# Курс «Алгоритмы и алгоритмические языки»

# Лекция 16

## *Двоичные деревья поиска*

- Структура для представления узла двоичного дерева:

```
struct BT_node {
  int key;
  struct BT_node *left;
  struct BT_node *right;
  struct BT_node *parent;
}
```

- Ключи в двоичном дереве поиска хранятся с соблюдением свойства упорядоченности:

  Пусть *x* – произвольный узел двоичного дерева поиска. Если узел *y* принадлежит левому поддереву, то

  $$key[y] \le key[x],$$

  если *y* находится в правом поддереве узла *x*, то

  $$key[y] \ge key[x].$$

## *Двоичные деревья поиска: вставка*

- **На входе**: указатель *root* на корень дерева и указатель *node* на новый узел, у которого есть значение ключа, а все поля с указателями имеют значение *NULL*.

```
void Btinsert (struct BT_node *root,
               struct BT_node *node) {
  struct BT_node *p, *q, *s;
  p = root, q = NULL;
  while (p) {
    q = p;
    p = (node->key < p->key) ? p->left : p->right;
  }
  node->parent = q;
  /* подвешиваем node к корню, либо к q слева или справа*/
  if (q == NULL)
    root = node;
  else if (node->key < q->key)
    q->left = node;
  else
    q->right = node;
}
```

## *Двоичные деревья поиска: удаление*

- **На входе**: указатель на корень *root* дерева *T* и указатель на узел *n* дерева *T*.

  **На выходе**: двоичное дерево *T* с удаленным узлом *n* (ключи нового дерева по-прежнему упорядочены).

- Необходимо рассмотреть три случая: (1) у узла *n* нет детей (листовой узел); (2) у узла *n* только один ребенок; (3) у узла *n* два ребенка.
  - (1) просто удаляем узел *n*;
  - (2) вырезаем узел *n*, соединив единственного ребенка узла *n* с родителем узла *n*.
  - (3) находим *succ*(*n*) и удаляем его, поместив ключ *succ*(*n*) в узел *n*.

## *Двоичные деревья поиска: удаление*

- Шаг 1: если у *n* меньше двух детей, удаляем *n*, иначе удаляем *succ*(*n*); устанавливаем указатель *y* на удаляемый узел.
- Шаг 2: находим ребенка удаляемого узла и устанавливаем на него указатель *x*.
- Шаг 3: подвешиваем ребенка *y* (указатель *x*) к родителю *y*; если у *y* нет родителя, новым корнем дерева становится *x*; устанавливаем в соответствующем поле родителя указатель на *x*, полностью исключая *y* из дерева.
- Шаг 4: если удаляемый узел *succ*(*n*), заменяем данные узла *n* на данные узла *succ*(*n*).

## *Двоичные деревья поиска: удаление*

```
struct BT_node BTdelete (struct BT_node **root,
                         struct BT_node *n) {
  struct BT_node *x, *y;
  /* y – указатель на удаляемый узел n */
  y = (! n->left || ! n->right) ? n : BT_succ (n);
  /* x – указатель на ребенка y, либо NULL */
  x = y->left ? y->left : y->right;
  /* если x – ребенок y, вырезаем y */
  if (x)
    x->parent = y->parent;
  /* если у y нет родителя, новым корнем дерева становится x */
  if (! y->parent)
    *root = x;
  else {
    /* x присоединяется к y->parent с требуемой стороны */
    if (y == y->parent->left)
      y->parent->left = x;
    else
      y->parent->right = x;
  }
  <...>
```

## *Двоичные деревья поиска: удаление*

```
struct BT_node BTdelete (struct BT_node **root,
                         struct BT_node *n) {
  struct BT_node *x, *y;
   <...>
  /* если удалялся не узел n, а succ(n), необходимо
     заменить данные узла n на данные узла succ(n) */
  if (y != n)
    n->key = y->key;
  /* функция возвращает указатель удаленного узла, что
   дает возможность использовать этот узел в других
   структурах, либо очистить занимаемую им память */
  return y;
}
```

- Среднее время выполнения $O(h)$, где $h$ – высота дерева.

## *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;
```

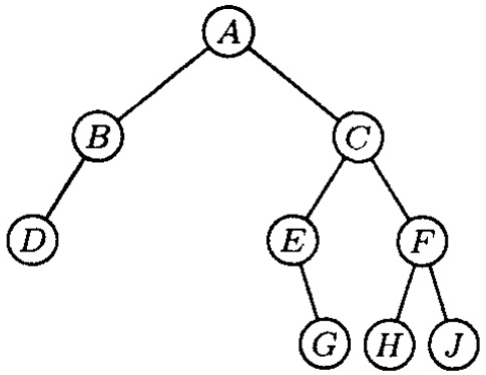



Рис. 1. Двоичное дерево

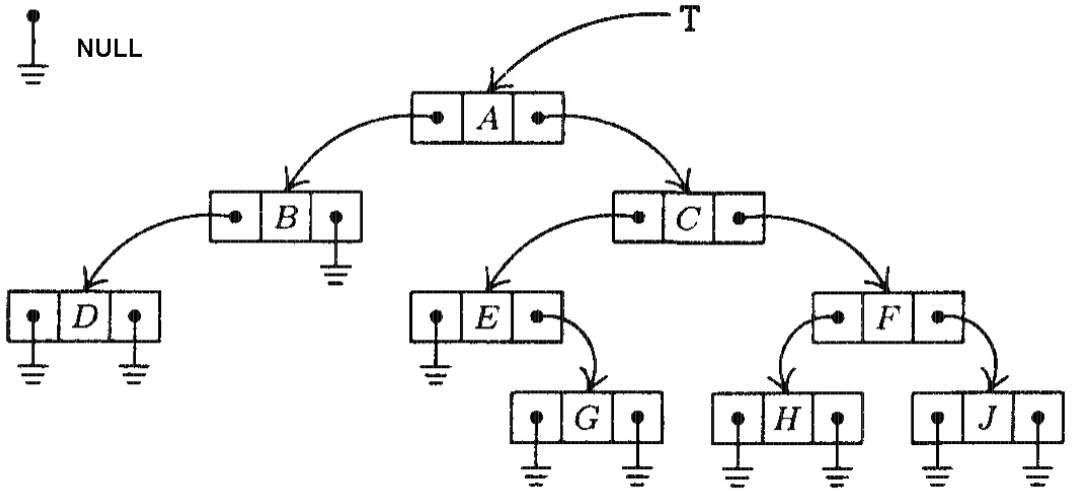



Рис. 2 Представление дерева с рис.1 в компьютере.

## *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***

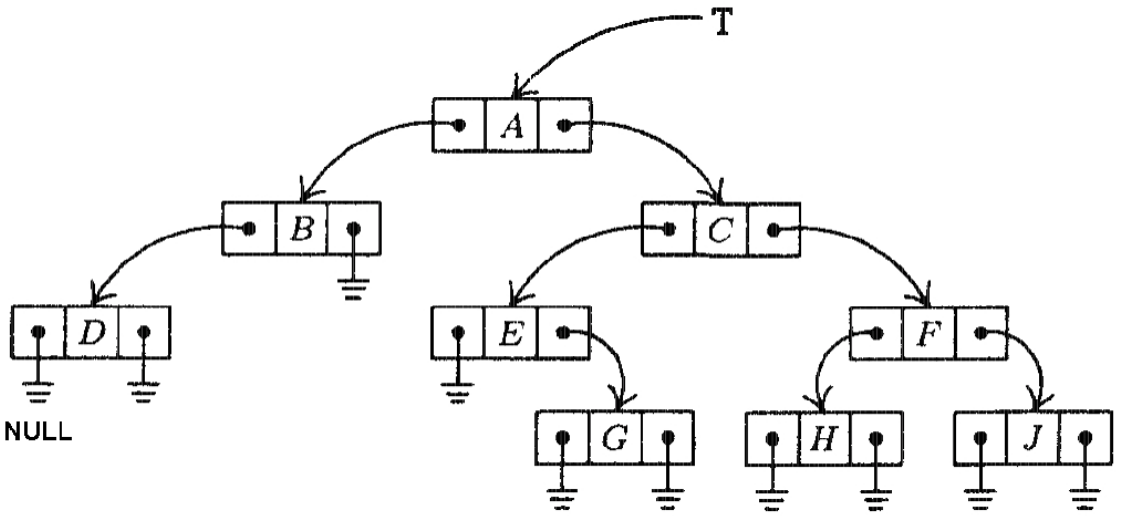
(1) Обход в глубину в *прямом порядке*:

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

Порядок обработки узлов дерева на рисунке

**A B 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***

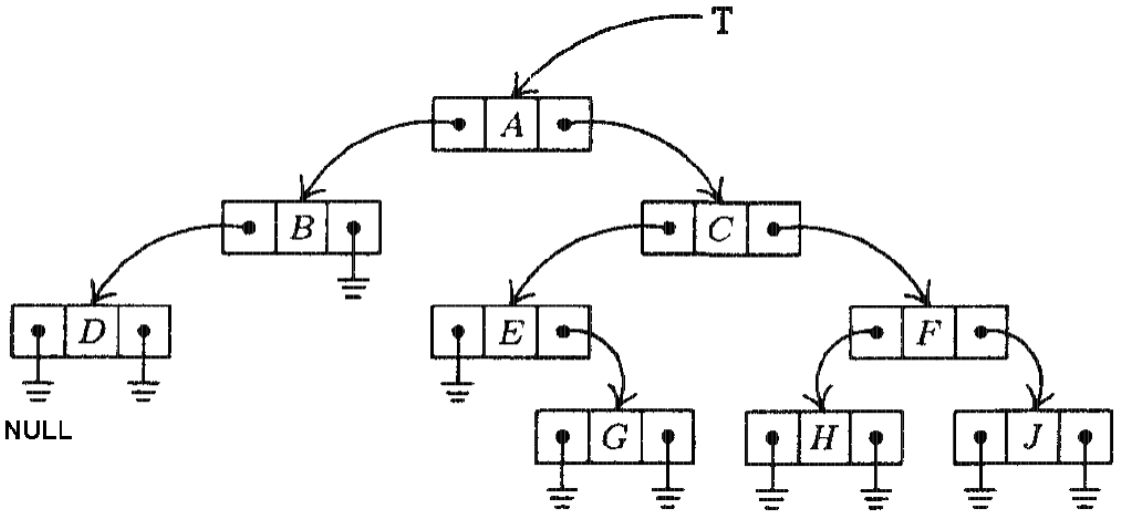
(2) Обход в глубину в *обратном порядке*:

- 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*

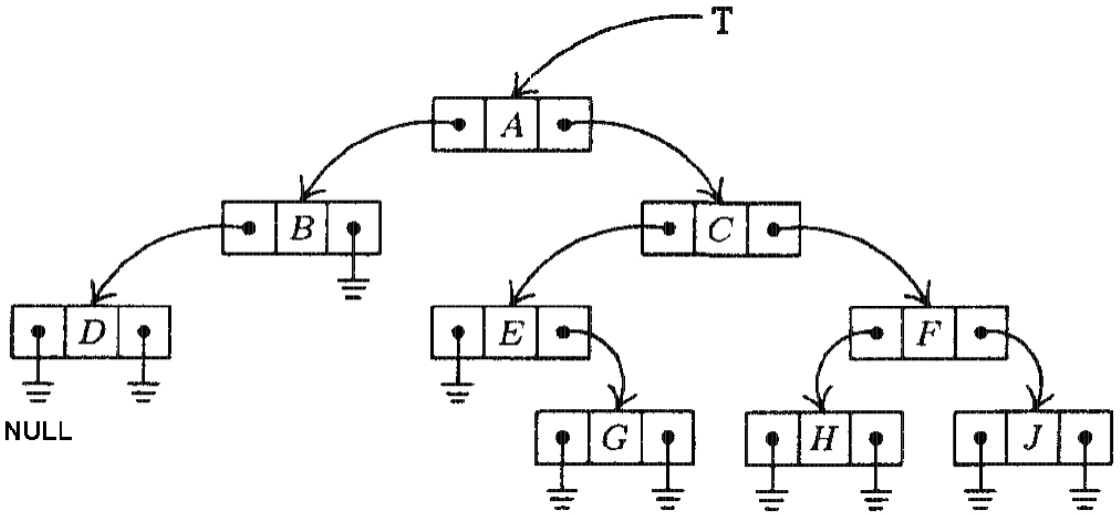
- ***Различные способы обхода двоичного дерева***

(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**



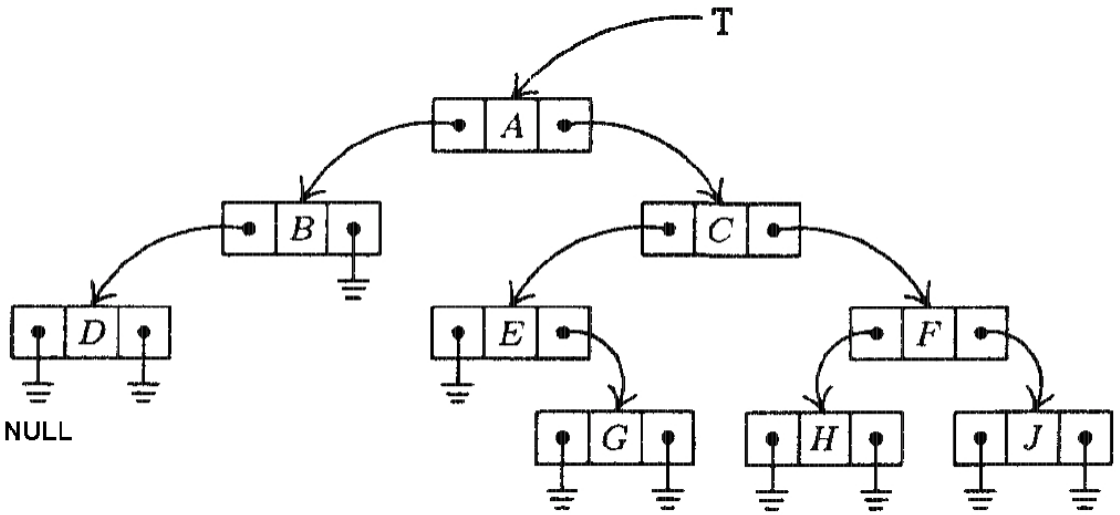
## *Двоичное дерево*

- ***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:

**A B C 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).

  **T** – указатель на корень дерева;

  **P** – указатель на корень обрабатываемого (текущего) поддерева;

  **Stack[D]** – массив,на котором моделируется стек,

  **D** – глубина стека,

  **bottom = Stack = Stack[0]** – указатель дна стека;

  **top** – указатель вершины стека;

## *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 **P** на корень дерева: **`P = T.`**

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

(3) [Продолжение ветви]. Затолкнуть **P** в стек: **`Stack[top] = P; top += 1;`**

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

(4) [К обработке правой ветви]. Вытолкнуть верхний элемент стека в **`P: P = Stack[top]; top -= 1;`** Если **`P == NULL`**, выполнение алгоритма прекращается, иначе обработать данные узла, на который указывает **P,** и перейти к шагу (5).

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

## *Двоичное дерево*

- Нерекурсивная функция обхода двоичного дерева

```
void inorder (node r, int *order) {
  node *P;
  int Stack[D];
  int top = 0, i = 0;
  Stack[0] = NULL;
  P = r;
  while (P) {
    while (P) {
      Stack[top++] = P;
      P = P->left;
    }
    P = Stack[top--];
    if (P) {
      order[i++] = P->info;
      P = P->right;
    }
  }
}
```