

**Курс «Алгоритмы и алгоритмические языки»
1 семестр 2015/2016**

Лекция 20

Двоичное дерево

- ◇ Двоичное дерево – набор узлов, который:
 - ◆ либо пуст (пустое дерево),
 - ◆ либо разбит на три непересекающиеся части:
узел, называемый *корнем*,
двоичное дерево, называемое *левым поддеревом*, и
двоичное дерево, называемое *правым поддеревом*.
- ◇ Двоичное дерево не является частным случаем обычного дерева, хотя у этих структур много общего. Основные отличия:
 - (1) Пустое дерево является двоичным деревом, но не является обычным деревом.
 - (2) Двоичные деревья $(A(B, NULL))$ и $(A(NULL, B))$ различны, а обычные деревья – одинаковы.
- ◇ Термины: узлы, ветви, корень, листья, высота

Двоичное дерево

- Представление двоичного дерева в памяти компьютера
Описание узла двоичного дерева на Си:

```
typedef struct bin_tree {  
    char info;  
    struct bin_tree *left;  
    struct bin_tree *right;  
} node;
```

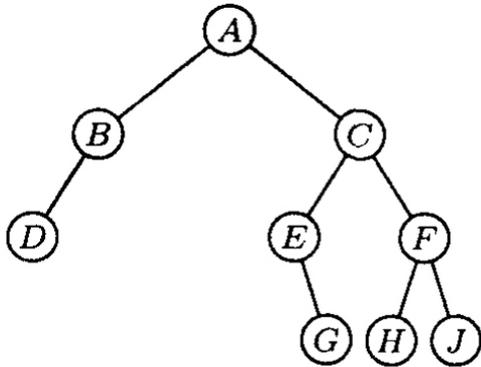


Рис. 1. Двоичное дерево

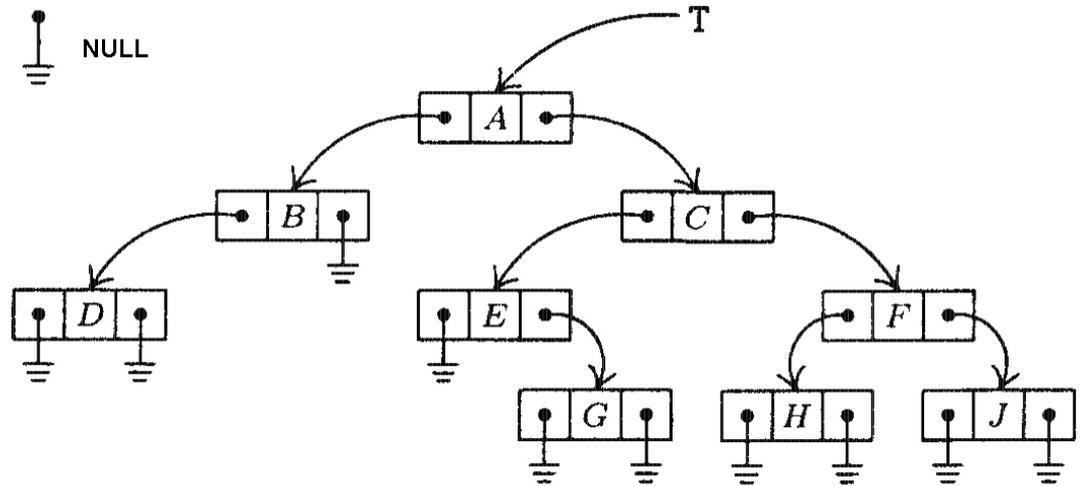


Рис. 2 Представление дерева с рис.1 в компьютере.

Двоичное дерево

◇ *Обход двоичного дерева.*

Обход дерева позволяет выполнить топологическую сортировку узлов дерева и расположить их в линейном одномерном массиве, порядок узлов дерева в котором таков, что их можно обрабатывать в цикле

```
for (i = 0; i < N; i++)
```

(топологический порядок)

Двоичное дерево

◆ Различные способы обхода двоичного дерева

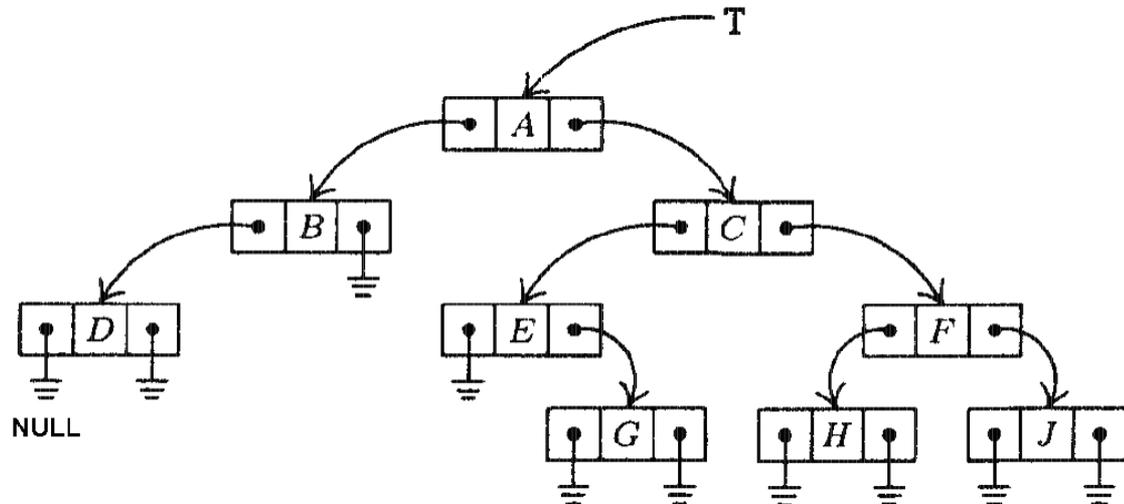
(1) Обход в глубину в *прямом порядке*:

- ◆ обработать корень,
- ◆ обойти левое поддерево,
- ◆ обойти правое поддерево.

Порядок обработки узлов дерева на рисунке

А В D C E G F H J

Линейная последовательность узлов, полученная при прямом обходе, отражает «спуск» информации от корня дерева к листьям.



Двоичное дерево

◆ Различные способы обхода двоичного дерева

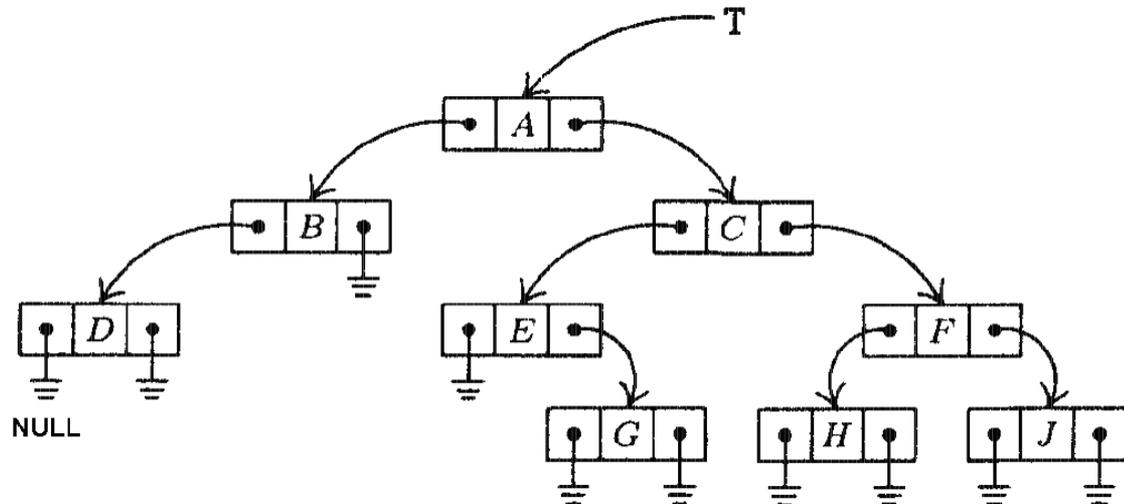
(2) Обход в глубину в *обратном порядке*:

- ◆ обойти левое поддерево,
- ◆ обойти правое поддерево,
- ◆ обработать корень.

Порядок обработки узлов дерева на рисунке:

D B G E H J F C A

Линейная последовательность узлов, полученная при обратном обходе, отражает «подъем» информации от листьев к корню дерева.



Двоичное дерево

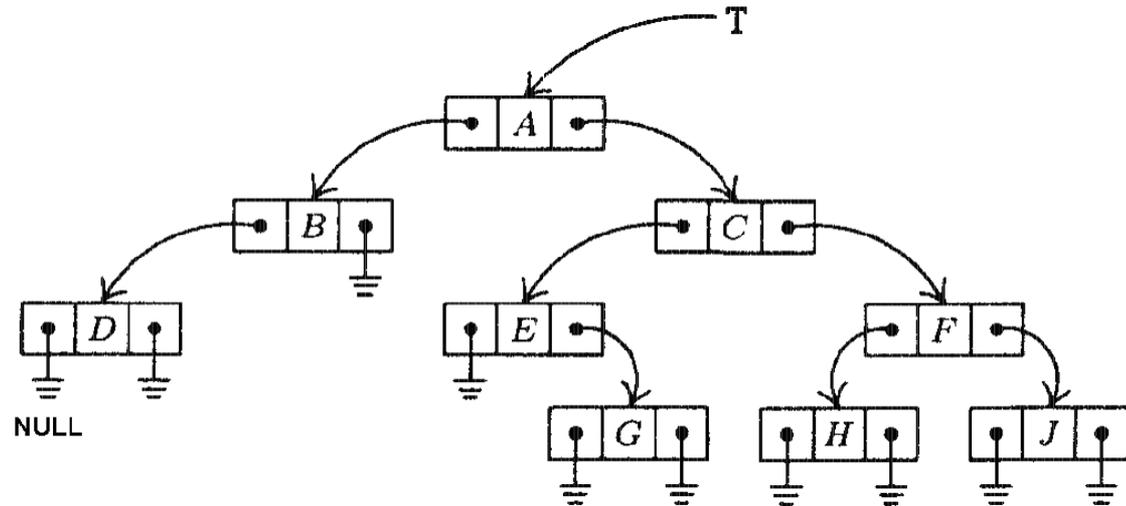
◆ *Различные способы обхода двоичного дерева*

(3) Симметричный обход в глубину (обход в симметричном порядке):

- ◆ обойти левое поддерево,
- ◆ обработать корень.
- ◆ обойти правое поддерево,

Порядок обработки узлов дерева на рисунке:

D B A E G C H F J



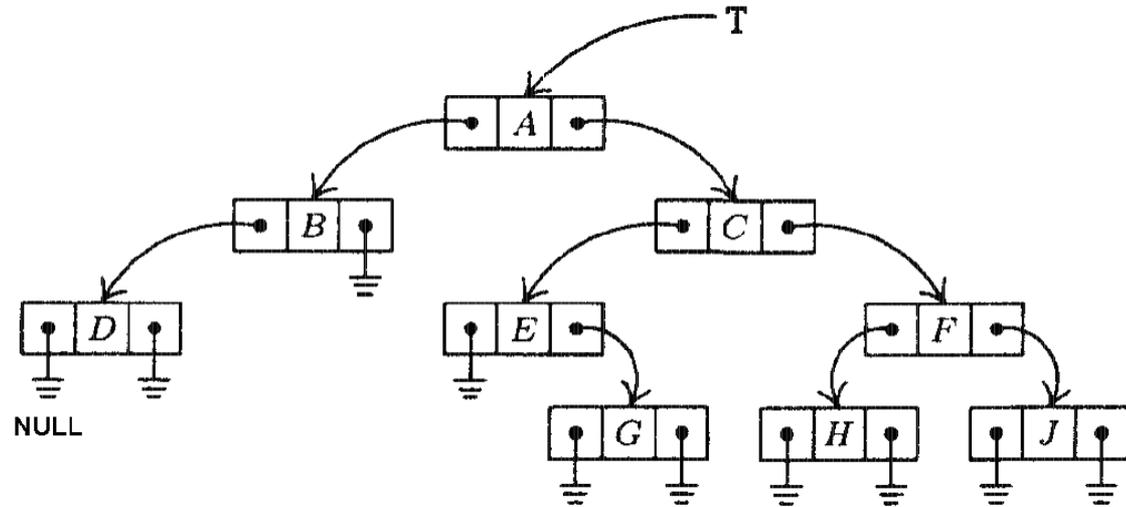
Двоичное дерево

◆ Различные способы обхода двоичного дерева

- (4) Обход двоичного дерева в ширину:
узлы дерева обрабатываются «по уровням»
(уровень составляют все узлы, находящиеся на
одинаковом расстоянии от корня)

Порядок обработки узлов дерева на рисунке:

А В С D E F G H J



Двоичное дерево

- ◆ Функции, реализующие обходы двоичного дерева, позволяют по указателю каждого узла дерева P вычислить указатели узлов

P_next_pre , P_next_post и P_next_in ,
 P_pred_pre , P_pred_post и P_pred_in .

- ◆ Рекурсивные Си-функции обхода двоичного дерева в глубину

```
(1) void preorder (node * r) {  
    if (r == NULL)  
        return;  
    if (r->info)  
        printf ("%c", r->info);  
    preorder (r->left);  
    preorder (r->right);  
}
```

Двоичное дерево

- ◆ Рекурсивные Си-функции обхода двоичного дерева в глубину

```
(2) void postorder (node *r) {  
    if (r == NULL)  
        return;  
    postorder (r->left);  
    postorder (r->right);  
    if (r->info)  
        printf ("%c", r->info);  
}
```

```
(3) void inorder (node *r) {  
    if (r == NULL)  
        return;  
    inorder (r->left);  
    if (r->info)  
        printf ("%c", r->info);  
    inorder (r->right);  
}
```

Двоичное дерево

- ◆ Нерекурсивная функция обхода двоичного дерева (управление стеком ведется не автоматически, а в самой функции).
 - `r` – указатель на корень дерева;
 - `t` – указатель на корень обрабатываемого (текущего) поддерева;
 - `stack` – массив, на котором моделируется стек,
 - `depth` – глубина стека,
 - `top` – указатель вершины стека;
- ◆ Стек требуется для ручного сохранения параметров функции, локальных переменных и точки возврата (если рекурсивных вызовов функции несколько).
- ◆ В функции `inorder` нет локальных переменных, а второй из двух рекурсивных вызовов хвостовой, что позволяет не сохранять его параметры в стеке
 - ◆ Поэтому сохраняется только параметр функции

Двоичное дерево

◆ Нерекурсивная функция обхода двоичного дерева

Алгоритм:

(1) [Инициализация]. Сделать стек пустым, т.е. затолкнуть `NULL` на дно стека: `stack[0] = NULL;`
установить указатель стека на дно стека: `top = 0;`
установить указатель `t` на корень дерева: `t = r.`

(2) [Конец ветви]. Если `t == NULL`, перейти к (4).

(3) [Продолжение ветви]. Затолкнуть `t` в стек:
`stack[++top] = t;`

установить `t = t->left` и вернуться к шагу (2).

(4) [К обработке правой ветви]. Вытолкнуть верхний элемент стека в `t`: `t = stack[top]; top--;`

Если `t == NULL`, выполнение алгоритма

прекращается, иначе обработать данные узла, на который указывает `t`, и перейти к шагу (5).

(5) [Начало обработки правой ветви]. Установить `t = t->right` и вернуться к шагу (2).

Двоичное дерево

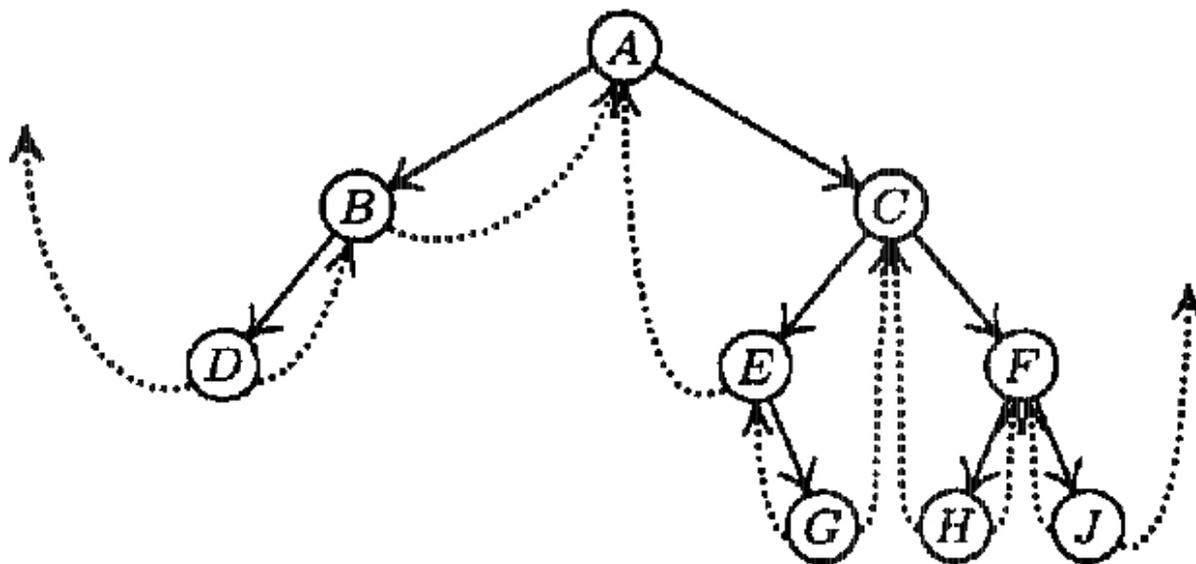
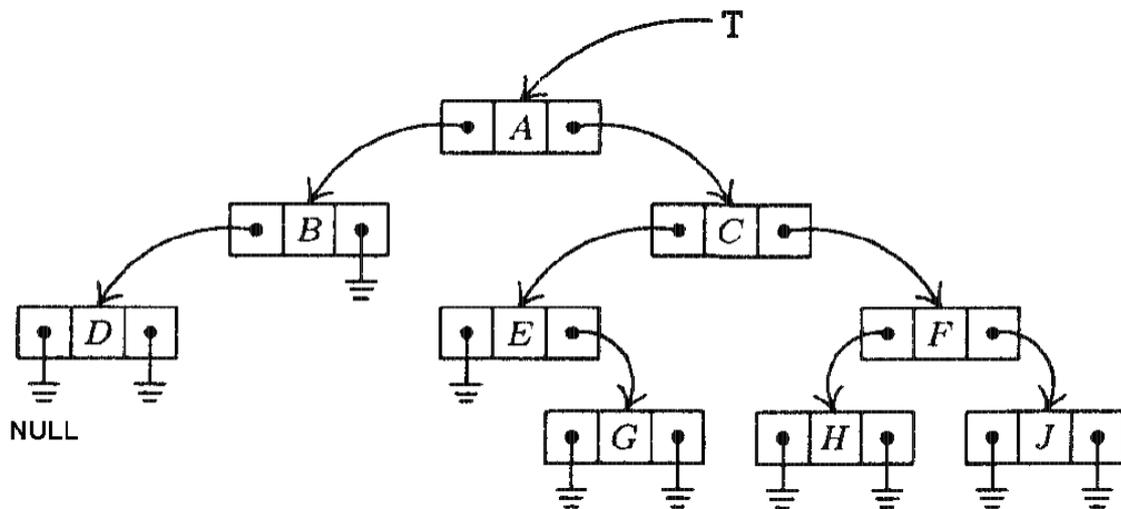
◆ Нерекурсивная функция обхода двоичного дерева

```
int inorder (node *r, char *order) {
    node *t = r;
    node *stack[depth];           // depth = ?
    int top = 0, i = 0;

    if (!t)
        return 0;
    stack[0] = NULL;             //Шаг 1
    while (1) {
        while (t) {              //Шаг 2
            stack[++top] = t;    //Шаг 3
            t = t->left;
        }
        t = stack[top--];        //Шаг 4
        if (t) {
            order[i++] = t->info; //обработка
            t = t->right;         //Шаг 5
        } else                   //t == NULL
            break;               //Шаг 4
    }
    return i;
}
```

Двоичное дерево

◇ Прошитое двоичное дерево



Рассмотрим двоичное дерево на верхнем рисунке. У этого дерева нулевых указателей, больше, чем ненулевых: 10 против 8. Это – типичный случай.

Будем записывать вместо нулевых указателей указатели на родителей (или более далеких предков) соответствующих узлов (такие указатели называются *нитями*). Это позволит при обходе дерева не использовать стек.

Двоичное дерево

- ◇ *Прошитое двоичное дерево*
- ◇ Описание узла прошитого двоичного дерева

```
typedef struct bin_tree {  
    char info;  
    struct bin_tree *left;  
    struct bin_tree *right;  
    char left_tag;  
    char right_tag;  
} threaded_node;
```

Нити устанавливаются таким образом, чтобы указывать на предшественников (левые нити) или последователей (правые нити) текущего узла при соответствующем обходе дерева. Например, в случае симметричного обхода:

<i>Обычное дерево</i>	<i>Прошитое дерево</i>
<code>P->left == NULL</code>	<code>P->left_tag == 1, P->left == P_pred_in</code>
<code>P->left == Q</code>	<code>P->left_tag == 0, P->left == Q</code>
<code>P->right == NULL</code>	<code>P->right_tag == 1, P->right == P_next_in</code>
<code>P->right == Q</code>	<code>P->right_tag == 0, P->right == Q</code>

Двоичное дерево

◇ *Прошитоое двоичное дерево*

◇ Нити существенно упрощают алгоритмы обхода двоичных деревьев. Например, для вычисления для каждого узла `p` указатель узла `P_next_in` можно использовать следующий простой алгоритм:

```
threaded_node * next_in (threaded_node *p) {
    threaded_node *q = p->right;
    if (p->right_tag == 1)
        return q;
    while (q->left_tag == 0) //q != NULL
        q = q->left;        //q->left != NULL
    return q;
}
```

◇ Функция `next_in` фактически реализует симметричный обход дерева, так как позволяет для произвольного узла дерева `P` найти `P_next_in`. Многократно применяя эту функцию, можно вычислить топологический порядок узлов двоичного дерева, соответствующий симметричному обходу.

Двоичное дерево

◇ *Прошитоое двоичное дерево*

◇ Аналогичным образом можно вычислить `P_next_pred` и `P_next_post`.

Применяя функции `next_pred` (либо `next_post`), можно вычислить топологический порядок узлов, соответствующий прямому (либо обратному) обходу.

◇ **Замечания**

- (1) С помощью обычного представления невозможно для произвольного узла `P` вычислить `P_next_in`, не вычисляя всей последовательности узлов.
- (2) Функции `next_in` не требуется стек ни в явной, ни в неявной (рекурсия) форме.

Двоичное дерево

◆ *Прошитое двоичное дерево*

◆ Сравнение функций `inorder()` и `next_in()` позволяет сделать следующие выводы:

- ◆ Если `p` – произвольно выбранный узел дерева, то следующий фрагмент функции `next_in()`:

```
q = p->right;  
if (p->right_tag == 1)  
    return q;
```

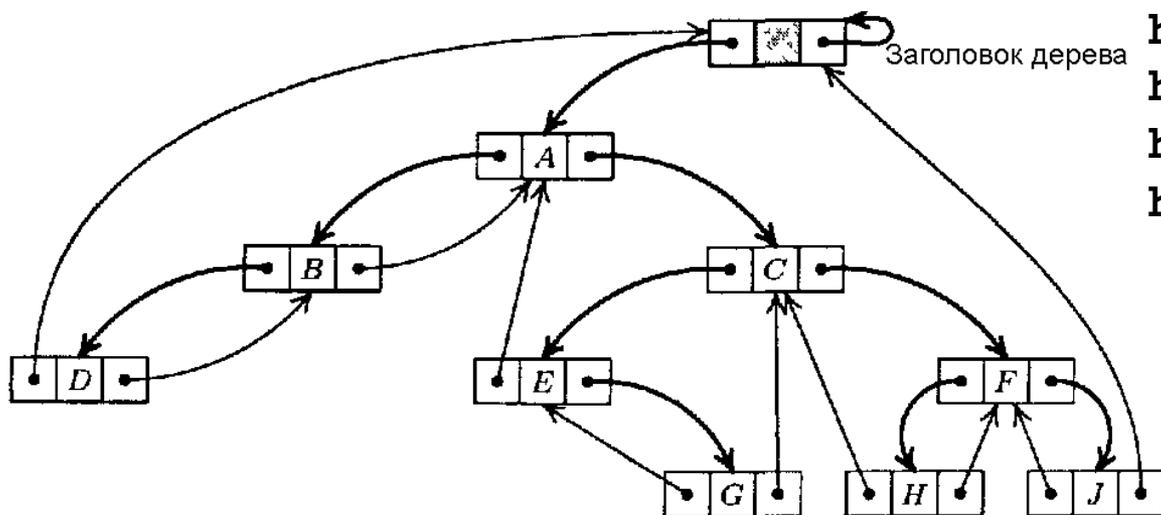
выполняется только один раз.

- ◆ Обход прошитога дерева выполняется быстрее, так как для него не нужны операции со стеком.
- ◆ Для `inorder()` требуется больше памяти, чем для `next_in()`, из-за массива `stack[depth]`. `depth` стараются взять не очень большим, но `depth` не может быть меньше высоты двоичного дерева.
Нельзя допускать переполнение стека деревьев.

Двоичное дерево

◆ Прошитоое двоичное дерево

В функции `inorder()` используется указатель `r` на корень двоичного дерева. Желательно, применив функцию `next_in()` к корню `r`, получить указатель на самый первый узел дерева для выбранного порядка обхода. Для этого к дереву добавляется еще один узел – заголовок дерева (`header`).



поля структуры

```
typedef struct bin_tree
{
    char info;
    struct bin_tree *left;
    struct bin_tree *right;
    char left_tag;
    char right_tag;
} threaded_node;
threaded_node *header;
```

заполняются в заголовке следующим образом

```
header->left_tag = 0;
header->right_tag = 0;
header->left = r;
header->right = header;
```

На рисунке дуги дерева показаны более жирными линиями, 19 чем нити.