Московский государственный университет им. М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Алгоритмы и алгоритмические языки
Лекция 19

16 ноября 2019 г.

Топологическая сортировка на Си

```
#include <stdio.h>
#include <stdlib.h>
typedef struct ldr { /* дескриптор ведущего узла */
  char key;
  int count;
  struct ldr *next;
  struct trl *trail;
} leader;
typedef struct trl { /* дескриптор ведомого узла */
  struct ldr *id;
  struct trl *next;
} trailer;

leader *head, *tail; /* два вспомогательных узла */
int lnum; /* счётчик ведущих узлов */
```

Топологическая сортировка на Си: поиск по ключу

```
leader *find (char w) {
  leader *h = head;
  /* барьер на случай отсутствия w */
  tail->key = w;
  while (h->key != w)
    h = h->next;
  if (h == tail) {
    /* генерация нового ведущего узла */
    tail = malloc (sizeof (leader));
    /* старый tail становится новым элементом списка */
    lnum++;
    h->count = 0;
    h->trail = NULL;
    h->next = tail;
  }
  return h;
}
```

Топологическая сортировка на Си: формирование списка

```
void init_list() {
  leader *p, *q;
  trailer *t;
  char x, y;

  head = (leader *) malloc (sizeof (leader));
  tail = head;
  lnum = 0;                  /* начальная установка */
  while (1) {
    if (scanf ("%c␣%c", &x, &y) != 2)
      break;
    /* включение пары в список */
    p = find (x);
    q = find (y);
    <...>
```

Топологическая сортировка на Си: формирование списка

```
      <...>
      /* коррекция списка */
      t = malloc (sizeof (trailer));
      t->id = q;
      t->next = p->trail;
      p->trail = t;
      q->count += 1;
    }
}
```

Топологическая сортировка на Си: новый список

```
void sort_list() {
  leader *p, *q;
  trailer *t;
  /*  В выходной список включаются все узлы с count == 0 */
  p = head;
  head = NULL; /* голова выходного списка */
  while (p != tail) {
    q = p;
    p = q->next;
    if (q->count == 0) {
      /* включение q в выходной список */
      q->next = head;
      head = q;
    }
  }
<...>
```

Топологическая сортировка на Си: новый список

```
q = head; /* есть ведущий узел -> head != NULL */
while (q != NULL) {
  printf ("%c\n", q->key);
  lnum--;
  t = q->trail;
  q = q->next;
  while (t != NULL) {
    p = t->id;
    p->count -= 1;
    if (p->count == 0) {
      p->next = q;  // достаточно для
      q = p;        // правильной сортировки
    }
    t = t->next;
  }
}
/* lnum == 0 */
```

Топологическая сортировка на Си: основная функция

```
int main (void) {
  init_list ();
  sort_list ();
  return 0;
}
```

Дома. Что поменяется, если узлы идентифицируются не одним символом, а именем (строкой)? Сделайте нужные изменения в коде. Добавьте определение циклов в исходных данных.

Сортировка: постановка задачи

Сортировка — это упорядочение наборов однотипных данных, для которых определено отношение линейного порядка (например, $<$, «меньше») по возрастанию или по убыванию. Здесь будут рассматриваться целочисленные данные и отношение порядка $<$.

Различают *внешнюю* и *внутреннюю* сортировку. Рассматривается только внутренняя сортировка: сортируемый массив находится в основной памяти компьютера. Внешняя сортировка применяется к записям на внешних файлах.

Сортировка: что есть в Си

```
#include <stdlib.h>
void qsort (void *buf, size_t num, size_t size,
      int(*compare)(const void *, const void *));
```

Функция **qsort** сортирует (по возрастанию) массив с указателем **buf**, используя алгоритм быстрой сортировки Ч.Э.Р. Хоара, который считается одним из лучших алгоритмов сортировки общего назначения.

Параметр **num** задает количество элементов массива **buf**, параметр **size** — размер (в байтах) элемента массива **buf**. Параметр **int(*compare)(const void *,const void *)** задаёт правило сравнения элементов массива **num**. Функция сравнивает аргументы и возвращает:

- целое < 0, если $arg1 < arg2$,
- целое = 0, если $arg1 = arg2$,
- целое > 0, если $arg1 > arg2$.

Простейший алгоритм сортировки

Сведение сортировки к задаче нахождения максимального (минимального) из n чисел. Нахождение максимума n чисел (n сравнений). Числа содержатся в массиве **`int a[n];`**

```
max = a[0];
for (i = 1; i < n; i++)
  if (a[i] > max)
    max = a[i];
```

Алгоритм сортировки: находим максимальное из n чисел, получаем последний элемент отсортированного массива (n сравнений); находим максимальное из $n-1$ оставшихся чисел, получаем предпоследний элемент отсортированного массива (еще $n-1$ сравнений); и так далее.

Общее количество сравнений: $1+2+\ldots+n-1+n = n(n-1)/2$. Сложность алгоритма $O(n^2)$.

Три общих метода внутренней сортировки

- сортировка *обменами*: рассматриваются соседние элементы сортируемого массива и при необходимости меняются местами;
- сортировка *выборкой*: идея описана на предыдущем слайде;
- сортировка *вставками*: сначала сортируются два элемента массива, потом выбирается третий элемент и вставляется в нужную позицию относительно первых двух и т.д.

Сортировка обменами (пузырьком)

Общее количество сравнений (действий): $n(n-1)/2$, так как внешний цикл выполняется $(n-1)$ раз, а внутренний — в среднем $n/2$ раза.

```
void bubble_sort (int *a, int n) {
  int i, j, tmp;
  for (j = 1; j < n; ++j)
    for (i = n - 1; i >= j; --i) {
      if (a[i - 1] > a[i]) {
        tmp = a[i - 1];
        a[i - 1] = a[i];
        a[i] = tmp;
      }
    }
}
```

Сортировка вставками

Количество сравнений зависит от степени перемешанности массива a. Если массив a уже отсортирован, количество сравнений равно $n - 1$. Если массив a отсортирован в обратном порядке (наихудший случай), количество сравнений имеет порядок n^2.

```
void insert_sort (int *a, int n) {
  int i, j, tmp;

  for (j = 1; j < n; ++j) {
    tmp = a[j];
    for (i = j - 1; i >= 0 && tmp < a[i]; i--)
      a[i + 1] = a[i];
    a[i + 1] = tmp;
  }
}
```

Оценка сложности алгоритмов сортировки

Скорость сортировки определяется количеством сравнений и количеством обменов (обмены занимают больше времени). Эти показатели интересны для худшего и лучшего случаев, а также интересно их среднее значение.

Кроме скорости, оценивается «естественность» алгоритма сортировки: *естественным* считается алгоритм, который на уже отсортированном массиве работает минимальное время, а на не отсортированном работает тем дольше, чем больше степень его неупорядоченности.

Важным показателем является и объем дополнительной памяти для хранения промежуточных данных алгоритма. Для рекурсивных алгоритмов расход памяти связан с необходимостью сохранять в автоматической памяти (стеке) локальные переменные и параметры.

Оценка сложности алгоритмов сортировки

Теорема. Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений $C_S \geqslant O(n \log_2 n)$.

Доказательство. Сначала покажем, что

$$C_S \geqslant \log_2(n!) \tag{1}$$

Алгоритм S можно представить в виде двоичного дерева сравнений. Так как любая перестановка индексов рассматриваемого массива может быть ответом в алгоритме, она должна быть приписана хотя бы одному листу дерева сравнений. Таким образом, дерево сравнений будет иметь не менее $n!$ листьев.

Оценка сложности алгоритмов сортировки

Теорема. Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений $C_S \geqslant O(n \log_2 n)$.

Доказательство. Сначала покажем, что

$$C_S \geqslant \log_2(n!) \tag{1}$$

Для высоты h_m двоичного дерева с m листьями имеет место оценка: $h_m \geqslant \log_2 m$.

Любое двоичное дерево высоты h можно достроить до полного двоичного дерева высоты h 2^h листьев. Применив полученную оценку к дереву сравнений, получим искомую оценку (1).

Оценка сложности алгоритмов сортировки

Далее, применим к $\log_2 n!$ формулу Стирлинга

$$n! = \sqrt{2\pi n} n^n e^{-n} e^{\theta(n)},$$

где $|\theta(n)| \leqslant \frac{1}{12n}$. Подставляя и логарифмируя, имеем

$$\log_2 n! = \frac{1}{2}\log_2 2\pi n + n\log_2 n - n + \theta(n),$$

$$\log_2 n! \geqslant O(n \log_2 n).$$

Быстрая сортировка

```
static void QuickSort (int *a, int left, int right) {
  /* comp -- компаранд, i, j -- значения индексов */
  int comp, tmp, i, j;
  i = left; j = right;
  comp = a[(left + right)/2];
  do {
    while (a[i] < comp && i < right)
      i++;
    while (comp < a[j] && j > left)
      j--;
    if (i <= j) {
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
      i++, j--;
    }
  } while (i <= j);
```

Быстрая сортировка

```
static void QuickSort (int *a, int left, int right) {
  ...
  if (left < j)
    QuickSort (a, left, j);
  if (i < right)
    QuickSort (a, i, right);
}
```

Программа быстрой сортировки.

```
void qsort (int *a, int n) {
  QuickSort (a, 0, n - 1);
}
```

Нужно, чтобы значение компаранда было таким, чтобы он попал в середину результирующей последовательности. Мы пытаемся угадать, какой из элементов массива имеет такое значение. Чем лучше мы угадаем, тем быстрее выполнится алгоритм.

Быстрая сортировка

Покажем, что цикл `do-while` действительно строит нужное нам разбиение массива `a[]`.

- В процессе работы цикла индексы `i` и `j` не выходят за пределы отрезка `[left, right]`, так как в циклах `while` выполняются соответствующие проверки.
- В момент окончания работы цикла `do-while` $j \leqslant right$, так как части разбиения не могут быть пустыми: хотя бы один элемент массива `a[]` (в крайнем случае `a[right]`) содержится в правой части разбиения.
- Аналогично, в момент окончания работы цикла `do-while` $i \geqslant left$.
- В момент окончания работы цикла `do-while` любой элемент подмассива `a[left..j]` не больше любого элемента подмассива `a[i..right]`, что очевидно.

Быстрая сортировка. Пример разделения массива

Работа цикла `do-while` на примере: `5 3 2 6 4 1 3 7`.

- Пусть в качестве первого компаранда выбран первый элемент массива — 5 (`a[left]`).
  Во время первого прохода цикла `do-while` после выполнения обоих циклов `while` получим:
  `(5) 3 2 6 4 1 {3} 7` (в круглых скобках элемент с индексом i, в фигурных — элемент с индексом j).
- Поскольку $i < j$, элементы, выделенные скобками, нужно поменять местами: `3 (3) 2 6 4 {1} 5 7`.
- В результате второго прохода цикла `do-while` получим: до обмена — `3 3 2 (6) 4 {1} 5 7`;
  после обмена — `3 3 2 1 ({4}) 6 5 7`.
- Третий проход лишь увеличивает i.

Теперь массив a состоит из двух подмассивов `3 3 2 1 4` и `6 5 7`, причём $i = 5$, $j = 4$. Нужно рекурсивно применить метод к этим подмассивам.

Быстрая сортировка. Выбор компаранда

При выборе компаранда можно брать первый элемент, значение которого больше значения следующего элемента. Для результирующих подмассивов из примера компаранды заключены в квадратные скобки:
`3 [3] 2 1 4` и `[6] 5 7`.

Оценка времени работы быстрой сортировки (Θ-нотация).
Если $f(n)$ и $g(n)$ — некоторые функции, то запись $g(n) = \Theta(f(n))$ означает, что найдутся такие константы $c_1, c_2 > 0$ и такое n_0, что для всех $n \geqslant n_0$ выполняются соотношения

$$0 \leqslant c_1 f(n) \leqslant g(n) \leqslant c_2 f(n),$$

т.е. при больших n $f(n)$ хорошо описывает поведение $g(n)$.

Быстрая сортировка. Оценка времени работы

Время выполнения цикла `do-while` – $\Theta(n)$, где $n = right - left + 1$.

Для алгоритма QuickSort максимальное (наихудшее) время выполнения $T_{max}(n) = \Theta(n_2)$. Наихудшее время: при каждом Partition массив длины n разбивается на подмассивы длины 1 и $n - 1$.

Для $T_{max}(n)$ имеет место соотношение $T_{max}(n) = T_{max}(n-1) + \Theta(n)$. Очевидно, что $T_{max}(1) = \Theta(1)$. Следовательно,

$$T_{max}(n) = T_{max}(n-1) + \Theta(n) = n(n-1)/2 = \Theta(n^2).$$

Если исходный массив a отсортирован в порядке убывания, время его сортировки в порядке возрастания с помощью алгоритма QuickSort будет $\Theta(n^2)$.

Быстрая сортировка. Оценка времени работы

Минимальное и среднее время выполнения алгоритма QuickSort $T_{mean}(n) = \Theta(n \log n)$ с разными константами: чем ближе разбиение на подмассивы к сбалансированному, тем константы меньше.

Доказательство использует теорему о рекуррентных оценках из Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. М.: МЦНМО, 1999. ISBN 5-900916-37-5, с. 66-73.

Рекуррентное соотношение для минимального (наилучшего) времени сортировки $T_{min}(n)$ имеет вид

$$T_{min}(n) = 2T_{min}(n/2) + \Theta(n),$$

так как минимальное время получается тогда, когда на каждом шаге удается выбрать компаранд, который делит массив на два подмассива одинаковой длины $\lceil n/2 \rceil$. Применяя ту же теорему, получаем $T_{min}(n) = \Theta(n \log n)$.

Быстрая сортировка. Оценка времени работы

Рекуррентное соотношение для $T(n)$ в общем случае, когда на каждом шаге массив делится в отношении $q : (n - q)$, причем q равновероятно распределено между 1 и n, также можно решить и установить, что $T(n) = \Theta(n \log n)$ (та же книга, с. 160-164).

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:

- пустое дерево является двоичным деревом, но не является обычным деревом;
- двоичные деревья `(A(B,NULL))` и `(A(NULL,B))` различны, а обычные деревья — одинаковы.

Термины: узлы, ветви, корень, листья, высота.

Описание узла двоичного дерева на Си

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

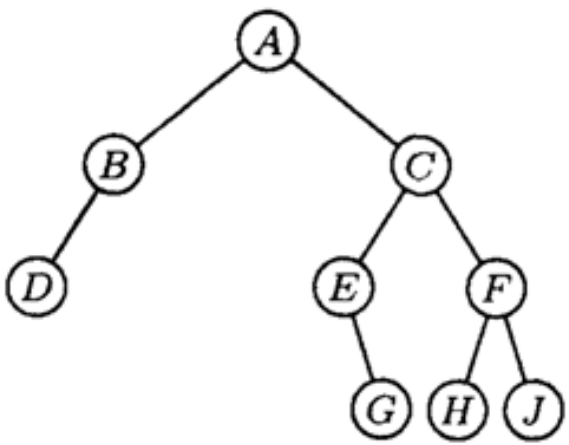


Рис. 1. Двоичное дерево

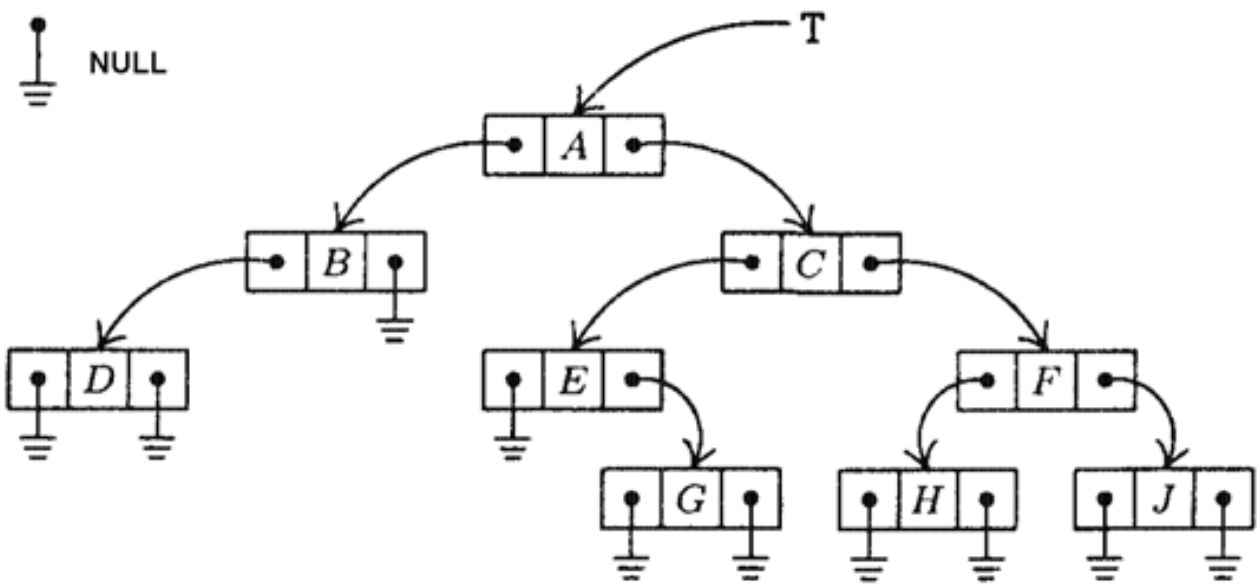


Рис. 2 Представление дерева с рис.1 в компьютере.

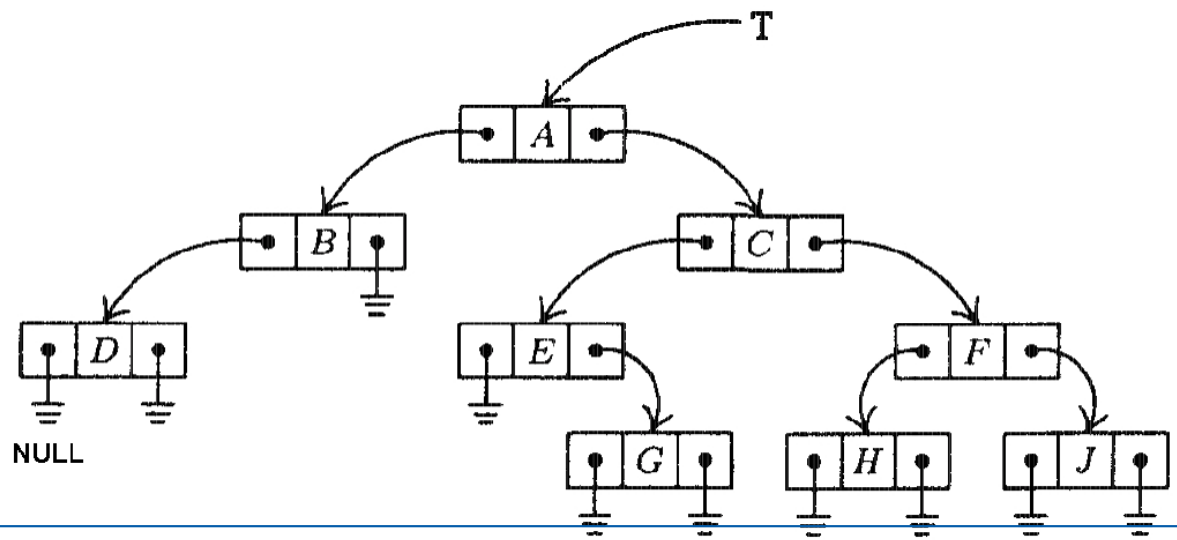
Способы обхода двоичного дерева

Обход в глубину в *прямом порядке*:

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

Порядок обработки узлов дерева: A B D C E G F H J.

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



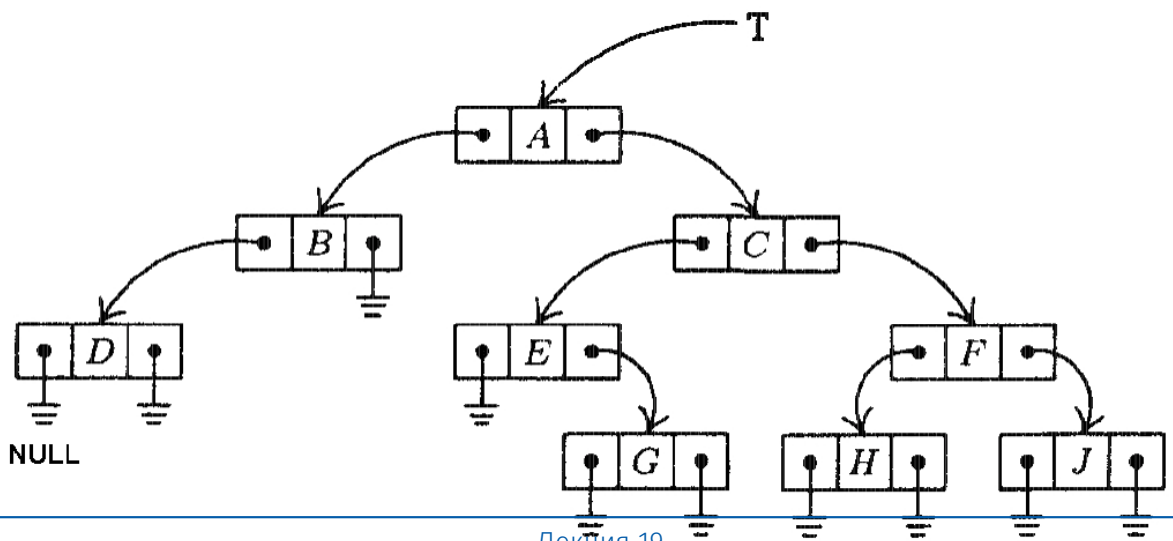
Способы обхода двоичного дерева

Обход в глубину в *обратном порядке*:

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

Порядок обработки узлов дерева: D B G E H J F C A.

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

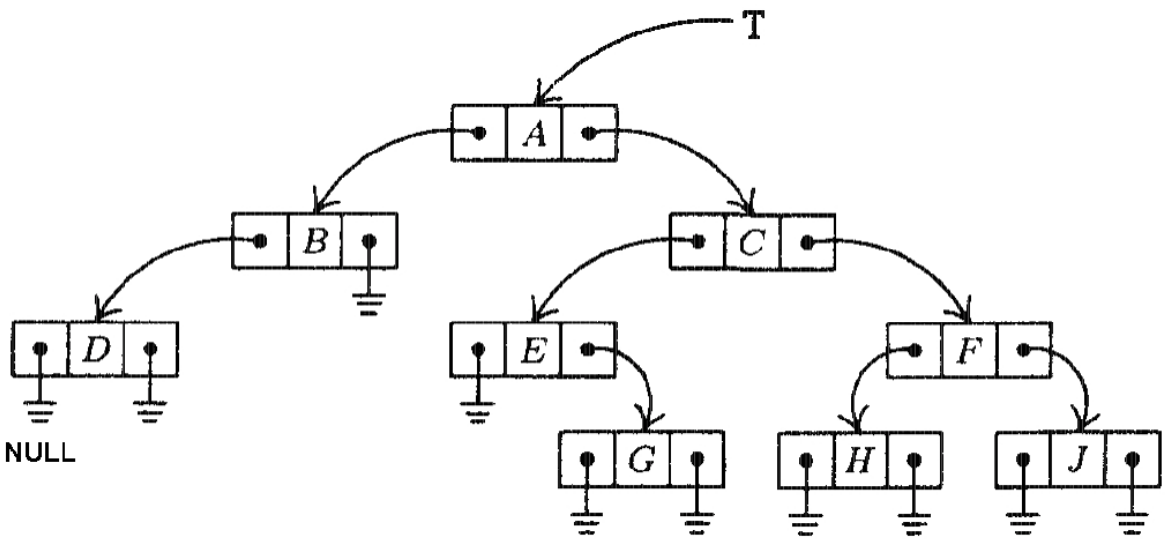


Способы обхода двоичного дерева

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

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

Порядок обработки узлов дерева: D B A E G C H F J.



Способы обхода двоичного дерева

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

Порядок обработки узлов дерева: A B C D E F G H J.