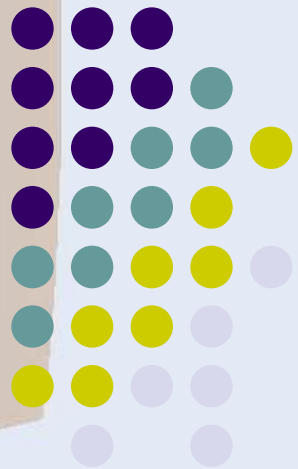
Лекция 5.1

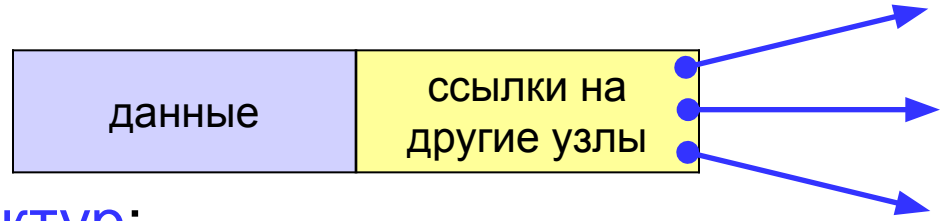
Динамические структуры данных



Динамические структуры данных

Строение: набор узлов, объединенных с помощью ссылок.

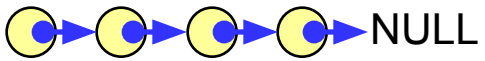
Как устроен узел:



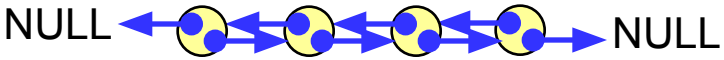
Типы структур:

списки

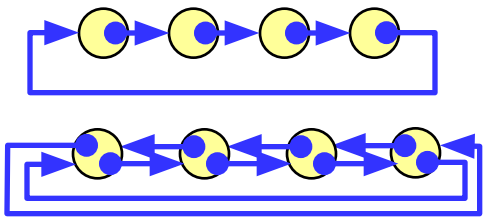
односвязный



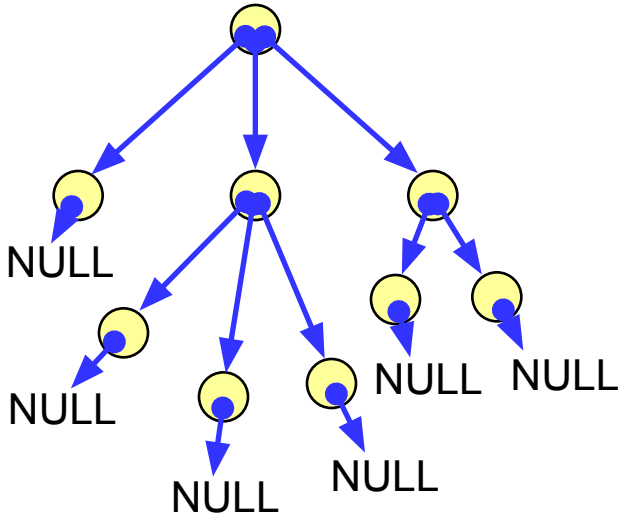
двунаправленный (двусвязный)



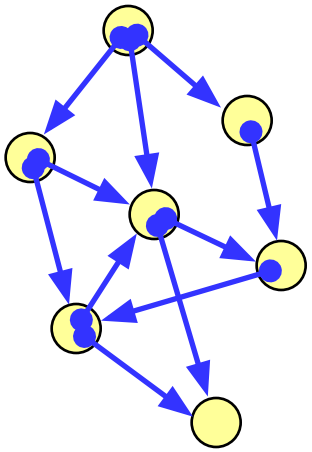
циклические списки (кольца)



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.

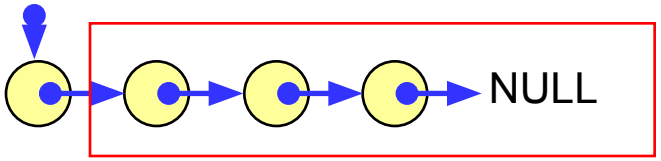
Алгоритм:

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!

Структура узла:

```
struct Node {
  char word[40];   // слово
  int  count;      // счетчик повторений
  Node *next;      // ссылка на следующий элемент
  };
```

Указатель на эту структуру:

```
typedef Node *PNode;
```

Адрес начала списка:

```
PNode Head = NULL;
```

Для доступа к списку достаточно знать адрес его головы!

Что нужно уметь делать со списком?

1. Создать новый узел.
2. Добавить узел:
   a) в начало списка;
   b) в конец списка;
   c) после заданного узла;
   d) 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.

возвращает адрес созданного узла

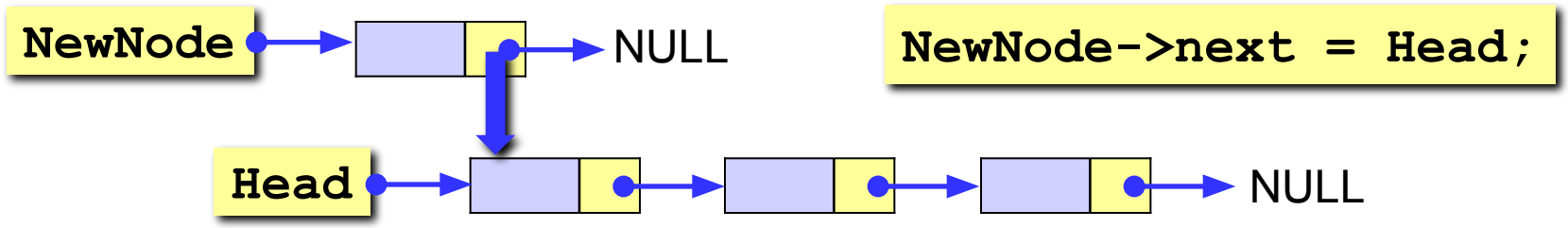
новое слово

```
PNode CreateNode ( char NewWord[] )
{
  PNode NewNode = new Node;
  strcpy(NewNode->word, NewWord);
  NewNode->count = 1;
  NewNode->next = NULL;
  return NewNode;
}
```

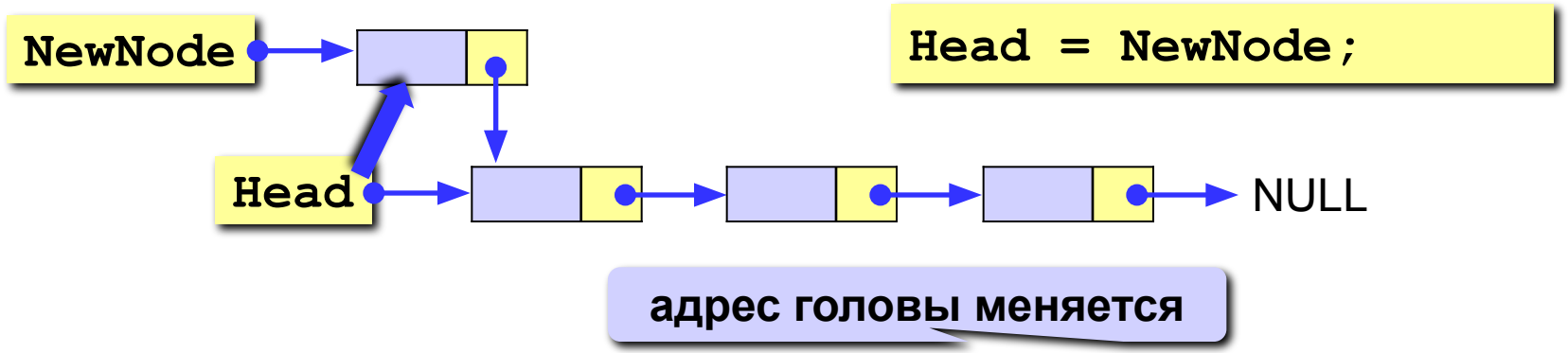
Если память выделить не удалось?

Добавление узла в начало списка

1) Установить ссылку нового узла на голову списка:



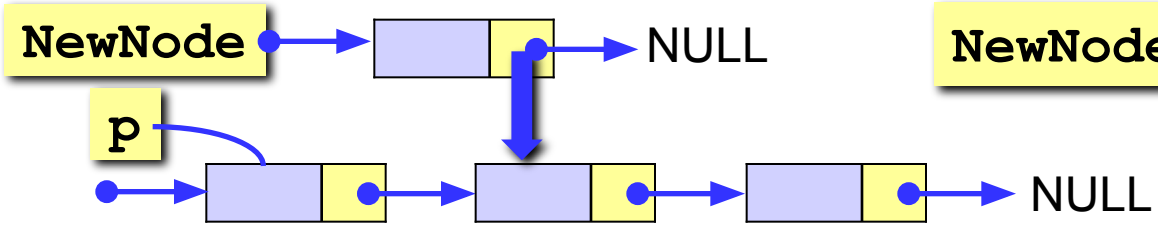
2) Установить новый узел как голову списка:




```
void AddFirst (PNode & Head, PNode NewNode)
{
  NewNode->next = Head;
  Head = NewNode;
}
```

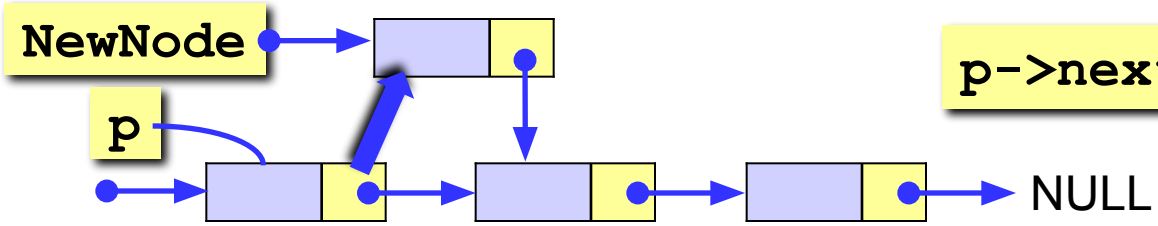
Добавление узла после заданного

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




```
NewNode->next = p->next;
```

2) Установить ссылку узла p на новый узел:



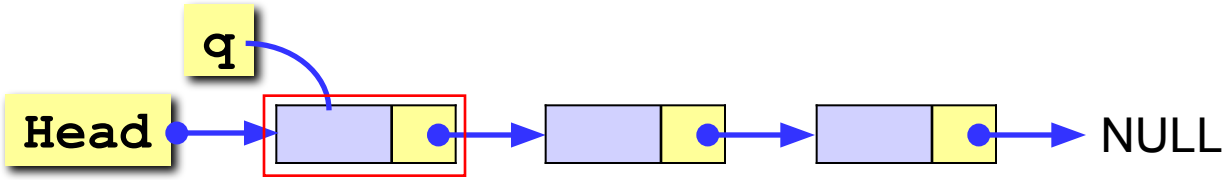

```
p->next = NewNode;
```

```
void AddAfter (PNode p, PNode NewNode)
{
   NewNode->next = p->next;
   p->next = NewNode;
}
```

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 `NULL` (дошли до конца списка), то стоп;
3) выполнить действие над узлом с адресом `q` ;
4) перейти к следующему узлу, `q->next`.

```
...
PNode q = Head;          // начали с головы
while ( q != NULL ) {    // пока не дошли до конца
  ...                    // делаем что-то хорошее с q
  q = q->next;           // переходим к следующему узлу
  }
...
```

Добавление узла в конец списка

Задача: добавить новый узел в конец списка.

Алгоритм:

1) найти последний узел `q`, такой что `q->next` равен `NULL`;
2) добавить узел после узла с адресом `q` (процедура `AddAfter`).

Особый случай: добавление в пустой список.

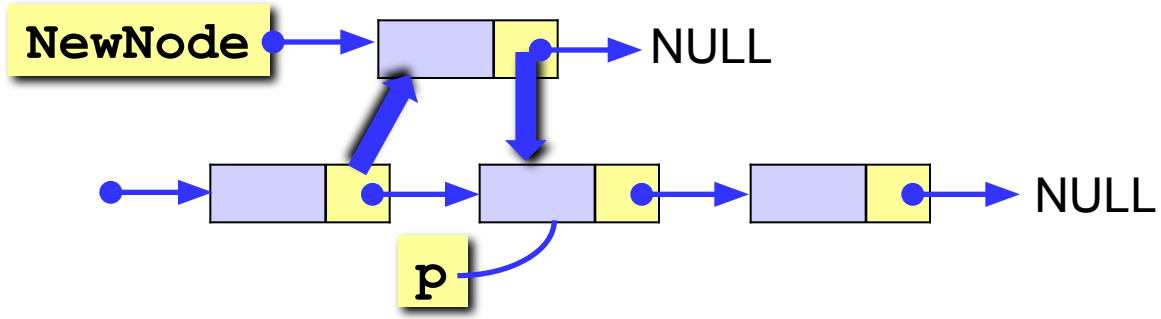
```
void AddLast ( PNode &Head, PNode NewNode )
{
PNode q = Head;
if ( Head == NULL ) {
    AddFirst( Head, NewNode );
    return;
    }
while ( q->next ) q = q->next;
AddAfter ( q, NewNode );
}
```

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).

```
void AddBefore ( PNode & Head, PNode p, PNode NewNode )
{
PNode q = Head;
if ( Head == p ) {
    AddFirst ( Head, NewNode );
    return;
    }
while ( q && q->next != p ) q = q->next;
if ( q ) AddAfter(q, NewNode);
}
```

особый случай – добавление в начало списка

ищем узел, следующий за которым – узел `p`

добавить узел после узла `q`

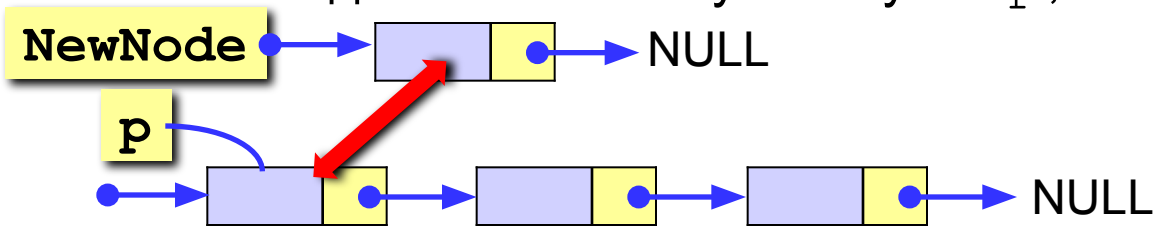
Что плохо?

Добавление узла перед заданным (II)

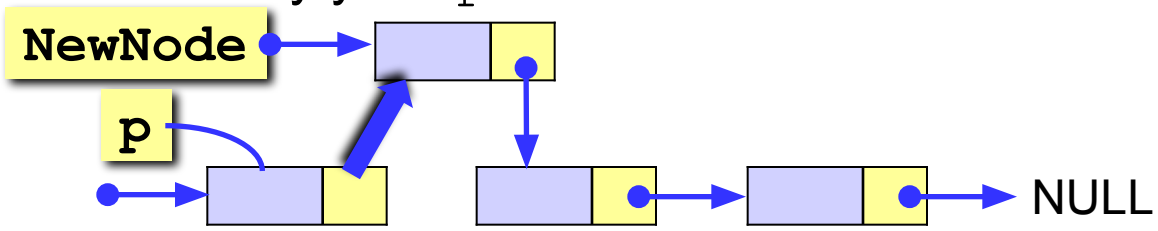
Задача: вставить узел перед заданным без поиска предыдущего.

Алгоритм:

1) поменять местами данные нового узла и узла `p`;



2) установить ссылку узла `p` на `NewNode`.




```
void AddBefore2 ( PNode p, PNode NewNode )
{
  Node temp;
  temp = *p; *p = *NewNode;
  *NewNode = temp;
  p->next = NewNode;
}
```

Так нельзя, если `p = NULL` 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 `NULL`.

Алгоритм: проход по списку.

результат – адрес узла

ищем это слово

```
PNode Find ( PNode Head, char NewWord[] )
{
  PNode q = Head;
  while ( q && strcmp ( q->word, NewWord) )
    q = q->next;
  return q;
}
```

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);

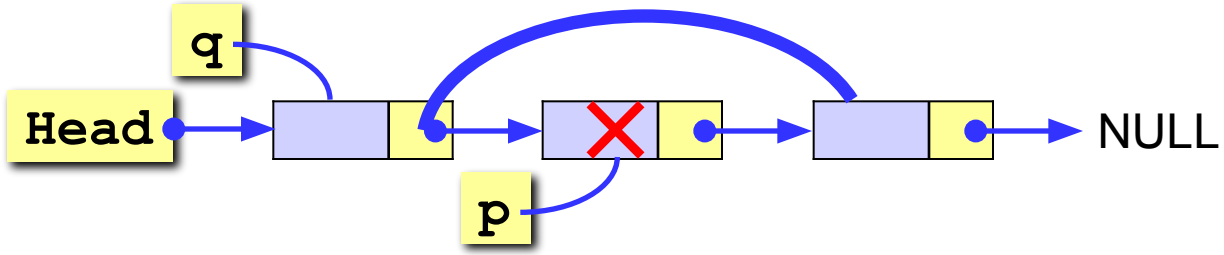
выход: адрес узла, перед которым нужно вставить это слово или `NULL`, если слово нужно вставить в конец списка.

```
PNode FindPlace ( PNode Head, char NewWord[] )
{
  PNode q = Head;
  while ( q  &&  strcmp(NewWord, q->word) > 0 )
    q = q->next;
  return q;
}
```

слово `NewWord` стоит по алфавиту *до* `q->word`

Удаление узла

Проблема: нужно знать адрес предыдущего узла `q`.




```
void DeleteNode ( PNode &Head, PNode p )
{
PNode q = Head;
if ( Head == p )
    Head = p->next;
else {
    while ( q  &&  q->next != p )
      q = q->next;
    if ( q == NULL ) return;
    q->next = p->next;
    }
delete p;
}
```

особый случай: удаляем первый узел

ищем предыдущий узел, такой что `q->next == p`

освобождение памяти

Алфавитно-частотный словарь

Алгоритм:

1) открыть файл на чтение;

read, чтение

```
FILE *in;
in = fopen ( "input.dat", "r" );
```

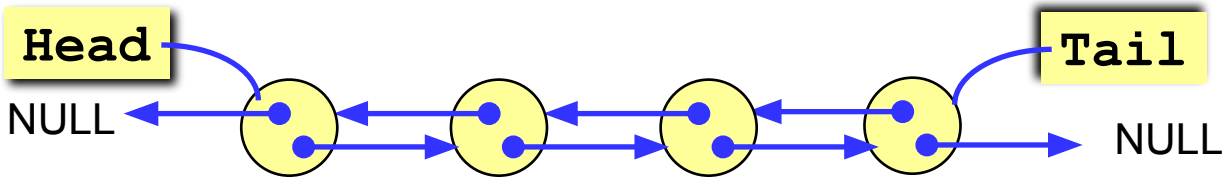
2) прочитать слово:

вводится только одно слово (до пробела)!

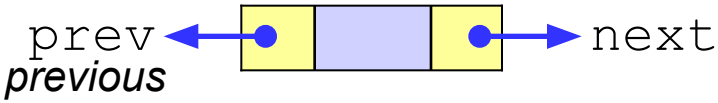
```
char word[80];
...
n = fscanf ( in, "%s", word );
```

3) если файл закончился (`n!=1`), 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) вывести список слов, используя проход по списку.

Двусвязные списки



Структура узла:




```
struct Node {
  char word[40];   // слово
  int  count;      // счетчик повторений
  Node *next;      // ссылка на следующий элемент
  Node *prev;      // ссылка на предыдущий элемент
  };
```

Указатель на эту структуру:

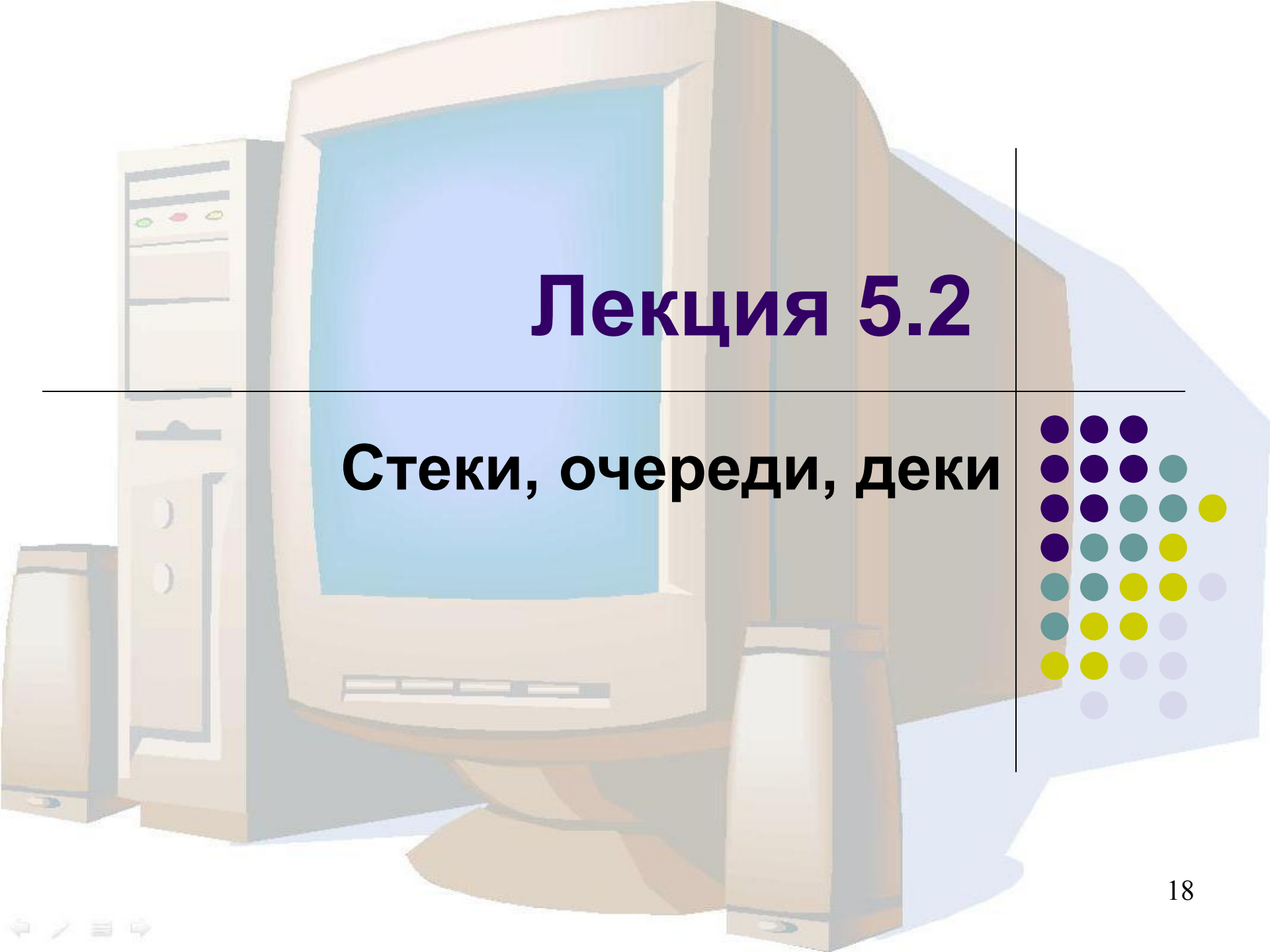
```
typedef Node *PNode;
```

Адреса «головы» и «хвоста»:

```
PNode Head = NULL;
PNode Tail = NULL;
```

можно двигаться в обе стороны

нужно правильно работать с двумя указателями вместо одного



Лекция 5.2

Стеки, очереди, деки

Стек





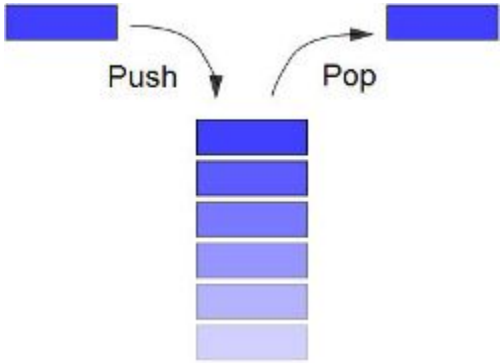
Стек – это линейная структура данных, в которой добавление и удаление элементов возможно только с одного конца (вершины стека). *Stack* = кипа, куча, стопка (англ.)

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

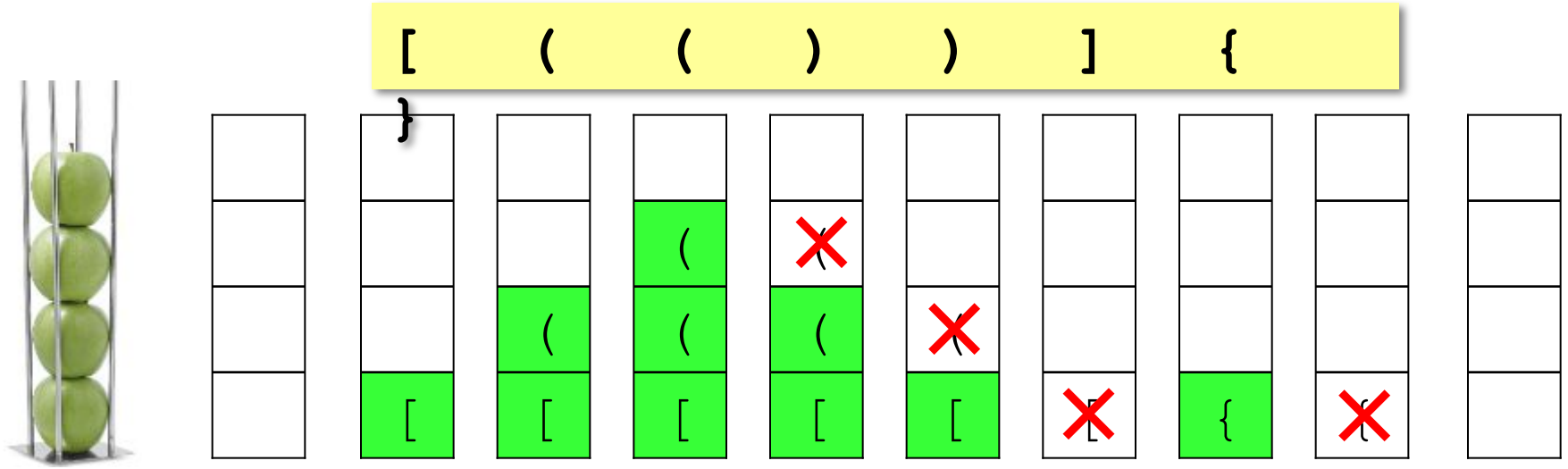
(	(	)	)	)	(
1	2	1	0	-1	0

(	(	)	)	(
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;
struct Stack {
   char data[MAXSIZE]; // стек на 100 символов
   int  size;          // число элементов
   };
```

Добавление элемента:

```
int Push ( Stack &S, char x )
{
if ( S.size == MAXSIZE ) return 0;
S.data[S.size] = x;
S.size ++;
return 1;
}
```

ошибка: переполнение стека

добавить элемент

нет ошибки

Реализация стека (массив)

Снятие элемента с вершины:

```
char Pop ( Stack &S )
{
if ( S.size == 0 ) return char(255);
S.size --;
return S.data[S.size];
}
```

ошибка: стек пуст

Пустой или нет?

```
int isEmpty ( Stack &S )
{
if ( S.size == 0 )
     return 1;
else return 0;
}
```

```
int isEmpty ( Stack &S )
{
return (S.size == 0);
}
```

Программа

```
void main()
{
  char br1[3] = { '(', '[', '{' };
  char br2[3] = { ')', ']', '}' };
  char s[80], upper;
  int i, k, error = 0;
  Stack S;
  S.size = 0;
  printf("Введите выражение со скобками > ");
  gets ( s );
  ...  // здесь будет основной цикл обработки
  if ( ! error && (S.size == 0) )
       printf("\nВыражение правильное\n");
  else printf("\nВыражение неправильное\n");
}
```

открывающие скобки

закрывающие скобки

то, что сняли со стека

признак ошибки

Обработка строки (основной цикл)

```
for ( i = 0; i < strlen(s); i++ )
  {
  for ( k = 0; k < 3; k++ )
    {
    if ( s[i] == br1[k] )  // если открывающая скобка
      {
      Push ( S, s[i] ); // втолкнуть в стек
      break;
      }
    if ( s[i] == br2[k] ) // если закрывающая скобка
      {
      upper = Pop ( S );  // снять верхний элемент
      if ( upper != br1[k] ) error = 1;
      break;
      }
    }
  if ( error ) break;
  }
```

цикл по всем символам строки s

цикл по всем видам скобок

ошибка: стек пуст или не та скобка

была ошибка: дальше нет смысла проверять

Реализация стека (список)

Структура узла:

```
struct Node {
    char data;
       Node *next;
       };
typedef Node *PNode;
```

Добавление элемента:

```
void Push (PNode &Head, char x)
{
    PNode NewNode = new Node;
    NewNode->data = x;
    NewNode->next = Head;
    Head = NewNode;
}
```

Реализация стека (список)

Снятие элемента с вершины:

```
char Pop (PNode &Head) {
  char x;
  PNode q = Head;
  if ( Head == NULL ) return char(255);
  x = Head->data;
  Head = Head->next;
  delete q;
  return x;
}
```

стек пуст

Изменения в основной программе:

```
Stack S;
S.size = 0;

...
if ( ! error && (S.size == 0) )
     printf("\nВыражение правильное\n");
else printf("\nВыражение неправильное \n");
```

PNode S = NULL;

(S == NULL)

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
```

польская нотация,
Jan Łukasiewicz (1920)

скобки не нужны, можно однозначно вычислить!

Постфиксная запись (знак операции после операндов)

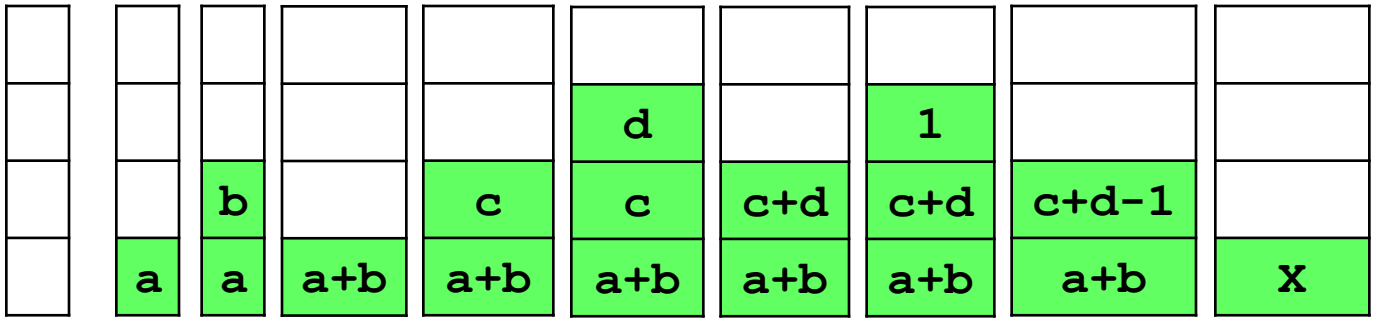
```
a b + c + d - 1 /
```

обратная польская нотация,
F. L. BauerF. L. Bauer and E. W. Dijkstra

Вычисление выражений

Постфиксная форма:

X =	a	b	+	c	d	+	1	-	/



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*)

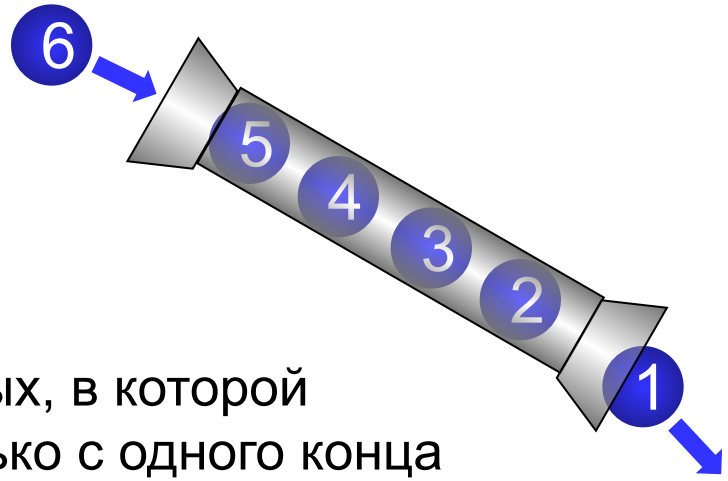
Используется для

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 *Ассмеблер*).

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).

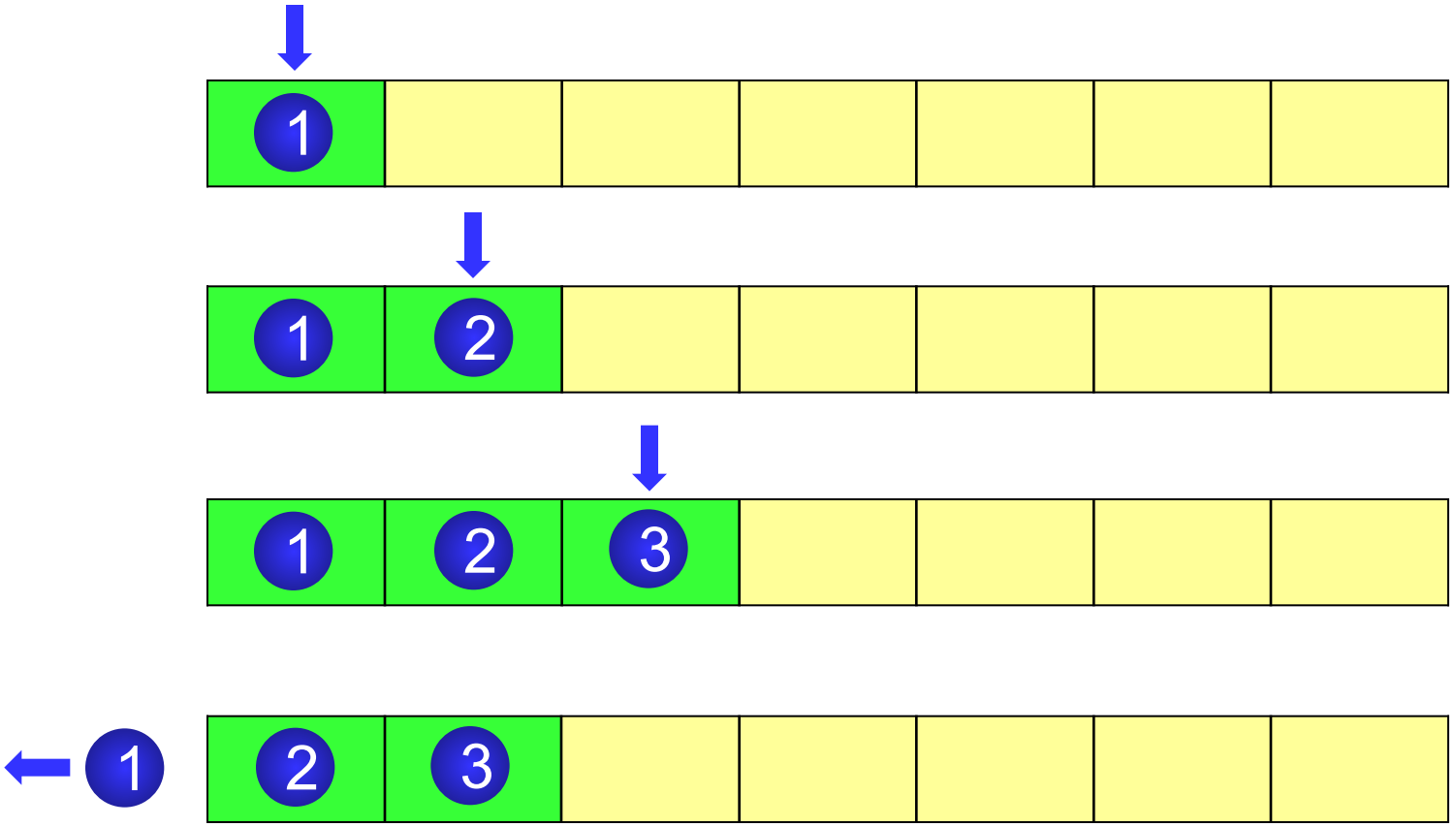
FIFO = *First In – First Out*

«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).*

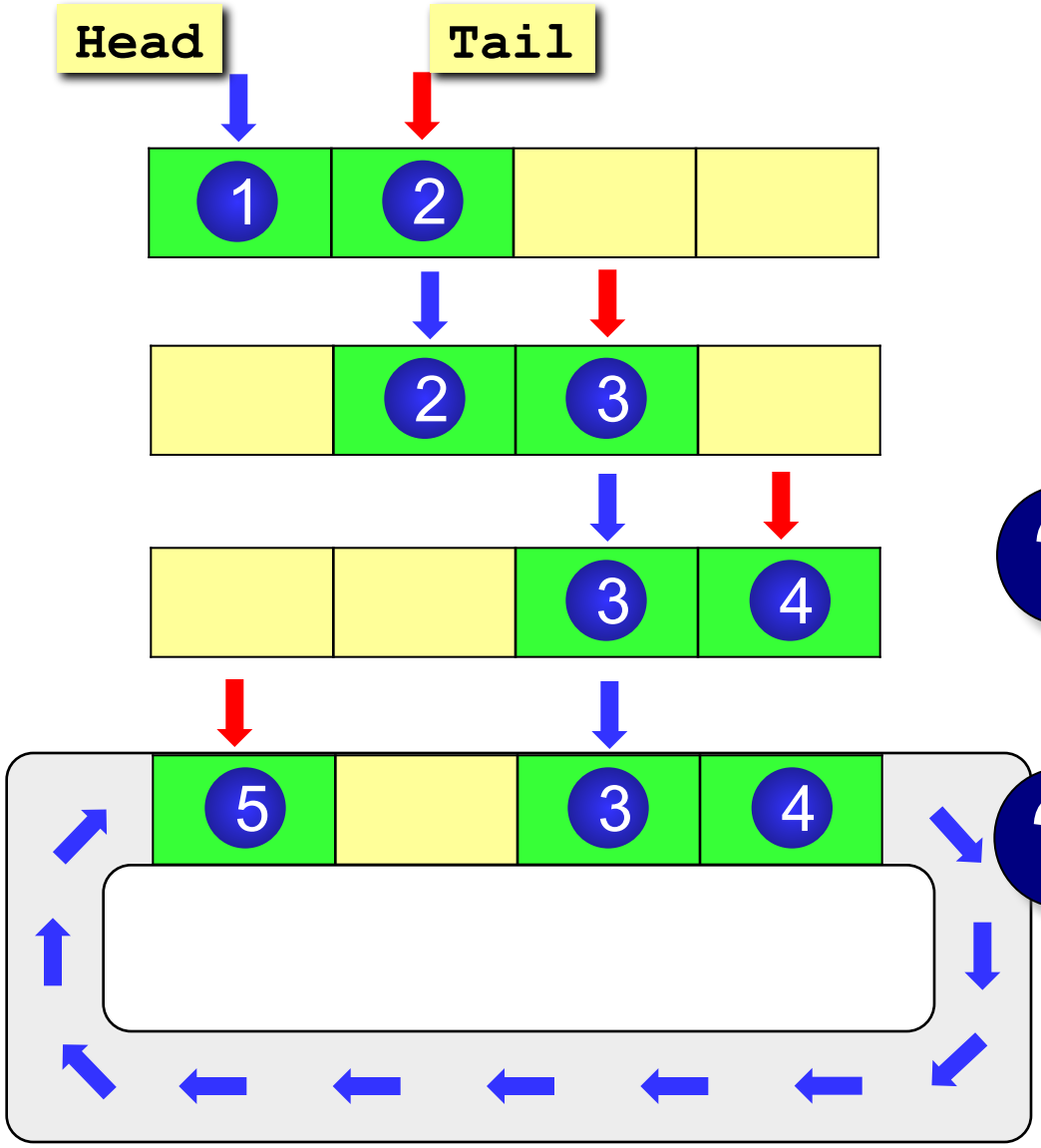
Реализация очереди (массив)

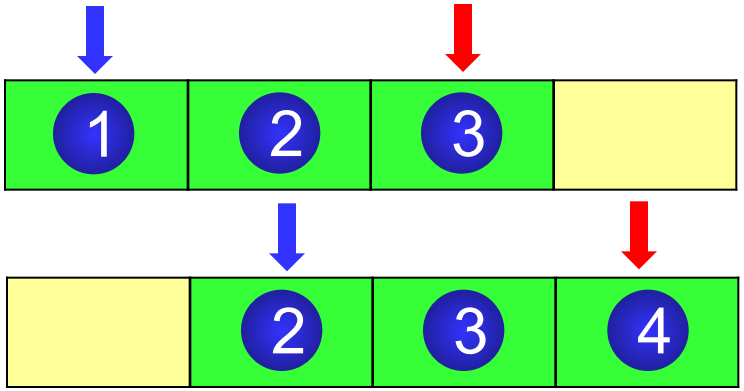


самый простой способ

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

Реализация очереди (кольцевой массив)



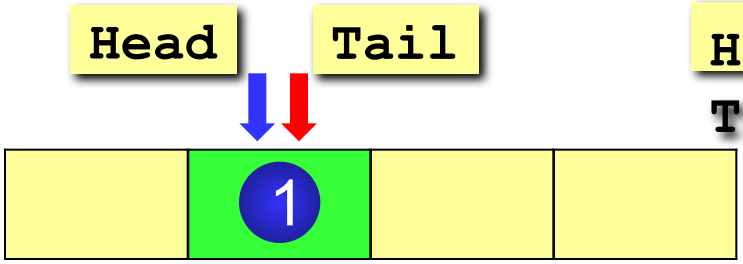


Сколько элементов можно хранить в такой очереди?

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

Реализация очереди (кольцевой массив)

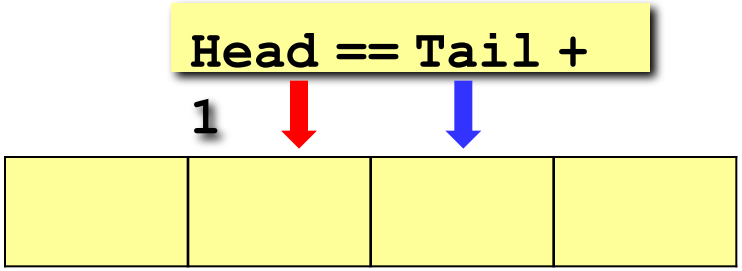
В очереди 1 элемент:

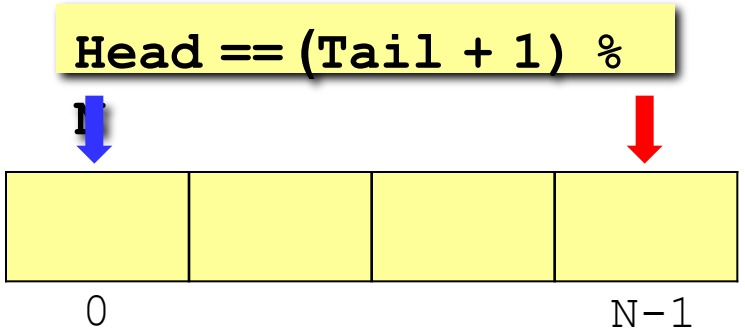


Head == Tail

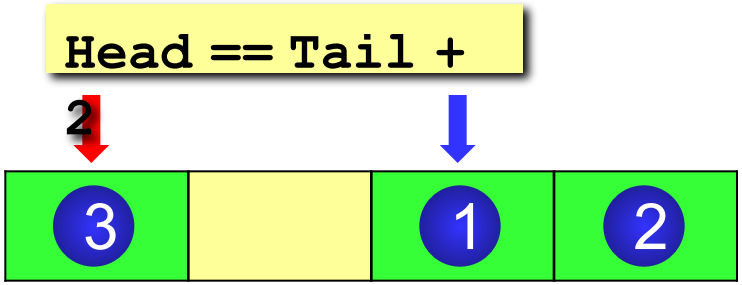
размер массива

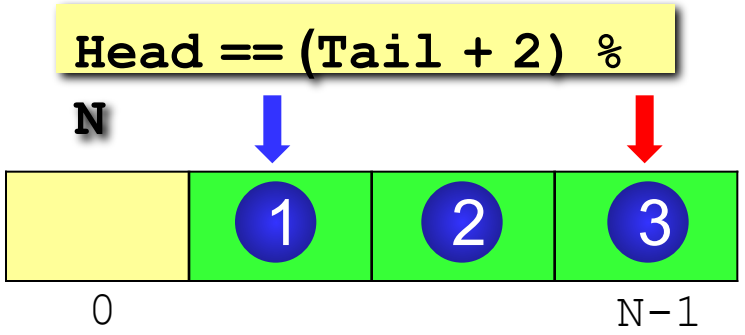
Очередь пуста:





Очередь полна:





Реализация очереди (кольцевой массив)

Структура данных:

```
const MAXSIZE = 100;
struct Queue {
    int data[MAXSIZE];
    int head, tail;
    };
```

замкнуть в кольцо

Добавление в очередь:

```
int PushTail ( Queue &Q, int x )
{
    if ( Q.head == (Q.tail+2) % MAXSIZE )
        return 0;
    Q.tail = (Q.tail + 1) % MAXSIZE;
    Q.data[Q.tail] = x;
    return 1;
}
```

очередь полна, не добавить

удачно добавили

Реализация очереди (кольцевой массив)

Выборка из очереди:

```
int Pop ( Queue &Q )
{
   int temp;
   if ( Q.head == (Q.tail + 1) % MAXSIZE )
     return 32767;
   temp = Q.data[Q.head];
   Q.head = (Q.head + 1) % MAXSIZE;
   return temp;
}
```

очередь пуста

взять первый элемент

удалить его из очереди

Реализация очереди (списки)

Структура узла:

```
struct Node {
    int  data;
    Node *next;
    };
typedef Node *PNode;
```

Тип данных «очередь»:

```
struct Queue {
    PNode Head, Tail;
    };
```

Реализация очереди (списки)

Добавление элемента:

```
void PushTail ( Queue &Q, int x )
{
    PNode NewNode;
    NewNode = new Node;
    NewNode->data = x;
    NewNode->next = NULL;
    if ( Q.Tail )
         Q.Tail->next = NewNode;
    Q.Tail = NewNode;
    if ( Q.Head == NULL )
         Q.Head = Q.Tail;
}
```

создаем новый узел

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

если в списке ничего не было, …

Реализация очереди (списки)

Выборка элемента:

```
int Pop ( Queue &Q )
{
   PNode top = Q.Head;
   int x;
   if ( top == NULL )
      return 32767;
   x = top->data;
   Q.Head = top->next;
   if ( Q.Head == NULL )
        Q.Tail = NULL;
   delete top;
   return x;
}
```

если список пуст, ...

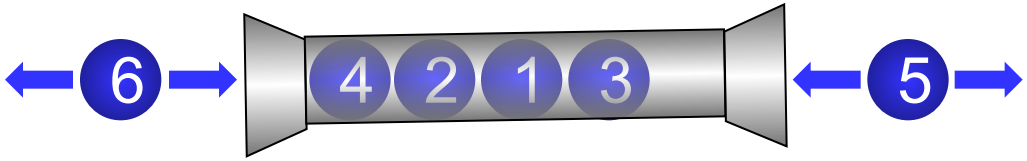
запомнили первый элемент

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.

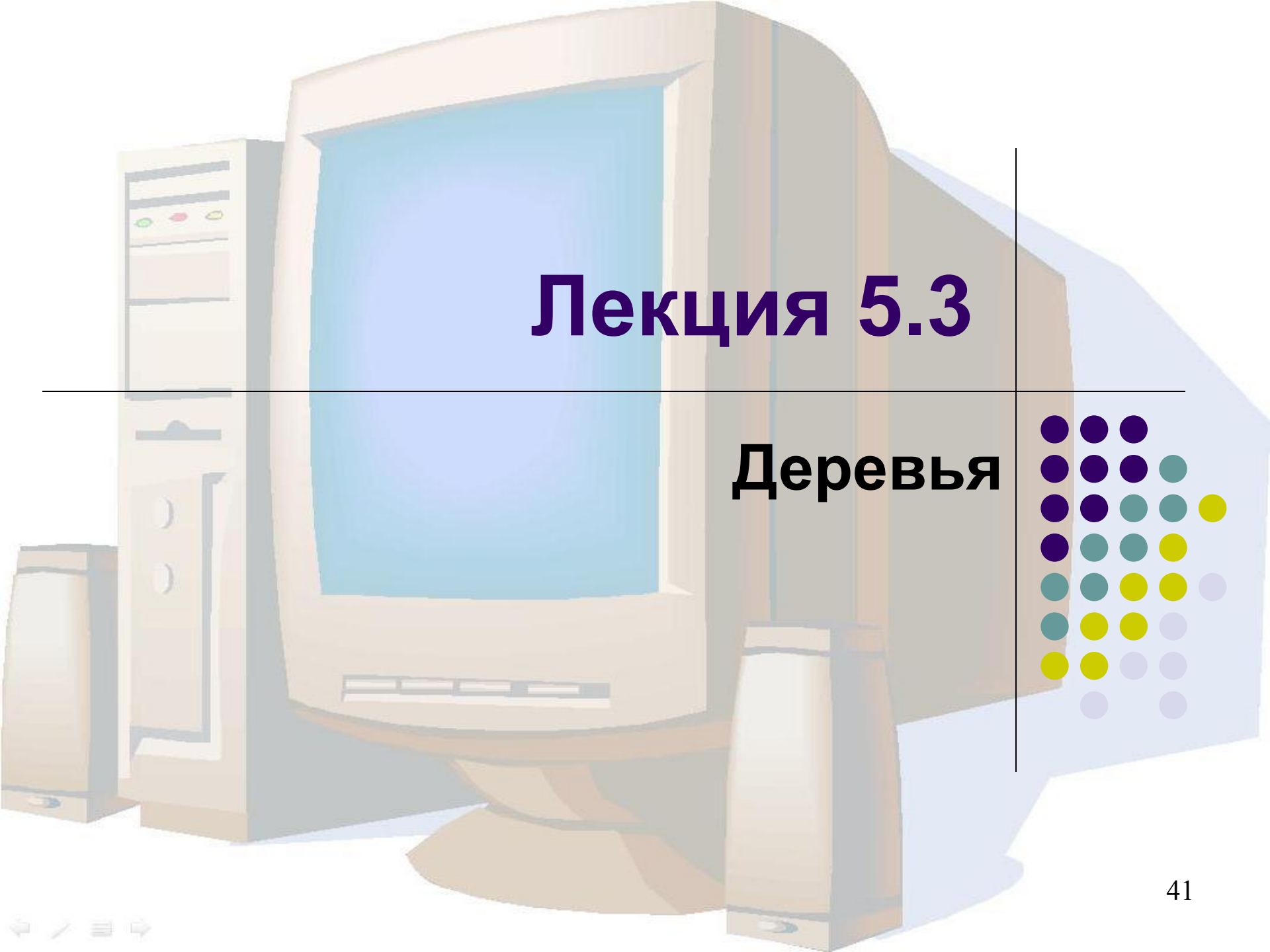


Операции с деком:

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:

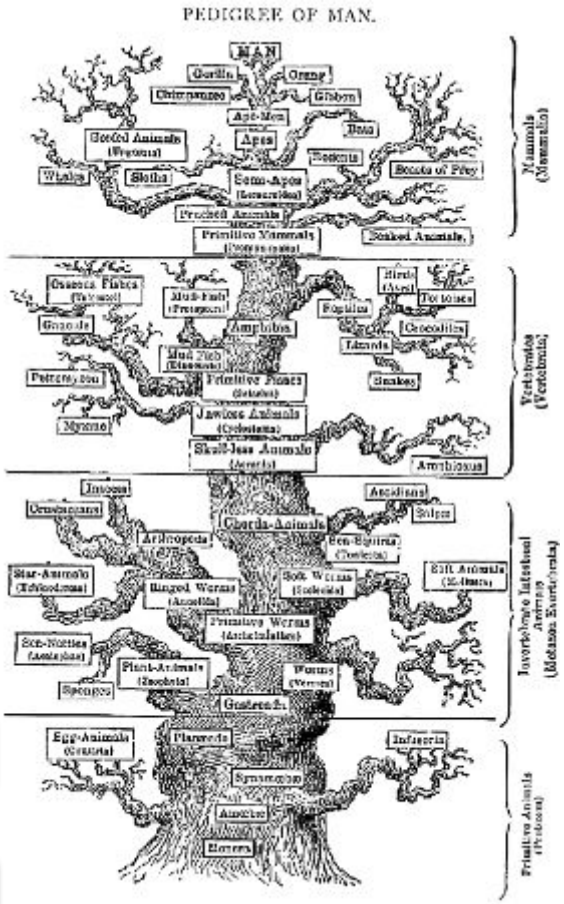
1) кольцевой массив*;*
2) двусвязный список.

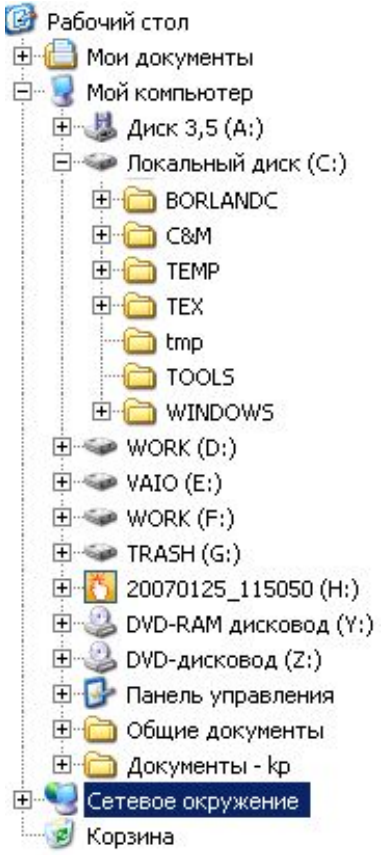


Лекция 5.3

Деревья

Деревья





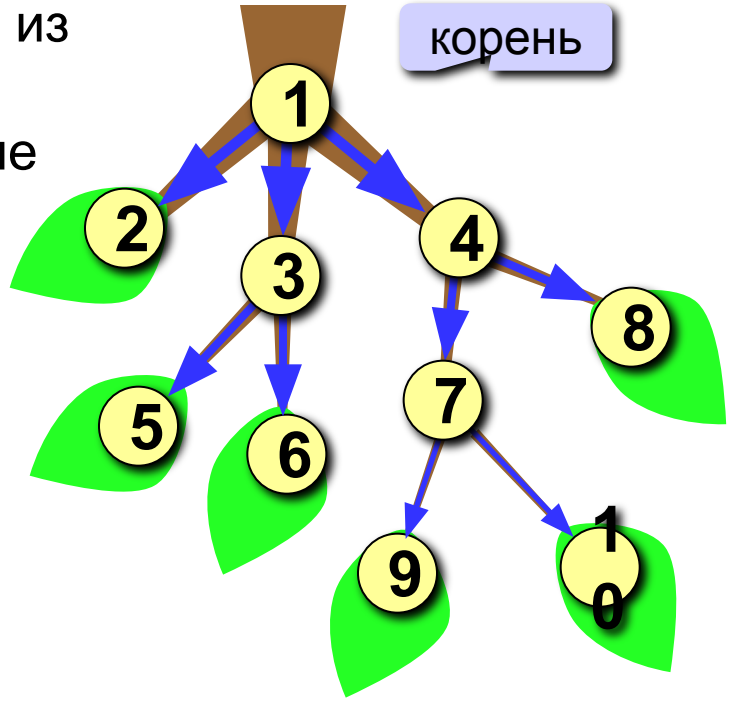
Что общего во всех примерах?

Деревья

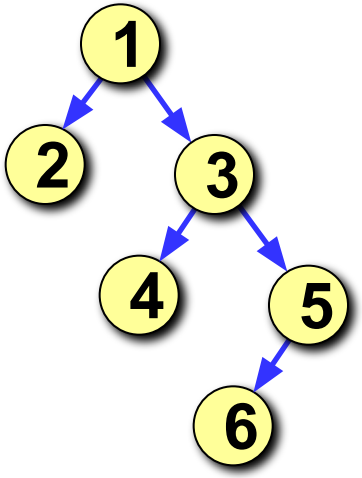
Дерево – это структура данных, состоящая из узлов и соединяющих их направленных ребер (дуг), причем в каждый узел (кроме корневого) ведет ровно одна дуга.

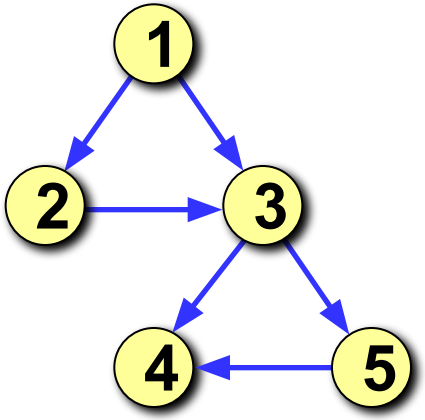
Корень – это начальный узел дерева.

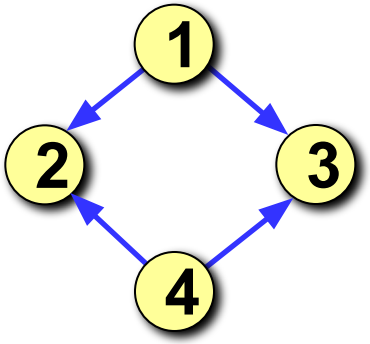
Лист – это узел, из которого не выходит ни одной дуги.



Какие структуры – не деревья?







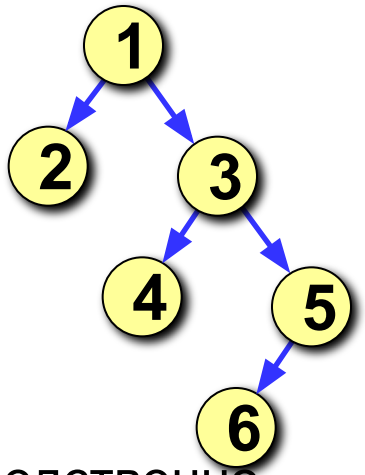
Деревья

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.

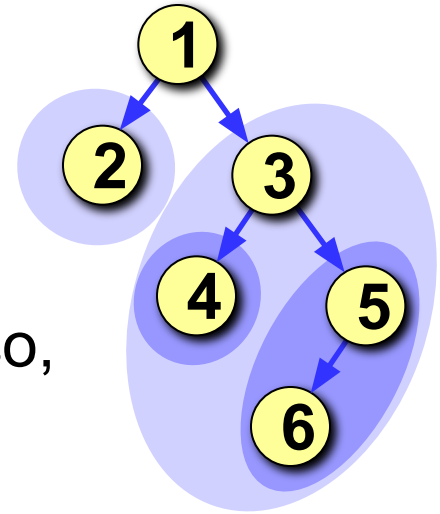
Брат узла x *(sibling)* – это узел, у которого тот же родитель, что и у узла x.

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).

Двоичные деревья

Применение:

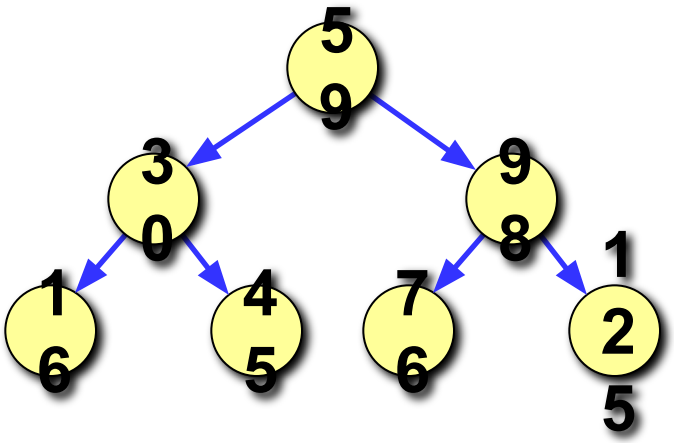
1) поиск данных в специально построенных деревьях (базы данных);
2) сортировка данных;
3) вычисление арифметических выражений;
4) кодирование (метод Хаффмана).

Структура узла:

```
struct Node {
  int  data;              // полезные данные
  Node *left, *right;     // ссылки на левого
                          // и правого сыновей
  };
typedef Node *PNode;
```

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 …?

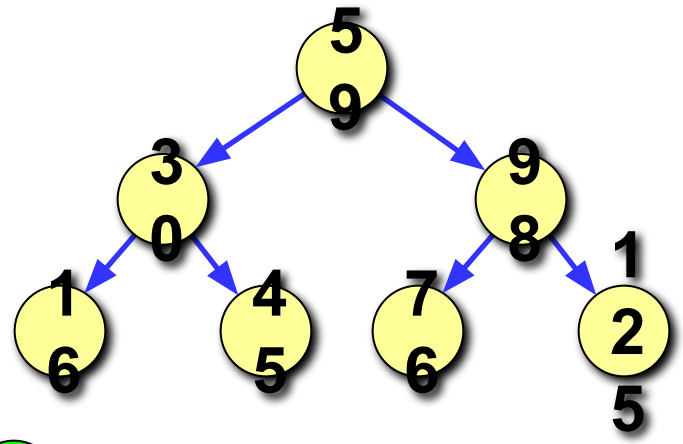
Двоичные деревья поиска

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 ~ $\log_2 N$.

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 NULL (не нашли)
//---------------------------------------
PNode Search (PNode Tree, int x)
{
if ( ! Tree ) return NULL;
if ( x == Tree->data )
     return Tree;
if ( x < Tree->data )
  return Search(Tree->left, x);
else
  return Search(Tree->right, x);
}
```

дерево пустое: ключ не нашли…

нашли, возвращаем адрес корня

искать в левом поддереве

искать в правом поддереве

Как построить дерево поиска?

```
//-----------------------------------------
// Функция AddToTree – добавить элемент к дереву
// Вход: Tree - адрес корня,
//         x  - что добавляем
//-----------------------------------------
void AddToTree (PNode &Tree, int x)
{
if ( ! Tree ) {
   Tree = new Node;
   Tree->data = x;
   Tree->left = NULL;
   Tree->right = NULL;
   return;
   }
if ( x < Tree->data )
     AddToTree ( Tree->left, x );
else AddToTree ( Tree->right, x );
}
```

адрес корня может измениться

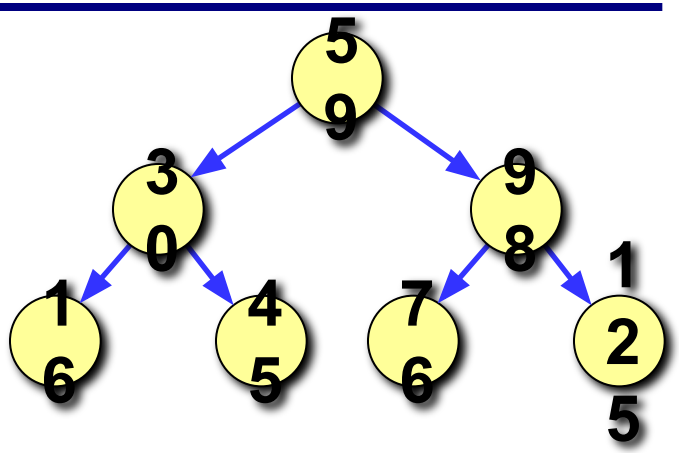
дерево пустое: создаем новый узел (корень)

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»):

16 30 45 59 76 98 125

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

125 98 76 59 45 30 16

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

59 30 16 45 98 76 125

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

16 45 30 76 125 98 59

Обход дерева – реализация

```
//-----------------------------------------
// Функция LKP – обход дерева в порядке ЛКП
//                (левый – корень – правый)
// Вход: Tree - адрес корня
//-----------------------------------------
void LKP( PNode Tree )
{
if ( ! Tree ) return;
LKP ( Tree->left );
printf ( "%d ", Tree->data );
LKP ( Tree->right );
}
```

обход этой ветки закончен

обход левого поддерева

вывод данных корня

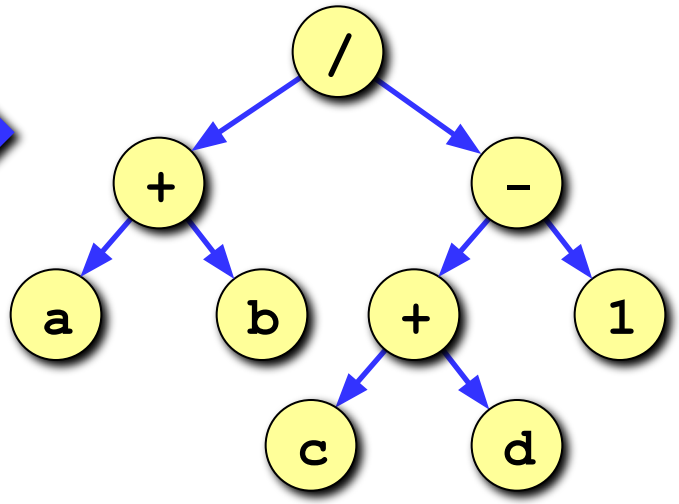
обход правого поддерева

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 1
```

польская нотация, Jan Łukasiewicz (1920)

скобки не нужны, можно однозначно вычислить!

Постфиксная запись, ЛПК (знак операции после операндов)

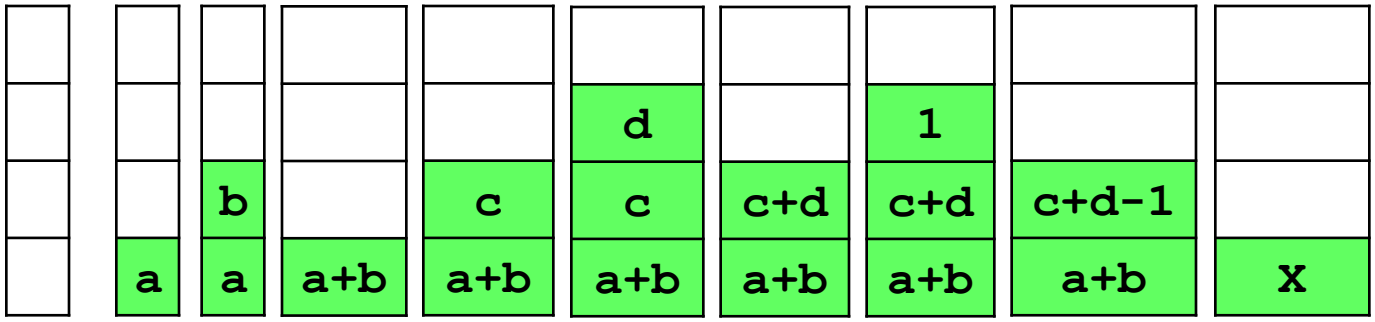
```
a b + c d + 1 - /
```

обратная польская нотация, F. L. BauerF. L. Bauer and E. W. Dijkstra

Вычисление выражений

Постфиксная форма:

X =	a	b	+	c	d	+	1	-	/



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 `+ - * \`. Вычислить это выражение.

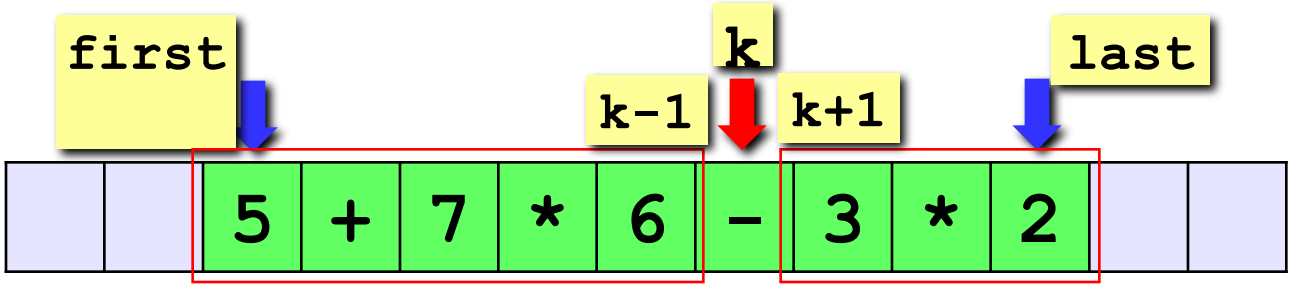
Алгоритм:

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.

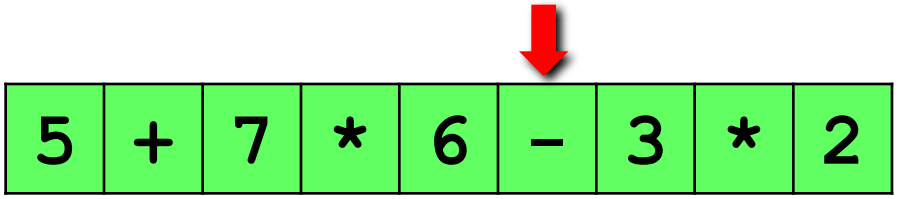
Построение дерева



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
//-----------------------------------------
int Priority ( char c )
{
  switch ( c ) {
    case '+': case '-':
      return 1;
    case '*': case '/':
      return 2;
    }
  return 100;
}
```

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
//-----------------------------------------
int LastOperation ( char Expr[], int first, int last )
{
  int MinPrt, i, k, prt;
  MinPrt = 100;
  for( i = first; i <= last; i++ ) {
    prt = Priority ( Expr[i] );
    if ( prt <= MinPrt ) {
      MinPrt = prt;
      k = i;
      }
    }
  return k;
}
```

проверяем все символы

нашли операцию с минимальным приоритетом

вернуть номер символа

Построение дерева

Структура узла

```
struct Node {
  char data;
  Node *left, *right;
  };
typedef Node *PNode;
```

Создание узла для числа (без потомков)

```
PNode NumberNode ( char c )
{
  PNode Tree = new Node;
  Tree->data = c;
  Tree->left = NULL;
  Tree->right = NULL;
  return Tree;
}
```

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
//-----------------------------------------
PNode MakeTree ( char Expr[], int first, int last )
{
 PNode Tree;
 int k;
 if ( first == last )
   return NumberNode ( Expr[first] );
 k = LastOperation ( Expr, first, last );
 Tree = new Node;
 Tree->data  = Expr[k];
 Tree->left  = MakeTree ( Expr, first, k-1 );
 Tree->right = MakeTree ( Expr, k+1, last );
 return Tree;
}
```

осталось только число

новый узел: операция

Вычисление выражения по дереву

```
//-----------------------------------------
// Функция CalcTree - вычисление по дереву
// Вход:  адрес дерева
// Выход: значение выражения
//-----------------------------------------
int CalcTree (PNode Tree)
{
 int num1, num2;
 if ( ! Tree->left ) return Tree->data - '0';
 num1 = CalcTree( Tree->left);
 num2 = CalcTree(Tree->right);
 switch ( Tree->data ) {
   case '+': return  num1+num2;
   case '-': return  num1-num2;
   case '*': return  num1*num2;
   case '/': return  num1/num2;
   }
 return 32767;
}
```

вернуть число, если это лист

вычисляем операнды (поддеревья)

выполняем операцию

некорректная операция

Основная программа

```
//-----------------------------------------
// Основная программа: ввод и вычисление
// выражения с помощью дерева
//-----------------------------------------
void main()
{
  char s[80];
  PNode Tree;
  printf ( "Введите выражение > " );
  gets(s);
  Tree = MakeTree ( s, 0, strlen(s)-1 );
  printf ( "= %d \n", CalcTree ( Tree ) );
  getch();
}
```

Дерево игры

Задача.

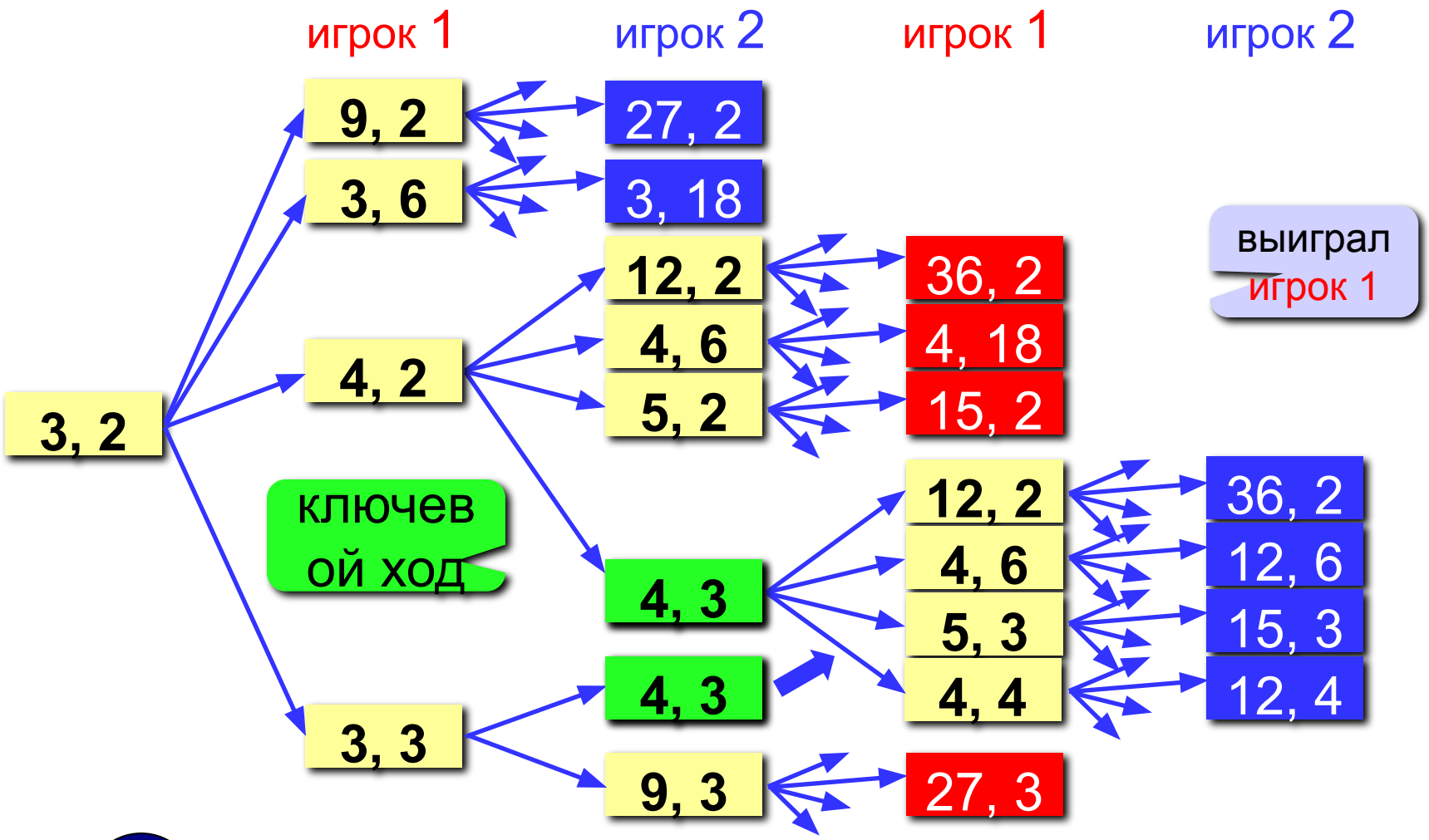
Перед двумя игроками лежат две кучки камней, в первой из которых 3, а во второй – 2 камня. У каждого игрока неограниченно много камней.

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?

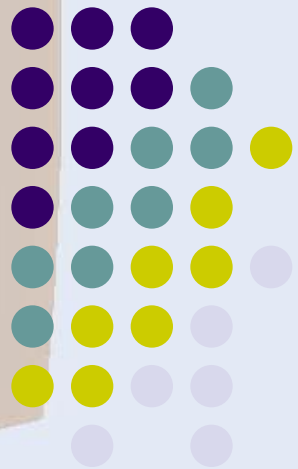
Дерево игры



При правильной игре выиграет игрок 2!

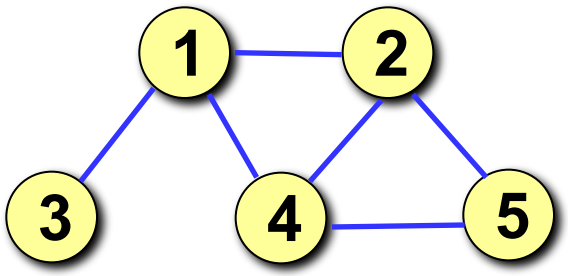
Лекция 5.4

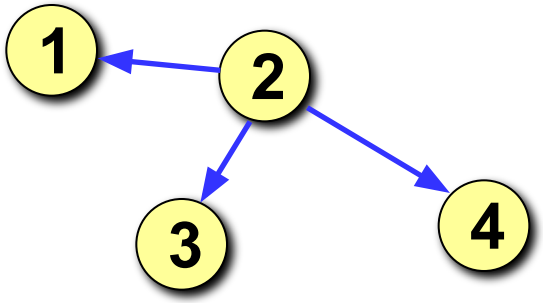
Графы



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).

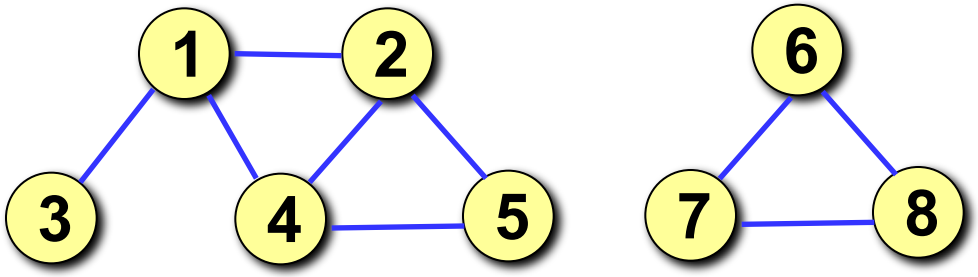
Дерево – это граф?

Да, без циклов!

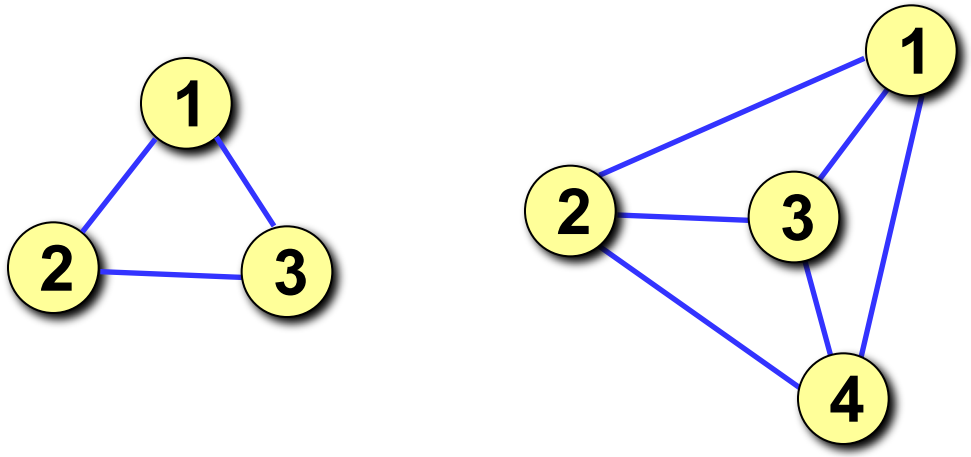
Определения

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

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

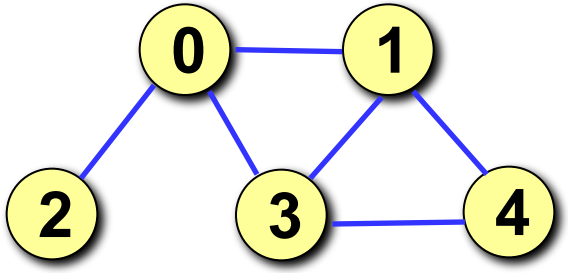


Полный граф – это граф, в котором проведены все возможные ребра (`n` вершин → `n(n-1)/2` ребер).



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.

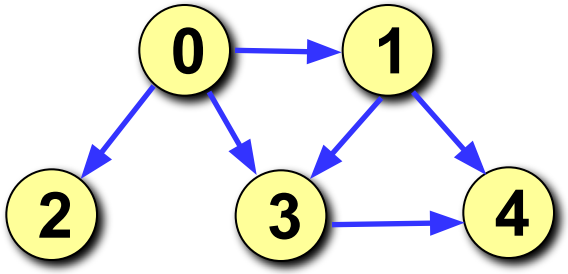


	0	1	2	3	4
0	0	1	1	1	0
1	1	0	0	1	1
2	1	0	0	0	0
3	1	1	0	0	1
4	0	1	0	1	0

Список смежности

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

! Симметрия!

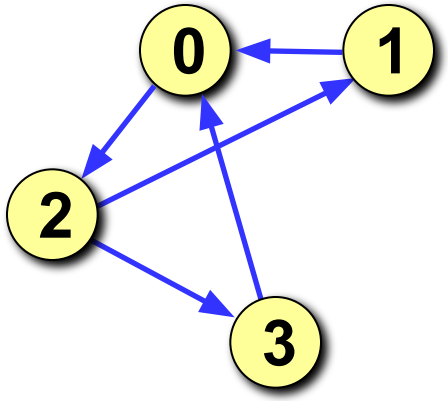


	0	1	2	3	4
0	0	1	1	1	0
1	0	0	0	1	1
2	0	0	0	0	0
3	0	0	0	0	1
4	0	0	0	0	0

0	1	2	3		
1	3	4			
2					
3	4				
4					

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).



$M =$

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

M^2`[i][j]=1`, если

`M[i][0]=1` и `M[0][j]=1` или
`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`

строка `i`

столбец `j`

логическое умножение

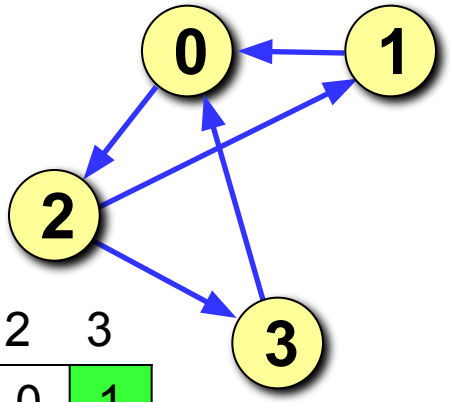
логическое сложение

Как обнаружить цепи и циклы?

Логическое умножение матрицы на себя:

матрица путей длины 2

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



$M^2 =$

0	0	1	0
1	0	0	0
0	1	0	1
1	0	0	0

$\otimes$

0	0	1	0
1	0	0	0
0	1	0	1
1	0	0	0

$=$

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

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

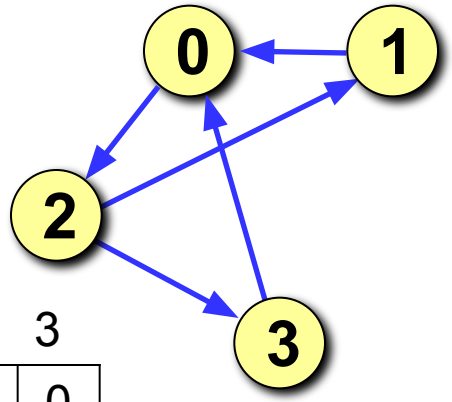
маршрут 2-1-0

маршрут 2-3-0

Как обнаружить цепи и циклы?

Матрица путей длины 3:

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



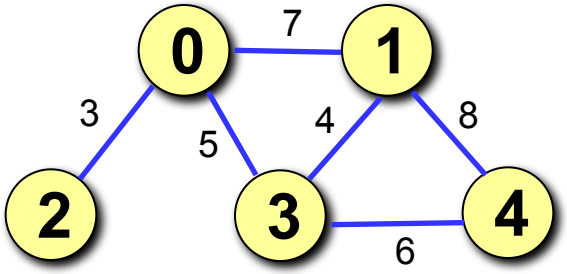
$$M^3 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

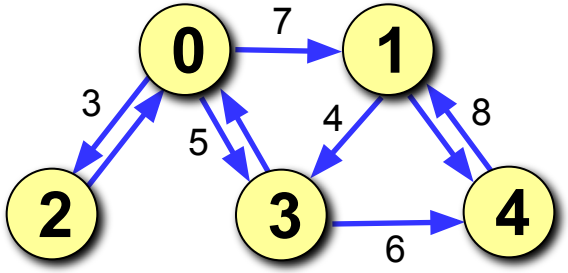
на главной диагонали – циклы!

$$M^4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

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.





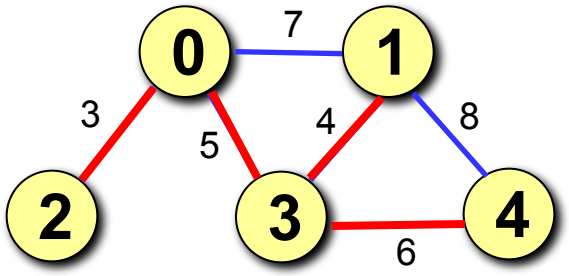
	0	1	2	3	4
0	0	7	3	5	∞
1	7	0	∞	4	8
2	3	∞	0	∞	∞
3	5	4	∞	0	6
4	∞	8	∞	6	0

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



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

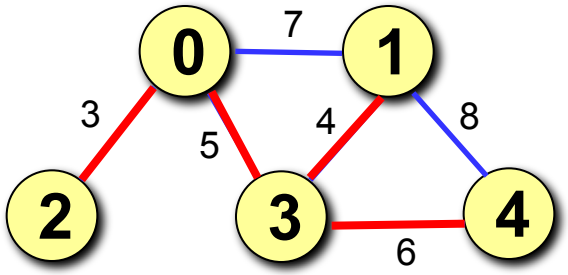
Жадный алгоритм

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.

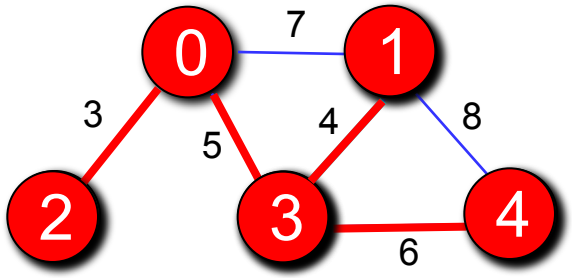


В задаче Прима-Краскала жадный алгоритм дает оптимальное решение!

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»:

```
struct rebro {
  int i, j;     // номера вершин
  };
```

Основная программа:

весовая матрица

цвета вершин

```
const N = 5;
void main()
{
int W[N][N], Color[N], i, j,
    k, min, col_i, col_j;
rebro Reb[N-1];
...// здесь надо ввести матрицу W
for ( i = 0; i < N; i ++ ) // раскрасить вершины
   Color[i] = i;
...// основной алгоритм – заполнение массива Reb
...// вывести найденные ребра (массив Reb)
}
```

Реализация алгоритма Прима-Краскала

Основной алгоритм:

```
for ( k = 0; k < N-1; k ++ ) {
  min = 30000; // большое число
  for ( i = 0; i < N-1; i ++ )
    for ( j = i+1; j < N; j ++ )
     if ( Color[i] != Color[j] &&
             W[i][j] < min ) {
       min = W[i][j];
       Reb[k].i = i;
          Reb[k].j = j;
          col_i = Color[i];
          col_j = Color[j];
          }
  for ( i = 0; i < N; i ++ )
    if ( Color[i] == col_j ) Color[i] = col_i;
  }
```

нужно выбрать N-1 ребро

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 = 0; k < N-1; k ++ ) {
  ...
  for ( i = 0; i < N-1; i ++ )
    for ( j = i+1; j < N; j ++ )
   ...
  }
```

три вложенных цикла, в каждом число шагов <=N

Количество операций:

O(N^3) растет не быстрее, чем N^3

Требуемая память:

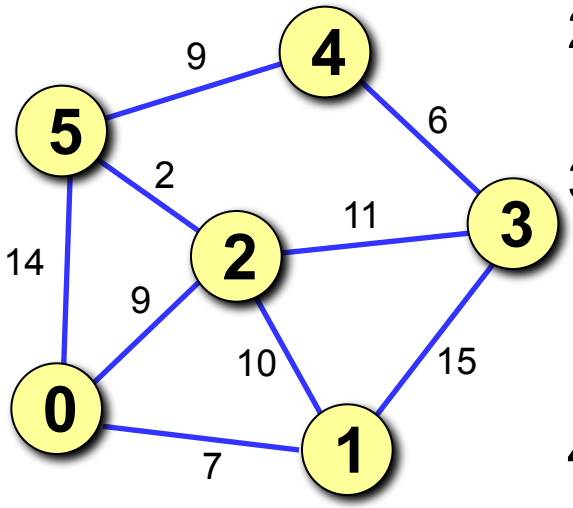
```
int W[N][N], Color[N];
rebro Reb[N-1];
```

➡ **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.

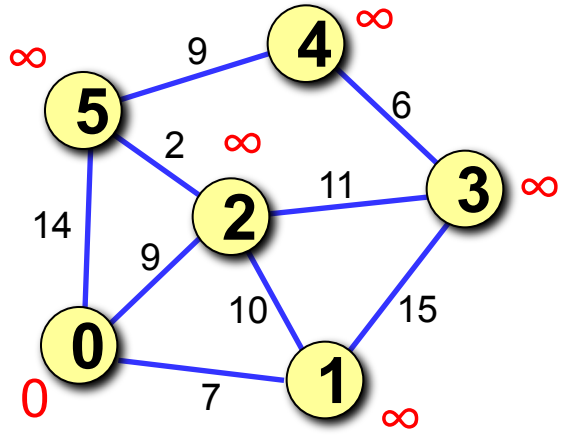
Та же задача: дан связный граф с `N` вершинами, веса ребер заданы матрицей `W`. Найти кратчайшие расстояния от заданной вершины до всех остальных.

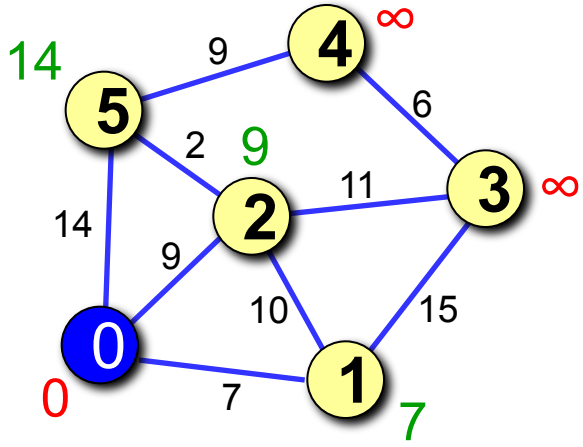


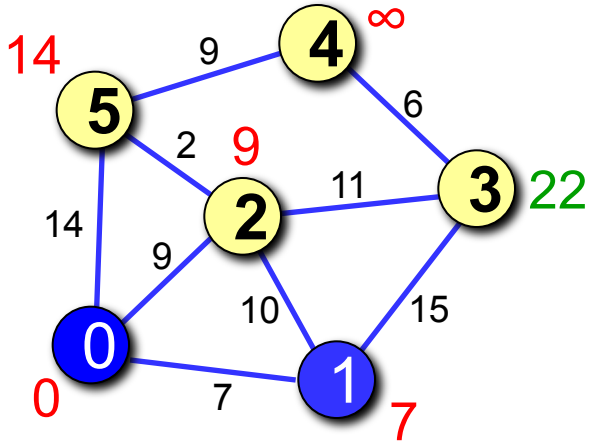
Алгоритм Дейкстры (*E.W. Dijkstra*, 1959)

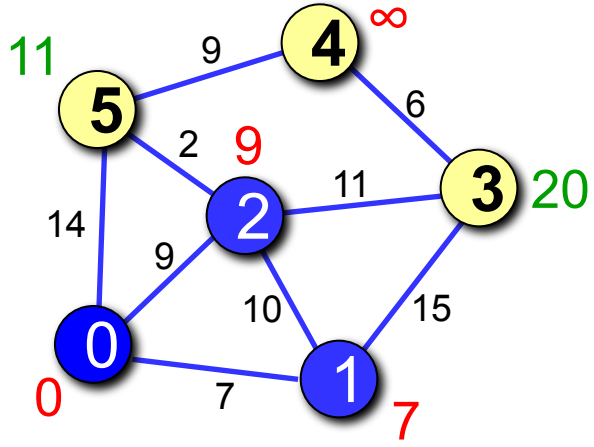
1) присвоить всем вершинам метку ∞;
2) среди нерассмотренных вершин найти вершину `j` с наименьшей меткой;
3) для каждой необработанной вершины `i`: если путь к вершине `i` через вершину `j` меньше существующей метки, заменить метку на новое расстояние;
4) если остались необработанны вершины, перейти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

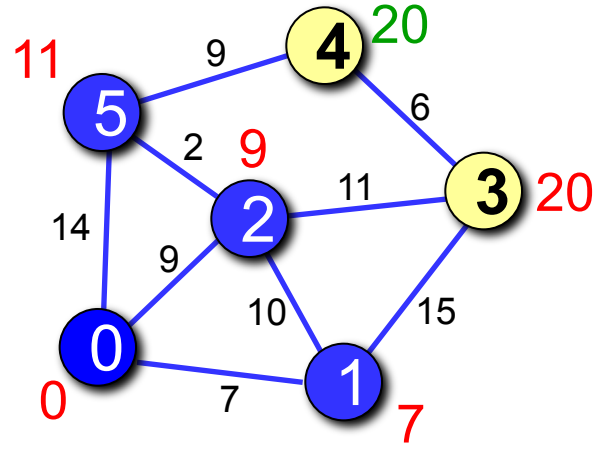
Алгоритм Дейкстры

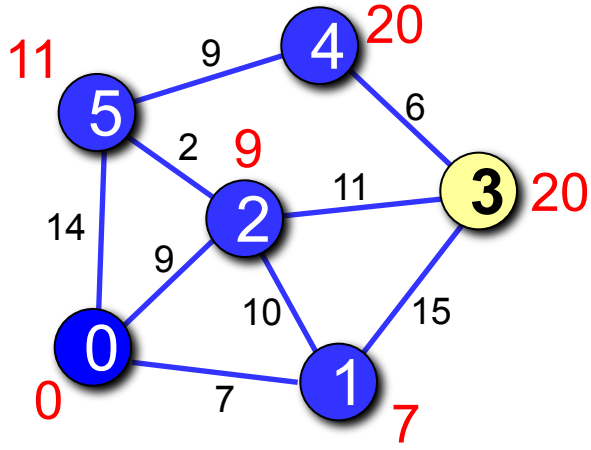












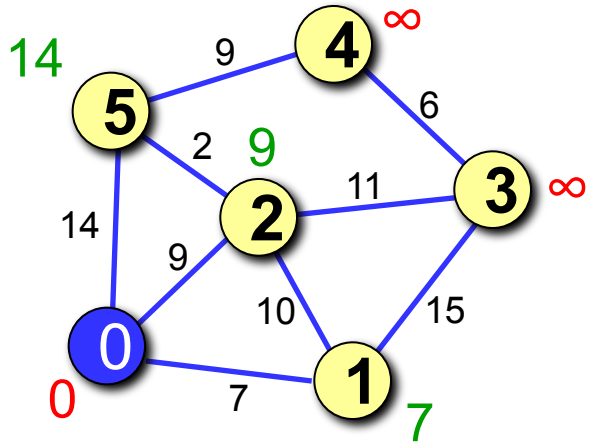
Реализация алгоритма Дейкстры

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`.



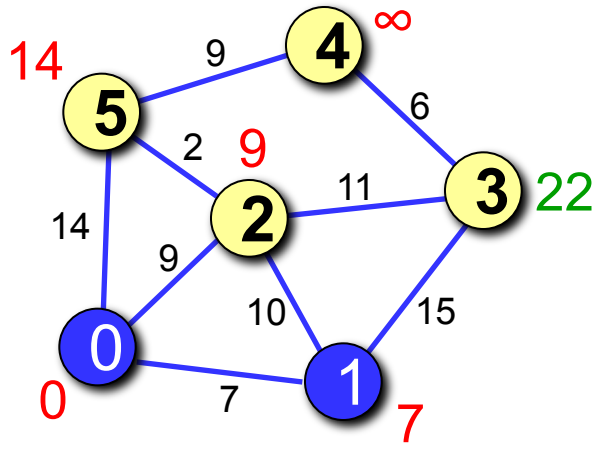
	0	1	2	3	4	5
a	**1**	**0**	**0**	**0**	**0**	**0**
b	**0**	**7**	**9**	**∞**	**∞**	**14**
c	**0**	**0**	**0**	**0**	**0**	**0**

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`.

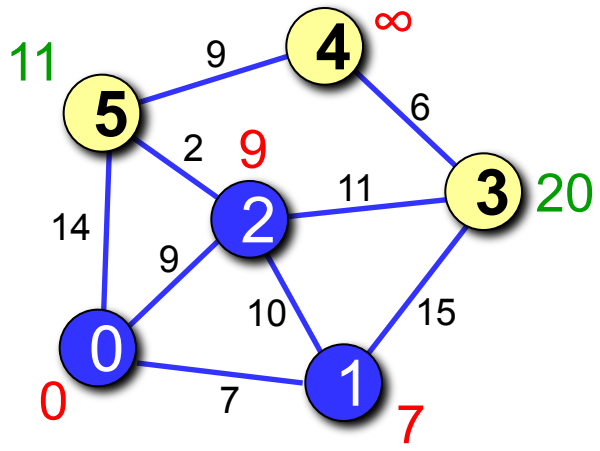
Шаг 1:



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

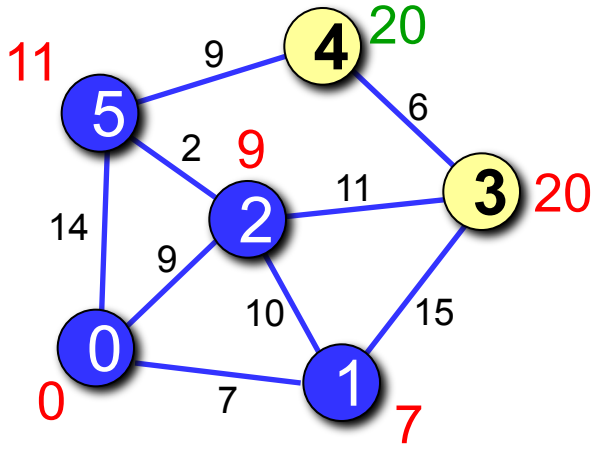
Реализация алгоритма Дейкстры

Шаг 2:



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

Шаг 3:



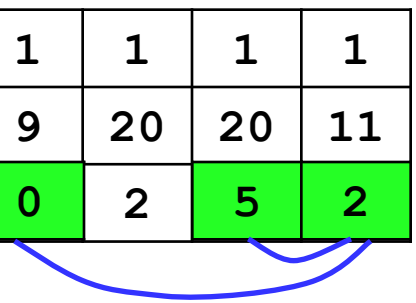
	0	1	2	3	4	5
a	1	1	1	0	0	1
b	0	7	9	20	20	11
c	0	0	0	2	5	2

Дальше массивы не изменяются!

Как вывести маршрут?

Результат работа алгоритма Дейкстры:

	0	1	2	3	4	5
a	1	1	1	1	1	1
b	0	7	9	20	20	11
c	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]!=x` присвоить `z=c[z]` и вывести `z`.

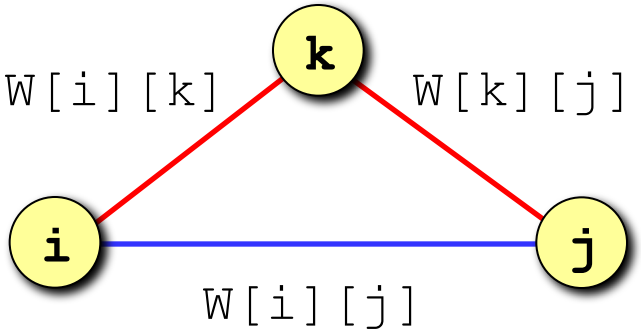
Сложность алгоритма Дейкстры:

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 = 0; k < N; k ++ )
  for ( i = 0; i < N; i ++ )
    for ( j = 0; j < N; j ++ )
      if ( W[i][j] > W[i][k] + W[k][j] )
           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 = 0; i < N; i ++ )
  for ( j = 0; j < N; j ++ )
    c[i][j] = i;
...
for ( k = 0; k < N; k ++ )
  for ( i = 0; i < N; i ++ )
    for ( j = 0; j < N; j ++ )
      if ( W[i][j] > W[i][k] + W[k][j] )
        {
        W[i][j] = W[i][k] + W[k][j];
        c[i][j] = c[k][j];
        }
```

`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? $O(N^3)$

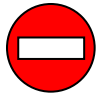
Задача коммивояжера

Задача коммивояжера. Коммивояжер (бродячий торговец) должен выйти из первого города и, посетив по разу в неизвестном порядке города 2, 3, . . . 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 ($i=1,\ldots,N$). 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.