритет или 100, если не операция
}

```
function Priority ( c: char ): integer;  
begin  
  case ( c ) of  
    '+', '-': Result := 1;  
    '*', '/': Result := 2;  
    else      Result := 100;  
  end;  
end;
```

сложение и
вычитание:
приоритет 1

умножение и
деление:
приоритет 2

это вообще не
операция

Номер последней операции

{ Функция LastOperation – номер последней операции
Вход: строка, номера первого и последнего
символов рассматриваемой части

Выход: номер символа – последней операции }

```
function LastOperation ( Expr: string;  
                        first, last: integer): integer;  
var MinPrt, i, k, prt: integer;  
begin  
    MinPrt := 100;  
    for i:=first to last do begin  
        prt := Priority ( Expr[i] );  
        if prt <= MinPrt then begin  
            MinPrt := prt;  
            k := i;  
        end;  
    end;  
    Result := k;  
end;
```

проверяем все
СИМВОЛЫ

нашли операцию с
МИНИМАЛЬНЫМ
приоритетом

вернуть номер
СИМВОЛА

Построение дерева

Структура узла

```
type PNode = ^Node;  
  Node = record  
    data: char;  
    left, right: PNode;  
  end;
```

Создание узла для числа (без потомков)

```
function NumberNode(c: char): PNode;  
begin  
  New(Result);  
  Result^.data := c;  
  Result^.left := nil;  
  Result^.right := nil;  
end;
```

ОДИН СИМВОЛ,
ЧИСЛО

возвращает адрес
созданного узла

Построение дерева

{ Функция MakeTree – построение дерева

Вход: строка, номера первого и последнего
символов рассматриваемой части

Выход: адрес построенного дерева }

```
function MakeTree ( Expr: string;  
                   first, last: integer): PNode;  
  
var k: integer;  
begin  
    if first = last then begin  
        Result := NumberNode ( Expr[first] );  
        Exit;  
    end;  
    k := LastOperation ( Expr, first, last );  
    New(Result);  
    Result^.data := Expr[k];  
    Result^.left := MakeTree ( Expr, first, k-1 );  
    Result^.right := MakeTree ( Expr, k+1, last );  
end;
```

ОСТАЛОСЬ
ТОЛЬКО ЧИСЛО

новый узел: операция

Вычисление выражения по дереву

```
{ Функция CalcTree - вычисление по дереву
  Вход:  адрес дерева
  Выход: значение выражения
}
function CalcTree(Tree: PNode): integer;
var num1, num2: integer;
begin
  if Tree^.left = nil then begin
    Result := Ord(Tree^.data) - Ord('0');
    Exit;
  end;
  num1 := CalcTree(Tree^.left);
  num2 := CalcTree(Tree^.right);
  case Tree^.data of
    '+': Result := num1+num2;
    '-': Result := num1-num2;
    '*': Result := num1*num2;
    '/': Result := num1 div num2;
    else Result := MaxInt;
  end;
end;
```

вернуть число,
если это ЛИСТ

ВЫЧИСЛЯЕМ
операнды
(поддерева)

ВЫПОЛНЯЕМ
операцию

некорректная
операция

Основная программа

```
{ Ввод и вычисление выражения с помощью  
дерева }  
program qq;  
var Tree: PNode;  
    Expr: string;  
{ процедуры и функции }  
begin  
    write('Введите выражение > ');  
    readln( Expr );  
    Tree := MakeTree( Expr, 1, Length(Expr) );  
    writeln(' = ', CalcTree(Tree) );  
end.
```

Дерево игры

Задача.

Перед двумя игроками лежат две кучки камней, в первой из которых 3, а во второй – 2 камня. У каждого игрока неограниченно много камней.

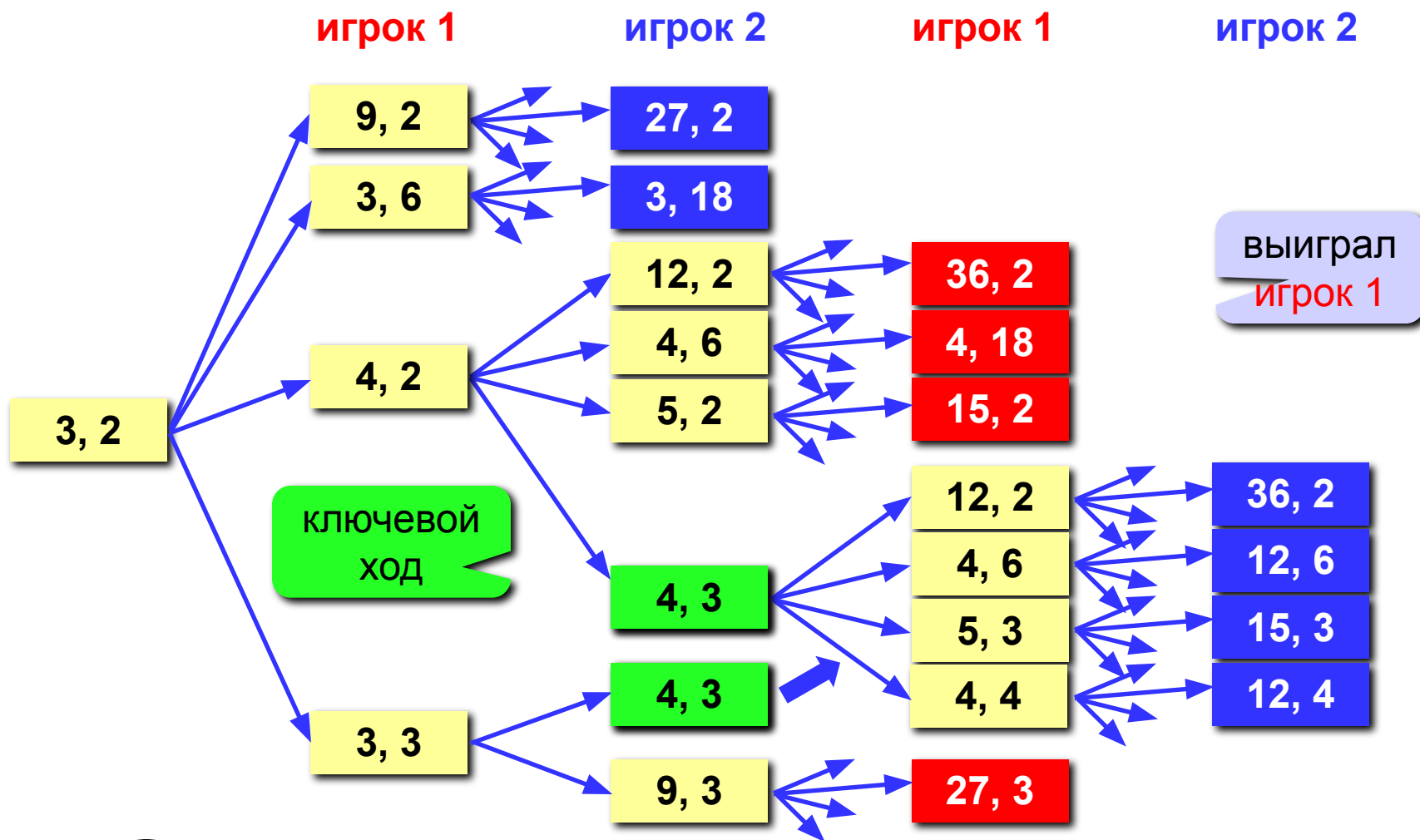
Игроки ходят по очереди. Ход состоит в том, что игрок или **увеличивает в 3 раза** число камней в какой-то куче, или **добавляет 1 камень** в какую-то кучу.

Выигрывает игрок, после хода которого общее число камней в двух кучах становится **не менее 16**.

Кто выигрывает при безошибочной игре – игрок, делающий первый ход, или игрок, делающий второй ход? Как должен ходить выигрывающий игрок?



Дерево игры



При правильной игре выиграет **игрок 2**!

Задания

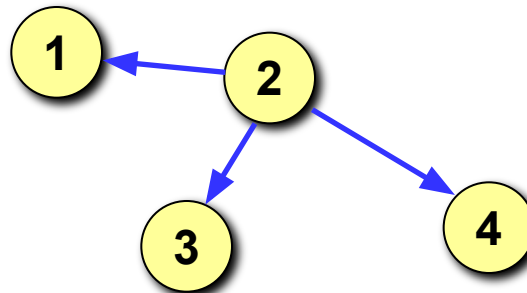
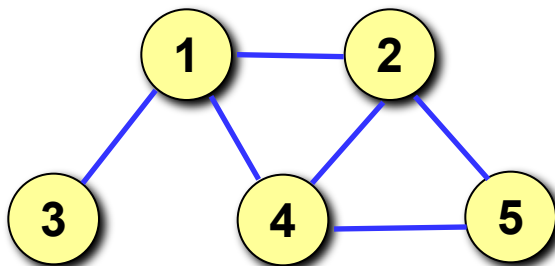
- «4»:** «Собрать» программу для вычисления правильного арифметического выражения, включающего только однозначные числа и знаки операций $+$, $-$, $*$, $/$.
- «5»:** То же самое, но допускаются также многозначные числа и скобки.
- «6»:** То же самое, что и на «5», но с обработкой ошибок (должно выводиться сообщение).

Динамические структуры данных (язык Паскаль)

Тема 7. Графы

Определения

Граф – это набор вершин (узлов) и соединяющих их ребер (дуг).



Направленный граф (ориентированный, орграф) – это граф, в котором все дуги имеют направления.

Цепь – это последовательность ребер, соединяющих две вершины (в орграфе – **путь**).

Цикл – это цепь из какой-то вершины в нее саму.

Взвешенный граф (сеть) – это граф, в котором каждому ребру приписывается вес (длина).



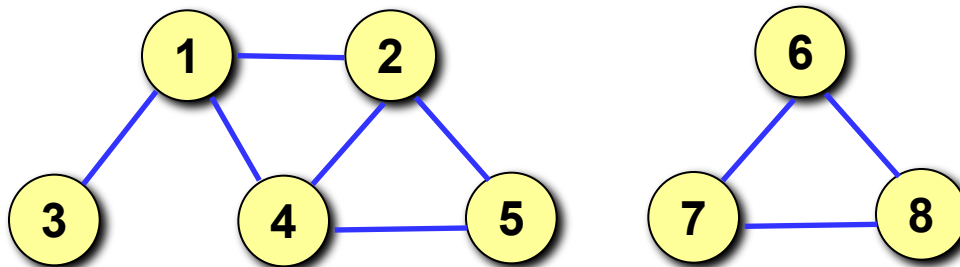
Дерево – это граф?

Да, без циклов!

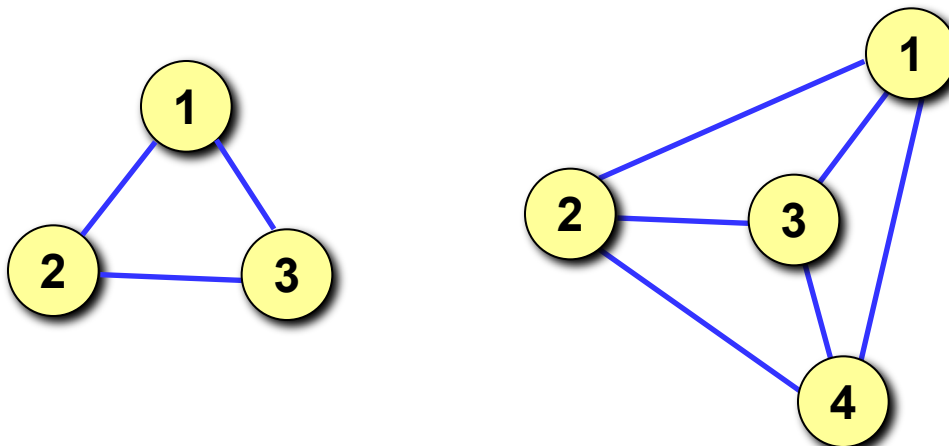
Определения

Связный граф – это граф, в котором существует цепь между каждой парой вершин.

k-связный граф – это граф, который можно разбить на **k** связных частей.

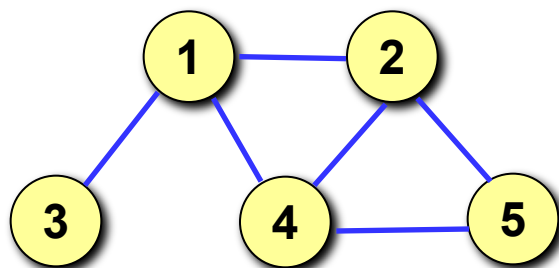


Полный граф – это граф, в котором проведены все возможные ребра (n вершин $\rightarrow n(n-1)/2$ ребер).



Описание графа

Матрица смежности – это матрица, элемент $M[i][j]$ которой равен 1, если существует ребро из вершины i в вершину j , и равен 0, если такого ребра нет.



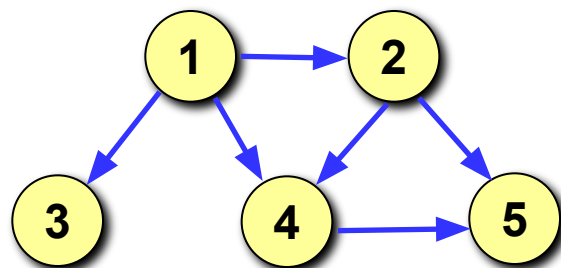
	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	1	1
3	1	0	0	0	0
4	1	1	0	0	1
5	0	1	0	1	0

Список смежности

1	2	3	4		
2	1	4	5		
3	1				
4	1	2	5		
5	2	4			



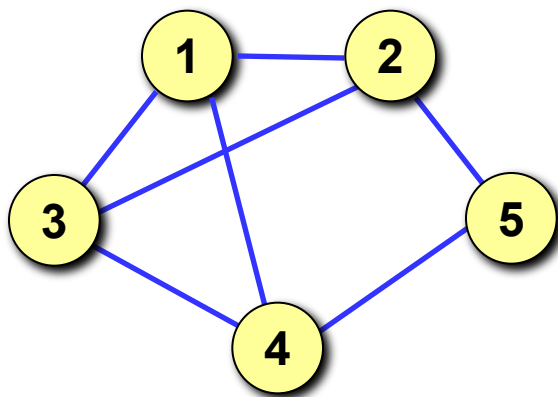
Симметрия!



	1	2	3	4	5
1	0	1	1	1	0
2	0	0	0	1	1
3	0	0	0	0	0
4	0	0	0	0	1
5	0	0	0	0	0

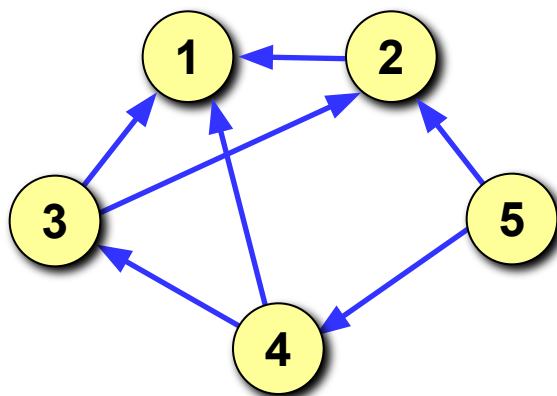
1	2	3	4		
2	4	5			
3					
4	5				
5					

Матрица и список смежности



	1	2	3	4	5
1					
2					
3					
4					
5					

1					
2					
3					
4					
5					

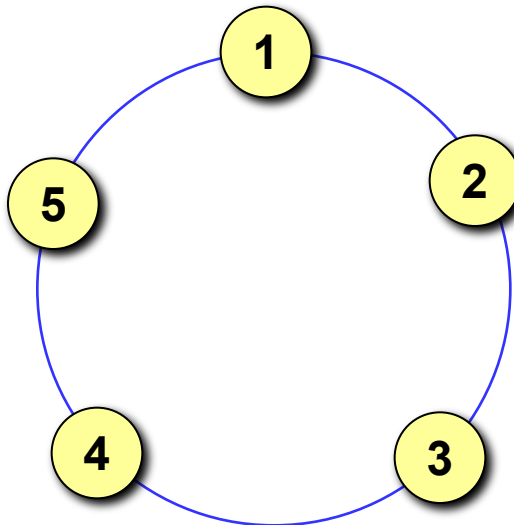


	1	2	3	4	5
1					
2					
3					
4					
5					

1					
2					
3					
4					
5					

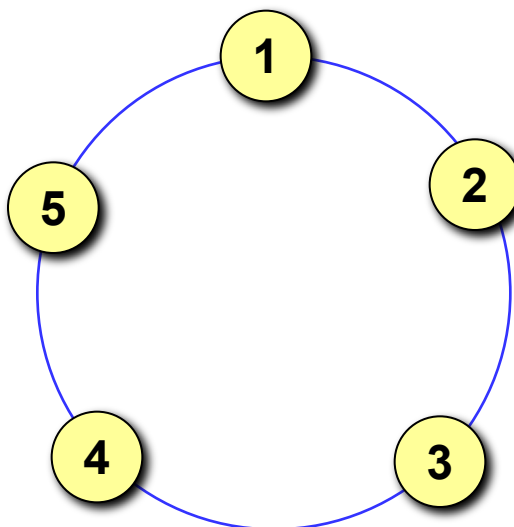
Построения графа по матрице смежности

	1	2	3	4	5
1	0	0	1	0	0
2	0	0	1	0	1
3	1	1	0	1	0
4	0	0	1	0	1
5	0	1	0	1	0



1					
2					
3					
4					
5					

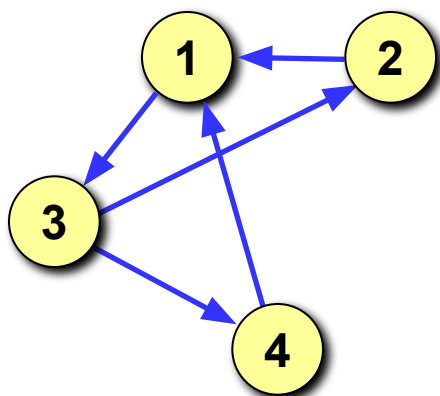
	1	2	3	4	5
1	0	0	1	1	1
2	0	1	0	1	0
3	0	1	0	1	0
4	1	1	0	0	0
5	0	1	1	0	0



1					
2					
3					
4					
5					

Как обнаружить цепи и циклы?

Задача: определить, существует ли цепь длины k из вершины i в вершину j (или цикл длиной k из вершины i в нее саму).



$M =$

	1	2	3	4
1	0	0	1	0
2	1	0	0	0
3	0	1	0	1
4	1	0	0	0

$M^2[i][j]=1$, если

$M[i][1]=1$	и	$M[1][j]=1$	или
$M[i][2]=1$	и	$M[2][j]=1$	или
$M[i][3]=1$	и	$M[3][j]=1$	или
$M[i][4]=1$	и	$M[4][j]=1$	

строка i

логическое
умножение

столбец j

логическое
сложение

Как обнаружить цепи и циклы?

Логическое умножение матрицы на себя:

матрица путей
длины 2

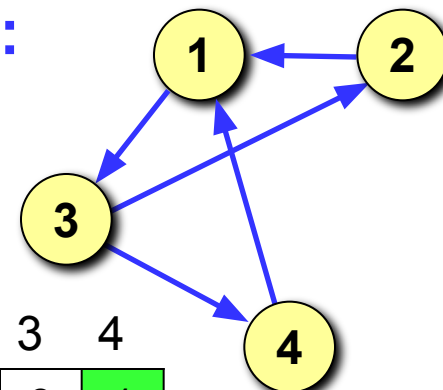
$$M^2 = M \otimes M$$

$$M^2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

$$M^2[3][1] = 0 \cdot 0 + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 = 1$$

маршрут 3-2-1

маршрут 3-4-1



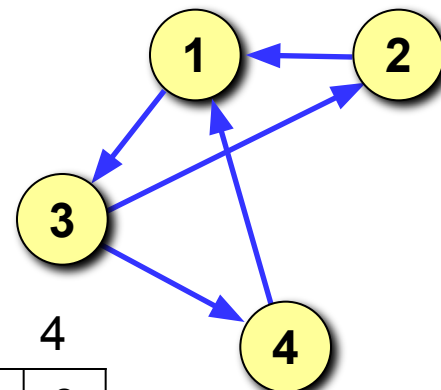
Как обнаружить цепи и циклы?

Матрица путей длины 3:

$$M^3 = M^2 \otimes M$$

$$M^3 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

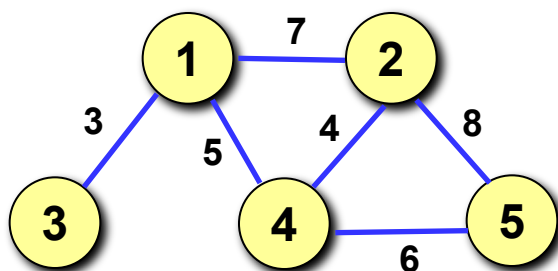
$$M^4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$



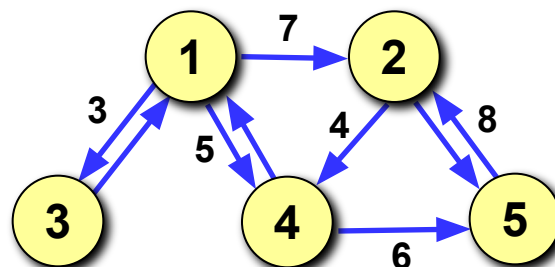
на главной
диагонали –
циклы!

Весовая матрица

Весовая матрица – это матрица, элемент $W[i, j]$ которой равен весу ребра из вершины i в вершину j (если оно есть), или равен ∞ , если такого ребра нет.



	1	2	3	4	5
1	0	7	3	5	∞
2	7	0	∞	4	8
3	3	∞	0	∞	∞
4	5	4	∞	0	6
5	∞	8	∞	6	0

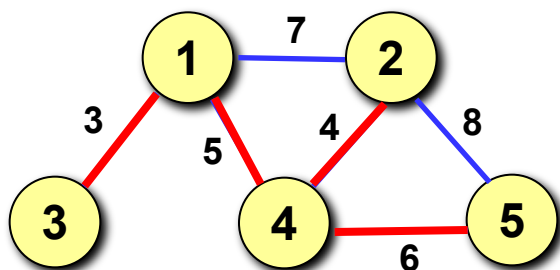


	1	2	3	4	5
1	0	7	3	5	∞
2	∞	0	∞	4	8
3	3	∞	0	∞	∞
4	5	∞	∞	0	6
5	∞	8	∞	∞	0

Задача Прима-Краскала

Задача: соединить N городов телефонной сетью так, чтобы длина телефонных линий была минимальная.

Та же задача: дан связный граф с N вершинами, веса ребер заданы весовой матрицей W . Нужно найти набор ребер, соединяющий все вершины графа (**остовное дерево**) и имеющий наименьший вес.



	1	2	3	4	5
1	0	7	3	5	∞
2	7	0	∞	4	8
3	3	∞	0	∞	∞
4	5	4	∞	0	6
5	∞	8	∞	6	0

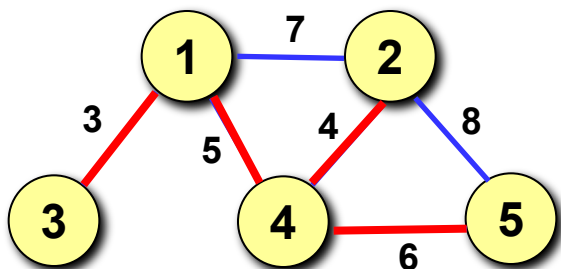
Жадный алгоритм

Жадный алгоритм – это многошаговый алгоритм, в котором на каждом шаге принимается решение, лучшее в данный момент.



В целом может получиться не оптимальное решение (последовательность шагов)!

Шаг в задаче Прима-Краскала – это выбор еще невыбранного ребра и добавление его к решению.



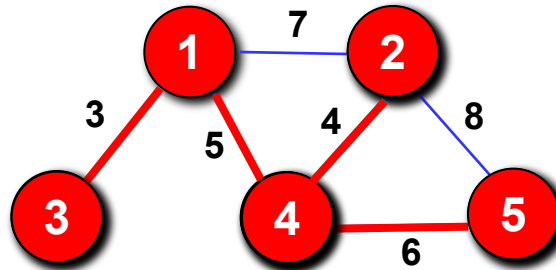
В задаче Прима-Краскала жадный алгоритм дает оптимальное решение!

Реализация алгоритма Прима-Краскала

Проблема: как проверить, что

- 1) ребро не выбрано, и
- 2) ребро не образует цикла с выбранными ребрами.

Решение: присвоить каждой вершине свой цвет и перекрашивать вершины при добавлении ребра.



Алгоритм:

- 1) покрасить все вершины в разные цвета;
- 2) сделать **$N-1$** раз в цикле:
 - выбрать ребро (i, j) минимальной длины из всех ребер, соединяющих вершины разного цвета;
 - перекрасить все вершины, имеющие цвет j , в цвет i .
- 3) вывести найденные ребра.

Реализация алгоритма Прима-Краскала

Структура «ребро»:

```
type rebro = record
    i, j: integer; { номера вершин }
end;
```

Основная программа:

весовая
матрица

цвета
вершин

```
const N = 5;
var W: array[1..N,1..N] of integer;
    Color: array[1..N] of integer;
    i, j, k, min, col_i, col_j: integer;
    Reb: array[1..N-1] of rebro;
begin
    ... { здесь надо ввести матрицу W }
    for i:=1 to N do { раскрасить в разные цвета }
        Color[i] := i;
    ... { основной алгоритм - заполнение массива Reb }
    ... { вывести найденные ребра (массив Reb) }
end.
```


Реализация алгоритма Прима-Краскала

Основной алгоритм:

```
for k:=1 to N-1 do begin
  min := MaxInt;
  for i:=1 to N do
    for j:=i+1 to N do
      if (Color[i] <> Color[j]) and
        (W[i,j] < min) then begin
        min := W[i,j];
        Reb[k].i := i;
        Reb[k].j := j;
        col_i := Color[i];
        col_j := Color[j];
      end;
  for i:=1 to N do
    if Color[i] = col_j then
      Color[i] := col_i;
end;
```

нужно выбрать
всего **N-1** ребер

цикл по всем
парам вершин

учитываем только
пары с разным
цветом вершин

запоминаем ребро и
цвета вершин

перекрашиваем
вершины цвета col_j

Сложность алгоритма

Основной цикл:

```
for k:=1 to N-1 do begin
    ...
    for i:=1 to N do
        for j:=i+1 to N do
            ...
end;
```

три вложенных
цикла, в каждом
число шагов $\leq N$

Количество операций:

$O(N^3)$ растёт не быстрее, чем N^3

Требуемая память:

```
var W: array[1..N,1..N] of integer;
    Color: array[1..N] of integer;
    Reb: array[1..N-1] of rebro;
```



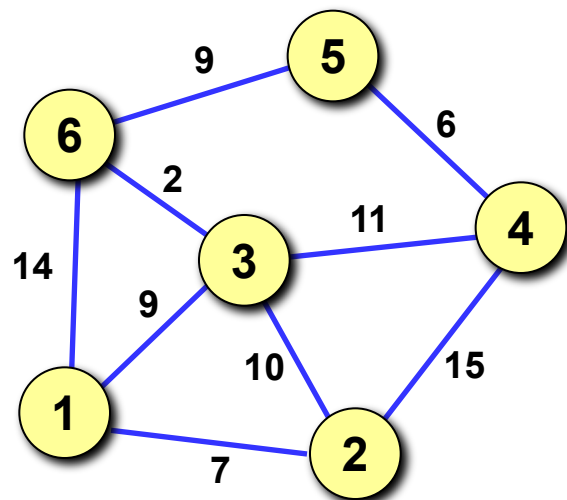
$O(N^2)$

Кратчайшие пути (алгоритм Дейкстры)

Задача: задана сеть дорог между городами, часть которых могут иметь одностороннее движение. Найти кратчайшие расстояния от заданного города до всех остальных городов.

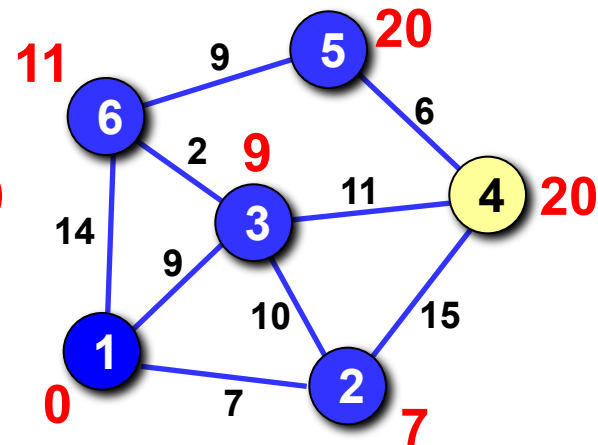
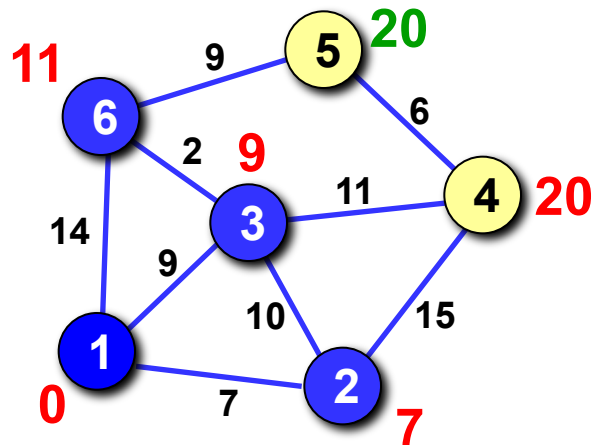
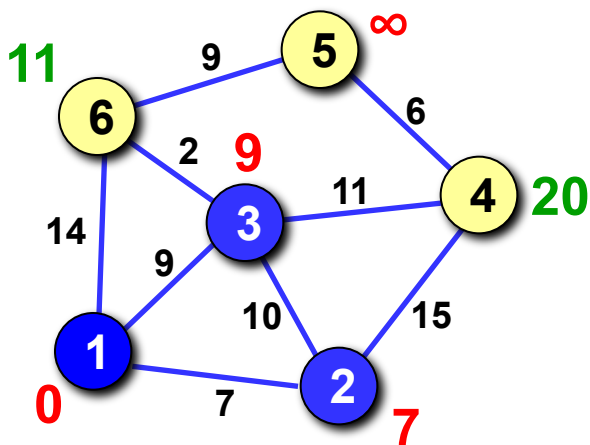
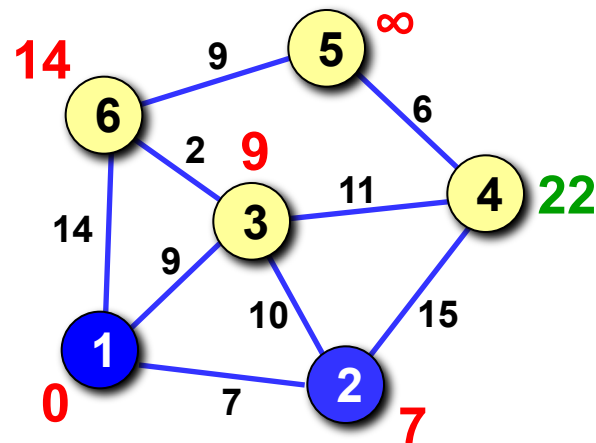
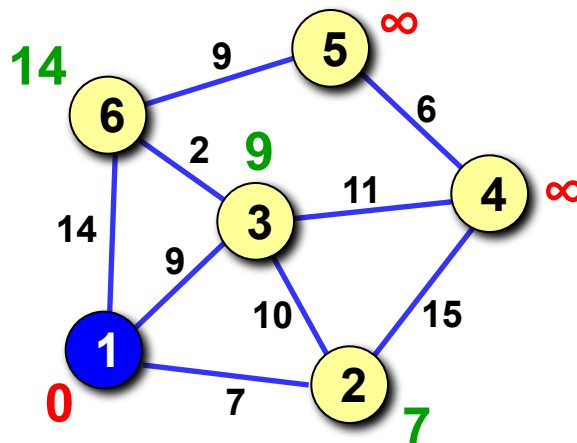
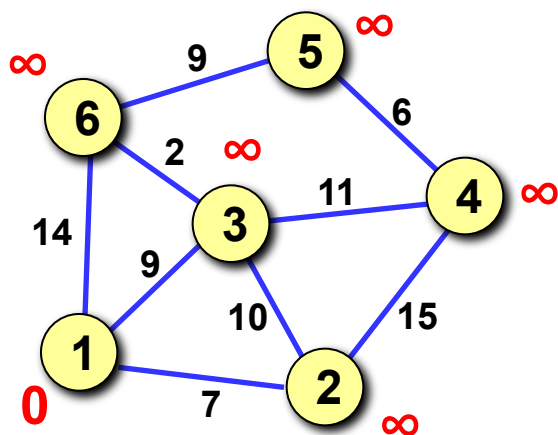
Та же задача: дан связный граф с N вершинами, веса ребер заданы матрицей W . Найти кратчайшие расстояния от заданной вершины до всех остальных.

Алгоритм Дейкстры (E.W. Dijkstra, 1959)



- 1) присвоить всем вершинам метку ∞ ;
- 2) среди нерассмотренных вершин найти вершину j с наименьшей меткой;
- 3) для каждой необработанной вершины i :
если путь к вершине i через вершину j меньше существующей метки, заменить метку на новое расстояние;
- 4) если остались необработанные вершины, перейти к шагу 2;
- 5) метка = минимальное расстояние.

Алгоритм Дейкстры



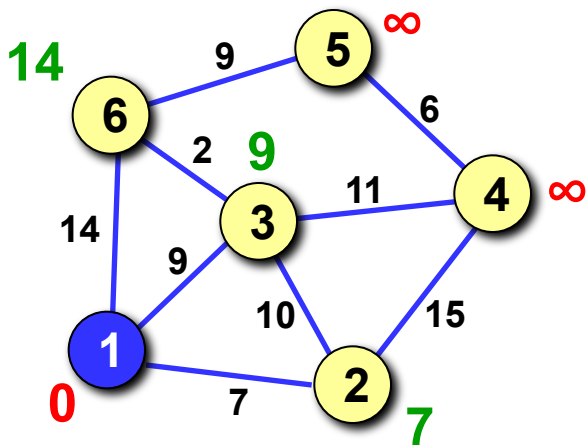
Реализация алгоритма Дейкстры

Массивы:

- 1) массив a , такой что $a[i]=1$, если вершина уже рассмотрена, и $a[i]=0$, если нет.
- 2) массив b , такой что $b[i]$ – длина текущего кратчайшего пути из заданной вершины x в вершину i ;
- 3) массив c , такой что $c[i]$ – номер вершины, из которой нужно идти в вершину i в текущем кратчайшем пути.

Инициализация:

- 4) заполнить массив a нулями (вершины не обработаны);
- 5) записать в $b[i]$ значение $W[x][i]$;
- 6) заполнить массив c значением x ;
- 7) записать $a[x]=1$.



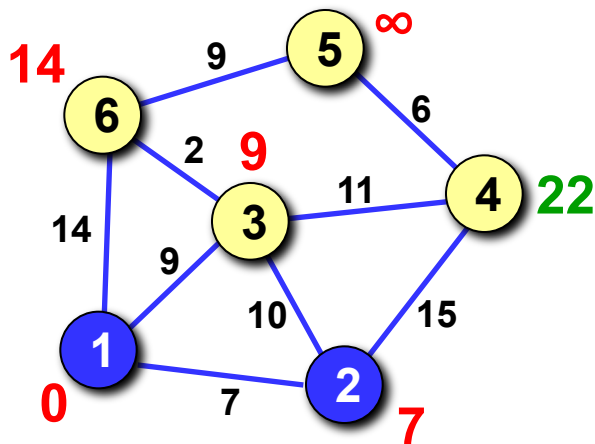
	1	2	3	4	5	6
a	1	0	0	0	0	0
b	0	7	9	∞	∞	14
c	0	0	0	0	0	0

Реализация алгоритма Дейкстры

Основной цикл:

- 1) если все вершины рассмотрены, то стоп.
- 2) среди всех нерассмотренных вершин ($a[i]=0$) найти вершину j , для которой $b[i]$ – минимальное;
- 3) записать $a[j] := 1$;
- 4) для всех вершин k : если путь в вершину k через вершину j короче, чем найденный ранее кратчайший путь, запомнить его: записать $b[k] := b[j] + W[j, k]$ и $c[k] = j$.

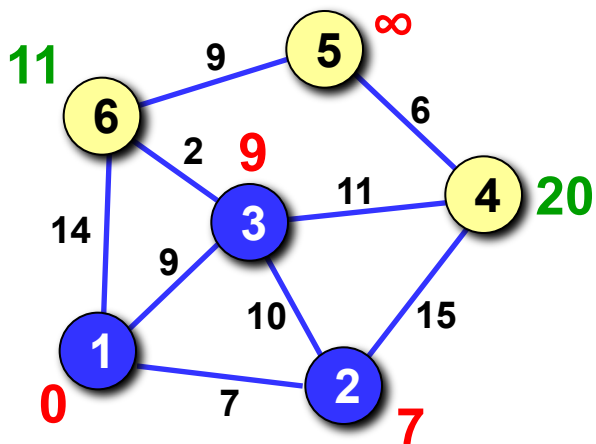
Шаг 1:



	1	2	3	4	5	6
a	1	1	0	0	0	0
b	0	7	9	22	∞	14
c	0	0	0	1	0	0

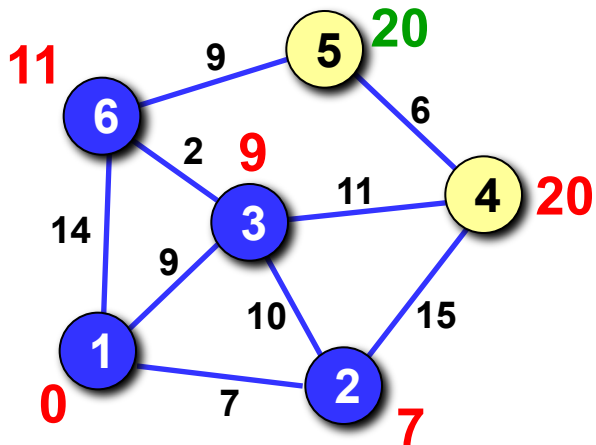
Реализация алгоритма Дейкстры

Шаг 2:



	1	2	3	4	5	6
a	1	1	1	0	0	0
b	0	7	9	20	∞	11
c	0	0	0	2	0	2

Шаг 3:



	1	4	3	4	5	6
a	1	1	1	0	0	1
b	0	7	9	20	20	11
c	0	0	0	2	5	2



Дальше массивы не
изменяются!

Как вывести маршрут?

Результат работа алгоритма Дейкстры:

	1	2	3	4	5	6
а	1	1	1	1	1	1
б	0	7	9	20	20	11
с	0	0	0	2	5	2

длины путей

Маршрут из вершины 0 в вершину 4:



Вывод маршрута в вершину i (использование массива c):

- 1) установить $z := i$;
- 2) пока $c[i] \neq x$ присвоить $z := c[z]$ и вывести z .

Сложность алгоритма Дейкстры:

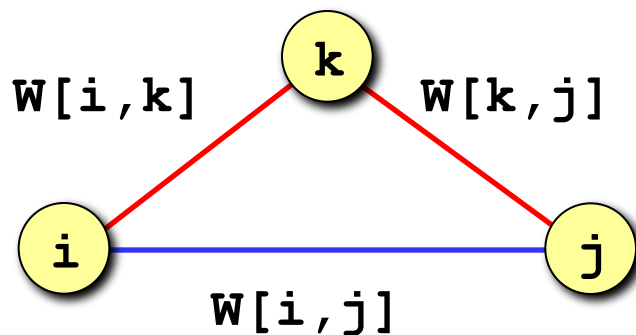
два вложенных цикла по N шагов

$O(N^2)$

Алгоритм Флойда-Уоршелла

Задача: задана сеть дорог между городами, часть которых могут иметь одностороннее движение. Найти **все кратчайшие расстояния**, от каждого города до всех остальных городов.

```
for k:=1 to N
  for i:=1 to N
    for j:=1 to N
      if  $W[i,j] > W[i,k] + W[k,j]$  then
         $W[i,j] := W[i,k] + W[k,j]$ ;
```



Если из вершины **i** в вершину **j** короче ехать через вершину **k**, мы едем через вершину **k**!



Нет информации о маршруте, только кратчайшие расстояния!

Алгоритм Флойда-Уоршелла

Версия с запоминанием маршрута:

```
for i:=1 to N
  for j:=1 to N
    c[i,j] := i;
  ...
for k:=1 to N
  for i:=1 to N
    for j:=1 to N
      if W[i,j] > W[i,k] + W[k,j] then begin
        W[i,j] := W[i,k] + W[k,j];
        c[i,j] := c[k,j];
      end;
```

i -ая строка строится так же, как массив c в алгоритме Дейкстры

в конце цикла $c[i, j]$ – предпоследняя вершина в кратчайшем маршруте из вершины i в вершину j



Какова сложность алгоритма?

$O(N^3)$

Задача коммивояжера

Задача коммивояжера. Коммивояжер (бродячий торговец) должен выйти из первого города и, посетив по разу в неизвестном порядке города $2, 3, \dots, N$, вернуться обратно в первый город. В каком порядке надо обходить города, чтобы замкнутый путь (тур) коммивояжера был кратчайшим?



Это NP-полная задача, которая строго решается только перебором вариантов (пока)!

Точные методы:

- 1) простой перебор;
- 2) метод ветвей и границ;
- 3) метод Литтла;
- 4) ...



большое время счета для больших N

$O(N!)$

Приближенные методы:

- 5) метод случайных перестановок (*Matlab*);
- 6) генетические алгоритмы;
- 7) метод муравьиных колоний;
- 8) ...



не гарантируется оптимальное решение

Другие классические задачи

Задача на минимум суммы. Имеется N населенных пунктов, в каждом из которых живет p_i школьников ($i=1, \dots, N$). Надо разместить школу в одном из них так, чтобы общее расстояние, проходимое всеми учениками по дороге в школу, было минимальным.

Задача о наибольшем потоке. Есть система труб, которые имеют соединения в N узлах. Один узел S является источником, еще один – стоком T . Известны пропускные способности каждой трубы. Надо найти наибольший поток от источника к стоку.

Задача о наибольшем паросочетании. Есть M мужчин и N женщин. Каждый мужчина указывает несколько (от 0 до N) женщин, на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от 0 до M), за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.

Конец фильма
